CODE: Ether to RS485

```
// Using SPI bus
// Hardware description:
// ENC28J60 Ethernet controller
// ~CS connects inverted version of RB9 (pin 18)
// MOSI connects to RP7/SDO1 (output fn 7) (pin 16)
// SCL connectes to RP6/SCLK1OUT (output fn 8) (pin 15)
// MISO connects to  to RB8 (pin 17)

//------------------------------------------------------------------------------
// Device includes and assembler directives
//------------------------------------------------------------------------------

#include <p33FJ128MC802.h>
#define FCY 40000000UL                            // instruction cycle rate
#include <libpic30.h>                            // __delay32
#include <stdio.h>
#include<string.h>

//#include "enc28j60.h"

#define BAUD_19200 129                           // brg for low-speed, 40 MHz clock
#define BAUD_91000 26
#define BAUD_250000 9


char* msg;
unsigned char datapacket[685];
unsigned char rx_buffer[3078];
int rx_buffer_index[6];
int ii = 0;
int k = 0;
int uart2_index;
unsigned char uart2_rcv[4];
int universe_no = 0;

// ENC28J60 Ethernet functions for 33FH128MC802
// Buffer is configured as follows
// Receive buffer starts at 0x0000 (bottom 6666 bytes of 8K space)
// Transmit buffer at 01A0A (top 1526 bytes of 8K space)

// --------------------------------------------------------------------------------
//  Defines
// --------------------------------------------------------------------------------

// Pins
#define PIN_ETHER_CS LATBbits.LATB9

#define bool unsigned char

#define ETHER_UNICAST        0x80
#define ETHER_BROADCAST      0x01
#define ETHER_MULTICAST      0x02
#define ETHER_HASHTABLE      0x04
#define ETHER_MAGICPACKET    0x08
#define ETHER_PATTERNMATCH   0x10

#define ETHER_HALFDUPLEX     0x00
#define ETHER_FULLDUPLEX     0x40

// Ether registers
#define ERDPTL          0x00
```

```
#define ERDPTH          0x01
#define EWRPTL          0x02
#define EWRPTH          0x03
#define ETXSTL          0x04
#define ETXSTH          0x05
#define ETXNDL          0x06
#define ETXNDH          0x07
#define ERXSTL          0x08
#define ERXSTH          0x09
#define ERXNDL          0x0A
#define ERXNDH          0x0B
#define ERXRDPTL  0x0C
#define ERXRDPTH  0x0D
#define ERXWRPTL  0x0E
#define ERXWRPTH  0x0F
#define EIE           0x1B
#define EIR           0x1C
#define RXERIF  0x01
#define TXERIF  0x02
#define TXIF    0x08
#define PKTIF   0x40
#define ESTAT       0x1D
#define CLKRDY  0x01
#define TXABORT 0x02
#define ECON2           0x1E
#define PKTDEC  0x40
#define ECON1       0x1F
#define RXEN    0x04
#define TXRTS   0x08
#define ERXFCON         0x38
#define EPKTCNT     0x39
#define MACON1          0x40
#define MARXEN  0x01
#define RXPAUS  0x04
#define TXPAUS  0x08
#define MACON2          0x41
#define MARST   0x80
#define MACON3          0x42
#define FULDPX  0x01
#define FRMLNEN 0x02
#define TXCRCEN 0x10
#define PAD60   0x20
#define MACON4          0x43
#define MABBIPG         0x44
#define MAIPGL          0x46
#define MAIPGH          0x47
#define MACLCON1  0x48
#define MACLCON2  0x49
#define MAMXFLL         0x4A
#define MAMXFLH         0x4B
#define MICMD       0x52
#define MIIRD   0x01
#define MIREGADR    0x54
#define MIWRL       0x56
#define MIWRH       0x57
#define MIRDL       0x58
#define MIRDH       0x59
#define MAADR1      0x60
#define MAADR0      0x61
#define MAADR3      0x62
#define MAADR2      0x63
#define MAADR5      0x64
#define MAADR4      0x65
```

```
#define MISTAT      0x6A
#define MIBUSY  0x01
#define ECOCON      0x75

// Ether phy registers
#define PHCON1       0x00
#define PDPXMD 0x0100
#define PHCON2       0x10
#define HDLDIS 0x0100
#define PHLCON       0x14

// ------------------------------------------------------------------------------
//  Globals
// ------------------------------------------------------------------------------

static unsigned char nxt_pckt_l = 0x00;
static unsigned char nxt_pckt_h = 0x00;
unsigned char sequence_id = 1;
unsigned long sum;
unsigned char mac_addr[6] = {1,2,3,4,5,6};
unsigned char ipv4_addr[4];
int process_flag;
int rs_485goer=0;
// ------------------------------------------------------------------------------
//  Macros
// ------------------------------------------------------------------------------

#define LOBYTE(x) ((x) & 0xFF)
#define HIBYTE(x) (((x) >> 8) & 0xFF)

// ------------------------------------------------------------------------------
//  Structures
// ------------------------------------------------------------------------------

// C30 compiler is little endian
// Network byte order is big endian
// Must unsigned charerpret unsigned ints in reverse order

struct enc28j60_frame // 4-bytes
{
  unsigned int size;
  unsigned int status;
  unsigned char data;
} *enc28j60;

struct ether_frame // 14-bytes
{
  unsigned char dest_add[6];
  unsigned char source_add[6];
  unsigned int frame_type;
  unsigned char data;
} *ether;

struct _ip // minimum 20 bytes
{
  unsigned char rev_size;
  unsigned char type_of_service;
  unsigned int length;
  unsigned int id;
  unsigned int flags_and_ofs;
  unsigned char ttl;
  unsigned char protocol;
  unsigned int header_checksum;
  unsigned char source_ip[4];
  unsigned char dest_ip[4];
} *ip;

struct _icmp
{
  unsigned char type;
  unsigned char code;
  unsigned int check;
  unsigned int id;
  unsigned int seq_no;
  unsigned char data;
} *icmp;

struct _arp
{
  unsigned int hard_type;
  unsigned int prot_type;
  unsigned char hard_size;
  unsigned char prot_size;
  unsigned int op;
  unsigned char source_add[6];
  unsigned char source_ip[4];
  unsigned char dest_add[6];
  unsigned char dest_ip[4];
} *arp;

struct _udp // 8 bytes
{
  unsigned int source_port;
  unsigned int dest_port;
  unsigned int length;
  unsigned int check;
  unsigned char  data;
} *udp;

struct _acn
{
unsigned char preamble[2];
unsigned char postamble[2];
unsigned char acn_packet_identifier[12];
unsigned char rlp_fl[2];
unsigned char rlp_vector[4];
unsigned char rlp_CID[16];
unsigned char flp_fl[2];
unsigned char flp_vector[4];
unsigned char flp_priority;
unsigned char flp_reserved[2];
unsigned char flp_sequence_number;
unsigned char flp_options;
unsigned char flp_universe[2];
unsigned char dmp_fl[2];
unsigned char dmp_vector;
unsigned char dmp_address_and_data;
unsigned char dmp_first_property_address[2];
unsigned char dmp_address_increment[2];
unsigned char dmp_property_value_count[2];
unsigned char dmp_property_values[514];
}*acn;
```

```
// ------------------------------------------------------------------------------
//  Functions
// ------------------------------------------------------------------------------

void spi_write(unsigned char data)
{
  SPI1BUF = data;
}

unsigned char spi_read()
{
  while(!SPI1STATbits.SPIRBF);
  return SPI1BUF;
}

void ether_cs_on()
{
  PIN_ETHER_CS = 0;
  __delay32(100);
}

void ether_cs_off()
{
  __delay32(100);
  PIN_ETHER_CS = 1;
}

void ether_write_reg(unsigned char reg, unsigned char data)
{
  ether_cs_on();
  spi_write(0x40 | (reg & 0x1F));
  spi_read();
  spi_write(data);
  spi_read();
  ether_cs_off();
}

unsigned char ether_read_reg(unsigned char reg)
{
  unsigned char data;
  ether_cs_on();
  spi_write(0x00 | (reg & 0x1F));
  spi_read();
  spi_write(0);
  data = spi_read();
  ether_cs_off();
  return data;
}

void ether_set_reg(unsigned char reg, unsigned char mask)
{
  ether_cs_on();
  spi_write(0x80 | (reg & 0x1F));
  spi_read();
  spi_write(mask);
  spi_read();
  ether_cs_off();
}

void ether_clear_reg(unsigned char reg, unsigned char mask)
{
  ether_cs_on();
  spi_write(0xA0 | (reg & 0x1F));
  spi_read();
  spi_write(mask);
  spi_read();
  ether_cs_off();
}

void ether_set_bank(unsigned char reg)
{
  ether_clear_reg(ECON1, 0x03);
  ether_set_reg(ECON1, reg >> 5);
}

void ether_write_phy(unsigned char reg, unsigned int data)
{
  ether_set_bank(MIREGADR);
  ether_write_reg(MIREGADR, reg);
  ether_write_reg(MIWRL, data & 0xFF);
  ether_write_reg(MIWRH, (data >> 8) & 0xFF);
}

unsigned int ether_read_phy(unsigned char reg)
{
  unsigned int data, data2;
  ether_set_bank(MIREGADR);
  ether_write_reg(MIREGADR, reg);
  ether_write_reg(MICMD, ether_read_reg(MICMD) | MIIRD);
  __delay_us(1024);
  while ((ether_read_reg(MISTAT) | MIBUSY) != 0);
  ether_write_reg(MICMD, ether_read_reg(MICMD) & ~MIIRD);
  data = ether_read_reg(MIRDL);
  data2 = ether_read_reg(MIRDH);
  data |= (data2 << 8);
  return data;
}

void ether_write_mem_start()
{
  ether_cs_on();
  spi_write(0x7A);
  spi_read();
}

void ether_write_mem(unsigned char data)
{
  spi_write(data);
  spi_read();
}

void ether_write_mem_stop()
{
  ether_cs_off();
}

void ether_read_mem_start()
{
  ether_cs_on();
  spi_write(0x3A);
  spi_read();
}

unsigned char ether_read_mem()
{
  spi_write(0);
  return spi_read();
}

void ether_read_mem_stop()
{
  ether_cs_off();
}

void ether_spi_select()
{
  // disable spi
  SPI1STATbits.SPIEN = 0;

  // 28j60 expects low idle on clk
  // data in to 28j60 clocked on rising edge
  // data out of 28j60 clocked on falling edge
  // all transistions on falling edge (from active to inactive)
  // smp=bit9 sample on middle of data out (0) or end of data out (1)
  // ckp=bit6 clock idle low (0) or high (1)
  // cke=bit8 MOSI updated on idle-to-active (0) or active-to-idle (1)
  // enable spi clock and data out
  // 8-bit mode, master mode
  // 16:1 primary, 1:1 secondary prescale
  // clk idle high, change dout on idle to active edge
  //  SPI1CON1 = 0x007D;
  // 16:1 primary prescale, 1:1 secondary prescale
  SPI1CON1bits.PPRE = 2;
  SPI1CON1bits.SPRE = 6;
  // 8 bit
  SPI1CON1bits.MODE16 = 0;
  // sample on rising edge since 28j60 updates on falling edge
  SPI1CON1bits.SMP = 0;
  // idle state of clock is low
  SPI1CON1bits.CKP = 0;
  // data out updates on active-to-idle edge (falling) since 28j60 samples on rising edge
  SPI1CON1bits.CKE = 1;
  // master mode
  SPI1CON1bits.MSTEN = 1;

  // frame and enhanced buffer disabled
  SPI1CON2 = 0;

  // enable spi, continue spi on idle, clr overflow
  SPI1STATbits.SPIROV = 0;
  SPI1STATbits.SPISIDL = 0;
  SPI1STATbits.SPIEN = 1;
}

// Initializes ethernet device
// Uses order suggested in Chapter 6 of datasheet except 6.4 OST which is first here
void ether_init(unsigned char mode)
{
  // configure spi for 28j60 mode
  ether_spi_select();

  // make sure that oscillator start-up timer has expired
  while ((ether_read_reg(ESTAT) & CLKRDY) == 0) {}

  // disable transmission and reception of packets
  ether_clear_reg(ECON1, RXEN);
  ether_clear_reg(ECON1, TXRTS);

  // initialize receive buffer space
```

```
ether_set_bank(ERXSTL);
ether_write_reg(ERXSTL, LOBYTE(0x0000));
ether_write_reg(ERXSTH, HIBYTE(0x0000));
ether_write_reg(ERXNDL, LOBYTE(0x1A09));
ether_write_reg(ERXNDH, HIBYTE(0x1A09));

// initialize receiver write and read ptrs
// at startup, will write from 0 to 1A08 only and will not overwrite rd ptr
ether_write_reg(ERXWRPTL, LOBYTE(0x0000));
ether_write_reg(ERXWRPTH, HIBYTE(0x0000));
ether_write_reg(ERXRDPTL, LOBYTE(0x1A09));
ether_write_reg(ERXRDPTH, HIBYTE(0x1A09));
ether_write_reg(ERDPTL, LOBYTE(0x0000));
ether_write_reg(ERDPTH, HIBYTE(0x0000));

// setup receive filter
// always check CRC, use OR mode
ether_set_bank(ERXFCON);
ether_write_reg(ERXFCON, (mode | 0x20) & 0xBF);
// bring mac out of reset
ether_set_bank(MACON2);
ether_write_reg(MACON2, 0);

// enable mac rx, enable pause control for full duplex
ether_write_reg(MACON1, TXPAUS | RXPAUS | MARXEN);

// enable padding to 60 bytes (no runt packets)
// add crc to tx packets, set full or half duplex
if ((mode & ETHER_FULLDUPLEX) != 0)
  ether_write_reg(MACON3, FULDPX | FRMLNEN | TXCRCEN | PAD60);
else
  ether_write_reg(MACON3, FRMLNEN | TXCRCEN | PAD60);

// leave MACON4 as reset

// set maximum rx packet size
ether_write_reg(MAMXFLL, LOBYTE(1518));
ether_write_reg(MAMXFLH, HIBYTE(1518));

// set back-to-back unsigned charer-packet gap to 9.6us
if ((mode & ETHER_FULLDUPLEX) != 0)
  ether_write_reg(MABBIPG, 0x15);
else
  ether_write_reg(MABBIPG, 0x12);

// set non-back-to-back unsigned charer-packet gap registers
ether_write_reg(MAIPGL, 0x12);
ether_write_reg(MAIPGH, 0x0C);

// leave collision window MACLCON2 as reset

// setup mac address
ether_set_bank(MAADR0);
ether_write_reg(MAADR5, mac_addr[0]);
ether_write_reg(MAADR4, mac_addr[1]);
ether_write_reg(MAADR3, mac_addr[2]);
ether_write_reg(MAADR2, mac_addr[3]);
ether_write_reg(MAADR1, mac_addr[4]);
ether_write_reg(MAADR0, mac_addr[5]);

// initialize phy duplex
if ((mode & ETHER_FULLDUPLEX) != 0)
  ether_write_phy(PHCON1, PDPXMD);
  else
    ether_write_phy(PHCON1, 0);

  // disable phy loopback if in half-duplex mode
  ether_write_phy(PHCON2, HDLDIS);

  // set LEDA (link status) and LEDB (tx/rx activity)
  // stretch LED on to 40ms (default)
  ether_write_phy(PHLCON, 0x0472);

  // enable reception
  ether_set_reg(ECON1, RXEN);
}

// Returns TRUE if packet received
unsigned char ether_kbhit()
{
  return ((ether_read_reg(EIR) & PKTIF) != 0);
}

// Returns up to max_size characters in data buffer
// Returns number of bytes copied to buffer
// Contents written are 16-bit size, 16-bit status, payload excl crc
unsigned int ether_get_packet(unsigned char data[], unsigned int max_size)
{
  unsigned int i = 0, size, tmp;

  // enable read from FIFO buffers
  ether_read_mem_start();

  // get next pckt information
  nxt_pckt_l = ether_read_mem();
  nxt_pckt_h = ether_read_mem();

  // calc size
  // don't return crc, instead return size + status, so size is correct
  size = ether_read_mem();
  data[i++] = size;
  tmp = ether_read_mem();
  data[i++] = tmp;
  size |= (tmp << 8);

  // copy status + data
  if (size > max_size)
    size = max_size;
  while (i < size)
    data[i++] = ether_read_mem();

  // end read from FIFO buffers
  ether_read_mem_stop();

  // advance read ptr
  ether_set_bank(ERXRDPTL);
  ether_write_reg(ERXRDPTL, nxt_pckt_l); // hw ptr
  ether_write_reg(ERXRDPTH, nxt_pckt_h);
  ether_write_reg(ERDPTL, nxt_pckt_l); // dma rd ptr
  ether_write_reg(ERDPTH, nxt_pckt_h);

  // decrement packet counter so that PKTIF is maunsigned charained correctly
   ether_set_reg(ECON2, PKTDEC);

  return size;
}
```

```
// Returns TRUE is rx buffer overflowed after correcting the problem
unsigned char ether_is_overflow()
{
  unsigned char err;
  err = (ether_read_reg(EIE) & RXERIF) != 0;
  if (err)
    ether_clear_reg(EIE, RXERIF);
  return err;
}

// Writes a packet
bool ether_put_packet(void *data, unsigned int size)
{
  unsigned int i;

  // clear out any tx errors
  if ((ether_read_reg(EIR) & TXERIF) != 0)
  {
    ether_clear_reg(EIR, TXERIF);
    ether_set_reg(ECON1, TXRTS);
    ether_clear_reg(ECON1, TXRTS);
  }

  // set DMA start address
  ether_set_bank(EWRPTL);
  ether_write_reg(EWRPTL, LOBYTE(0x1A0A));
  ether_write_reg(EWRPTH, HIBYTE(0x1A0A));

  // start FIFO buffer write
  ether_write_mem_start();

  // write control byte
  ether_write_mem(0);

  // write data
  for (i = 0; i < size; i++)
  {
    ether_write_mem(*(unsigned char*) data);
    data++;
  }

  // stop write
  ether_write_mem_stop();

  // request transmit
  ether_write_reg(ETXSTL, LOBYTE(0x1A0A));
  ether_write_reg(ETXSTH, HIBYTE(0x1A0A));
  ether_write_reg(ETXNDL, LOBYTE(0x1A0A+size));
  ether_write_reg(ETXNDH, HIBYTE(0x1A0A+size));
  ether_clear_reg(EIR, TXIF);
  ether_set_reg(ECON1, TXRTS);

  // wait for completion
  while ((ether_read_reg(ECON1) & TXRTS) != 0);

  // determine success
 return ((ether_read_reg(ESTAT) & TXABORT) == 0);
}

// Calculate sum of words
// Must use ether_crc to complete 1's compliment addition
void word_sum(void *data, unsigned int size_in_bytes)
```

```
{
  unsigned int i;
  unsigned char phase = 0;
  unsigned int data_temp;
  for (i = 0; i < size_in_bytes; i++)
  {
    if (phase)
    {
      data_temp = *(unsigned char*)data;
      sum += data_temp << 8;
    }
    else
      sum += *(unsigned char*)data;
    phase = 1 - phase;
    data++;
  }
}

// Completes 1's compliment addition by folding carries back unsigned charo field
unsigned int ether_crc()
{
  unsigned int result;
  // this is based on rfc1071
  while ((sum >> 16) > 0)
    sum = (sum & 0xFFFF) + (sum >> 16);
  result = sum & 0xFFFF;
  return ~result;
}

// Converts from host to network order and vice versa
unsigned int htons(unsigned int value)
{
  return ((value & 0xFF00) >> 8) + ((value & 0x00FF) << 8);
}

#define ntohs htons

// Determines whether packet is IP datagram
unsigned char ether_is_ip(unsigned char data[])
{
  unsigned char ok;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  ok = (ether->frame_type == 0x0008);
  if (ok)
  {
    sum = 0;
    word_sum(&ip->rev_size, (ip->rev_size & 0xF) * 4);
    ok = (ether_crc() == 0);
  }
  return ok;
}

unsigned char ether_is_MAC(unsigned char data[])
{
  unsigned char ok;
  int i;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  for (i = 0; i < 6; i++)
  {
    if(ether->dest_add[i] != mac_addr[i])
```

```
      return 0;
  }

  return 1;
}



// Determines whether packet is unicast to this ip
// Must be an IP packet
bool ether_is_ip_unicast(unsigned char data[])
{
  unsigned char i = 0;
  bool ok = 1;//TRUE;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  while (ok && (i < 4))
  {
    ok = (ip->dest_ip[i] == ipv4_addr[i]);
    i++;
  }
  return ok;
}

// Determines whether packet is ping request
// Must be an IP packet
unsigned char ether_is_ping_req(unsigned char data[])
{
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  icmp = (void*)ip + ((ip->rev_size & 0xF) * 4);
  return (ip->protocol == 0x01 && icmp->type == 8);
}

// Sends a ping response given the request data
void ether_send_ping_resp(unsigned char data[])
{
  unsigned char i, tmp;
  unsigned int icmp_size;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  icmp = (void*)ip + ((ip->rev_size & 0xF) * 4);
  // swap source and destination fields
  for (i = 0; i < 6; i++)
  {
    tmp = ether->dest_add[i];
    ether->dest_add[i] = ether->source_add[i];
    ether->source_add[i] = tmp;
  }
  for (i = 0; i < 4; i++)
  {
    tmp = ip->dest_ip[i];
    ip->dest_ip[i] = ip ->source_ip[i];
    ip->source_ip[i] = tmp;
  }
  // this is a response
  icmp->type = 0;
  // calc icmp checksum
  sum = 0;
  word_sum(&icmp->type, 2);
  icmp_size = ntohs(ip->length);
  icmp_size -= 24; // sub ip header and icmp code, type, and check
  word_sum(&icmp->id, icmp_size);
  icmp->check = ether_crc();
  // send packet
  ether_put_packet(ether, 14 + ntohs(ip->length));
}

// Determines whether packet is ARP
#define ARP_INVALID 0
#define ARP_REQUEST 1
#define ARP_RESPONSE 2
unsigned char ether_is_arp(unsigned char data[])
{
  unsigned char ok;
  unsigned char i = 0;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  arp = (void*)&ether->data;
  ok = (ether->frame_type == 0x0608);
  while (ok && (i < 4))
  {
    ok = (arp->dest_ip[i] == ipv4_addr[i]);
    i++;
  }
  return ok;
}

// Sends an ARP response given the request data
void ether_send_arp_resp(unsigned char data[])
{
  unsigned char i, tmp;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  arp = (void*)&ether->data;
  // set op to response
  arp->op = 0x0200;
  // swap source and destination fields
  for (i = 0; i < 6; i++)
  {
    arp->dest_add[i] = arp->source_add[i];
    ether->dest_add[i] = ether->source_add[i];
    ether->source_add[i] = arp->source_add[i] = mac_addr[i];
  }
  for (i = 0; i < 4; i++)
  {
    tmp = arp->dest_ip[i];
    arp->dest_ip[i] = arp->source_ip[i];
    arp->source_ip[i] = tmp;
  }
  // send packet
  ether_put_packet(ether, 42);
}

// Sends an ARP request
void ether_send_arp_req(unsigned char data[], unsigned char ip[])
{
  unsigned char i;
  ether = (void*)data;
  arp = (void*)&ether->data;
  // fill ethernet frame
  for (i = 0; i < 6; i++)
  {
     ether->dest_add[i] = 0xFF;
     ether->source_add[i] = mac_addr[i];
  }
  ether->frame_type = 0x0608;
  // fill arp frame
  arp->hard_type = 0x0100;
  arp->prot_type = 0x0008;
  arp->hard_size = 6;
  arp->prot_size = 4;
  arp->op = 0x0100;
  for (i = 0; i < 6; i++)
  {
    arp->source_add[i] = mac_addr[i];
    arp->dest_add[i] = 0xFF;
  }
  for (i = 0; i < 4; i++)
  {
    arp->source_ip[i] = ipv4_addr[i];
    arp->dest_ip[i] = ip[i];
  }
  // send packet
  ether_put_packet(data, 42);
}

// Determines whether packet is UDP datagram
// Must be an IP packet
unsigned char ether_is_udp(unsigned char data[])
{
  unsigned char ok;
  unsigned int tmp_int;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  udp = (void*)ip + ((ip->rev_size & 0xF) * 4);
  ok = (ip->protocol == 0x11);
  if (ok)
  {
    // 32-bit sum over pseudo-header
    sum = 0;
    word_sum(ip->source_ip, 8);
    tmp_int = ip->protocol;
    sum += (tmp_int & 0xff) << 8;
    word_sum(&udp->length, 2);
    // add udp header and data
    word_sum(udp, ntohs(udp->length));
    ok = (ether_crc() == 0);
  }
  return ok;
}

// Gets pounsigned charer to UDP payload of frame
unsigned char* ether_get_udp_data(unsigned char data[])
{
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  udp = (void*)ip + ((ip->rev_size & 0xF) * 4);
  return &udp->data;
}

void ether_calc_ip_checksum()
{
  // 32-bit sum over ip header
```

```
  sum = 0;
  word_sum(&ip->rev_size, 10);
  word_sum(ip->source_ip, ((ip->rev_size & 0xF) * 4) - 12);
  ip->header_checksum = ether_crc();
}

// Send responses to a udp datagram
// destination port, ip, and hardware address are extracted from provided data
// uses destination port of received packet as destination of this packet
void ether_send_udp_resp(unsigned char data[], unsigned char* udp_data, unsigned char
udp_size)
{
  unsigned char *copy_data;
  unsigned char i, tmp;
  unsigned int tmp_int;
  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  udp = (void*)&ether->data + ((ip->rev_size & 0xF) * 4);
  // swap source and destination fields
  for (i = 0; i < 6; i++)
  {
    tmp = ether->dest_add[i];
    ether->dest_add[i] = ether->source_add[i];
    ether->source_add[i] = tmp;
  }
  for (i = 0; i < 4; i++)
  {
    tmp = ip->dest_ip[i];
    ip->dest_ip[i] = ip->source_ip[i];
    ip->source_ip[i] = tmp;
  }
  // set source port of resp will be dest port of req
  // dest port of resp will be left at source port of req
  // unusual nomenclature, but this allows a different tx
  // and rx port on other machine
  udp->source_port = udp->dest_port;
  // adjust lengths
  ip->length = htons(((ip->rev_size & 0xF) * 4) + 8 + udp_size);
  // 32-bit sum over ip header
  sum = 0;
  word_sum(&ip->rev_size, 10);
  word_sum(ip->source_ip, ((ip->rev_size & 0xF) * 4) - 12);
  ip->header_checksum = ether_crc();
  udp->length = htons(8 + udp_size);
  // copy data
  copy_data = &udp->data;
  for (i = 0; i < udp_size; i++)
    copy_data[i] = udp_data[i];
  // 32-bit sum over pseudo-header
  sum = 0;
  word_sum(ip->source_ip, 8);
  tmp_int = ip->protocol;
  sum += (tmp_int & 0xff) << 8;
  word_sum(&udp->length, 2);
  // add udp header except crc
  word_sum(udp, 6);
  word_sum(&udp->data, udp_size);
  udp->check = ether_crc();

  // send packet with size = ether + udp hdr + ip header + udp_size
  ether_put_packet(ether, 22 + ((ip->rev_size & 0xF) * 4) + udp_size);
}
```

```
unsigned int ether_get_id()
{
  return htons(sequence_id);
}

void ether_inc_id()
{
  sequence_id++;
}

// Determines if the IP address is valid
bool ether_is_valid_ip()
{
  return ipv4_addr[0] || ipv4_addr[1] || ipv4_addr[2] || ipv4_addr[3];
}

// Sets IP address
void ether_set_ip_address(unsigned char a, unsigned char b,
  unsigned char c, unsigned char d)
{
  ipv4_addr[0] = a;
  ipv4_addr[1] = b;
  ipv4_addr[2] = c;
  ipv4_addr[3] = d;
}


unsigned char process_data(unsigned char data[])
{
  int i;
  unsigned char ok;
  unsigned int tmp_int;
  unsigned char temp[12] = {0x41, 0x53, 0x43, 0x2d, 0x45, 0x31, 0x2e, 0x31, 0x37, 0x00,
0x00, 0x00};
  unsigned char rlp_temp_vector[4] = {0x00, 0x00, 0x00, 0x04};
  unsigned char flp_temp_vector[4] = {0x00, 0x00, 0x00, 0x02};


  enc28j60 = (void*)data;
  ether = (void*)&enc28j60->data;
  ip = (void*)&ether->data;
  udp = (void*)ip + ((ip->rev_size & 0xF) * 4);
  acn = (void*)&udp->data;

  ok = (acn->preamble[0] == 0x00);
  ok = (acn->preamble[1] == 0x10);
  if(ok){
  ok = (acn->postamble[0] == 0x00);
  ok = (acn->postamble[1] == 0x00);}
  if(ok)
  {
    for(i=0;i<12;i++){
      if(acn->acn_packet_identifier[i] != temp[i])
          return 0;
     }
  }


  if(ok)
  {
    for(i=0;i<4;i++){
      if(acn->rlp_vector[i] != rlp_temp_vector[i])
```

```
          return 0;
    }
  }

  if(ok)
  ok = ((acn -> rlp_fl[1]) == 0x72);

  if(ok)
  {
    for(i=0;i<4;i++){
      if(acn->flp_vector[i] != flp_temp_vector[i])
         return 0;
     }
 }

  if(ok)
 ok = ((acn -> flp_fl[1]) == 0x72);

  if(ok)
  ok = (acn->dmp_vector == 0x02);


  if(ok)
  ok = ((acn -> dmp_fl[0]) == 0x72);

  if(ok)
  ok = (acn -> dmp_address_and_data == 0xa1);

  if(ok){
  ok = (acn -> dmp_first_property_address[0] == 0x00);
  ok = (acn -> dmp_first_property_address[1]== 0x00);}
  if(ok){
  ok = (acn -> dmp_address_increment[0] == 0x01);
ok = (acn -> dmp_address_increment[1] == 0x00);
            }
  return 1;
}



unsigned char clear_flag;

unsigned char is_universe_no()
{
   unsigned char ok = 1;
if(datapacket[159]== LOBYTE(universe_no)&& datapacket[160]== HIBYTE(universe_no))
return ok;

uart2_puts("\n\rACN packet, unmatched universes!");

return !ok;

}

//-------------------------------------------------------------------------------
// Subroutines
//-------------------------------------------------------------------------------

// Initialize Hardware
void init_hw()
{
  LATAbits.LATA3 = 1;                        // write 1 to shutdown latch
  LATBbits.LATB9 = 0;
  TRISAbits.TRISA3 = 0;
  TRISAbits.TRISA4 = 0;
  TRISBbits.TRISB5 = 0;                      // make A0 pin an output
  TRISBbits.TRISB9 = 0;                      // make CS pin an output

  RPOR3bits.RP6R = 8;                        // assign SCLK1OUT to RP6
  RPOR2 = 7;                                 // assign SDO1 to RP7
  RPINR20bits.SDI1R = 8;                     // assign SDI1 to RP8

  PLLFBDbits.PLLDIV =38;                     // pll feedback divider = 40;
  CLKDIVbits.PLLPRE = 0;                     // pll pre divider = 2
  CLKDIVbits.PLLPOST = 0;                    // pll post divider = 2
  TRISBbits.TRISB10 = 0;
  TRISBbits.TRISB13 = 0;
  //TRISBbits.TRISB11 = 0;
  RPOR6bits.RP12R = 3;
  RPOR5bits.RP10R = 5;
  TRISBbits.TRISB7 = 1;                      // INT0 digital input

  RPINR19bits.U2RXR = 11;
  INTCON2bits.INT0EP = 1; // INT0 Polarity on neg edge
  IFS0bits.INT0IF = 0;    // Clear flag
  IEC0bits.INT0IE = 1;    // Enable INT0

}

int h;
int store[638];
int j;
int flag = 0;
int rx_count = 0;
int tx_count = 0;
//unsigned char tx_buffer[1026];

void __attribute__((interrupt, no_auto_psv)) _INT0Interrupt (void)
{
IFS0bits.INT0IF = 0;    // Clear flag
IEC0bits.INT0IE = 0;    // Enable INT0

//ether_set_bank(EPKTCNT);
//clear_flag = ether_read_reg(EPKTCNT);
//ether_write_reg(EPKTCNT,clear_flag - 1);
ether_clear_reg(ESTAT,0x80);
ether_clear_reg(EIE,0x80);
ether_clear_reg(EIE,0x40);
ether_get_packet(datapacket,684);
if (ether_is_ip(datapacket)){
      if(ether_is_MAC(datapacket)){
            if (ether_is_udp(datapacket)){
                            if(process_data(datapacket) ){
                              if(is_universe_no()){
                                    rx_buffer_index[k++] = ii;
                                    rx_count++;
                                    for(j=0;j<513;j++){
                                                rx_buffer[j+ii] = datapacket[171+j];
                                                //tx_buffer[j+ii] = datapacket[171+j];
                                    }
                                    ii = j + ii;
                                    if(rx_count == 2){
                                          send_rs485(rx_buffer_index[tx_count]);
                                          tx_count++;
                                                rx_count = 0;
```

```
                                            if(tx_count == 6)
                                                  tx_count =0;
                                      }
                                            if(k == 6){
                                                  k=0;
                                            ii = 0;
                                      }

                                      }
                          }
                    }
              }

        }
        clear_flag++;
        IFS0bits.INT0IF = 0;    // Clear flag
        IEC0bits.INT0IE = 1;    // Enable INT0

        //ether_set_bank(EPKTCNT);
        //clear_flag = ether_read_reg(EPKTCNT);
        //ether_write_reg(EPKTCNT,clear_flag - 1);
        ether_clear_reg(ESTAT,0x80);
        ether_set_reg(EIE,0x80);
        ether_set_reg(EIE,0x40);
}



void uart1_init(unsigned int baud_rate)
{
  U1BRG = baud_rate;
  U1MODE = 0x8001;
  U1STA = 0x0400;
 }


void uart2_init(unsigned int baud_rate)
{
  U2BRG = baud_rate;
  U2MODE = 0x8000;
  U2STA = 0x0400;
  IEC1bits.U2RXIE = 1;
  IFS1bits.U2RXIF = 0;
 }



void uart1_putc(char dmx_char)
{
    // make sure buffer is empty
    while(U1STAbits.UTXBF);
    // write character
    U1TXREG = dmx_char;

}

void uart2_putc(char str)
{

    // make sure buffer is empty
    while(U2STAbits.UTXBF);
    // write character
    U2TXREG = str;
```

```
}


void uart2_puts(char str[])
{
  int i;
  for (i = 0; i < strlen(str); i++)
  {
    // make sure buffer is empty
    while(U2STAbits.UTXBF);
    // write character
    U2TXREG = str[i];
  }
}

void uart1_puts(char str[])
{
  int i;
  for (i = 0; i < strlen(str); i++)
  {
    // make sure buffer is empty
    while(U1STAbits.UTXBF);
    // write character
    U1TXREG = str[i];
  }
}

char uart2_getc()
{
  // clear out any overflow error condition
  if (U2STAbits.OERR == 1)
    U2STAbits.OERR = 0;
  // wait until character is ready
  while(!U2STAbits.URXDA);
  return U2RXREG;
}


void __attribute__((interrupt, no_auto_psv)) _U2RXInterrupt (void)
{
    //int temp;
    IFS1bits.U2RXIF=0;
    IEC1bits.U2RXIE = 0;
    uart2_rcv[uart2_index++]=U2RXREG;
    if(uart2_index==4 && uart2_rcv[3]==13)
    {
      uart2_index=0;
      uart2_puts("\n\rTwo digits have been entered, enter new universe number if wish to
change");
        universe_no = atoi(uart2_rcv);
    }
    IFS1bits.U2RXIF = 0;
    IEC1bits.U2RXIE = 1;

}


void send_rs485(int tx_index)
{
  int i;
  rs_485goer++;
  LATBbits.LATB13 = 1;
  uart1_init(BAUD_91000);
  uart1_putc(0x00);
  while(!U1STAbits.TRMT);
  uart1_init(BAUD_250000);
  uart1_putc(0x00);
  while(!U1STAbits.TRMT);
  for(i = tx_index;i<tx_index +513;i++){
      uart1_putc(rx_buffer[i]);
      while(!U1STAbits.TRMT);
  }
  LATBbits.LATB13 = 0;
}



//--------------------------------------------------------------------------------
// Main
//--------------------------------------------------------------------------------

int main(void)
{

  int tx_packet_no = 0;
  int j;
  char *ptr;

  //unsigned char data[128];
  init_hw();

  LATBbits.LATB5 = 1;
  __delay32(10000);
  LATBbits.LATB5 = 0;

  uart2_init(BAUD_19200);
  __delay32(4000000);

  __delay32(10);
  //TRISBbits.TRISB8 = 1;
  LATBbits.LATB9 = 1;
  __delay32(10);

  ether_spi_select();
  __delay32(4000000);
  //ether_set_bank();
  ether_clear_reg(ESTAT,0x80);
  ether_set_reg(EIE,0x80);
  ether_set_reg(EIE,0x40);
  IFS0bits.INT0IF = 0;    // Clear flag
  IEC0bits.INT0IE = 1;
  uart2_puts("Ready !");
  uart2_puts("\n\rEnter the two digits of the universe number and press the return
key:");
  uart2_puts(&uart2_rcv);

  // init ethernet interface
  ether_init(ETHER_UNICAST | ETHER_BROADCAST | ETHER_HALFDUPLEX);
  ether_set_ip_address(192,168,1,2);

  // flash phy leds
  ether_write_phy(PHLCON, 0x0880);
  __delay32(16000000);
  ether_write_phy(PHLCON, 0x0990);
  __delay_ms(250);
  while(1);
  return 0;
}
```