```
/*
    ModbusMaster.c

Description:

    This code uses the serial port of the KFlop+KAnalog to connect to the DirectAutomation Koyo
Click PLC (C0-00DD1-D)
    This is a low cost solution to add 8 remote inputs and 6 remote outputs.
    Virtual bits 48-55 are used for the inputs of the PLC.
    Virtual bits 56-61 are sent to the outputs of the PLC.
    It is easy to configure this code to use other virtual bits on the kflop or other memory
areas of the PLC.
    At 38400 baud (fastest allowed on the PLC) the inputs and outputs are read/written at
~40Hz. If you only need just
        input or just output, ~80hz.
    Note that the KFlop does not currently support parity. The PLC is connected through its
port 2 which is configured
        to use no parity, 38400 baud, 8 bit data, 1 stop bit.


Implementation:

    An array of 16bit registers (MBRegisters) is used to make Modbus commands easier to specify.

    RegLoad() and RegUnload() routines are used to marshal (move and format) data between
MBRegisters and
    KFlop memory.

    There are two lists of PLC commands: Connection and Monitor
    The connection list is used to read/write one time information, like SW version numbers.
    The monitor list is used for the continual IO calls.
    The Connection list is sent and then the Monitor list is looped through forever.
    A command timeout flags a disconnect, and the very next successful command starts back at
the start of the connection list.
    This allows a PLC to be stopped and restarted or disconnect-reconnect, and the ModbusMaster
will reinit.

    Console printing is designed to display information when anomalies occur, like disconnects
or unknown commands.

Pseudocode for main loop:

    When data will be written to the PLC, call RegLoad to marshal data from KFlop to MBRegisters.
    Build the command using data from MBRegisters.
    Send the command.
    Wait for PLC response.
        Resend for retryCount times.
            if failed, then flag disconnect.
    When the PLC responds, if data is received
        place read information into MBRegisters and call RegUnload to marshal data to KFlop
memory.
    Move on to the next command in the list.

*/

#include "KMotionDef.h"

#include "ModBusMaster.h"
```

```
unsigned short MBRegisters[N_MB_REGISTERS];

// Constants
int ModbusMaster_MaxRetry=3;
double ModbusMaster_Timeout=0.5; //seconds for no response for a send command
double ModbusMaster_ResponseTime; //seconds for last PLC response
double ModbusMaster_CommandSentTime;

// status and performance counters
double ModbusMaster_MonitorStartTime=0; // start of most recent monitor cycle
double ModbusMaster_MonitorCycleTime=0; // seconds to call all commands in Monitor list
int ModbusMaster_TallyConnections=0;    // Number of times Connection list has been sent
int ModbusMaster_TallyCommands=0;   // Commands since connection
int ModbusMaster_TallyRetries=0;    // Retries since connection

// statuses, counters, etc
int ModbusMaster_List=0;    // 0=connect, 1=monitor
int ModbusMaster_Connected=0;
int ModbusMaster_MonitorIndex=0;
int ModbusMaster_ConnectIndex=0;
int ModbusMaster_Idle=0;    // 0=idle, 1=await reply
int ModbusMaster_Retry=0;

double ModbusMaster_LastInTime=0;
double ModbusMaster_EndOfPacketWait=3.5*10/9600; // wait 3.5 characters after a packet
9600=baud rate)
char ModbusMaster_packetBuild[256]; // max length of modbus frame
int ModbusMaster_packetSize=0;

typedef enum
{
    MBERROR_NONE = 0,
    // Modbus codes; reported with Modbus error packet
    MBERROR_ILLEGAL_FUNCTION = 1,
    MBERROR_ILLEGAL_DATA_ADDRESS = 2,   // used
    MBERROR_ILLEGAL_DATA_VALUE = 3, // used
    MBERROR_SLAVE_DEVICE_FAILURE = 4,
    MBERROR_ACKNOWLEDGE = 5,
    MBERROR_SLAVE_DEVICE_BUSY = 6,
    MBERROR_NEGATIVE_ACKNOWLEDGE = 7,
    MBERROR_MEMORY_PARITY_ERROR = 8,
    // internal codes
    INTERROR_WRONG_DEVICE = 9,
    INTERROR_CHECKSUM = 10,
    INTERROR_TIMEOUT = 11,
} MBErrors;


typedef struct ModbusMaster_sCmds
{
    char *start; // has "dev,cmd,adrhi,adrlo,lenhi,lenlo" bytes of modbus command. Not data and
Not checksum.
    int len; // length of start string. commonly 6
    int reg; // reg# is the start index into the MBRegisters array for the command
} ModbusMaster_Cmds;

ModbusMaster_Cmds *ModbusMaster_SentPtr; // pointer to current command. used to send, resend,
```

```
interpret response.

ModbusMaster_Cmds ModbusMaster_ConnectList[] =
{
    // string is "dev,cmd,adrhi,adrlo,lenhi,lenlo" bytes of modbus command. bytelen, data, and
checksum are added.
    {"\x01\x04\xF0\x00\x00\x10", 6, 2}, // Collect PLC firmware info block MBRegisters[10] for
16 registers
    {0,0,0} // end flag
};

ModbusMaster_Cmds ModbusMaster_MonitorList[] =
{
    // string is "dev,cmd,adrhi,adrlo,lenhi,lenlo" bytes of modbus command. bytelen, data, and
checksum are added as necessary.
    {"\x01\x04\xE0\x00\x00\x01", 6, 0}, // Read inputs to MBRegisters[0]
    {"\x01\x10\xE2\x00\x00\x01", 6, 1}, // Write outputs from MBRegisters[1]
    {0,0,0} // end flag
};

void ModbusMaster_Init()
{
    printf("\nModbus Master Init\n");
    EnableRS232Cmds(RS232_BAUD_38400);
    DoRS232Cmds = FALSE;  // turn off processing RS232 input as commands
    ModbusMaster_LastInTime=Time_sec();
    ModbusMaster_EndOfPacketWait=3.5*10.0/38400; // wait 3.5 characters after a packet
    ModbusMaster_packetSize=0;

    ModbusMaster_Idle=0;
    ModbusMaster_SentPtr=&ModbusMaster_ConnectList[0];

    int c;
    for (c=0;c<N_MB_REGISTERS;c++)
        MBRegisters[c]=0;

    // make the register static arrays available to the other threads
    persist.UserData[PERSIST_MBREG_BLOCK_ADR]=(int)MBRegisters;
    //d printf("persist.UserData[%d]<=%08X\n",PERSIST_RWREG_BLOCK_ADR,MBRWRegisters); //debug
}

char* strncpy(char *dst,char* src,int len)
{
    int i;
    for (i=0;i<len;i++)
        dst[i]=src[i];
    return dst;
}

// marshal and move values read from PLC/Slave into MBRegisters to KFlop memory
void ModbusMaster_RegUnload()
{
    // Move 8 PLC inputs to virtual bits via MBRegisters[0]
    // Note use SetStateBit which is Atomic and Thread Safe

    int i;

    for (i=0; i<8; i++)
```

```
        SetStateBit(48+i,(MBRegisters[0]>>i)&1);  // 8 input bits
}

// marshal and move values to be sent to PLC/Slave into MBRegisters
void ModbusMaster_RegLoad()
{
    // Move 6 virtual bits to PLC outputs via MBRegisters[1]
    MBRegisters[1] = (VirtualBits>>8)&0x3F; // the six bits after the 8 input bits
}

static 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
} ;

static 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
} ;


unsigned short CRC16(unsigned char *puchMsg,unsigned short usDataLen)
{
    unsigned char uchCRCHi = 0xff;
    unsigned char uchCRCLo = 0xff;
    unsigned int uIndex;
```

```
    while(usDataLen--)
    {
        uIndex = uchCRCLo ^ *puchMsg++;
        uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex];
        uchCRCHi = auchCRCLo[uIndex];
    }
    return (uchCRCHi<<8|uchCRCLo);
}

void ModbusMaster_NextCmd(MBErrors ecode)
{
    if (ecode) printf("ModbusMaster_NextCmd(%d)\n",ecode); //debug
    ModbusMaster_Idle=0; // ready to send a new command
    if (INTERROR_TIMEOUT==ecode)
    {
        if (ModbusMaster_List)
            ModbusMaster_Connected=0;
    }
    if (ModbusMaster_List)
    {
        ModbusMaster_MonitorIndex++;
        if (!ModbusMaster_MonitorList[ModbusMaster_MonitorIndex].start)
        {
            ModbusMaster_MonitorIndex=0;
            ModbusMaster_MonitorCycleTime=Time_sec()-ModbusMaster_MonitorStartTime;
            ModbusMaster_MonitorStartTime=Time_sec();
        }
    }
    else
    {
        ModbusMaster_ConnectIndex++;
        if (!ModbusMaster_ConnectList[ModbusMaster_ConnectIndex].start)
            ModbusMaster_List=1;    // continue monitor list, do not restart here
    }
    if (INTERROR_TIMEOUT!=ecode&&0==ModbusMaster_Connected)
    {
        ModbusMaster_Connected=1;
        ModbusMaster_List=0;
        ModbusMaster_ConnectIndex=0;

        ModbusMaster_TallyConnections++;
        ModbusMaster_TallyCommands=0;
        ModbusMaster_TallyRetries=0;
    }

    if (!ModbusMaster_List)
        ModbusMaster_SentPtr=&ModbusMaster_ConnectList[ModbusMaster_ConnectIndex];
    else
        ModbusMaster_SentPtr=&ModbusMaster_MonitorList[ModbusMaster_MonitorIndex];
}

void ModbusMaster_Send(int verbose)
{
    // send the command currently pointed to by ModbusMaster_SentPtr
    //  printf("ModbusMaster_Send(%d)\n",verbose);

    char *chp;
    int x;
```

```
    unsigned char *xp;

    if (!ModbusMaster_List)
        printf("List:%d ConnectIndex:%d MonitorIndex:%d\n",
                ModbusMaster_List,ModbusMaster_ConnectIndex,ModbusMaster_MonitorIndex);

    if (!ModbusMaster_SentPtr->start)
    {
        printf("ModbusMaster_Send: Tried to execute at end list\n");
        ModbusMaster_NextCmd(MBERROR_NONE);
    }

    strncpy(ModbusMaster_packetBuild,ModbusMaster_SentPtr->start,ModbusMaster_SentPtr->len);
    chp=&ModbusMaster_packetBuild[ModbusMaster_SentPtr->len];
    switch(ModbusMaster_packetBuild[1])
    {
        case 0x10: // RW Write
            ModbusMaster_RegLoad();
            *chp++=ModbusMaster_packetBuild[5]*2;
            for (x=0;x<ModbusMaster_packetBuild[5];x++)
            {
                *chp++=(MBRegisters[x+ModbusMaster_SentPtr->reg]>>8)&0xFF;
                *chp++=MBRegisters[x+ModbusMaster_SentPtr->reg]&0xFF;
            }
            break;
        case 0x03:  // RO Read
        case 0x04: // RW Read
            break;
        default:
            printf("Unexpected default: ModbusMaster_Send(), Function %d\n",
ModbusMaster_packetBuild[1]); //debug
            break;
    }
    int csum=CRC16(ModbusMaster_packetBuild,chp-ModbusMaster_packetBuild);
    *chp++=csum&0xFF;
    *chp++=(csum>>8)&0xFF;

    if (verbose) printf("Tx:"); //debug
    for (xp=ModbusMaster_packetBuild;xp<chp;xp++)
    {
        RS232_PutChar(*xp);
        if (verbose) printf("%02x;",*xp); //debug
    }
    if (verbose) printf("\n"); //debug

    ModbusMaster_LastInTime=Time_sec();
    ModbusMaster_Idle=1;
}

MBErrors Process_Data(unsigned char *Buffer, unsigned char Count)
{
    int regndx;
    int cnt;
    int x;
    unsigned short CRC = (((Buffer[Count-1]<<8)&0xFF00)|(Buffer[Count-2]&0xFF));    // Received
CRC
    unsigned short Recalculated_CRC = CRC16(Buffer,Count-2);    // Computed CRC
    if(Recalculated_CRC != CRC)
```

```
    {
        printf("Count %d\n",Count);
        printf("Checksum: Theirs:%04X Mine:%04X, %d chars\n",CRC,Recalculated_CRC,Count); //debug
        return INTERROR_CHECKSUM;
    }

    //d printf("Packet %d\n",Function);    //debug

    switch(Buffer[1])
    {
        case 0x03:
        case 0x04:
            regndx=ModbusMaster_SentPtr->reg;
            cnt=Buffer[2];
            for (x=0;x<cnt;x+=2)
                MBRegisters[regndx++]=((Buffer[3+x]<<8)&0xFF00)|(Buffer[4+x]&0x00FF);
            ModbusMaster_RegUnload();
            break;
        case 0x10:
            // no action on successful write
            break;
        default:
            printf("Unexpected default: Process_Data(), Buffer[1]=%d\n",Buffer[1]);
            break;
    }

    return MBERROR_NONE;    //We made it to the end, return
}



void ModbusMaster_Monitor()
{
    char c;

    if (pRS232RecIn != pRS232RecOut)
    {
        ModbusMaster_LastInTime=Time_sec();
        while (pRS232RecIn != pRS232RecOut) // data in buffer
        {
            c=RS232_GetChar();
            if (ModbusMaster_packetSize<255)
                ModbusMaster_packetBuild[ModbusMaster_packetSize++]=c;
            //d printf("%02x,",c&0xFF); //debug
        }
    }
    else
    {
        if (ModbusMaster_LastInTime+ModbusMaster_EndOfPacketWait<Time_sec() &&
ModbusMaster_packetSize)
        {
            int rtrn=Process_Data(ModbusMaster_packetBuild,ModbusMaster_packetSize);
            ModbusMaster_packetSize=0;  // ready for next packet
            if (!rtrn)
            {
                ModbusMaster_TallyCommands++;
                ModbusMaster_NextCmd(rtrn);
                return;
```

```
            }
            printf("Error=%d, %f\n",rtrn,ModbusMaster_LastInTime+ModbusMaster_EndOfPacketWait-
Time_sec());  //debug
            //d printf("\n",c); //debug
        }
        if (ModbusMaster_LastInTime+ModbusMaster_Timeout<Time_sec())    // retry test
        {
            ModbusMaster_Retry++;
            ModbusMaster_TallyRetries++;
            printf("ModbusMaster_Retry:%d\n",ModbusMaster_Retry); //debug
            if (ModbusMaster_Retry>ModbusMaster_MaxRetry)
            {
                //d printf("Failed Monitor message %d\n",ModbusMaster_MonitorIndex); //debug
                //ModbusMaster_ConnectIndex=0;  // reset connection
                ModbusMaster_NextCmd(INTERROR_TIMEOUT);
            }
            else
            {
                ModbusMaster_Send(1);
            }
        }
    }
}


void ModbusMaster_Loop()
{
    //d printf("ModbusMaster_Loop: ModbusMaster_Idle=%d\n",ModbusMaster_Idle); //debug
    if(!ModbusMaster_Idle)
    {
        int x;
        //if (ModbusMaster_LastInTime+ModbusMaster_EndOfPacketWait>Time_sec())
        //  ;
        //printf("ModbusMaster_Loop: ModbusMaster_Idle=%d\n",ModbusMaster_Idle); //debug
        ModbusMaster_Send(0);
        //for (x=0;x<16;x++)
        //{
        //  printf("%04d ",(unsigned)MBRegisters[x]);
        //  if ((x&7)==7) printf("\n");
        //}
        ModbusMaster_Retry=0;
    }
    else
        ModbusMaster_Monitor();
}



main()
{
    ModbusMaster_Init();
    int reportsecs=10;
    double starttime;;
    double MonitorStartTime=0;  // start of most recent monitor cycle
    double MonitorCycleTime=0;  // seconds to call all commands in Monitor list
    int TallyConnections=0; // Number of times Connecion list has been sent
    int TallyCommands=0;    // Commands since connection
    int TallyRetries=0; // Retries since connection
```

```
    starttime=Time_sec();
    TallyCommands=ModbusMaster_TallyCommands;

    while(1)
    {
        ModbusMaster_Loop();
        if (starttime+reportsecs<Time_sec())
        {
            printf("\nSeconds: %d\n",reportsecs);
            printf("ModbusMaster_MonitorCycleTime=%f (%f/s)\n",ModbusMaster_MonitorCycleTime,1.0
/ModbusMaster_MonitorCycleTime);
            printf("ModbusMaster_TallyCommands/s=%0.1f\n",(ModbusMaster_TallyCommands-
TallyCommands)/(double)reportsecs);
            printf("ModbusMaster_TallyConnections=%d\n",ModbusMaster_TallyConnections);
            printf("ModbusMaster_TallyRetries=%d\n",ModbusMaster_TallyRetries);

            starttime=Time_sec();
            TallyCommands=ModbusMaster_TallyCommands;
        }
    }
}
```