```
/*
 * Program to answer requests on the celestron AUX serial bus like the
 * GPS module from Celstron.
 *
 * in order to avoid the bus errors in communication with the motor
 * control (Error-16 and Error-17) the pin 8 and 9 are used
 * for AUX Communication.
 *
 * to read the GPS messages the HW input RX0 is used, because no input
 * from the serial is expected - output (for debugging purposes) can
 * then still be send to the USB port of the Arduino.
 *
 * CAUTION: as this is used for the programming the Arduino shield,
 * it MUST BE DISCONNECTED during the loading of a sketch to hte board!
 *
 */
#include "define.h"
#include <SoftwareSerial.h>
#include <TinyGPS++.h>
#include <EEPROM.h>

#define SERIAL_DBG

SoftwareSerial   celestron_aux(8,9);

TinyGPSPlus gps;
TinyGPSCustom fixQuality(gps, "GPGGA", 6); // 0=invalid, 1=GPS, 2=DGPS,
etc...
TinyGPSCustom satellitesInView(gps, "GPGSV", 3);

const double GPS_MULT_FACTOR = 46603.37778;  // = 2^24 / 360

// Packet structure on Celetron AUX port
#define PK_MAX_LEN 12
unsigned char packet[PK_MAX_LEN];   // Is 12 enough?
enum pk_state { PREAMBLE_WAIT, LENGTH_WAIT, DATA, CKSUM, DONE, VALID };
enum pk_state pkstate;
int pklen;
int pkidx;
int16_t cksum_accumulator;

void setup()
{
// Celestron Code set-up
  pkstate = PREAMBLE_WAIT;
  pklen = 0;
  pkidx = 0;

// serial set-up

  celestron_aux .begin(19200); // AUX bus runs at 19200

  Serial        .begin(9600);
```

```
#ifdef SERIAL_DBG
  Serial.println("\nReady");
#endif
};


void loop()
{
  uint8_t c;
  char cc;
  int32_t lat, lng;
  uint8_t *latBytePtr, *lngBytePtr;

  // Read, forward and echo chars from gps_ser
  while (Serial.available()) {
    cc = Serial.read();
    gps.encode(cc);  // feed the stream into the GPS parser
  }

  // Read and echo chars from celestron_aux
  while (celestron_aux.available()) {
    c = celestron_aux.read();
#ifdef SERIAL_DBG
    if (c==0x3b) Serial.write('\n');
    Serial.print( c, HEX ); // for debugging purposes
    Serial.write(' ');
#endif
    packet_decode(c);
  };

  // check if the packet is valid
  if (pkstate != VALID) return;

  // check if the packet is for me
  if (packet[2] != DEV_GPS ) {
    pkstate = 0;
    pkidx = 0;
    pklen = 0;
    return;
  }
  // YES - it's for me
#ifdef SERIAL_DBG
    Serial.print( "- GPS - " ); // for debugging purposes
#endif

  uint8_t dest = packet[1];

  switch(packet[3])
  {
    case GPS_LINKED:
    case GPS_TIME_VALID:
#ifdef SERIAL_DBG
    Serial.println("gps_LINKED/Time_valid");
```

```
#endif
    if (fixQuality.value() > 0)
      pk_send(dest, packet[3], 1);
    else
      pk_send(dest, packet[3], 0);
    break;

    case GPS_GET_TIME:
    pk_send(dest, GPS_GET_TIME, gps.time.hour(), gps.time.minute(),
gps.time.second());
#ifdef SERIAL_DBG
    Serial.println("gps_get_time");
#endif
    break;

    case GPS_GET_HW_VER:
#ifdef SERIAL_DBG
    Serial.println("gps_get_hw_ver");
#endif
    pk_send(dest, GPS_GET_HW_VER, GPS_HW_VER);
    break;

    case GPS_GET_YEAR:
    pk_send(dest, GPS_GET_YEAR, (gps.date.year() >> 8) & 0xff, gps.date.
year() & 0xff);
#ifdef SERIAL_DBG
    Serial.println("gps_get_year");
#endif
    break;

    case GPS_GET_DATE:
    pk_send(dest, GPS_GET_DATE, gps.date.month(), gps.date.day());
#ifdef SERIAL_DBG
    Serial.println("gps_get_date");
#endif
    break;

    case GPS_GET_LAT:
        lat = (int32_t) (gps.location.lat() * GPS_MULT_FACTOR);
        latBytePtr = (uint8_t*)&lat;
#ifdef SERIAL_DBG
        Serial.println("gps_get_lat");
#endif
        pk_send(dest, GPS_GET_LAT, latBytePtr[2], latBytePtr[1],
latBytePtr[0]);
        break;

    case GPS_GET_LONG:
        lng = (int32_t) (gps.location.lng() * GPS_MULT_FACTOR);
        lngBytePtr = (uint8_t*)&lng;
#ifdef SERIAL_DBG
        Serial.println("gps_get_long");
#endif
```

```
        pk_send(dest, GPS_GET_LONG, lngBytePtr[2], lngBytePtr[1],
lngBytePtr[0]);
        break;

    case GPS_GET_SAT_INFO:
//        String satellitesInViewString(satellitesInView.value());
//        pk_send(dest, GPS_GET_SAT_INFO, satellitesInViewString.toInt(),
gps.satellites.value());
    break;

    case GPS_GET_RCVR_STATUS:
    break;

    case GPS_GET_COMPASS:
    break;

    case GPS_GET_VER:
    pk_send(dest, GPS_GET_VER, 0, 1);  // Version 0.1
#ifdef SERIAL_DBG
    Serial.print(" gps_get_ver\n");
#endif
    break;
  }

  pkstate = PREAMBLE_WAIT;
  pklen = 0;
  pkidx = 0;
}

void packet_decode(int8_t c)
{
  switch (pkstate)
  {
    case PREAMBLE_WAIT:
    if (c == 0x3b || c==0x76) {
      pkstate = LENGTH_WAIT;
    }
    break;

    case LENGTH_WAIT:
    if (c < PK_MAX_LEN) {
      pklen = c;
      packet[0] = c;
      pkidx = 1;
      pkstate = DATA;
    }
    else
      pkstate = PREAMBLE_WAIT;
    break;

    case DATA:
    packet[pkidx] = c;
    pkidx++;
```

```
    if (pkidx == pklen + 1)
      pkstate = CKSUM;
    break;

    case CKSUM:
    if (pk_checksum(c))
      pkstate = VALID;
    else
      pkstate = PREAMBLE_WAIT;
    break;
  }
}

bool pk_checksum(int8_t target)
{
  int sum = 0;
  for (int i = 0; i <= pklen; i++) sum += packet[i];
  int8_t chk = (-sum) & 0xff;
  return (target == chk);
}

inline void cksum_init()
{
  cksum_accumulator = 0;
}

inline void cksum_update(uint8_t b)
{
  cksum_accumulator += b;
}

inline int8_t cksum_final()
{
  return (-cksum_accumulator) & 0xff;
}

// Send a 1-byte response
void pk_send(uint8_t dest, uint8_t id, uint8_t byte0)
{
  cksum_init();
  // Send preamble
  celestron_aux.write(0x3b);
  // Send length 4
  cksum_update(0x04);
  celestron_aux.write(0x04);
  // Send src
  cksum_update(DEV_GPS);
  celestron_aux.write((uint8_t)DEV_GPS);
  // Send dest
  cksum_update(dest);
  celestron_aux.write(dest);
  // Send message id
  cksum_update(id);
```

```
  celestron_aux.write(id);
  // Send byte0
  cksum_update(byte0);
  celestron_aux.write(byte0);
  // Send checksum
  celestron_aux.write(cksum_final());
}

// Send a 2-byte response
void pk_send(uint8_t dest, uint8_t id, uint8_t byte0, uint8_t byte1)
{
  cksum_init();
  // Send preamble
  celestron_aux.write(0x3b);
  // Send length 5
  cksum_update(0x05);
  celestron_aux.write(0x05);
  // Send src
  cksum_update(DEV_GPS);
  celestron_aux.write((uint8_t)DEV_GPS);
  // Send dest
  cksum_update(dest);
  celestron_aux.write(dest);
  // Send message id
  cksum_update(id);
  celestron_aux.write(id);
  // Send byte0
  cksum_update(byte0);
  celestron_aux.write(byte0);
  // Send byte1
  cksum_update(byte1);
  celestron_aux.write(byte1);
  // Send checksum
  celestron_aux.write(cksum_final());
}

// Send a 3-byte response
void pk_send(uint8_t dest, uint8_t id, uint8_t byte0, uint8_t byte1,
uint8_t byte2)
{
  cksum_init();
  // Send preamble
  celestron_aux.write(0x3b);
  // Send length 6
  cksum_update(0x06);
  celestron_aux.write(0x06);
  // Send src
  cksum_update(DEV_GPS);
  celestron_aux.write((uint8_t)DEV_GPS);
  // Send dest
  cksum_update(dest);
  celestron_aux.write(dest);
  // Send message id
```

```
  cksum_update(id);
  celestron_aux.write(id);
  // Send byte0
  cksum_update(byte0);
  celestron_aux.write(byte0);
  // Send byte1
  cksum_update(byte1);
  celestron_aux.write(byte1);
  // Send byte2
  cksum_update(byte2);
  celestron_aux.write(byte2);
  // Send checksum
  celestron_aux.write(cksum_final());
}
```