```
/*
  Dwe_Heater_Pilot

  created 04 Jan 2026
  by ARo

*/
//#include "define.h" // included below
#define GPS_GET_LAT     0x01
#define GPS_GET_LONG    0x02
#define GPS_GET_DATE    0x03
#define GPS_GET_YEAR    0x04
#define GPS_GET_SAT_INFO  0x07
#define GPS_GET_RCVR_STATUS 0x08
#define GPS_GET_TIME    0x33
#define GPS_TIME_VALID  0x36
#define GPS_LINKED      0x37
#define GPS_GET_HW_VER  0x55
#define GPS_GET_COMPASS 0xa0
#define GPS_GET_VER     0xfe

#define GPS_N   0x0b
#define GPS_NE  0x09
#define GPS_E   0x0d
#define GPS_SE  0x0c
#define GPS_S   0x0e
#define GPS_SW  0x06
#define GPS_W   0x07
#define GPS_NW  0x03

#define GPS_HW_VER   0xab    // Motorola HW version

#define DEV_MAIN  0x01
#define DEV_HC    0x04
#define DEV_HC1   0x0D
#define DEV_AZM   0x10
#define DEV_ALT   0x11
#define DEV_FOCUS 0x12  // Fousser ID
#define DEV_DEW   0x17  // Smart Dew Heater Controller
#define DEV_GPS   0xb0  // GPS Device
#define DEV_RTC   0xb2  // Realtime Clock
#define DEV_WIFI  0xb5  // WiFi
#define DEV_SSSWI 0xb8
#define DEV_SSAG  0xb9  // StarSense Autoguider
#define DEV_DEWBB 0xbb  // Dew Controller ID during detection

#define RTC_GET_LAT   0x01
#define RTC_GET_LONG  0x02
#define RTC_SET_LAT   0x81
#define RTC_SET_LONG  0x82
#define RTC_GET_DATE  0x03
#define RTC_GET_YEAR  0x04
#define RTC_GET_TIME  0x33
#define RTC_SET_DATE  0x83
#define RTC_SET_YEAR  0x84
#define RTC_SET_TIME  0xb1

#define DEW_QUERY_INPUT_POWER          0x00
#define DEW_GET_LED_BRIGHTNESS         0x02
#define DEW_SET_INPUT_LIMIT            0x03
#define DEW_QUERY_INPUT_LIMITS         0x04
#define DEW_BB_OPCODE_05               0x05
#define DEW_GET_NUM_PORTS              0x10
#define DEW_QUERY_PORT                 0x11
#define DEW_QUERY_HEATER               0x12 // channel 0,1; Numbering is weired with more
than 2
```

```
#define DEW_DEW_UNKNOWN_13            0x13
#define DEW_ENABLE_PORT               0x14
#define DEW_QUERY_ENV_SENSORS         0x18 // get Temp, Dewpoint, rel Hum.
#define DEW_SET_AUTO_AGGR             0x16
#define DEW_SET_MANUAL_PWM            0x17
#define DEW_RECALIBRATE_ENV_SENSOR    0x19
#define DEW_QUERY_ENV_CALIBRATING     0x1a  // "is calibrated?"
#define DEW_SET_LED_BRIGHTNESS        0x20
#define DEW_DEW_UNKNOWN_21            0x21 // CWPI during startup

// All devices ops:
#define DEV_PROGRAM_ENA     0x8a
#define DEV_GET_VERSION     0xfe
#define DEV_GET_MODEL       0x05
#define DEV_BAD_OP          0xf0  // Devices return 'f0 xx' when 'xx' is an unsupported
opcode
// end define.h

#include "Adafruit_INA3221.h"
#include "Adafruit_SHT31.h"
#include <ADS1015_WE.h>
#include <Wire.h>
#include <math.h>
#include <EEPROM.h>
#include <SoftwareSerial.h>

#define DEBUG

// Packet structure on Celetron AUX port
#define PK_MAX_LEN 16
unsigned char packet[PK_MAX_LEN]; // Is 16 enough?
enum pk_state { PREAMBLE_WAIT, LENGTH_WAIT, DATA, CKSUM, DONE, VALID };
enum pk_state pkstate;
int pklen;
int pkidx;
int16_t cksum_accumulator;

// Create the Serial port for the AUX bus
SoftwareSerial  celestron_aux(8,9);
// Create an INA3221 object
Adafruit_INA3221 ina3221;
// Create an ADS1015 object
ADS1015_WE adc = ADS1015_WE(0x48);
// Create SHT31 Object
Adafruit_SHT31 sht31 = Adafruit_SHT31();

// Paramters for the Dew Heater that shall be saved
//uint8_t aggression[] = {5,5};   // medium agression of 0 to 10
//uint8_t heating[]    = {64,64}; // manual quarter heating of 0 to 255
uint8_t delta        = 2;       // tube alway 2 degrees warmer than dew point
//uint8_t led_brightness = 128;   // dew heater LED
//uint8_t enabled_12V_port = 0; // 12V enabled cache enable status
double dew_input_current_limit = 3.0; // configurable power limit

/* These constants won't change. They're used to give names to the pins used
     Channel 1:
     PWM     -> Arduino Nano Pin  10
     ADC     -> ADS1x15 channel   0
     CURRENT -> INA3221  channel  0

     Channel 2:
     PWM     -> Arduino Nano Pin  11
     ADC     -> ADS1x15 channel   1
     CURRENT -> INA3221  channel  1

*/
```

```
const int ee_aggression[]    = {0, 1};
const int ee_heating[]       = {2, 3};
const int ee_delta           = 4;
const int ee_led_brightness  = 5;
const int ee_enabled_12V_port= 6;
const int ee_dew_input_currrent_limit = 7*sizeof(int);

const int temp_adc_idx[]     = {ADS1015_COMP_0_GND, ADS1015_COMP_1_GND}; // Thermistor
channels
const int PWM_Pin[]          = {10, 11}; // PWM pins to controll heater
const int current_idx[]      = {1, 2}; // channels of INA3221
const int recalibration_counter = 512;
const int num_channels = 2; // only 2 channels for the dew heater

// hold the actual status of the peripherials (DewRings, Thermistors)
int therm[num_channels]   ={0,0};
int dewring[num_channels] ={0,0};

//
// SETUP
//
void setup() {
// use the (internal) LED (pin 13 ) to show if we have dewrings connected
  pinMode(LED_BUILTIN, OUTPUT);
// Set up Serial Communication towards USB
  Serial.begin(115200);
  while(!Serial) delay(10); // wait Serial to settle
  Serial.println("\n\nDew Heater Control starting up.");

// prepare pin for PWM output
  pinMode(PWM_Pin[0], OUTPUT);
  pinMode(PWM_Pin[1], OUTPUT);

// set up the I2C bus
  Wire.begin();
  delay(100);
// for INA3221 Current read out
  if (!ina3221.begin(0x40, &Wire))  // can use other I2C addresses or buses
    Serial.println("Failed to find INA3221 chip");
  else {
    Serial.println("INA3221 Found!");
    ina3221.setAveragingMode(INA3221_AVG_16_SAMPLES);
    // Set shunt resistances for all INA3221 channels to 0.1 ohms
    for (uint8_t i = 0; i < 3; i++) {
      ina3221.setShuntResistance(i, 0.1);
    }
  };

// for SHT31 Ambient Temp & Humidity Readout
  if (! sht31.begin(0x44))    // Set to 0x45 for alternate i2c addr
    Serial.println("Couldn't find SHT31");
  else Serial.println("SHT31 found!");
  sht31.heater(false); // should be off when measuring

// for ADC readout (Dew ring PTCs and 12V power measurement)
  if(!adc.init(true)) // passing true will tell the lib that an ADS1015 is used
    Serial.println("ADS1015 not connected!");
  else Serial.println("ADC1015 found!");
  adc.setVoltageRange_mV(ADS1015_RANGE_6144);

//check for the peripherials that are actually present
  delay(200);
  for (int i=0; i<num_channels; i++) {
    if(check_thermistor(i)) {
      therm[i] = 1;
      Serial.print("Found Thermistor Channel: ");
```

```
      Serial.println(i);
    } else therm[i]=0;

    if(check_dewring(i)) {
      dewring[i] = 1;
      Serial.print("Found DewRing Channel: ");
      Serial.println(i);
      digitalWrite(LED_BUILTIN, HIGH); // any Dew-Ring "lights the LED"
    } else {
      dewring[i] = 0;
      analogWrite(PWM_Pin[i], 0); // switch off channel
      EEPROM.update(ee_heating[i], 0x00);
      //heating[i]    = 0;
      EEPROM.update(ee_aggression[i], 0x00); // no dewring-> zero agression/
    };
  }
//Celestron Code set-up
  pkstate = PREAMBLE_WAIT;
  pklen = 0;
  pkidx = 0;
// AUX serial set-up
  celestron_aux.begin(19200); // AUX bus runs at 19200

  Serial.println("Start-up competed.");

#ifdef DEBUG
  Serial.print("Aggression ");
  Serial.print(EEPROM.read(ee_aggression[0]),HEX);
  Serial.print(" ");Serial.println(EEPROM.read(ee_aggression[1]), HEX);
  Serial.print("Heating    ");
  Serial.print(EEPROM.read(ee_heating[0]),HEX);
  Serial.print(" ");Serial.println(EEPROM.read(ee_heating[1]), HEX);
#endif
}

//
//    LOOP
//
float voltage, current;
float t,h, dp, tt;
int pwm, i;
int counter = 0;

void loop() {
  uint8_t c;
  uint8_t r_len = 0;
  uint8_t out_bytes[PK_MAX_LEN];

  // always read ambient Humidity & Temp via I2C
  t = sht31.readTemperature();
  h = sht31.readHumidity();
  dp = calc_dewpoint( t, h);
//  Serial.print("Temp *C  = "); Serial.println(t);
//  Serial.print("DewPt *C = "); Serial.println(dp);

// loop over all channels and set PWN from aggression or heating
  for (int hw_ch=0; hw_ch<num_channels;hw_ch++) {
    if (therm[i]) {
      voltage = readChannel(temp_adc_idx[hw_ch]); // read out ADC, if there, for dew ring
temp
      tt = voltage_to_degreesC (voltage);
    } else {
      tt = t - delta; // no thermistor, take ambient temp-delta
    }
    if (EEPROM.read(ee_aggression[hw_ch]) != 0) { // aggression != 0 means AUTO mode
      if ((tt - dp) < float(delta)) {
```

```
        pwm = EEPROM.read(ee_aggression[hw_ch])*25;
      //  Serial.print("Auto Heating "); Serial.println(pwm);
      }
      else
        pwm = 0;
    } else    // aggression = 0 means MANUAL mode
      pwm = EEPROM.read(ee_heating[hw_ch]);

    analogWrite(PWM_Pin[hw_ch], pwm); // switch power level accordingly
  };

// handle re-calibratoin timing here
  if (counter == 0)
    sht31.heater(false);  // switch off when counter is zero
  if (sht31.isHeaterEnabled()) { // only decrease, if heating is on
    counter--; };

// Read and echo chars from celestron_aux
  while (celestron_aux.available()) {
    c = celestron_aux.read();
    packet_decode(c);
  };
// check if the packet is valid
  if (pkstate != VALID) return;

#ifdef DEBUG
Serial.print("Len:"); Serial.print(packet[0], HEX);
Serial.print("  Src:"); Serial.print(packet[1], HEX);
Serial.print("  Dest:"); Serial.print(packet[2], HEX);
Serial.print("  ID:"); Serial.print(packet[3], HEX);
Serial.print(" Data: ");
for (i=4; i<=(packet[0]+1); i++) {
  Serial.print(packet[i], HEX); Serial.print(" ");
};
Serial.println(".");
#endif

// check if the packet is for me
  if (packet[2] != DEV_DEWBB && packet[2] != DEV_DEW) {
    pkstate = 0;
    pkidx = 0;
    pklen = 0;
    return;
  }
#ifdef DEBUG
    Serial.print( F("- DEW - ") ); // for debugging purposes
#endif

  uint8_t dest=packet[1], src=packet[2];
  switch(packet[3])
  {
    case DEV_GET_VERSION: {
      r_len = 0;
      out_bytes[r_len++] = 0x01;
      out_bytes[r_len++] = 0x01;
      out_bytes[r_len++] = 0x04;
      out_bytes[r_len++] = 0xf6;
      pk_send_all(dest, src, DEV_GET_VERSION, r_len, out_bytes);
#ifdef DEBUG
      Serial.println("DEV GET_VERSION");
#endif
      break;
    }
    case DEV_GET_MODEL:{
      r_len = 0;
      out_bytes[r_len++] = 0x01;
```

```
      out_bytes[r_len++] = 0x07;
      pk_send_all(dest, src, DEV_GET_VERSION, r_len, out_bytes);
#ifdef DEBUG
      Serial.println("DEV GET_MODEL");
#endif
      break;
    }
    case DEW_QUERY_PORT: {   // simplified version, no VAR,  ports
      byte i_port = packet[4];
#ifdef DEBUG
      Serial.println("DEW QUERY_PORT");
#endif
      if (i_port == 0x00 || i_port == 0x01) { // normal dew heater ports
        pk_send(dest, src, DEW_QUERY_PORT, i_port+1);
        break;
      }
      if (i_port == 0x02) { // one 12V port
        r_len = 0;
        out_bytes[r_len++] = 0x11; //
        out_bytes[r_len++] = EEPROM.read(ee_enabled_12V_port);
        out_bytes[r_len++] =0x00;
        r_len += double_to_2bytes(out_bytes+r_len,  0.00); // watts
        r_len += double_to_2bytes(out_bytes+r_len, 12.00); // volts
        pk_send_all(dest, src, DEW_QUERY_PORT, r_len, out_bytes);
        break;
      }
      pk_send(dest, src, DEW_QUERY_PORT, i_port, 0x00);
      break;
    }
    case DEW_GET_NUM_PORTS: {
      uint8_t ports = num_channels;
      if (packet[1]== DEV_HC1) ports +=1;
      pk_send(dest, src, DEW_GET_NUM_PORTS, ports);
#ifdef DEBUG
      Serial.println("DEW GET_NUM_PORTS");
#endif
      break;
    }
    case DEW_QUERY_ENV_SENSORS: {
      r_len = 0;
      r_len += double_to_4bytes(out_bytes+r_len, (double) t);
      r_len += double_to_4bytes(out_bytes+r_len, (double) dp);
      out_bytes[r_len++] = (uint8_t)h+0.5;
      pk_send_all(dest, src, DEW_QUERY_ENV_SENSORS, r_len, out_bytes);
#ifdef DEBUG
      Serial.print("DEW QUERY_ENV_SENSORS ");
      Serial.print(t);   Serial.print(" ");
      Serial.print(dp);  Serial.print(" ");
      Serial.println(h);
#endif
      break;
    }
    case DEW_RECALIBRATE_ENV_SENSOR: {
      byte p1 = packet[4];
#ifdef DEBUG
      Serial.println("DEW RECALIBRATE_ENV_SENSORS");
#endif
      if (p1) {
#ifdef DEBUG
        Serial.println("Begin ENV SENSOR recalibration");
#endif
        counter = recalibration_counter;
        sht31.heater(true); // code to start heater goes here!
      }
      pk_send(dest,src,DEW_RECALIBRATE_ENV_SENSOR, p1);
      break;
```

```
    }
    case DEW_QUERY_ENV_CALIBRATING: {
#ifdef DEBUG
      Serial.println("DEW QUERY_ENV_CALIBRATING");
#endif
      pk_send(dest, src, DEW_QUERY_ENV_CALIBRATING, (uint8_t) (counter!=0)); // calibrated
      break;
    }
    case DEW_QUERY_INPUT_POWER: {
      r_len = 0;
      double bus_voltage = readChannel(ADS1015_COMP_3_GND)*3.0; // Channel3 monitors power
voltage
      r_len += double_to_2bytes(out_bytes+r_len, bus_voltage);
      double bus_current = (ina3221.getCurrentAmps(current_idx[0])
                          + ina3221.getCurrentAmps(current_idx[1])) * 1;
      r_len += double_to_2bytes(out_bytes+r_len, (double)bus_current);
      out_bytes[r_len++] = 0x00; // ???
      out_bytes[r_len++] = 0x00;
      pk_send_all(dest, src, DEW_QUERY_INPUT_POWER, r_len, out_bytes);
#ifdef DEBUG
      Serial.println("DEW QUERY_INPUT_POWER");
#endif
      break;
    }
    case DEW_SET_INPUT_LIMIT: {
      dew_input_current_limit = (double)(((uint16_t)packet[4]) << 8 | packet[5]) / 1000.0;
      r_len = 0;
      r_len += double_to_2bytes(out_bytes+r_len, (double) dew_input_current_limit);
      pk_send_all(dest, src, DEW_SET_INPUT_LIMIT, r_len, out_bytes);
#ifdef DEBUG
      Serial.print("DEW SET_INPUT_LIMIT -  ");
      Serial.println(dew_input_current_limit);
#endif
      break;
    }
    case DEW_QUERY_INPUT_LIMITS: {
      r_len = 0;
      r_len += double_to_2bytes(out_bytes+r_len, (double) dew_input_current_limit);
      r_len += double_to_2bytes(out_bytes+r_len, (double) 15.0);
      pk_send_all(dest, src, DEW_QUERY_INPUT_LIMITS, r_len, out_bytes);
#ifdef DEBUG
      Serial.println("DEW QUERY_INPUT_LIMITS");
#endif
      break;
    }
    case DEW_QUERY_HEATER:{
      byte i_chan = packet[4];
      r_len = 0;
      out_bytes[r_len++] = i_chan+1;
      out_bytes[r_len++] = (EEPROM.read(ee_aggression[i_chan]) !=0); // true, if AUTO,
false if MANUAL
      out_bytes[r_len++] = (uint8_t) EEPROM.read(ee_heating[i_chan]); // PWM
      double amps = (double) ina3221.getCurrentAmps(current_idx[i_chan]);
      r_len += double_to_2bytes(out_bytes+r_len, (double) amps);
      out_bytes[r_len++] = EEPROM.read(ee_aggression[i_chan]); // agresssion 1-10
      double tt = voltage_to_degreesC (readChannel(temp_adc_idx[i_chan]));
      r_len += double_to_4bytes(out_bytes+r_len, (double) tt);
      pk_send_all(dest, src, DEW_QUERY_HEATER, r_len, out_bytes);
#ifdef DEBUG
      Serial.print("DEW QUERY_HEATER ");
      Serial.println(i_chan);
#endif
      break;
    }
    case DEW_SET_AUTO_AGGR: {
      byte i_chan = packet[4]; // channel 0 or 1
```

```
      byte i_agg  = packet[5]; // this is the aggression value between 0 and 10 for
automatic heating
      EEPROM.update(ee_aggression[i_chan], (i_agg > 10) ? 10 : i_agg); // needs to be <=
10
      pk_send(dest, src, DEW_SET_AUTO_AGGR, i_chan, i_agg);
#ifdef DEBUG
      Serial.println("DEW SET_AUTO_AGGR");
      Serial.println(i_chan); Serial.println(i_agg);
#endif
      break;
    }
    case DEW_SET_MANUAL_PWM: {
      byte i_chan = packet[4]; // channel 0 or 1
      byte i_pwm  = packet[5]; // this is the PWM value between 0 and 255 for MANUAL
heating
      EEPROM.update(ee_aggression[i_chan], 0x00);  // switch to MANUAL mode with
aggression to zero
      EEPROM.update(ee_heating[i_chan], i_pwm); // (i_pwm>255) ? 255 : i_pwm; // always
less 256
      pk_send(dest, src, DEW_SET_MANUAL_PWM, i_chan, i_pwm);
#ifdef DEBUG
      Serial.print("DEW SET_MANUAL_PWM Chan ");
      Serial.print(i_chan); Serial.print(" ");
      Serial.println(i_pwm);
#endif
      break;
    }
    case DEW_ENABLE_PORT:{
      byte i_port = packet[4];
      // hard coded for 2 heate and one (fake) 12 port
      if (i_port==2) EEPROM.update(ee_enabled_12V_port, packet[5]);
      pk_send(dest, src, DEW_ENABLE_PORT, packet[4], packet[5]);
#ifdef DEBUG
      Serial.println("DEW ENABLE_PORT");
#endif
      break;
    }
    case DEW_DEW_UNKNOWN_13:{
      pk_send(dest, src, DEW_DEW_UNKNOWN_13, packet[4], packet[5]);
#ifdef DEBUG
      Serial.println("DEW DEW_UNKNOWN_13");
#endif
      break;
    }
    case DEW_DEW_UNKNOWN_21:{
      pk_send(dest, src, DEW_DEW_UNKNOWN_21, packet[4], packet[5]);
#ifdef DEBUG
      Serial.println("DEW DEW_UNKNOWN_21");
#endif
      break;
    }
    case DEW_GET_LED_BRIGHTNESS: {
#ifdef DEBUG
      Serial.println("DEW GET_LED_BRIGHTNESS");
#endif
      pk_send(dest, src, DEW_GET_LED_BRIGHTNESS, (uint8_t)EEPROM.read(ee_led_brightness));
      break;
    }
    case DEW_SET_LED_BRIGHTNESS: {
      EEPROM.update(ee_led_brightness, packet[4]);
#ifdef DEBUG
      Serial.println("DEW SET_LED_BRIGHTNESS");
#endif
      pk_send(dest, src, DEW_SET_LED_BRIGHTNESS, (uint8_t)EEPROM.read(ee_led_brightness));
      break;
    }
```

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

// utilities for AUX handling
static byte double_to_4bytes (byte out[4], double val)
{
   long v = val * 1000;
   out[3] = v;
   v >>= 8;
   out[2] = v;
   v >>= 8;
   out[1] = v;
   v >>= 8;
   out[0] = v;
   return 4;
}

static byte double_to_2bytes (byte out[2], double val)
{
   long v = val * 1000;
   out[1] = v;
   out[0] = v >> 8;
   return 2;
}

// check if thermistor is present
boolean check_thermistor(int channel) {
  float voltage;
  if (channel == 1)
    voltage = readChannel(ADS1015_COMP_1_GND);
  else if (channel==2)
    voltage = readChannel(ADS1015_COMP_2_GND);
  else if (channel==3)
    voltage = readChannel(ADS1015_COMP_3_GND);
  else if (channel==0)
    voltage = readChannel(ADS1015_COMP_0_GND);
  else
  return(false);
  if (voltage>3.0 or voltage<0.65) return(false); // strange reading, dont use
return(true);
}

// check if Dewring is present
boolean check_dewring(int channel) {
  float current, voltage, dew_power;
  analogWrite(PWM_Pin[channel], 50); delay(100);     // power up dewring
  voltage = ina3221.getBusVoltage(current_idx[channel]);          // measure
  current = ina3221.getCurrentAmps(current_idx[channel]) * 1000;  // measure
  analogWrite(PWM_Pin[channel], 0);                        // power down again

  dew_power=voltage*current;
  if ((dew_power)<45.0) return(false);  // 45mW are about the usage by the LED
  return(true);
}

float readChannel(ADS1015_MUX channel) {
  float voltage = 0.0;
  adc.setCompareChannels(channel);
  // setCompareChannels_nonblock(channel);
  adc.startSingleMeasurement();
  while(adc.isBusy())delay(0);
  voltage = adc.getResult_V(); // alternative: getResult_mV for Millivolt
  return voltage;
```

```
}

///// calculate thermistor temperature from Voltage value

#include<math.h>
static double voltage_to_degreesC (float v_in)
{
  const double R1      = 10000.0;      // voltage divider resistor value
  const double Beta    = 3950.0;       // Beta value.  FIXME? Celestron measured as 3435
  const double Koffset = 273.15;       // Degrees Kelvin for zero degrees Celsius
  const double Ro      = 10000.0;      // Resistance of Thermistor at 25 degree Celsius
  const double v_max   = 5.0; //3.28;
        double Rt      = R1 * v_in / (v_max - v_in);
        double Tk      = 1.0 / ( (1.0 / (25.0 + Koffset)) + (log(Rt / Ro) / Beta)); //
degrees Kelvin
  return Tk - Koffset;  // degrees Celsius
}
// calculates dew point
// input:   humidity [%RH], temperature in C
// output:  dew point in C
double calc_dewpoint(float t, float h)
{
  float logEx, dew_point;
  logEx = 0.66077 + 7.5 * t / (237.3 + t) + (log10(h) - 2);
  dew_point = (logEx - 0.66077) * 237.3 / (0.66077 + 7.5 - logEx);
  return dew_point;
}

/*
 * Auxbus handling routines
 */

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++;
// Version 0.1
  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 (long) response of variable length
void pk_send_all(uint8_t dest, uint8_t src, uint8_t id, uint8_t len, uint8_t bytes[])
{
  cksum_init();// Send header bytes
  celestron_aux.write(0x3b);
  cksum_update(3+len); // length = len + Src + Dest + ID
  celestron_aux.write((uint8_t)3+len);
  cksum_update(src);
  celestron_aux.write((uint8_t)src);
  cksum_update(dest);
  celestron_aux.write(dest);
  cksum_update(id);
  celestron_aux.write(id);
  for (int i=0; i<len; i++) { // Send all message bytes
    cksum_update(bytes[i]);
    celestron_aux.write(bytes[i]);
  };
  celestron_aux.write(cksum_final());  // Send checksum
}
// Send a 1-byte response
void pk_send(uint8_t dest, uint8_t src, uint8_t id, uint8_t byte0)
{
  cksum_init();// Send preamble
  celestron_aux.write(0x3b);
  cksum_update(0x04);
  celestron_aux.write(0x04); // Send length 4
  cksum_update(src);
  celestron_aux.write((uint8_t)src);
  cksum_update(dest);
  celestron_aux.write(dest);
  cksum_update(id);
  celestron_aux.write(id);
  cksum_update(byte0);
  celestron_aux.write(byte0); // send the byte
  celestron_aux.write(cksum_final());
}
// Send a 2-byte response
void pk_send(uint8_t dest, uint8_t src, uint8_t id, uint8_t byte0, uint8_t byte1)
{
  cksum_init();
// Send preamble
  celestron_aux.write(0x3b);
```

```
  cksum_update(0x05);
  celestron_aux.write(0x05); // Send length 5
  cksum_update(src);
  celestron_aux.write((uint8_t)src);
  cksum_update(dest);
  celestron_aux.write(dest);
  cksum_update(id);
  celestron_aux.write(id);
  cksum_update(byte0);
  celestron_aux.write(byte0); // send byte 0
  cksum_update(byte1);
  celestron_aux.write(byte1); // send byte 1
  celestron_aux.write(cksum_final());
}
```