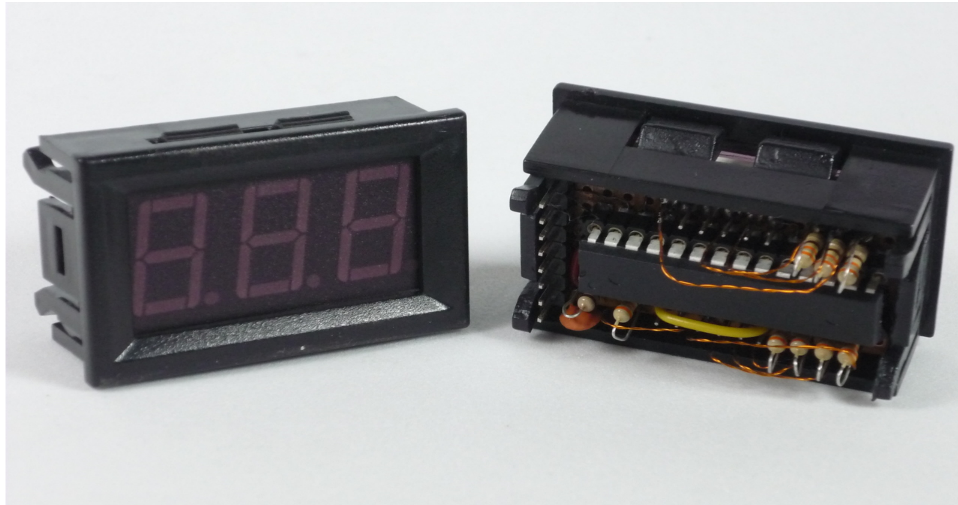
PANEL MOUNT 7 SEGMENT DISPLAY
WITH EMBEDDED ARDUINO

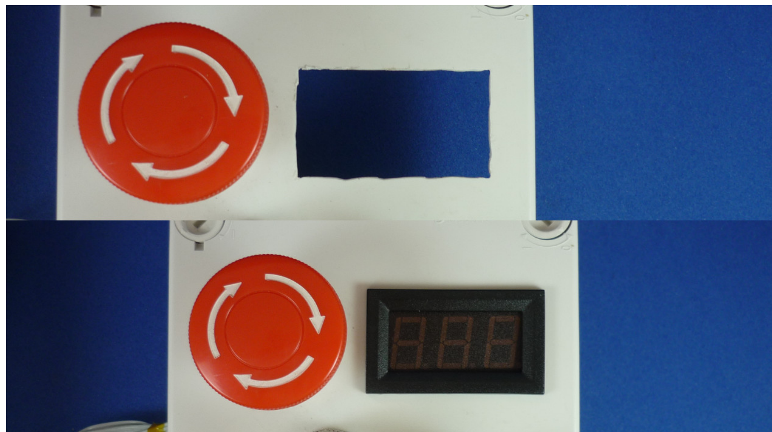


PANEL MOUNT 7 SEGMENT DISPLAY

Is not always an easy task to get a good looking finished project if it includes a 7 segment display!.

The main problem to get an elegant finish is the tight tolerance needed for the rectangular hole cut where the display will be exposed, only accomplished using CNC machining or laser cutting services.

Cheap panel meters solved the problem using an enclosure (usually bigger than 1/32 DIN), those enclosures has two advantages: Requires a rectangular cut, which could have some imperfections and can be done using home tools like drill and hand saw, because such enclosures incorporates a bezel that covers imperfections once installed. The other advantage is there is no need for additional screws to fix it to the panel/chassis.



SMART VISUALIZATION WITH EMBEDDED ARDUINO

There are 2 kinds of devices in the market: Specific meters for panel mounting (voltage, current, RPM, etc.) and "smart" displays usually using serial protocols like I2C, SPI, shift registers, some with ATmega microcontroller included and compatible with Arduino ecosystem!. However the last ones usually came in a PCB-shaped form factor and are not easily installed on a panel/ chassis in an elegant way without using additional elements and resources.

This project brings the best characteristics of all of the aforementioned devices:

- Low cost.
- Panel/chassis mount with good looking aspect.
- Needs simply tools for installation.
- Only 2 I/O pins required for communication.
- Built in processing power to offload the main processor/microcontroller.
- Programmable with the Arduino suite.
- Could be used as serial slave display or as a standalone controller with the I/O pins available.
- Open source hardware and software (schematics and code available for download).
- Can be built with through-hole components.
- The design can be accommodated on a single side PCB (some wire jumpers needed).

PROTOTYPE BUILDING

The foundation for this project was the enclosure of a ready made cheap panel meter, easily available in local markets. Because the idea was to use through-hole components there is no much space to waste inside the enclosure, so a minimal part count Arduino-based system like Pro Mini (ATmega168 or ATmega328) with internal 8 MHz clock was chosen because only needs one external resistor and capacitor around reset pin.

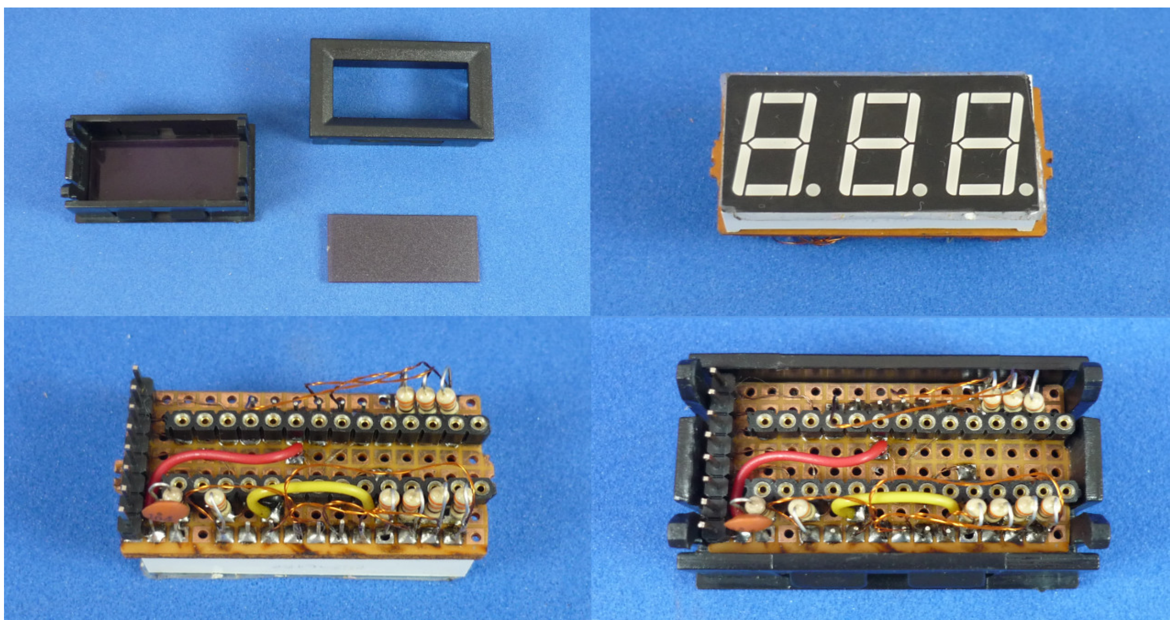
To burn the bootloader, an AVR chip programmer is needed, also some Arduino boards could be programmed as ISP using the appropriate sketch. To modify the fuses that manages the internal 8MHz clock, sometimes an oscillator circuit is needed: A crystal and two capacitors or a square wave clock generator. It depends on how the clock fuses were initially set-up.The bootloader used was ATmegaBOOT_168_pro_8MHz.hex, see documentation in the bottom part of the article for the respective boards.txt file that has the lines than must be included in the actual boards.txt file to execute the burn. Finally to compile sketches for this device the option for Ardino Pro Mini @8MHz tools->board option was used

The display used was a seven segment 3 digit multiplexed(could be common anode or cathode) of dimensions according to the available space inside the case. A current liming resistor of 330 ohms for every segment were used. (7 + decimal point = 8). The circuit was built on a universal circuit board cut exactly as the original PCB of the meter, leaving also the 2 little tabs on the

sides for pressure fixing. There is a crude example of a single side board(some jumper wires required!) PCB below in the documentation zone.

Due to space constrains only 5 I/O pins are available to the user in any of the following configurations:(2) UART (TX/RX),(2) SPI (SDA/SCL) and (1) analog pin A3. All (5) I/O pins as digital or a mix of them.

The test firmware is based on the SevenSeg library for display multiplexing, character mapping and string interpretation tasks.



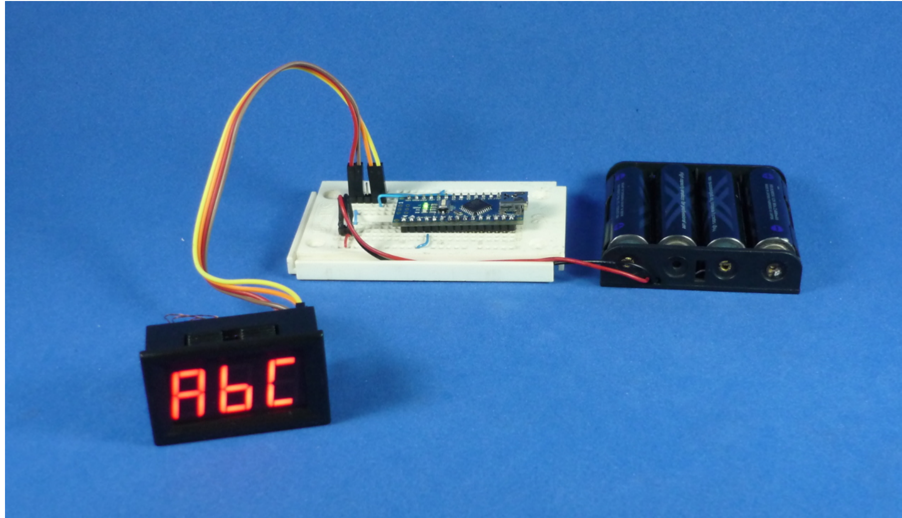
SERIAL SLAVE

To minimize the I/O pin count used for communication with the main processor/microcontroller, a two wire serial protocol must be used, in this specific case I2C, because only needs two signals (SDA/SCL) also support multiple devices on the same bus, using the same pair of wires. This protocol is available in almost any microcontroller in the market. In the experimental firmware every byte sent through I2C represents an ASCII character, so there is no need for a specialized library in the master. The device is almost a "serial terminal".

The first byte transmitted is a "command", at the moment the available ones are:

0x27 Display character string.
0x25 Change I2C address stored in EEPROM.
0x26 Change displayed message blink rate (useful for alarms!).

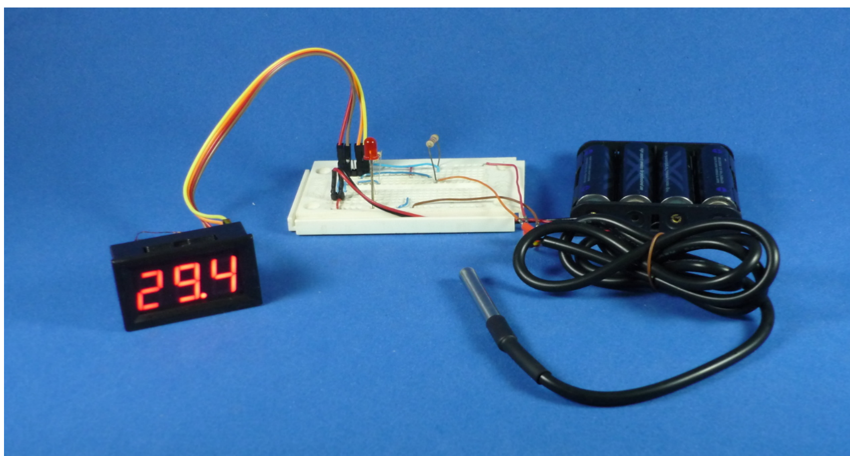
UART (TX/RX) I/O pins are also available, so some sort of byte oriented serial protocol could be implemented, like for example Modbus RTU (adding an RS-232 or RS-485 to TTL chip)



STANDALONE CONTROLLER

The circuit is basically a derivation of an Arduino Pro Mini, so it can be programmed for any other purpose (than a serial display) using an USB to TTL interface. With the available 5 I/O pins (a lot of I/O pins were used for the 7 segment display) some interesting projects could be built:

- Voltmeter or Ammeter, adding a precision shunt resistor and/or voltage divider.
- RPM counter using a digital pin as counting input.
- Display for I2C or 1-Wire digital sensors.
- Universal instrumentation displays for 4-20 mA or 0-10 V signals.
- Byte oriented serial protocol analyzer (Modbus, Mewtocol, DF1, etc.).



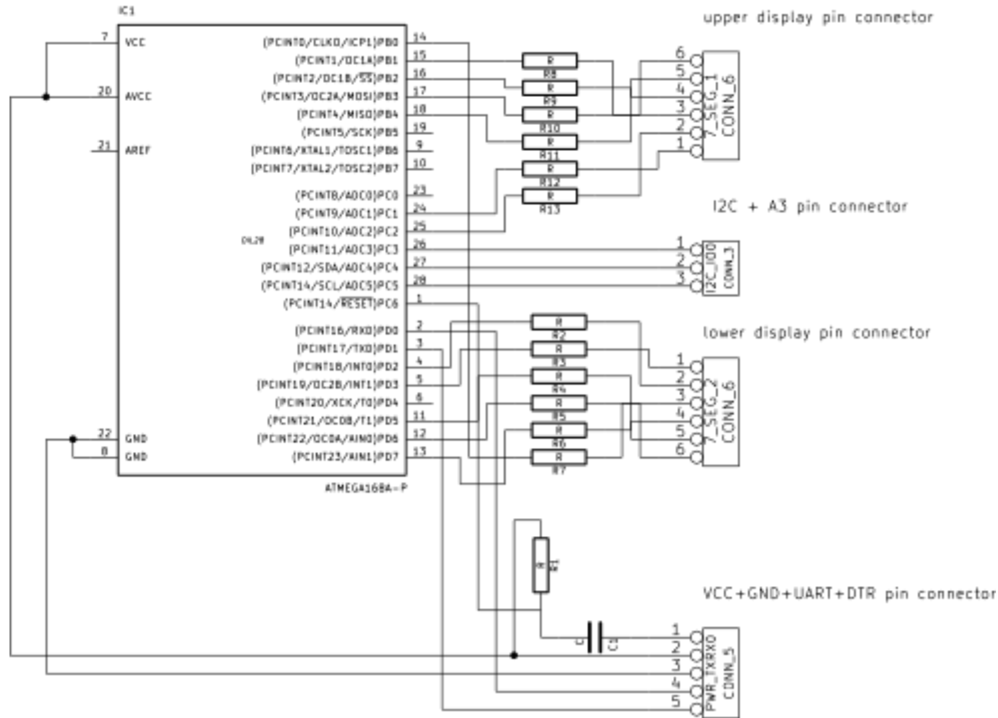
TESTS AND CONCLUSIONS

- The circuit required a lot of patience, however, was done using only through-hole components and single side universal circuit board.
- The rectangular orifice cut was done with a drill and a hacksaw. Precision cut isn't very important because the case's bezel will cover the imperfections.
- No screws required to keep the panel case in place. Only hand pressure is needed for installing & removing
- When using an I2C master different than Arduino, is possible that the R/W bit in the I2C address field must be written, so the address could differ from the one stored in EEPROM.
- RPM counter using a digital pin as counting input.
- Is feasible to design a single side PCB for the circuit, however, some jumper wires will be required.
- Due to display multiplexing, power consumption is quite low (around 25 mA), so it could be powered by batteries for portable applications.

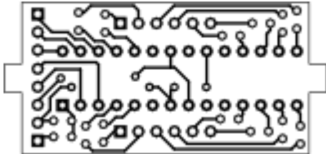
LINKS

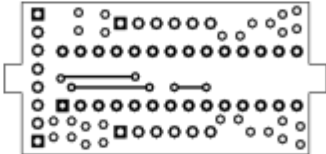
Video that shows all the components and working tests : **youtu.be/f5e__fcaucE**
Arduino Pro Mini Clone: **http://s.click.aliexpress.com/e/NvRnMzV**
Panel meter: **http://s.click.aliexpress.com/e/QBYFUFA**
RPM counter**: http://s.click.aliexpress.com/e/NrjUba2**
AVR ISP programmer: **http://s.click.aliexpress.com/e/uzf6ubq**
3 digit 7 segment display: **http://s.click.aliexpress.com/e/AYNjaQR**
1/4W 330 ohm resistors: **http://s.click.aliexpress.com/e/Mb2rVBy**
Universal circuit board: **http://s.click.aliexpress.com/e/iU3zvzb**
1-wire temperature sensor: **http://s.click.aliexpress.com/e/qna2buV**

SCHEMATIC FOR THE PANEL MOUNT 7 SEGMENT DISPLAY WITH EMBEDDED ARDUINO



PCB FOR THE PANEL MOUNT 7 SEGMENT DISPLAY WITH EMBEDDED ARDUINO





SMART_7_SEGMENT.ino

```
// *************************************
//            Absolutelyautomation.com
//
//  Panel mount 7 segment display
//  with embedded Arduino
//  Working as serial slave
//
//**************************************


#include <SevenSeg.h>
#include <Wire.h>
#include <EEPROM.h>


// Pin definition for each segment
int sega = 12;
int segb = 10;
int segc = 7;
int segd = 5;
int sege = 3;
int segf = 11;
int segg = 2;
int segp = 6;

// Pin definition for each common
int digit1 = 13;
int digit2 = A0;
int digit3 = A1;

String dispvalue = "------";

int FWMajorVersion=0;
int FWMinorVersion=1;

#define MAX_I2C_BUFF     20
#define MAX_CHARS        6

#define EEPROM_I2C_ADD   0x00
#define I2C_ADD_DEFAULT  0x08
#define CMD_CHNGADD      0x25
#define CMD_BLNKRATE     0x26
#define CMD_DISPTEXT     0x27

SevenSeg disp( sega, segb, segc, segd, sege, segf, segg );
const int numOfDigits = 3;
int digitPins[numOfDigits] = { digit1, digit2, digit3  };

int firstboot;
int i2cadd;
int i2c_buffer[MAX_I2C_BUFF];
```

```
int i2c_bytesrcvd;
int i2c_rcvdone;
int blinkrate=0;
int blinkstate;
long previousMillis = 0;

String astring = "A00";
String fstring = "F00";

void setup() {

  // display setup

  pinMode(sega, OUTPUT);
  pinMode(segb, OUTPUT);
  pinMode(segc, OUTPUT);
  pinMode(segd, OUTPUT);
  pinMode(sege, OUTPUT);
  pinMode(segf, OUTPUT);
  pinMode(segg, OUTPUT);
  pinMode(segp, OUTPUT);

  pinMode(digit1, OUTPUT);
  pinMode(digit2, OUTPUT);
  pinMode(digit3, OUTPUT);


  pinMode(13, OUTPUT);

  disp.setDPPin(segp);
  //disp.setCommonCathode();
  disp.setCommonAnode();
  disp.setDigitPins( numOfDigits, digitPins );
  disp.setRefreshRate(300);
  disp.setTimer(1);  // Don't use with servo functions at the same time!
  //disp.setTimer(2);  // Don't use with tone() function at the same time!
  disp.startTimer();

 // I2C slave setup

  i2cadd = EEPROM.read(EEPROM_I2C_ADD);
  if(i2cadd == 0xFF)
  {
    EEPROM.write(EEPROM_I2C_ADD, I2C_ADD_DEFAULT); // EEPROM address erased,
writting default value
    i2cadd = EEPROM.read(EEPROM_I2C_ADD);
  }
  Wire.begin(i2cadd); // join I2C bus using the address stored in  EEPROM
  Wire.onReceive(receiveEvent); // event register

}

// infinite loop:
void loop() {
```

```
  if(!firstboot){
    disp.write("8.8.8.");
    delay(500);
    disp.write("");
    delay(500);
    fstring = "U"+String(FWMajorVersion,DEC)+"."+String(FWMinorVersion,DEC);
    disp.write(fstring);// FW version = MajorVersion.MinorVersion
    delay(500);
    disp.write("");
    delay(500);

    if(i2cadd > 0x0f)
    {
      astring = "A"+String(i2cadd, HEX);
    }
    else
    {
      astring = "A0"+String(i2cadd, HEX);
    }

    disp.write(astring);// direccion i2c
    delay(500);
    disp.write("");
    delay(500);
    firstboot=1;
  }


  //
****************************************************************************
  // I2C command processing
  //
****************************************************************************
  if(i2c_rcvdone== 1)
  {
    // change I2C address
    if(i2c_buffer[0]==CMD_CHNGADD && i2c_bytesrcvd > 1)
    {
      // Validate new I2C address , write into  EEPROM
      // Reserved adress 0000XXX y 1111XXX
      if( i2c_buffer[1] > 7 &&  i2c_buffer[1] < 120)
        {
          EEPROM.write(EEPROM_I2C_ADD,i2c_buffer[1]); // Required power cycle
to take effect
          delay(1000); // just in case!
          blinkrate=5;
        }
    }
    // change blink rate
    if(i2c_buffer[0]==CMD_BLNKRATE && i2c_bytesrcvd > 1)
    {
      // Validate blink rate
      if( i2c_buffer[1] > 0 &&  i2c_buffer[1] < 10)
        {
```

```
        blinkrate = i2c_buffer[1];
      }
    }
    // Display received text string
    if(i2c_buffer[0]==CMD_DISPTEXT && i2c_bytesrcvd > 1)
    {
      int steps=0;
      int m=0;
      if(i2c_bytesrcvd > MAX_CHARS)
        steps=MAX_CHARS;
      else
        steps=i2c_bytesrcvd-1;

      dispvalue = "";

      while( steps > 0 )
      {
        unsigned char ch=i2c_buffer[m+1];
        dispvalue = dispvalue + char(ch);
        steps--;
        m++;
      }
    }
    i2c_rcvdone=0;
  }


  // ********************************************************
  //     Display data on to the 7 segment ( at the selected blink rate! )
  // ********************************************************

  if(blinkrate == 0)
  {
    blinkstate = HIGH;
  }
  else
  {
    unsigned long currentMillis = millis();
    unsigned long interval = 500/blinkrate;

    if(currentMillis - previousMillis > interval)
    {
      // store last time display blink
      previousMillis = currentMillis;
      // if ON then turn OFF and vice-versa:
      if (blinkstate == LOW)
      {
        blinkstate = HIGH;
      }
      else
      {
        blinkstate = LOW;
      }
    }
```

```
  }


  if(blinkstate == HIGH)
    disp.write(dispvalue);
  else
    disp.write("");


}

ISR(TIMER1_COMPA_vect)   {
  disp.interruptAction();
}

// function executed everytime a data is received from an I2C master
// function registered as an event, see setup()

void receiveEvent(int howMany)
{
  // First byte is a command: CHANGE I2C ADDRESS, BLINK SPEED, DISPLAY DATA

  i2c_bytesrcvd=0;
  while(Wire.available()) // I2C data available?
  {
    int i2c_byte = Wire.read(); // read byte and write to buffer
    i2c_buffer[i2c_bytesrcvd]=i2c_byte;
    i2c_bytesrcvd++;
  }
  i2c_rcvdone=1;

}
```

SevenSeg.h

```
/*
  SevenSeg v1.1
  SevenSeg.h - Library for controlling a 7-segment LCD
  Created by Sigvald Marholm, 02.06.2015.
*/

#ifndef SevenSeg_h
#define SevenSeg_h

#include "Arduino.h"

class SevenSeg
{

  public:

    // Constructor
    SevenSeg(int,int,int,int,int,int,int);

    // Low level functions for initializing hardware
    void setCommonAnode();
    void setCommonCathode();
    void setDigitPins(int,int *);
    void setActivePinState(int,int);
    void setDPPin(int);
    void setColonPin(int);
    void setSymbPins(int,int,int,int);

    // Low level functions for printing to display
    void clearDisp();
    void changeDigit(int);
    void changeDigit(char);
    void writeDigit(int);
    void writeDigit(char);
    void setDP();
    void clearDP();
    void setColon();
    void clearColon();
    void setApos();
    void clearApos();

    // Low level functions for controlling multiplexing
    void setDigitDelay(long int);   // Should I have this function?
    void setRefreshRate(int);
    void setDutyCycle(int);

    // High level functions for printing to display
    void write(long int);
    void write(int);
    void write(long int,int);
    void write(int, int);
```

```
    void write(char*);
    void write(String);
    void write(double);
    void write(double num, int point);
    void writeClock(int,int,char);
    void writeClock(int,int);
    void writeClock(int,char);
    void writeClock(int);

    // Timer control functions
    void setTimer(int);
    void clearTimer();
    void interruptAction();
    void startTimer();
    void stopTimer();

    // To clean up
//  void setPinState(int);    // I think this isn't in use. Its called
setActivePinState?
//  int getDigitDelay();      // How many get-functions should I make?

  private:

    // The pins for each of the seven segments (eight with decimal point)
    int _A;
    int _B;
    int _C;
    int _D;
    int _E;
    int _F;
    int _G;
    int _DP;      // -1 when decimal point not assigned

    // Variables used for colon and apostrophe symbols
    int _colonState;    // Whether colon is on (_segOn) or off (_segOff).
    int _aposState;     // Whether apostorphe is on (_segOn) or off (_segOff).
    int _colonSegPin;
    int _colonSegLPin;
    int _aposSegPin;
    int _symbDigPin;

    /* The colon/apostrophe handling needs some further explanation:
     *
     * colonSegPin is the segment pin for colon. I.e. some displays have a
separate segment for colon on one of the digits.
     * others have colon split across two digits: i.e. the upper part has a
separate segment on one digit, whereas the lower
     * part has uses the same segment pin but on another digit. It is assumed
that this segment pin is only used for colon,
     * and it is stored into colonSegPin by setColonPin(int). The functions
setColon() and clearColon() turns on/off this pin,
     * respectively.
     *
```

```
     * On some displays, colon is one or two separate free-standing LED(s)
with its own cathode and anode. In case of common
     * cathode, ground the cathod and treat the anode(s) as a segment pin. The
other way around in case of common anode. This
     * should make the method described above applicable.
     *
     * On other displays, the upper colon part, the lower colon part, as well
as an apostrophe, shares segments with the usual
     * segments (i.e. segments A, B and C) but is treated as a separate symbol
digit that must be multiplexed along with the
     * other digits. In this case the function setSymbPins(int,int,int,int) is
used to assign a pin to that digit, stored in
     * symbDigPin. The pin corresponding to the upper colon segment is stored
in colonSegPin, whereas the lower colon segment
     * is stored in colonSegLPin. aposSegPin holds the segment pin for the
apostrophe. symbDigPin being unequal to -1 is an
     * indication for multiplexing-related functions that it must multiplex
over _numOfDigits+1 digit pins. In this case, the
     * setColon(), clearColon(), setApos() and clearApos() does not directly
influence the pin, but the variable colonState and
     * aposState. In this case, the digit must be changed to the symbol digit
by issuing changeDigit('s') in order to show the
     * symbols.
     */

    // The pins for each of the digits
    int *_dig;
    int _numOfDigits;

    // Timing variables. Stored in microseconds.
    long int _digitDelay;           // How much time spent per display during
multiplexing.
    long int _digitOnDelay;         // How much on-time per display (used for
dimming), i.e. it could be on only 40% of digitDelay
    long int _digitOffDelay;        // digitDelay minus digitOnDelay
    int _dutyCycle;           // The duty cycle (digitOnDelay/digitDelay, here
in percent)
    // Strictly speaking, _digitOnDelay and _digitOffDelay holds redundant
information, but are stored so the computations only
    // needs to be made once. There's an internal update function to update
them based on the _digitDelay and _dutyCycle

    void updDelay();
    void execDelay(int);      // Executes delay in microseconds
    char iaExtractDigit(long int,int,int);
    long int iaLimitInt(long int);

    // Sets which values (HIGH or LOW) pins should have to turn on/off
segments or digits.
    // This depends on whether the display is Common Anode or Common Cathode.
    int _digOn;
    int _digOff;
    int _segOn;
    int _segOff;
```

```
    // Variables used by interrupt service routine to keep track of stuff
    int _timerDigit;          // What digit interrupt timer should update next
time
    int _timerPhase;          // What phase of the cycle it is to update, i.e.
phase 1 (on), or phase 0 (off). Needed for duty cycling.
    int _timerID;       // Values 0,1,2 corresponds to using timer0, timer1 or
timer2.
    long int _timerCounter;         // Prescaler of 64 is used since this is
available on all timers (0, 1 and 2).
                        // Timer registers are not sufficiently large. This
counter variable will extend upon the original timer.
                        // and increment by one each time.
    long int _timerCounterOnEnd;    // How far _timerCounter should count to
provide a delay approximately equal to _digitOnDelay
    long int _timerCounterOffEnd;   // How far _timerCounter should count to
provide a delay approximately equal to _digitOffDelay

    // What is to be printed by interruptAction is determined by these
variables
    long int _writeInt;       // Holds the number to be written in case of
int, fixed point, or clock
    int _writePoint;          // Holds the number of digits to use as decimals
in case of fixed point
//    float _writeFloat;            // Holds the float to write in case of
float. OBSOLETE: Float are converted to fixed point
    char *_writeStr;          // Holds a pointer to a string to write in case
of string
    char _writeMode;          // 'p' for fixed point, 'i' for integer, 'f' for
float, ':'/'.'/'_' for clock with according divisor symbol
    String _writeStrObj;

};

#endif
```

SevenSeg.cpp

```
/*
  SevenSeg v.1.0
  SevenSeg.h - Library for controlling a 7-segment display
  Created by Sigvald Marholm, 02.06.2015.
*/

#include "Arduino.h"
#include "SevenSeg.h"

// Constructor
SevenSeg::SevenSeg(int A,int B,int C,int D,int E,int F,int G){

  // Assume Common Anode (user must change this if false)
  setCommonAnode();

  // Set segment pins
  _A=A;
  _B=B;
  _C=C;
  _D=D;
  _E=E;
  _F=F;
  _G=G;
  _DP=-1;   // DP initially not assigned

  // Set all segment pins as outputs
  pinMode(_A, OUTPUT);
  pinMode(_B, OUTPUT);
  pinMode(_C, OUTPUT);
  pinMode(_D, OUTPUT);
  pinMode(_E, OUTPUT);
  pinMode(_F, OUTPUT);
  pinMode(_G, OUTPUT);

  // Assume no digit pins are used (i.e. it's only one hardwired digit)
  _numOfDigits=0;

  _colonState=_segOff;  // default off
  _aposState=_segOff;   // default off
  _colonSegPin=-1;      // -1 when not assigned
  _colonSegLPin=-1;     // -1 when not assigned
  _aposSegPin=-1; // -1 when not assigned
  _symbDigPin=-1; // -1 when not assigned

  // When no pins are used you need not multiplex the output and the delay is
superfluous
  // TBD: Needed for duty cycle control. Add option to differentiate between
0 and 1 digit pins
  _digitDelay=0;
  _digitOnDelay=0;
  _digitOffDelay=0;
```

```
  _dutyCycle=100;

  // Timer data (default values when no timer is assigned)
  _timerDigit=0;
  _timerPhase=1;
  _timerID=-1;
  _timerCounter=0;
  _timerCounterOnEnd=0;
  _timerCounterOffEnd=0;

  _writeInt=0;
  _writePoint=0;
//  _writeFloat=0;
  _writeStr=0;
  _writeMode=' ';

  // Clear display
  clearDisp();
}

void SevenSeg::setTimer(int timerID){

/*
  Assigns timer0, timer1 or timer2 solely to the task of multiplexing the
display (depending on
  the value of timerNumber).

  For an example of a 5 digit display with 100Hz refresh rate and able to
resolve duty cycle in
  10%-steps a timing of the following resolution is needed:

    1/( 100Hz * 5 digits * 0.1 ) = 200us

  It is sufficient, but the brightness should be adjustable to more than 10
values. Hence a
  resolution of 16us is selected by setting the prescaler to 64 and the
compare register to 3:

    interrupt delay = (64*(3+1))/16MHz = 16us

  The timerCounter variable is of type unsigned int having a maximum value of
65535. Incrementing
  this at each interrupt and taking action upon timerCounterOnEnd or
timerCounterOffEnd yields
  a maximum delay for something to happen:

    max delay for something to happen = 16us * (65535+1) = 1.04s

  which should be more than sufficient if you want to be able to look at your
display.
*/

  _timerID = timerID;
```

```
}

void SevenSeg::clearTimer(){

  stopTimer();
  _timerID = -1;

}

void SevenSeg::startTimer(){

  cli();  // Temporarily stop interrupts

  // See registers in ATmega328 datasheet

  if(_timerID==0){
    TCCR0A = 0;
    TCCR0B = 0;
    TCNT0 = 0;                            // Initialize counter value to 0
    OCR0A = 3;                            // Set Compare Match Register to 3
    TCCR0A |= (1<<WGM01);                 // Turn on CTC mode
    TCCR0B |= (1<<CS01) | (1<<CS00);            // Set prescaler to 64
    TIMSK0 |= (1<<OCIE0A);                // Enable timer compare interrupt
  }

  if(_timerID==1){
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1  = 0;                           // Initialize counter value to 0
    OCR1A = 3;                            // Set compare Match Register to 3
    TCCR1B |= (1 << WGM12);               // Turn on CTC mode
    TCCR1B |= (1 << CS11) | (1 << CS10);  // Set prescaler to 64
    TIMSK1 |= (1 << OCIE1A);              // Enable timer compare interrupt
  }

  if(_timerID==2){
    TCCR2A = 0;
    TCCR2B = 0;
    TCNT2  = 0;                           // Initialize counter value to 0
    OCR2A = 3;                            // Set Compare Match Register to 3
    TCCR2A |= (1 << WGM21);               // Turn on CTC mode
    TCCR2B |= (1 << CS22);                // Set prescaler to 64
    TIMSK2 |= (1 << OCIE2A);              // Enable timer compare interrupt
  }

  sei();  // Continue allowing interrupts

  // update delays to get reasonable values to _timerCounterOn/OffEnd.
  updDelay();
  _timerCounter=0;

}

void SevenSeg::stopTimer(){
```

```
  if(_timerID==0){
    TCCR0B = 0;
  }
  if(_timerID==1){
    TCCR1B = 0;
  }
  if(_timerID==2){
    TCCR2B = 0;
  }
}

void SevenSeg::setCommonAnode(){
  _digOn=HIGH;
  _digOff=LOW;
  _segOn=LOW;
  _segOff=HIGH;
}

void SevenSeg::setCommonCathode(){
  _digOn=LOW;
  _digOff=HIGH;
  _segOn=HIGH;
  _segOff=LOW;
}

void SevenSeg::clearDisp(){

  for(int i=0;i<_numOfDigits;i++){
    digitalWrite(_dig[i], _digOff);
  }
  digitalWrite(_A, _segOff);
  digitalWrite(_B, _segOff);
  digitalWrite(_C, _segOff);
  digitalWrite(_D, _segOff);
  digitalWrite(_E, _segOff);
  digitalWrite(_F, _segOff);
  digitalWrite(_G, _segOff);

  if(_DP!=-1){    // Clear DP too if assigned
    digitalWrite(_DP, _segOff);
  }

  if(_symbDigPin!=-1){
    digitalWrite(_symbDigPin, _digOff);
  }

}

/* OLD METHOD
   This method works, but you have to make a (static) array outside the object
and it
   will only be pointed to by this object. That's not very pretty since the
object doesn't
```

```
   actually contain all the information about the display. Further on you rely
on the array
   being declared in a persistent scope and that the user doesn't change it.
*/
void SevenSeg::setDigitPins(int numOfDigits, int *pDigitPins){
  _dig=pDigitPins;
  _numOfDigits=numOfDigits;

  for(int i=0;i<_numOfDigits;i++){
    pinMode(_dig[i],OUTPUT);
  }

  clearDisp();

  // Set the default refresh rate of 100 Hz. If the user wants another refresh
rate this
  // would have to be set after the setDigitPins function.
  setRefreshRate(100);
}

void SevenSeg::setDigitDelay(long int delay){
  _digitDelay=delay;
  updDelay();
}

void SevenSeg::setDutyCycle(int dc){
  _dutyCycle=dc;
  updDelay();
}

void SevenSeg::setActivePinState(int segActive, int digActive){
  _digOn = digActive;
  _digOff = !digActive;
  _segOn = segActive;
  _segOff = !segActive;
}

void SevenSeg::setRefreshRate(int freq){
  long int period = 1000000L/freq;
  long int digitDelay = period/_numOfDigits;

  if(_symbDigPin!=-1){  // Separate symbol pin in use. One more digit to
multiplex across.
    digitDelay = period/(_numOfDigits+1);
  }

  setDigitDelay(digitDelay);
}

/*
 * HIGH LEVEL WRITE-FUNCTIONS
 *
 * High-level write functions should work in the following way:
 *
```

```
 * writeClock(int aa, int bb) - writes a time in the following format aa:bb
(or aa.bb if no colon, or aabb if no dp).
 * writeClock(int aa, int bb, char c) - where char c specifies decimator ("_"
for no decimator)
 * writeClock(int bb) or writeClock(int bb,char c) - writes in the format
aa:bb (or similar) where aa is deduced from bb.
 * write(int a)
 * write(float a)
 * write(int a, int point)
 * write(char* a)
 *
 * For the timer interrupt mode, the value must simply be stored to a private
variable, and printed by interruptAction. For
 * storing, the following is needed:
 *
 * int _writePoint      - Stores the fixed point in case of writeFixed is
used.
 * int _writeInt  - Stores the number for write(int), writeFixed() and
writeClock() (convert to only one number bb for clock)
 * float _writeFloat    - Stores the float for write(float)
 * char* _writeStr      - Stores a pointer to a string to be written. This
pointer must be maintained outside the class
 * String _writeStrObj  - Stores a string object to be written.
 * char _writeMode      - Describes which writing function/mode is used:
 *                      i - write(int)
 *                      f - write(float)
 *                      p - writeFixed()
 *                      s - write(char* a)
 *                      o - write(String)
 *                      : - writeClock() with colon as decimator
 *                      . - writeClock() with period as decimator
 *                      _ - writeClock() with no decimator
 *
 * Further on, all functions should be using timer if _timerID!=-1. If not,
the should multiplex through the
 * display once and rely on the function being placed in a loop.
 *
 * Perhaps the write int/float functions need a separate private parsing
function to extract the digits? Here's the algorithm:
 *
 *   if num > (10^_numOfDigits-1) or num < (-10^_(numOfDigits-1)+1)     //
Store these values in object during digit pin assignment to save computation?
 *     Display a positive or negative overload
 *
 *   else display can handle number
 *
 *     num = 2468                                     // example
 *     digit_0 = num / (10^(_numOfDigits-1))                // 2
 *     digit_1 = (num / (10^(_numOfDigits-2)))%10           // 4
 *     i_th_digit = (num / (10^(_numOfDigits-1-i)))%10            // 6 and 8
for i=2 and 3
 *
 *     IMPROVED:
 *     num = 2468                               // exampmle
```

```
 *     digit_0 = num % 10;
 *     num /= 10;
 *     digit_1 = num % 10;
 *     num /= 10;
 *
 *    FIXED POINT:
 *    write similar to int, but write fp at correct position.
 *    num=1234, and fp=0 => 1234 (don't show .)
 *    num=1234, and fp=1 => 123.4
 *      num=1234, and fp=4 => 0.123 (option 1, too unpredictable and heavy)
 *    num=1234, and fp=4 => 1234 (simply don't show fp as it's invalid)
 *
 */

void SevenSeg::writeClock(int ss){

  writeClock(ss/60,ss%60);

}

void SevenSeg::writeClock(int ss, char c){

  writeClock(ss/60,ss%60,c);

}

void SevenSeg::writeClock(int mm, int ss){

  // Use ':' if assigned, '.' otherwise, or simply nothing if none assigned

  if(_colonSegPin!=-1){
    writeClock(mm,ss,':');
  } else if(_DP!=-1){
    writeClock(mm,ss,'.');
  } else {
    writeClock(mm,ss,'_');
  }

}

void SevenSeg::writeClock(int mm, int ss, char c){

  if(_timerID==-1){  // No timer assigned. MUX once.

    int num = mm*100+ss;

    // colon through symbpin? 1 if yes.
    int symbColon = (_symbDigPin!=-1);

    for(int i=_numOfDigits-1;i>=0;i--){
      changeDigit(i);
      int nextDigit = num % 10;
      writeDigit(nextDigit);       // Possible future update: don't write
insignificant zeroes
```

```
      if(c==':' && !symbColon) setColon();
      if((c=='.')&&(i==_numOfDigits-3)) setDP();  // Only set "." in the right
place
      num /= 10;
      execDelay(_digitOnDelay);
      if(c==':' && !symbColon) clearColon();
      if(c=='.') clearDP();
      writeDigit(' ');
      execDelay(_digitOffDelay);
    }

    if(symbColon && c==':'){
      changeDigit('s');
      setColon();
      execDelay(_digitOnDelay);
      clearColon();
      execDelay(_digitOffDelay);
    }

  } else {

    _writeMode=c;
    _writeInt=mm*100+ss;

  }

}

void SevenSeg::write(int num,int point){
  write((long int)num, point);
}

void SevenSeg::write(long int num,int point){

  if(_timerID==-1){  // No timer assigned. MUX once.

    // Compute the maximum positive and negative numbers possible to display
    // (TBD: Move this to a computation done on pin assignments?)
    long int maxNegNum=1;
    for(int i=1;i<=_numOfDigits-1;i++) maxNegNum*=10;
    long int maxPosNum=10*maxNegNum-1;
    maxNegNum=-maxNegNum+1;

    // TBD: Change to displaying OL (overload) or ---- or similar?
    if(num>maxPosNum) num=maxPosNum;
    if(num<maxNegNum) num=maxNegNum;

    if(point==0){    // Don't display decimal point if zero decimals used
      point=_numOfDigits;          // value if-sentence won't trigger on
    } else {
      point=_numOfDigits-point-1;  // Map number of decimal points to digit
number
    }
```

```
    // TBD: Fix minus

    int minus=0;
    if(num<0){
      num*=-1;
      minus=1;
    }

/* USED IN v1.0 - DOESN'T SUPPRESS LEADING ZEROS
    for(int i=_numOfDigits-1;i>=0;i--){
      changeDigit(i);
      int nextDigit = num % 10L;
      if(minus&&i==0) writeDigit('-');
      else writeDigit(nextDigit);       // TBD: Possible future update: don't
write insignificant zeroes
      if(point==i) setDP();
      num /= 10;
      execDelay(_digitOnDelay);
      writeDigit(' ');
      clearDP();
      execDelay(_digitOffDelay);
    }
*/

    for(int i=_numOfDigits-1;i>=0;i--){
        changeDigit(i);
        int nextDigit = num % 10L;
        if(num || i>point-1 || i==_numOfDigits-1){
            writeDigit(nextDigit);
        } else if(minus){
            writeDigit('-');
            minus=0;
        } else {
            writeDigit(' ');
        }
        if(point==i) setDP();
        num /= 10;
        execDelay(_digitOnDelay);
        writeDigit(' ');
        clearDP();
        execDelay(_digitOffDelay);
    }

  } else {  // Use timer

    if(point==0){    // Don't display decimal point if zero decimals used
      point=_numOfDigits;          // value if-sentence won't trigger on
    } else {
      point=_numOfDigits-point-1;  // Map number of decimal points to digit
number
    }

      _writeMode = 'p'; // Tell interruptAction that write(int,int) was used
(fixed point).
```

```
      _writeInt = iaLimitInt(num);  // Tell interruptAction to write this
number ...
        _writePoint = point;  // ... with this fixed point
  }

}

// Extracts digit number "digit" from "number" for use with ia -
interruptAction
char SevenSeg::iaExtractDigit(long int number, int digit, int point){

/* OLD VERSION WITHOU ZERO SUPPRESION (v1.0)
  if(number<0){
    if(digit==0) return '-';
    number*=-1;
  }
  for(int i=0;i<_numOfDigits-digit-1;i++) number/=10L;
  return (char)((number%10L)+48L);
*/

  long int old_number = number;
  int minus = 0;
  if(number<0){
    number*=-1;
    minus = 1;
  }

  if(digit!=_numOfDigits-1){
    for(int i=0;i<_numOfDigits-digit-1;i++) number/=10L;
  }

  if(digit>point-1 || digit==_numOfDigits-1 || number!=0) return (char)
((number%10L)+48L);
  else {
    if(iaExtractDigit(old_number,digit+1,point)!='-' && iaExtractDigit
(old_number,digit+1,point)!=' ' && minus) return '-';
    else return ' ';
  }

//  else if(iaExtractDigit(old_number,digit+1,point)=='-' && minus) return
'-';
//  else if(iaExtractDigit(old_number,digit+1,point)!=' ' && iaExtractDigit
(old_number,digit+1,point)!='-' && minus) return '-';
//  else ' ';
/*
  else if(iaExtractDigit(number,digit+1,point)=='0'){
    if(minus) return '-';
    else return ' ';
  } else return ' ';
  */

}

// Limits integer similar to how it's done in write(int,int)
```

```
long int SevenSeg::iaLimitInt(long int number){


    // Compute the maximum positive and negative numbers possible to display
    // (TBD: Move this to a computation done on pin assignments?)
    long int maxNegNum=1;
    for(int i=1;i<=_numOfDigits-1;i++) maxNegNum*=10;
    long int maxPosNum=10*maxNegNum-1;
    maxNegNum=-maxNegNum+1;

    // TBD: Change to displaying OL (overload) or ---- or similar?
    if(number>maxPosNum) number=maxPosNum;
    if(number<maxNegNum) number=maxNegNum;

    return number;

}

void SevenSeg::write(int num){
  write((long int)num);
}

void SevenSeg::write(long int num){

  if(_timerID==-1){  // No timer assigned. MUX once.

    write(num,0);

  } else {  // Use timer

      _writeMode = 'i'; // Tell interruptAction that write(int) is used.
      _writeInt = iaLimitInt(num);  // Tell interruptAction to write this int
  }

}
/*
void SevenSeg::writeDisplay(int A,int B,int C,int D,int colon){

  // Rewrite. Take caution to properly shut down all symbols for the duty
cycle control.
  // This can be done by turning off the digit rather than the symbol, but
that does not
  // work for symbols not connected to any digit (i.e. hardwired). Hence it
should happen
  // at a segment level.x

  int digits[4];
  digits[0]=A;
  digits[1]=B;
  digits[2]=C;
  digits[3]=D;
  for(int i=0;i<_numOfDigits;i++){
```

```
    if(_digitOnDelay!=0){           // delayMicroseconds(0) yields a large
delay for some reason. Hence the if-sentence.
      changeDigit(i);
      writeDigit(digits[i]);
      //if(colon) setColon();
      delayMicroseconds(_digitOnDelay);   // TBD: change to execDelay()
    }
    if(_digitOffDelay!=0){          // delayMicroseconds(0) yields a large
delay for some reason. Hence the if-sentence.
      changeDigit(' ');
      //writeDigit(' ');
      //clearColon();
      delayMicroseconds(_digitOffDelay);// TBD: change to execDelay()
    }
  }
}
*/
void SevenSeg::write(char *str){

  if(_timerID==-1){  // No timer assigned. MUX once.

    int i=0;
    int j=0;
    clearColon();
    while(str[i]!='\0'){
      changeDigit(j);
      writeDigit(str[i]);
      if(str[i+1]=='.'){
        setDP();
        i++;
      }
      execDelay(_digitOnDelay);
      writeDigit(' ');
      clearDP();
      execDelay(_digitOffDelay);
      i++;
      j++;
    }

  } else {  // Use timer
      _writeMode = 's'; // Tell interruptAction that write(char*) is used.
      _writeStr = str;  // Tell interruptAction to write this string
  }


}

void SevenSeg::write(String str){

  if(_timerID==-1){  // No timer assigned. MUX once.

    int i=0;
    int j=0;
    clearColon();
```

```
    while(i<str.length()){
      changeDigit(j);
      writeDigit(str[i]);
      if(str[i+1]=='.'){
        setDP();
        i++;
      }
      execDelay(_digitOnDelay);
      writeDigit(' ');
      clearDP();
      execDelay(_digitOffDelay);
      i++;
      j++;
    }

  } else {  // Use timer

      _writeMode = 'o'; // Tell interruptAction that write(String) is used.
      _writeStrObj = str;     // Tell interruptAction to write this string
  }


}

void SevenSeg::write(double num, int point){
    for(int i=0;i<point;i++) num*=10;
    long int intNum = (long int) num;
    double remainder = num - intNum;
    if(remainder>=0.5 && num>0) intNum++;
    if(remainder<=0.5 && num<0) intNum--;
    write(intNum,point);
}

void SevenSeg::write(double num){

    // Compute the maximum positive and negative numbers possible to display
    // (TBD: Move this to a computation done on pin assignments?)
    long int maxNegNum=1;
    for(int i=1;i<=_numOfDigits-1;i++) maxNegNum*=10;
    long int maxPosNum=10*maxNegNum-1;
    maxNegNum=-maxNegNum+1;

    if(num>maxPosNum) num=maxPosNum;
    if(num<maxNegNum) num=maxNegNum;

    int point=0;
    if(num<0&&num>-1){
      while(num*100<=maxPosNum && num*100>=maxNegNum && point<_numOfDigits-2){
        num*=10;
        point++;
      }
      if((int)num==0){
        point++; // The minus sign will disappear
      }
```

```
    } else if(num>0&&num<1){
      while(num*100<=maxPosNum && num*100>=maxNegNum && point<_numOfDigits-1){
        num*=10;
        point++;
      }
    } else {
      while(num*10<=maxPosNum && num*10>=maxNegNum){
        num*=10;
        point++;
      }
    }

    // Implementing correct round-off
    double rest=num;
    if(rest<0) rest*=-1;
    rest=rest-(long int)rest;
    if(rest>=0.5&&num>0) num++;
    if(rest>=0.5&&num<0) num--;

    if(_timerID==-1){

      write((long int)num,point);

    } else { // user timer

      // Adapting to another format
      point=point+1-_numOfDigits;

      _writeMode='f';
      _writePoint=-point;
      _writeInt=(long int)num;

    }
}

void SevenSeg::updDelay(){

  // On-time for each display is total time spent per digit times the duty
cycle. The
  // off-time is the rest of the cycle for the given display.

  long int temp = _digitDelay;            // Stored into long int since
temporary variable gets larger than 32767
  temp *= _dutyCycle;               // Multiplication in this way to prevent
multiplying two "shorter" ints.
  temp /= 100;                      // Division after multiplication to
minimize round-off errors.
  _digitOnDelay=temp;
  _digitOffDelay=_digitDelay-_digitOnDelay;

  if(_timerID!=-1){
    // Artefacts in duty cycle control appeared when these values changed
while interrupts happening (A kind of stepping in brightness appeared)
    cli();
```

```
    _timerCounterOnEnd=(_digitOnDelay/16)-1;
    _timerCounterOffEnd=(_digitOffDelay/16)-1;
    if(_digitOnDelay==0) _timerCounterOnEnd=0;
    if(_digitOffDelay==0) _timerCounterOffEnd=0;
    _timerCounter=0;
    sei();
  }
}

void SevenSeg::interruptAction(){

  // Increment the library's counter
  _timerCounter++;

  // Finished with on-part. Turn off digit, and switch to the off-phase
(_timerPhase=0)
  if((_timerCounter>=_timerCounterOnEnd)&&(_timerPhase==1)){
    _timerCounter=0;
    _timerPhase=0;

    writeDigit(' ');

    // If a write mode using . is used it is reasonable to assume that DP
exists. Clear it (eventhough it might not be on this digit).
    if(_writeMode=='p'||_writeMode=='.'||_writeMode=='s'||_writeMode=='f')
clearDP();
    if(_writeMode==':') clearColon();

  }

  // Finished with the off-part. Switch to next digit and turn it on.
  if((_timerCounter>=_timerCounterOffEnd)&&(_timerPhase==0)){
    _timerCounter=0;
    _timerPhase=1;

    _timerDigit++;

    //if(_timerDigit>=_numOfDigits) _timerDigit=0;
    if(_timerDigit>=_numOfDigits){
      if(_symbDigPin!=-1 && _timerDigit==_numOfDigits){  // Symbol pin in use.
Let _timerDigit=_numOfDigits be used for symbol mux.
      } else { // Finished muxing symbol digit, or symbol pin not in use
        _timerDigit=0;
      }
    }

    if(_timerDigit==_numOfDigits) changeDigit('s');
    else changeDigit(_timerDigit);
//    writeDigit(_timerDigit);

    if(_writeMode=='p'||_writeMode=='f'){ // Fixed point writing (or float)
      writeDigit(iaExtractDigit(_writeInt,_timerDigit,_writePoint));
      if(_writePoint==_timerDigit) setDP();
    }
```

```
    if(_writeMode=='i'){      // Fixed point writing
      writeDigit(iaExtractDigit(_writeInt,_timerDigit,_numOfDigits));
    }

    if(_writeMode==':'||_writeMode=='.'||_writeMode=='_'){

      // colon through symbpin? 1 if yes.
      int symbColon = (_symbDigPin!=-1);

      if(_timerDigit==_numOfDigits){      // Symbol digit
        setColon();
      } else {
        writeDigit(iaExtractDigit(_writeInt,_timerDigit,_numOfDigits));
        if(_writeMode==':' && !symbColon) setColon();
        if((_writeMode=='.')&&(_timerDigit==_numOfDigits-3)) setDP();  // Only
set "." in the right place
      }

    }

    if(_writeMode=='s'){

      // This algorithm must count to the correct letter i in _writeStr for
digit j, since the two may be unmatched
      // and it is impossible to know which letter to write without counting
      int i=0; // which digit
      int j=0; // which digit have it counted to
      while(_writeStr[i]!='\0' && j<_timerDigit){
        if(_writeStr[i+1]=='.'){
          i++;
        }
        i++;
        j++;
      }
      writeDigit(_writeStr[i]);
      if(_writeStr[i+1]=='.') setDP();

    }

    if(_writeMode=='o'){

      // This algorithm must count to the correct letter i in _writeStr for
digit j, since the two may be unmatched
      // and it is impossible to know which letter to write without counting
      int i=0; // which digit
      int j=0; // which digit have it counted to
      while(i<_writeStrObj.length() && j<_timerDigit){
        if(_writeStrObj[i+1]=='.'){
          i++;
        }
        i++;
        j++;
      }
```

```
      writeDigit(_writeStrObj[i]);
      if(_writeStrObj[i+1]=='.') setDP();

    }

  }


/*
  // If we're in the on-part of the cycle and has counted as many interrupts
corresponding to one on-phase
  if((_timerCounter>=_timerCounterOnEnd) && (_timerPhase==1)){
    _timerCounter=0;    // Reset the library's counter
    _timerPhase=0;      // Switch to off-phase

    // Turn off this digit
    writeDigit(' ');

  }

  // Similar for the off-phase.
  if((_timerCounter>=_timerCounterOffEnd) && (_timerPhase==0)){
    _timerCounter=0;
    _timerPhase=1;

    // Turn on the next digit
    _timerDigit++;
    if(_timerDigit>=_numOfDigits){
      _timerDigit=0;
    }
    changeDigit(_timerDigit);
    writeDigit(_timerDigit);

  }
*/
}

/*
void SevenSeg::setDigitPins(int numOfDigits, int *pDigitPins){

  if(_numOfDigits>0){
//    delete [] _dig;
    free(_dig);
  }

  _numOfDigits = numOfDigits;

//  _dig = new int[numOfDigits];
  _dig = (int*)malloc(_numOfDigits * sizeof(int));

//  memcpy(_dig, pDigitPins, numOfDigits);
  for(int i=0;i<_numOfDigits;i++){
    _dig[i]=pDigitPins[i];
  }
```

```
}
*/
void SevenSeg::changeDigit(int digit){


  // Turn off all digits/segments first.
  // If you swith on a new digit before turning off the segments you will get
  // a slight shine of the "old" number in the "new" digit.
  clearDisp();
  digitalWrite(_dig[digit], _digOn);

}

void SevenSeg::changeDigit(char digit){

  if(digit=='s'){
    // change to the symbol digit
    clearDisp();
    digitalWrite(_symbDigPin, _digOn);
    digitalWrite(_colonSegPin, _colonState);
    digitalWrite(_colonSegLPin, _colonState);
    digitalWrite(_aposSegPin, _aposState);
  }

  if(digit=' '){
    clearDisp();
  }

}

void SevenSeg::setDPPin(int DPPin){

  _DP=DPPin;
  pinMode(_DP, OUTPUT);

}

void SevenSeg::setDP(){

  digitalWrite(_DP, _segOn);

}

void SevenSeg::clearDP(){

  digitalWrite(_DP, _segOff);

}
/*
    void setDPPin(int);
    void setDP();
    void clearDP();
```

Characters: Colon, apostrophe, comma(DP), randomly assignable symbols?
Most symbols can be confined to one character, but colon should be able to assign in two parts (UC and LC).
Yet two colons are also present on some displays. And on some displays, colons and apostorphes are treated as a separate digit using an additional common cathode/anode.

I've decided to treat the symbols in the following way

DP is assigned as an eight segment for each digit, as this is the only way I've seen it done. Simple.
I want the functions setDPPin(int), setDP(), clearDP(). DP should be cleared at each changeDigit (i.e. in ClearDisp?)
In the parsing function it should be possible to write "1.2.3.4.".

Colon are treated in many different ways on many different displays. I want a function setColonPin() that are overloaded
and take most of the case. setColon() and clearColon() should turn it on or off. I'll explain the scenarios, and the syntax:

Colon may be split in two parts UC (upper colon) and LC (lower colon) or colon may be hardwired as one LED

1. Colon has its own cathodes and anodes. Ground the cathode if common cathode or tie anode to supply in case of common anode.
   Syntax: setColonPin(segPin) where segPin is the other pin. In this case the colon needs not be multiplexed. If split segment pin
   for UC/LC, the user joins them together. setColon()/clearColon() writes directly to segmentPin.
2. UC/LC is joined together using its own segment pin, and shares common anode/cathode with one of the digits.
   Syntax: setColonPin(segPin,digPin). The function will detect at what digit to type the colon based on digitPin, and store this in colonDigU and
   colonDigL. If no digit is tied to digitPin issue an error. colonState is a private member variable being set or cleared by setColon()/clearColon().
   writeDigit() checks colonState when on colonDigU or colonDigL digits and writes accordingly.
3. UC shares common anode/cathode digit pin with one of the digits, while LC shares with another digit. They are the same segment pin.
   Syntax: setColonPin(segPin,digLPin,digUPin). This works in the same way as above, except that different values are stored to DigU and DigL.
4. UC and LC are treated like seperate segments on a new "symbol" digit pin(!). In these cases there is usually also an apostrophe (A) segment.
   The UC and LC segments should be joined together into one segment (C).
   Syntax: setSymbolPin(SegCPin,SegAPin,digPin) digPin will be stored to symbDigPin, and the multiplexing must occur over one additional digit.
   This implies modification to i.e. setRefreshRate, setDigitDelay and maybe other functions. All functions must be checked.

Members needed in class:

```
  private:
    colonState      ;     // _segOn or _segOff.
  aposState;  // _segOn or _segOff.
```

```
        colonDigU;      // Which digits to activate colon at (one digit for UC
and one for LC)
      colonDigL;  // colonDigU==colonDigL==-1 means that it is treated as a
separate digit, case 1 or 4. (check negative numbers for int)
      symbDigPin; // If the colon is on a separate digit pin the symbol digit
pin number is stored here. Otherwise, it is -1. Non-zero values imply more
muxing.
        colonSegPin;
      aposSegPin;
      public:
      setColonPin(int);
        setColonPin(int,int);
      setColonPin(int,int,int);
        setSymbPin(int,int,int,int);
        setColon();
        clearColon();
      setApos();
      clearApos();

  The parsing function should parse colon to off except when any colon present
in string. Same with apostorphe. I.e. "34:07". I initially wanted to have
support for two
  colons since you need that on a watch. However, I've settled on only one
colon since there are almost none display available with two colons. If I find
one, and will
  use one, I will simply duplicate the colon stuff in my class.

  Actually, cases 1, 2 and 3 can be joined together! The colon can be turned
on irrespective of what digits they are on (at least as long as there are only
one colon).
  This simplifies the class:

      private:
        colonState;     // _segOn or _segOff.
      aposState;  // _segOn or _segOff.
      symbDigPin; // If the colon is on a separate digit pin the symbol digit
pin number is stored here. Otherwise, it is -1. Non-zero values imply more
muxing.
        colonSegPin;
      aposSegPin;
      public:
      setColonPin(int);
        setSymbPin(int,int,int,int);
        setColon();
        clearColon();
      setApos();
      clearApos();

*/

void SevenSeg::setColonPin(int colonPin){
  _colonSegPin=colonPin;
  pinMode(_colonSegPin,OUTPUT);
  digitalWrite(_colonSegPin, _colonState);
```

```
}

void SevenSeg::setSymbPins(int digPin, int segUCPin, int segLCPin, int
segAPin){
  _colonSegPin=segUCPin;
  _colonSegLPin=segLCPin;
  _aposSegPin=segAPin;
  _symbDigPin=digPin;
  pinMode(_colonSegPin,OUTPUT);
  pinMode(_colonSegLPin,OUTPUT);
  pinMode(_aposSegPin,OUTPUT);
  pinMode(_symbDigPin,OUTPUT);
  digitalWrite(_colonSegPin, _colonState);
  digitalWrite(_colonSegLPin, _colonState);
  digitalWrite(_aposSegPin, _aposState);
}

/*
The functions for setColon(), clearColon(), setApos(), clearApos() directly
sets or clears the
segment pins if no symbol pin is assigned. Since no symbol pin is assigned
colon (apos isn't set) has
a separate segment pin "Colon" and shares a digit pin with one or two other
digits
(in case it is split into UC and LC). In this case it makes sense to control
it just like other
segments; by setting and clearing the segment pin with setColon() or
clearColon() after the correct
digit is selected with changeDigit(). Compare with setDP()/clearDP().
Furthermore, it is not necessary
to identify WHICH digit the colon segments apply for since, if colon is turned
on, one may simply switch
on the segment pin for all digits. Nothing will be tied to the colon segment
pin for other digits than
those it applies to, hence it is sufficient to initialize this kind of
hardware with setColon(int colonPin).
Sometimes, a colon is present as one or two complete stand-alone LEDs. In this
case, the can be wired
up into one of these configurations to work.

In the other main case, a separate symbol pin is assigned for colon and
apostrophe. This is actually an
additional digit pin which must be muxed across. The segment pins are shared
with other segments such as
A-G. This is a compact way of allowing many symbols (colon and apostrophe)
while only adding one more pin.
This configuration is programmed with setSymbPins(int digPin, int segUCPin,
int segLCPin, int segAPin),
where digPin is the symbol digit pin and the other pins are the pins used for
segment UC, LC and apos.
If colon is present as one segment only, segUCPin and segULPin can be the same
value. Sometimes, colon
and apostorphe are present as stand-alone diodes with their own cathodes and
anodes not being connected
```

```
to anything else. In this case, join their cathodes or anodes (in case of
common cathode or anode respectively)
and connect their other terminal to one segment pin each to make the mentioned
configuration.
The behaviour set/clear behaviour of these digits are a bit different in this
case. The set/clear-funciton
only sets a flag to on or off. In order to type the characters you must mux to
the symbol digit by issuing
changeDigit('s'). This function will light up the appropriate symbols in
accordance with the flags.
*/

void SevenSeg::setColon(){
  _colonState=_segOn;
  if(_symbDigPin==-1){
    digitalWrite(_colonSegPin, _segOn);
  }
}

void SevenSeg::clearColon(){
  _colonState=_segOff;
  if(_symbDigPin==-1){
    digitalWrite(_colonSegPin, _segOff);
  }
}

void SevenSeg::setApos(){
  _aposState=_segOn;
  if(_symbDigPin==-1){
    digitalWrite(_aposSegPin, _segOn);
  }
}

void SevenSeg::clearApos(){
  _aposState=_segOff;
  if(_symbDigPin==-1){
    digitalWrite(_aposSegPin, _segOff);
  }
}

void SevenSeg::writeDigit(int digit){

  // Turn off all LEDs first to avoid running current through too many LEDs at
once.
  digitalWrite(_A, _segOff);
  digitalWrite(_B, _segOff);
  digitalWrite(_C, _segOff);
  digitalWrite(_D, _segOff);
  digitalWrite(_E, _segOff);
  digitalWrite(_F, _segOff);
  digitalWrite(_G, _segOff);

  if(digit==1){
    digitalWrite(_B, _segOn);
```

```
    digitalWrite(_C, _segOn);
  }

  if(digit==2){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_G, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_D, _segOn);
  }

  if(digit==3){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_G, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
  }

  if(digit==4){
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
  }

  if(digit==5){
    digitalWrite(_A, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
  }

  if(digit==6){
    digitalWrite(_A, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit==7){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
  }

  if(digit==8){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
```

```
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit==9){
    digitalWrite(_G, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
  }

  if(digit==0){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

}

void SevenSeg::writeDigit(char digit){

  // Turn off all LEDs first. Run writeDigit(' ') to clear digit.
  digitalWrite(_A, _segOff);
  digitalWrite(_B, _segOff);
  digitalWrite(_C, _segOff);
  digitalWrite(_D, _segOff);
  digitalWrite(_E, _segOff);
  digitalWrite(_F, _segOff);
  digitalWrite(_G, _segOff);

  if(digit=='-'){
    digitalWrite(_G, _segOn);
  }

  if(digit=='\370'){ // ASCII code 248 or degree symbol: '°'
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  // Digits are numbers. Write with writeDigit(int)
  if(digit>=48&&digit<=57)  writeDigit(digit-48);

  // Digits are small caps letters. Capitalize.
  if(digit>=97&&digit<=122) digit-=32;

  if(digit=='A'){
```

```
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='B'){
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='C'){
    digitalWrite(_A, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='D'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='E'){
    digitalWrite(_A, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='F'){
    digitalWrite(_A, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='G'){
/*
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_G, _segOn);
```

```
    // TBD: Really write G like a 9, when it can be written as almost G?
*/
    digitalWrite(_A, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='H'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='I'){
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='J'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
  }

  if(digit=='K'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='L'){
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='M'){
    digitalWrite(_A, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
  }

  if(digit=='N'){
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_G, _segOn);
  }
```

```
  if(digit=='O'){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='P'){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='Q'){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='R'){
    digitalWrite(_E, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='S'){
    digitalWrite(_A, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='T'){
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

 if(digit=='U'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
  }
```

```
  if(digit=='V'){
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
  }

  if(digit=='W'){
    digitalWrite(_B, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_F, _segOn);
  }

  if(digit=='X'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='Y'){
    digitalWrite(_B, _segOn);
    digitalWrite(_C, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_F, _segOn);
    digitalWrite(_G, _segOn);
  }

  if(digit=='Z'){
    digitalWrite(_A, _segOn);
    digitalWrite(_B, _segOn);
    digitalWrite(_D, _segOn);
    digitalWrite(_E, _segOn);
    digitalWrite(_G, _segOn);
  }
}

void SevenSeg::execDelay(int usec){

  if(usec!=0){    // delay() and delayMicroseconds() don't handle 0 delay

    if(usec<=16383)     delayMicroseconds(usec);      // maximum value for
delayMicroseconds();
    else          delay(usec/1000);

  }

}
```

keywords.txt

```
#######################################
# SevenSeg v1.1
# keywords.txt - Syntax Coloring Map
# Sascha Bruechert, 05.02.2015
#######################################

#######################################
# Datatypes (KEYWORD1)
#######################################
SevenSeg    KEYWORD1


#######################################
# Methods and Functions (KEYWORD2)
#######################################
digitPins   KEYWORD2
numOfDigits KEYWORD2

## Low level functions for initializing hardware
setCommonAnode    KEYWORD2
setCommonCathode  KEYWORD2
setDigitPins      KEYWORD2
setActivePinState KEYWORD2
setDPPin    KEYWORD2
setColonPin KEYWORD2
setSymbPins KEYWORD2

## Low level functions for printing to display
clearDisp   KEYWORD2
changeDigit KEYWORD2
writeDigit  KEYWORD2
setDP KEYWORD2
clearDP     KEYWORD2
setColon    KEYWORD2
clearColon  KEYWORD2
setApos     KEYWORD2
clearApos   KEYWORD2

## Low level functions for controlling multiplexing
setDigitDelay     KEYWORD2
setRefreshRate    KEYWORD2
setDutyCycle      KEYWORD2

## High level functions for printing to display
write KEYWORD2
writeClock  KEYWORD2

## Timer control functions
setTimer    KEYWORD2
clearTimer  KEYWORD2
startTimer  KEYWORD2
```

```
stopTimer	KEYWORD2
interruptAction	KEYWORD2

#######################################
# Instances (KEYWORD2)
#######################################

#######################################
# Constants (LITERAL1)
#######################################
```

STANDALONE_CONTROL_7_SEGMENT.ino

```
// ***************************************
//            Absolutelyautomation.com
//
//  PANEL MOUNT 7 SEGMENT DISPLAY
//  WITH EMBEDDED ARDUINO
//
//  Used as standalone temperature controller
//  with a 1-wire ds18b20 sensor
//
//***************************************


#include <SevenSeg.h>
#include <DallasTemperature.h>
#include <OneWire.h>


// Segments setup
int sega = 12;
int segb = 10;
int segc = 7;
int segd = 5;
int sege = 3;
int segf = 11;
int segg = 2;
int segp = 6;

// Commons setup
int digit1 = 13;
int digit2 = A0;
int digit3 = A1;

String dispvalue = "------";
char StrArray[4];

int FWMajorVersion=0;
int FWMinorVersion=1;


SevenSeg disp( sega, segb, segc, segd, sege, segf, segg );
const int numOfDigits = 3;
int digitPins[numOfDigits] = { digit1, digit2, digit3  };

int firstboot;

String astring = "A00";
String fstring = "F00";


// pin for the sensor data wire
#define ONE_WIRE_BUS A3
```

```
// Instance oneWire to start communication with any OneWire device(not only
Maxim/Dallas temperature)
OneWire oneWire(ONE_WIRE_BUS);
// Pass oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

float  TempInC  ; // Float Variable temp in C
int    RoundTempInC  ; //Integer variable temp in C


String stringTemp = "  ";

void setup() {

  // Display setup

  pinMode(sega, OUTPUT);
  pinMode(segb, OUTPUT);
  pinMode(segc, OUTPUT);
  pinMode(segd, OUTPUT);
  pinMode(sege, OUTPUT);
  pinMode(segf, OUTPUT);
  pinMode(segg, OUTPUT);
  pinMode(segp, OUTPUT);

  pinMode(digit1, OUTPUT);
  pinMode(digit2, OUTPUT);
  pinMode(digit3, OUTPUT);


  disp.setDPPin(segp);
  //disp.setCommonCathode();
  disp.setCommonAnode();
  disp.setDigitPins( numOfDigits, digitPins );
  disp.setRefreshRate(300);
  disp.setTimer(1);  // Don't use with servo functions at the same time!
  //disp.setTimer(2);  // Don't use with tone() at the same time!
  disp.startTimer();


 // One wire sensors
 sensors.begin();

 // control Output
 pinMode(A4, OUTPUT);

}

// infinite loop:
void loop() {
  if(!firstboot){
    disp.write("8.8.8.");
    delay(500);
    disp.write("");
```

```
    delay(500);
    fstring = "U"+String(FWMajorVersion,DEC)+"."+String(FWMinorVersion,DEC);
    disp.write(fstring);// FW version = MajorVersion.MinorVersion
    delay(500);
    disp.write("");
    delay(500);

    firstboot=1;
  }


  sensors.requestTemperatures();
  //TempInC=sensors.getTempFByIndex(0);
  TempInC=sensors.getTempCByIndex(0);

  // comparation not very reliable with float numbers, use with care!!
  if( TempInC > 99.9 )
  {
    dtostrf(TempInC, 3, 0, StrArray);
  }
  else
  {
    dtostrf(TempInC, 4, 1, StrArray);
  }
  dispvalue = StrArray;


  // Simplistic control example
  // comparation not very reliable with float numbers, use with care!!
  if( TempInC > 33.0 )
  {
    digitalWrite(A4,HIGH);
  }
  else
  {
    digitalWrite(A4,LOW);
  }


  // display refresh
  disp.write(dispvalue);

}

ISR(TIMER1_COMPA_vect)   {
  disp.interruptAction();
}
```

OneWire.h

```
#ifndef OneWire_h
#define OneWire_h

#include <inttypes.h>

#if ARDUINO >= 100
#include "Arduino.h"       // for delayMicroseconds, digitalPinToBitMask, etc
#else
#include "WProgram.h"      // for delayMicroseconds
#include "pins_arduino.h"  // for digitalPinToBitMask, etc
#endif

// You can exclude certain features from OneWire.  In theory, this
// might save some space.  In practice, the compiler automatically
// removes unused code (technically, the linker, using -fdata-sections
// and -ffunction-sections when compiling, and Wl,--gc-sections
// when linking), so most of these will not result in any code size
// reduction.  Well, unless you try to use the missing features
// and redesign your program to not need them!  ONEWIRE_CRC8_TABLE
// is the exception, because it selects a fast but large algorithm
// or a small but slow algorithm.

// you can exclude onewire_search by defining that to 0
#ifndef ONEWIRE_SEARCH
#define ONEWIRE_SEARCH 1
#endif

// You can exclude CRC checks altogether by defining this to 0
#ifndef ONEWIRE_CRC
#define ONEWIRE_CRC 1
#endif

// Select the table-lookup method of computing the 8-bit CRC
// by setting this to 1.  The lookup table enlarges code size by
// about 250 bytes.  It does NOT consume RAM (but did in very
// old versions of OneWire).  If you disable this, a slower
// but very compact algorithm is used.
#ifndef ONEWIRE_CRC8_TABLE
#define ONEWIRE_CRC8_TABLE 1
#endif

// You can allow 16-bit CRC checks by defining this to 1
// (Note that ONEWIRE_CRC must also be 1.)
#ifndef ONEWIRE_CRC16
#define ONEWIRE_CRC16 1
#endif

#define FALSE 0
#define TRUE  1

// Platform specific I/O definitions
```

```
#if defined(__AVR__)
#define PIN_TO_BASEREG(pin)             (portInputRegister(digitalPinToPort
(pin)))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
#define IO_REG_TYPE uint8_t
#define IO_REG_ASM asm("r30")
#define DIRECT_READ(base, mask)         (((*(base)) & (mask)) ? 1 : 0)
#define DIRECT_MODE_INPUT(base, mask)   ((*((base)+1)) &= ~(mask))
#define DIRECT_MODE_OUTPUT(base, mask)  ((*((base)+1)) |= (mask))
#define DIRECT_WRITE_LOW(base, mask)    ((*((base)+2)) &= ~(mask))
#define DIRECT_WRITE_HIGH(base, mask)   ((*((base)+2)) |= (mask))

#elif defined(__MK20DX128__)
#define PIN_TO_BASEREG(pin)             (portOutputRegister(pin))
#define PIN_TO_BITMASK(pin)             (1)
#define IO_REG_TYPE uint8_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (*((base)+512))
#define DIRECT_MODE_INPUT(base, mask)   (*((base)+640) = 0)
#define DIRECT_MODE_OUTPUT(base, mask)  (*((base)+640) = 1)
#define DIRECT_WRITE_LOW(base, mask)    (*((base)+256) = 1)
#define DIRECT_WRITE_HIGH(base, mask)   (*((base)+128) = 1)

#elif defined(__SAM3X8E__)
// Arduino 1.5.1 may have a bug in delayMicroseconds() on Arduino Due.
// http://arduino.cc/forum/index.php/topic,141030.msg1076268.html#msg1076268
// If you have trouble with OneWire on Arduino Due, please check the
// status of delayMicroseconds() before reporting a bug in OneWire!
#define PIN_TO_BASEREG(pin)             (&(digitalPinToPort(pin)->PIO_PER))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
#define IO_REG_TYPE uint32_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (((*((base)+15)) & (mask)) ? 1 : 0)
#define DIRECT_MODE_INPUT(base, mask)   ((*((base)+5)) = (mask))
#define DIRECT_MODE_OUTPUT(base, mask)  ((*((base)+4)) = (mask))
#define DIRECT_WRITE_LOW(base, mask)    ((*((base)+13)) = (mask))
#define DIRECT_WRITE_HIGH(base, mask)   ((*((base)+12)) = (mask))
#ifndef PROGMEM
#define PROGMEM
#endif
#ifndef pgm_read_byte
#define pgm_read_byte(addr) (*(const uint8_t *)(addr))
#endif

#elif defined(__PIC32MX__)
#define PIN_TO_BASEREG(pin)             (portModeRegister(digitalPinToPort
(pin)))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
#define IO_REG_TYPE uint32_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (((*(base+4)) & (mask)) ? 1 : 0)  //
PORTX + 0x10
#define DIRECT_MODE_INPUT(base, mask)   ((*(base+2)) = (mask))            //
TRISXSET + 0x08
```

```
#define DIRECT_MODE_OUTPUT(base, mask)  ((*(base+1)) = (mask))            //
TRISXCLR + 0x04
#define DIRECT_WRITE_LOW(base, mask)    ((*(base+8+1)) = (mask))          //
LATXCLR  + 0x24
#define DIRECT_WRITE_HIGH(base, mask)   ((*(base+8+2)) = (mask))          //
LATXSET + 0x28

#else
#error "Please define I/O register types here"
#endif


class OneWire
{
  private:
    IO_REG_TYPE bitmask;
    volatile IO_REG_TYPE *baseReg;

#if ONEWIRE_SEARCH
    // global search state
    unsigned char ROM_NO[8];
    uint8_t LastDiscrepancy;
    uint8_t LastFamilyDiscrepancy;
    uint8_t LastDeviceFlag;
#endif

  public:
    OneWire( uint8_t pin);

    // Perform a 1-Wire reset cycle. Returns 1 if a device responds
    // with a presence pulse.  Returns 0 if there is no device or the
    // bus is shorted or otherwise held low for more than 250uS
    uint8_t reset(void);

    // Issue a 1-Wire rom select command, you do the reset first.
    void select(const uint8_t rom[8]);

    // Issue a 1-Wire rom skip command, to address all on bus.
    void skip(void);

    // Write a byte. If 'power' is one then the wire is held high at
    // the end for parasitically powered devices. You are responsible
    // for eventually depowering it by calling depower() or doing
    // another read or write.
    void write(uint8_t v, uint8_t power = 0);

    void write_bytes(const uint8_t *buf, uint16_t count, bool power = 0);

    // Read a byte.
    uint8_t read(void);

    void read_bytes(uint8_t *buf, uint16_t count);

    // Write a bit. The bus is always left powered at the end, see
```

```
    // note in write() about that.
    void write_bit(uint8_t v);

    // Read a bit.
    uint8_t read_bit(void);

    // Stop forcing power onto the bus. You only need to do this if
    // you used the 'power' flag to write() or used a write_bit() call
    // and aren't about to do another read or write. You would rather
    // not leave this powered if you don't have to, just in case
    // someone shorts your bus.
    void depower(void);

#if ONEWIRE_SEARCH
    // Clear the search state so that if will start from the beginning again.
    void reset_search();

    // Setup the search to find the device type 'family_code' on the next call
    // to search(*newAddr) if it is present.
    void target_search(uint8_t family_code);

    // Look for the next device. Returns 1 if a new address has been
    // returned. A zero might mean that the bus is shorted, there are
    // no devices, or you have already retrieved all of them.  It
    // might be a good idea to check the CRC to make sure you didn't
    // get garbage.  The order is deterministic. You will always get
    // the same devices in the same order.
    uint8_t search(uint8_t *newAddr);
#endif

#if ONEWIRE_CRC
    // Compute a Dallas Semiconductor 8 bit CRC, these are used in the
    // ROM and scratchpad registers.
    static uint8_t crc8(const uint8_t *addr, uint8_t len);

#if ONEWIRE_CRC16
    // Compute the 1-Wire CRC16 and compare it against the received CRC.
    // Example usage (reading a DS2408):
    //    // Put everything in a buffer so we can compute the CRC easily.
    //    uint8_t buf[13];
    //    buf[0] = 0xF0;    // Read PIO Registers
    //    buf[1] = 0x88;    // LSB address
    //    buf[2] = 0x00;    // MSB address
    //    WriteBytes(net, buf, 3);    // Write 3 cmd bytes
    //    ReadBytes(net, buf+3, 10);  // Read 6 data bytes, 2 0xFF, 2 CRC16
    //    if (!CheckCRC16(buf, 11, &buf[11])) {
    //        // Handle error.
    //    }
    //
    // @param input - Array of bytes to checksum.
    // @param len - How many bytes to use.
    // @param inverted_crc - The two CRC16 bytes in the received data.
    //                        This should just point into the received data,
    //                        *not* at a 16-bit integer.
```

```
    // @param crc - The crc starting value (optional)
    // @return True, iff the CRC matches.
    static bool check_crc16(const uint8_t* input, uint16_t len, const uint8_t*
inverted_crc, uint16_t crc = 0);

    // Compute a Dallas Semiconductor 16 bit CRC.  This is required to check
    // the integrity of data received from many 1-Wire devices.  Note that the
    // CRC computed here is *not* what you'll get from the 1-Wire network,
    // for two reasons:
    //   1) The CRC is transmitted bitwise inverted.
    //   2) Depending on the endian-ness of your processor, the binary
    //      representation of the two-byte return value may have a different
    //      byte order than the two bytes you get from 1-Wire.
    // @param input - Array of bytes to checksum.
    // @param len - How many bytes to use.
    // @param crc - The crc starting value (optional)
    // @return The CRC16, as defined by Dallas Semiconductor.
    static uint16_t crc16(const uint8_t* input, uint16_t len, uint16_t crc =
0);
#endif
#endif
};

#endif
```

OneWire.cpp

```
/*
Copyright (c) 2007, Jim Studt  (original old version - many contributors
since)

The latest version of this library may be found at:
  http://www.pjrc.com/teensy/td_libs_OneWire.html

OneWire has been maintained by Paul Stoffregen (paul@pjrc.com) since
January 2010.  At the time, it was in need of many bug fixes, but had
been abandoned the original author (Jim Studt).  None of the known
contributors were interested in maintaining OneWire.  Paul typically
works on OneWire every 6 to 12 months.  Patches usually wait that
long.  If anyone is interested in more actively maintaining OneWire,
please contact Paul.

Version 2.2:
  Teensy 3.0 compatibility, Paul Stoffregen, paul@pjrc.com
  Arduino Due compatibility, http://arduino.cc/forum/index.php?topic=141030
  Fix DS18B20 example negative temperature
  Fix DS18B20 example's low res modes, Ken Butcher
  Improve reset timing, Mark Tillotson
  Add const qualifiers, Bertrik Sikken
  Add initial value input to crc16, Bertrik Sikken
  Add target_search() function, Scott Roberts

Version 2.1:
  Arduino 1.0 compatibility, Paul Stoffregen
  Improve temperature example, Paul Stoffregen
  DS250x_PROM example, Guillermo Lovato
  PIC32 (chipKit) compatibility, Jason Dangel, dangel.jason AT gmail.com
  Improvements from Glenn Trewitt:
  - crc16() now works
  - check_crc16() does all of calculation/checking work.
  - Added read_bytes() and write_bytes(), to reduce tedious loops.
  - Added ds2408 example.
  Delete very old, out-of-date readme file (info is here)

Version 2.0: Modifications by Paul Stoffregen, January 2010:
http://www.pjrc.com/teensy/td_libs_OneWire.html
  Search fix from Robin James
    http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1238032295/27#27
  Use direct optimized I/O in all cases
  Disable interrupts during timing critical sections
    (this solves many random communication errors)
  Disable interrupts during read-modify-write I/O
  Reduce RAM consumption by eliminating unnecessary
    variables and trimming many to 8 bits
  Optimize both crc8 - table version moved to flash

Modified to work with larger numbers of devices - avoids loop.
Tested in Arduino 11 alpha with 12 sensors.
26 Sept 2008 -- Robin James
```

```
http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1238032295/27#27

Updated to work with arduino-0008 and to include skip() as of
2007/07/06. --RJL20

Modified to calculate the 8-bit CRC directly, avoiding the need for
the 256-byte lookup table to be loaded in RAM.  Tested in arduino-0010
-- Tom Pollard, Jan 23, 2008

Jim Studt's original library was modified by Josh Larios.

Tom Pollard, pollard@alum.mit.edu, contributed around May 20, 2008

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Much of the code was inspired by Derek Yerger's code, though I don't
think much of that remains.  In any event that was..
    (copyleft) 2006 by Derek Yerger - Free to distribute freely.

The CRC code was excerpted and inspired by the Dallas Semiconductor
sample code bearing this copyright.
//---------------------------------------------------------------------------
// Copyright (C) 2000 Dallas Semiconductor Corporation, All Rights Reserved.
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the
// Software is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included
// in all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
// OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
// MERCHANTABILITY,  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
```

```
// IN NO EVENT SHALL DALLAS SEMICONDUCTOR BE LIABLE FOR ANY CLAIM, DAMAGES
// OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
// ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
// OTHER DEALINGS IN THE SOFTWARE.
//
// Except as contained in this notice, the name of Dallas Semiconductor
// shall not be used except as stated in the Dallas Semiconductor
// Branding Policy.
//--------------------------------------------------------------------------
*/

#include "OneWire.h"


OneWire::OneWire(uint8_t pin)
{
      pinMode(pin, INPUT);
      bitmask = PIN_TO_BITMASK(pin);
      baseReg = PIN_TO_BASEREG(pin);
#if ONEWIRE_SEARCH
      reset_search();
#endif
}


// Perform the onewire reset function.  We will wait up to 250uS for
// the bus to come high, if it doesn't then it is broken or shorted
// and we return a 0;
//
// Returns 1 if a device asserted a presence pulse, 0 otherwise.
//
uint8_t OneWire::reset(void)
{
      IO_REG_TYPE mask = bitmask;
      volatile IO_REG_TYPE *reg IO_REG_ASM = baseReg;
      uint8_t r;
      uint8_t retries = 125;

      noInterrupts();
      DIRECT_MODE_INPUT(reg, mask);
      interrupts();
      // wait until the wire is high... just in case
      do {
            if (--retries == 0) return 0;
            delayMicroseconds(2);
      } while ( !DIRECT_READ(reg, mask));

      noInterrupts();
      DIRECT_WRITE_LOW(reg, mask);
      DIRECT_MODE_OUTPUT(reg, mask);      // drive output low
      interrupts();
      delayMicroseconds(480);
      noInterrupts();
      DIRECT_MODE_INPUT(reg, mask); // allow it to float
```

```
	delayMicroseconds(70);
	r = !DIRECT_READ(reg, mask);
	interrupts();
	delayMicroseconds(410);
	return r;
}

//
// Write a bit. Port and bit is used to cut lookup time and provide
// more certain timing.
//
void OneWire::write_bit(uint8_t v)
{
	IO_REG_TYPE mask=bitmask;
	volatile IO_REG_TYPE *reg IO_REG_ASM = baseReg;

	if (v & 1) {
		noInterrupts();
		DIRECT_WRITE_LOW(reg, mask);
		DIRECT_MODE_OUTPUT(reg, mask);	// drive output low
		delayMicroseconds(10);
		DIRECT_WRITE_HIGH(reg, mask); // drive output high
		interrupts();
		delayMicroseconds(55);
	} else {
		noInterrupts();
		DIRECT_WRITE_LOW(reg, mask);
		DIRECT_MODE_OUTPUT(reg, mask);	// drive output low
		delayMicroseconds(65);
		DIRECT_WRITE_HIGH(reg, mask); // drive output high
		interrupts();
		delayMicroseconds(5);
	}
}

//
// Read a bit. Port and bit is used to cut lookup time and provide
// more certain timing.
//
uint8_t OneWire::read_bit(void)
{
	IO_REG_TYPE mask=bitmask;
	volatile IO_REG_TYPE *reg IO_REG_ASM = baseReg;
	uint8_t r;

	noInterrupts();
	DIRECT_MODE_OUTPUT(reg, mask);
	DIRECT_WRITE_LOW(reg, mask);
	delayMicroseconds(3);
	DIRECT_MODE_INPUT(reg, mask); // let pin float, pull up will raise
	delayMicroseconds(10);
	r = DIRECT_READ(reg, mask);
	interrupts();
	delayMicroseconds(53);
```

```
      return r;
}

//
// Write a byte. The writing code uses the active drivers to raise the
// pin high, if you need power after the write (e.g. DS18S20 in
// parasite power mode) then set 'power' to 1, otherwise the pin will
// go tri-state at the end of the write to avoid heating in a short or
// other mishap.
//
void OneWire::write(uint8_t v, uint8_t power /* = 0 */) {
    uint8_t bitMask;

    for (bitMask = 0x01; bitMask; bitMask <<= 1) {
      OneWire::write_bit( (bitMask & v)?1:0);
    }
    if ( !power) {
      noInterrupts();
      DIRECT_MODE_INPUT(baseReg, bitmask);
      DIRECT_WRITE_LOW(baseReg, bitmask);
      interrupts();
    }
}

void OneWire::write_bytes(const uint8_t *buf, uint16_t count, bool power /*
= 0 */) {
  for (uint16_t i = 0 ; i < count ; i++)
    write(buf[i]);
  if (!power) {
    noInterrupts();
    DIRECT_MODE_INPUT(baseReg, bitmask);
    DIRECT_WRITE_LOW(baseReg, bitmask);
    interrupts();
  }
}

//
// Read a byte
//
uint8_t OneWire::read() {
    uint8_t bitMask;
    uint8_t r = 0;

    for (bitMask = 0x01; bitMask; bitMask <<= 1) {
      if ( OneWire::read_bit()) r |= bitMask;
    }
    return r;
}

void OneWire::read_bytes(uint8_t *buf, uint16_t count) {
  for (uint16_t i = 0 ; i < count ; i++)
    buf[i] = read();
}
```

```
//
// Do a ROM select
//
void OneWire::select(const uint8_t rom[8])
{
    uint8_t i;

    write(0x55);           // Choose ROM

    for (i = 0; i < 8; i++) write(rom[i]);
}

//
// Do a ROM skip
//
void OneWire::skip()
{
    write(0xCC);           // Skip ROM
}

void OneWire::depower()
{
      noInterrupts();
      DIRECT_MODE_INPUT(baseReg, bitmask);
      interrupts();
}

#if ONEWIRE_SEARCH

//
// You need to use this function to start a search again from the beginning.
// You do not need to do it for the first search, though you could.
//
void OneWire::reset_search()
{
  // reset the search state
  LastDiscrepancy = 0;
  LastDeviceFlag = FALSE;
  LastFamilyDiscrepancy = 0;
  for(int i = 7; ; i--) {
    ROM_NO[i] = 0;
    if ( i == 0) break;
  }
}

// Setup the search to find the device type 'family_code' on the next call
// to search(*newAddr) if it is present.
//
void OneWire::target_search(uint8_t family_code)
{
   // set the search state to find SearchFamily type devices
   ROM_NO[0] = family_code;
   for (uint8_t i = 1; i < 8; i++)
      ROM_NO[i] = 0;
```

```
   LastDiscrepancy = 64;
   LastFamilyDiscrepancy = 0;
   LastDeviceFlag = FALSE;
}

//
// Perform a search. If this function returns a '1' then it has
// enumerated the next device and you may retrieve the ROM from the
// OneWire::address variable. If there are no devices, no further
// devices, or something horrible happens in the middle of the
// enumeration then a 0 is returned.  If a new device is found then
// its address is copied to newAddr.  Use OneWire::reset_search() to
// start over.
//
// --- Replaced by the one from the Dallas Semiconductor web site ---
//--------------------------------------------------------------------------
// Perform the 1-Wire Search Algorithm on the 1-Wire bus using the existing
// search state.
// Return TRUE  : device found, ROM number in ROM_NO buffer
//        FALSE : device not found, end of search
//
uint8_t OneWire::search(uint8_t *newAddr)
{
   uint8_t id_bit_number;
   uint8_t last_zero, rom_byte_number, search_result;
   uint8_t id_bit, cmp_id_bit;

   unsigned char rom_byte_mask, search_direction;

   // initialize for search
   id_bit_number = 1;
   last_zero = 0;
   rom_byte_number = 0;
   rom_byte_mask = 1;
   search_result = 0;

   // if the last call was not the last one
   if (!LastDeviceFlag)
   {
      // 1-Wire reset
      if (!reset())
      {
         // reset the search
         LastDiscrepancy = 0;
         LastDeviceFlag = FALSE;
         LastFamilyDiscrepancy = 0;
         return FALSE;
      }

      // issue the search command
      write(0xF0);

      // loop to do the search
      do
```

```
      {
         // read a bit and its complement
         id_bit = read_bit();
         cmp_id_bit = read_bit();

         // check for no devices on 1-wire
         if ((id_bit == 1) && (cmp_id_bit == 1))
            break;
         else
         {
            // all devices coupled have 0 or 1
            if (id_bit != cmp_id_bit)
               search_direction = id_bit;  // bit write value for search
            else
            {
               // if this discrepancy if before the Last Discrepancy
               // on a previous next then pick the same as last time
               if (id_bit_number < LastDiscrepancy)
                  search_direction = ((ROM_NO[rom_byte_number] &
rom_byte_mask) > 0);
               else
                  // if equal to last pick 1, if not then pick 0
                  search_direction = (id_bit_number == LastDiscrepancy);

               // if 0 was picked then record its position in LastZero
               if (search_direction == 0)
               {
                  last_zero = id_bit_number;

                  // check for Last discrepancy in family
                  if (last_zero < 9)
                     LastFamilyDiscrepancy = last_zero;
               }
            }

            // set or clear the bit in the ROM byte rom_byte_number
            // with mask rom_byte_mask
            if (search_direction == 1)
              ROM_NO[rom_byte_number] |= rom_byte_mask;
            else
              ROM_NO[rom_byte_number] &= ~rom_byte_mask;

            // serial number search direction write bit
            write_bit(search_direction);

            // increment the byte counter id_bit_number
            // and shift the mask rom_byte_mask
            id_bit_number++;
            rom_byte_mask <<= 1;

            // if the mask is 0 then go to new SerialNum byte rom_byte_number
and reset mask
            if (rom_byte_mask == 0)
            {
```

```
                rom_byte_number++;
                rom_byte_mask = 1;
            }
         }
      }
      while(rom_byte_number < 8);  // loop until through all ROM bytes 0-7

      // if the search was successful then
      if (!(id_bit_number < 65))
      {
         // search successful so set
LastDiscrepancy,LastDeviceFlag,search_result
         LastDiscrepancy = last_zero;

         // check for last device
         if (LastDiscrepancy == 0)
            LastDeviceFlag = TRUE;

         search_result = TRUE;
      }
   }

   // if no device found then reset counters so next 'search' will be like a
first
   if (!search_result || !ROM_NO[0])
   {
      LastDiscrepancy = 0;
      LastDeviceFlag = FALSE;
      LastFamilyDiscrepancy = 0;
      search_result = FALSE;
   }
   for (int i = 0; i < 8; i++) newAddr[i] = ROM_NO[i];
   return search_result;
  }

#endif

#if ONEWIRE_CRC
// The 1-Wire CRC scheme is described in Maxim Application Note 27:
// "Understanding and Using Cyclic Redundancy Checks with Maxim iButton
Products"
//

#if ONEWIRE_CRC8_TABLE
// This table comes from Dallas sample code where it is freely reusable,
// though Copyright (C) 2000 Dallas Semiconductor Corporation
static const uint8_t PROGMEM dscrc_table[] = {
      0, 94,188,226, 97, 63,221,131,194,156,126, 32,163,253, 31, 65,
    157,195, 33,127,252,162, 64, 30, 95,  1,227,189, 62, 96,130,220,
     35,125,159,193, 66, 28,254,160,225,191, 93,  3,128,222, 60, 98,
    190,224,  2, 92,223,129, 99, 61,124, 34,192,158, 29, 67,161,255,
     70, 24,250,164, 39,121,155,197,132,218, 56,102,229,187, 89,  7,
    219,133,103, 57,186,228,  6, 88, 25, 71,165,251,120, 38,196,154,
    101, 59,217,135,  4, 90,184,230,167,249, 27, 69,198,152,122, 36,
```

```
    248,166, 68, 26,153,199, 37,123, 58,100,134,216, 91,  5,231,185,
    140,210, 48,110,237,179, 81, 15, 78, 16,242,172, 47,113,147,205,
     17, 79,173,243,112, 46,204,146,211,141,111, 49,178,236, 14, 80,
    175,241, 19, 77,206,144,114, 44,109, 51,209,143, 12, 82,176,238,
     50,108,142,208, 83, 13,239,177,240,174, 76, 18,145,207, 45,115,
    202,148,118, 40,171,245, 23, 73,  8, 86,180,234,105, 55,213,139,
     87,  9,235,181, 54,104,138,212,149,203, 41,119,244,170, 72, 22,
    233,183, 85, 11,136,214, 52,106, 43,117,151,201, 74, 20,246,168,
    116, 42,200,150, 21, 75,169,247,182,232, 10, 84,215,137,107, 53};

//
// Compute a Dallas Semiconductor 8 bit CRC. These show up in the ROM
// and the registers.  (note: this might better be done without to
// table, it would probably be smaller and certainly fast enough
// compared to all those delayMicrosecond() calls.  But I got
// confused, so I use this table from the examples.)
//
uint8_t OneWire::crc8(const uint8_t *addr, uint8_t len)
{
      uint8_t crc = 0;

      while (len--) {
            crc = pgm_read_byte(dscrc_table + (crc ^ *addr++));
      }
      return crc;
}
#else
//
// Compute a Dallas Semiconductor 8 bit CRC directly.
// this is much slower, but much smaller, than the lookup table.
//
uint8_t OneWire::crc8(const uint8_t *addr, uint8_t len)
{
      uint8_t crc = 0;

      while (len--) {
            uint8_t inbyte = *addr++;
            for (uint8_t i = 8; i; i--) {
                  uint8_t mix = (crc ^ inbyte) & 0x01;
                  crc >>= 1;
                  if (mix) crc ^= 0x8C;
                  inbyte >>= 1;
            }
      }
      return crc;
}
#endif

#if ONEWIRE_CRC16
bool OneWire::check_crc16(const uint8_t* input, uint16_t len, const uint8_t*
inverted_crc, uint16_t crc)
{
    crc = ~crc16(input, len, crc);
    return (crc & 0xFF) == inverted_crc[0] && (crc >> 8) == inverted_crc[1];
```

```
}

uint16_t OneWire::crc16(const uint8_t* input, uint16_t len, uint16_t crc)
{
    static const uint8_t oddparity[16] =
        { 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0 };

    for (uint16_t i = 0 ; i < len ; i++) {
      // Even though we're just copying a byte from the input,
      // we'll be doing 16-bit computation with it.
      uint16_t cdata = input[i];
      cdata = (cdata ^ crc) & 0xff;
      crc >>= 8;

      if (oddparity[cdata & 0x0F] ^ oddparity[cdata >> 4])
          crc ^= 0xC001;

      cdata <<= 6;
      crc ^= cdata;
      cdata <<= 1;
      crc ^= cdata;
    }
    return crc;
}
#endif

#endif
```

keywords.txt

```
#######################################
# Syntax Coloring Map For OneWire
#######################################

#######################################
# Datatypes (KEYWORD1)
#######################################

OneWire     KEYWORD1

#######################################
# Methods and Functions (KEYWORD2)
#######################################

reset KEYWORD2
write_bit   KEYWORD2
read_bit    KEYWORD2
write KEYWORD2
write_bytes KEYWORD2
read  KEYWORD2
read_bytes  KEYWORD2
select      KEYWORD2
skip  KEYWORD2
depower     KEYWORD2
reset_search      KEYWORD2
search      KEYWORD2
crc8  KEYWORD2
crc16 KEYWORD2
check_crc16 KEYWORD2

#######################################
# Instances (KEYWORD2)
#######################################


#######################################
# Constants (LITERAL1)
#######################################
```

DallasTemperature.h

```
#ifndef DallasTemperature_h
#define DallasTemperature_h

#define DALLASTEMPLIBVERSION "3.7.7" // To be deprecated

// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.

// set to true to include code for new and delete operators
#ifndef REQUIRESNEW
#define REQUIRESNEW false
#endif

// set to true to include code implementing alarm search functions
#ifndef REQUIRESALARMS
#define REQUIRESALARMS true
#endif

#include <inttypes.h>
#include <OneWire.h>

// Model IDs
#define DS18S20MODEL 0x10  // also DS1820
#define DS18B20MODEL 0x28
#define DS1822MODEL  0x22
#define DS1825MODEL  0x3B
#define DS28EA00MODEL 0x42

// OneWire commands
#define STARTCONVO      0x44  // Tells device to take a temperature reading
and put it on the scratchpad
#define COPYSCRATCH     0x48  // Copy EEPROM
#define READSCRATCH     0xBE  // Read EEPROM
#define WRITESCRATCH    0x4E  // Write to EEPROM
#define RECALLSCRATCH   0xB8  // Reload from last known
#define READPOWERSUPPLY 0xB4  // Determine if device needs parasite power
#define ALARMSEARCH     0xEC  // Query bus for devices with an alarm condition

// Scratchpad locations
#define TEMP_LSB        0
#define TEMP_MSB        1
#define HIGH_ALARM_TEMP 2
#define LOW_ALARM_TEMP  3
#define CONFIGURATION   4
#define INTERNAL_BYTE   5
#define COUNT_REMAIN    6
#define COUNT_PER_C     7
#define SCRATCHPAD_CRC  8

// Device resolution
```

```
#define TEMP_9_BIT  0x1F //  9 bit
#define TEMP_10_BIT 0x3F // 10 bit
#define TEMP_11_BIT 0x5F // 11 bit
#define TEMP_12_BIT 0x7F // 12 bit

// Error Codes
#define DEVICE_DISCONNECTED_C -127
#define DEVICE_DISCONNECTED_F -196.6
#define DEVICE_DISCONNECTED_RAW -7040

typedef uint8_t DeviceAddress[8];

class DallasTemperature
{
public:

    DallasTemperature();
    DallasTemperature(OneWire*);

    void setOneWire(OneWire*);

    // initialise bus
    void begin(void);

    // returns the number of devices found on the bus
    uint8_t getDeviceCount(void);

    // returns true if address is valid
    bool validAddress(const uint8_t*);

    // returns true if address is of the family of sensors the lib supports.
    bool validFamily(const uint8_t* deviceAddress);

    // finds an address at a given index on the bus
    bool getAddress(uint8_t*, uint8_t);

    // attempt to determine if the device at the given address is connected to
the bus
    bool isConnected(const uint8_t*);

    // attempt to determine if the device at the given address is connected to
the bus
    // also allows for updating the read scratchpad
    bool isConnected(const uint8_t*, uint8_t*);

    // read device's scratchpad
    bool readScratchPad(const uint8_t*, uint8_t*);

    // write device's scratchpad
    void writeScratchPad(const uint8_t*, const uint8_t*);

    // read device's power requirements
    bool readPowerSupply(const uint8_t*);
```

```
    // get global resolution
    uint8_t getResolution();

    // set global resolution to 9, 10, 11, or 12 bits
    void setResolution(uint8_t);

    // returns the device resolution: 9, 10, 11, or 12 bits
    uint8_t getResolution(const uint8_t*);

    // set resolution of a device to 9, 10, 11, or 12 bits
    bool setResolution(const uint8_t*, uint8_t, bool
skipGlobalBitResolutionCalculation = false);

    // sets/gets the waitForConversion flag
    void setWaitForConversion(bool);
    bool getWaitForConversion(void);

    // sets/gets the checkForConversion flag
    void setCheckForConversion(bool);
    bool getCheckForConversion(void);

    // sends command for all devices on the bus to perform a temperature
conversion
    void requestTemperatures(void);

    // sends command for one device to perform a temperature conversion by
address
    bool requestTemperaturesByAddress(const uint8_t*);

    // sends command for one device to perform a temperature conversion by
index
    bool requestTemperaturesByIndex(uint8_t);

    // returns temperature raw value (12 bit integer of 1/128 degrees C)
    int16_t getTemp(const uint8_t*);

    // returns temperature in degrees C
    float getTempC(const uint8_t*);

    // returns temperature in degrees F
    float getTempF(const uint8_t*);

    // Get temperature for device index (slow)
    float getTempCByIndex(uint8_t);

    // Get temperature for device index (slow)
    float getTempFByIndex(uint8_t);

    // returns true if the bus requires parasite power
    bool isParasitePowerMode(void);

     // Is a conversion complete on the wire?
    bool isConversionComplete(void);
```

```
    int16_t millisToWaitForConversion(uint8_t);

#if REQUIRESALARMS

    typedef void AlarmHandler(const uint8_t*);

    // sets the high alarm temperature for a device
    // accepts a char.  valid range is -55C - 125C
    void setHighAlarmTemp(const uint8_t*, char);

    // sets the low alarm temperature for a device
    // accepts a char.  valid range is -55C - 125C
    void setLowAlarmTemp(const uint8_t*, char);

    // returns a signed char with the current high alarm temperature for a
device
    // in the range -55C - 125C
    char getHighAlarmTemp(const uint8_t*);

    // returns a signed char with the current low alarm temperature for a
device
    // in the range -55C - 125C
    char getLowAlarmTemp(const uint8_t*);

    // resets internal variables used for the alarm search
    void resetAlarmSearch(void);

    // search the wire for devices with active alarms
    bool alarmSearch(uint8_t*);

    // returns true if ia specific device has an alarm
    bool hasAlarm(const uint8_t*);

    // returns true if any device is reporting an alarm on the bus
    bool hasAlarm(void);

    // runs the alarm handler for all devices returned by alarmSearch()
    void processAlarms(void);

    // sets the alarm handler
    void setAlarmHandler(const AlarmHandler *);

    // The default alarm handler
    static void defaultAlarmHandler(const uint8_t*);

#endif

    // if no alarm handler is used the two bytes can be used as user data
    // example of such usage is an ID.
    // note if device is not connected it will fail writing the data.
    // note if address cannot be found no error will be reported.
    // in short use carefully
    void setUserData(const uint8_t*, int16_t );
    void setUserDataByIndex(uint8_t, int16_t );
```

```
    int16_t getUserData(const uint8_t* );
    int16_t getUserDataByIndex(uint8_t );

    // convert from Celsius to Fahrenheit
    static float toFahrenheit(float);

    // convert from Fahrenheit to Celsius
    static float toCelsius(float);

    // convert from raw to Celsius
    static float rawToCelsius(int16_t);

    // convert from raw to Fahrenheit
    static float rawToFahrenheit(int16_t);

#if REQUIRESNEW

    // initialize memory area
    void* operator new (unsigned int);

    // delete memory reference
    void operator delete(void*);

#endif

private:
    typedef uint8_t ScratchPad[9];

    // parasite power on or off
    bool parasite;

    // used to determine the delay amount needed to allow for the
    // temperature conversion to take place
    uint8_t bitResolution;

    // used to requestTemperature with or without delay
    bool waitForConversion;

    // used to requestTemperature to dynamically check if a conversion is
complete
    bool checkForConversion;

    // count of devices on the bus
    uint8_t devices;

    // Take a pointer to one wire instance
    OneWire* _wire;

    // reads scratchpad and returns the raw temperature
    int16_t calculateTemperature(const uint8_t*, uint8_t*);

    void    blockTillConversionComplete(uint8_t);

#if REQUIRESALARMS
```

```
    // required for alarmSearch
    uint8_t alarmSearchAddress[8];
    char alarmSearchJunction;
    uint8_t alarmSearchExhausted;

    // the alarm handler function pointer
    AlarmHandler *_AlarmHandler;

#endif

};
#endif
```

DallasTemperature.cpp

```
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.

#include "DallasTemperature.h"


#if ARDUINO >= 100
#include "Arduino.h"
#else
extern "C" {
#include "WConstants.h"
}
#endif

DallasTemperature::DallasTemperature() {}
DallasTemperature::DallasTemperature(OneWire* _oneWire)

#if REQUIRESALARMS
: _AlarmHandler(&defaultAlarmHandler)
#endif
{
    setOneWire(_oneWire);
}

bool DallasTemperature::validFamily(const uint8_t* deviceAddress){
    switch (deviceAddress[0]){
        case DS18S20MODEL:
        case DS18B20MODEL:
        case DS1822MODEL:
        case DS1825MODEL:
        case DS28EA00MODEL:
            return true;
        default:
            return false;
    }
}

void DallasTemperature::setOneWire(OneWire* _oneWire){

    _wire = _oneWire;
    devices = 0;
    parasite = false;
    bitResolution = 9;
    waitForConversion = true;
    checkForConversion = true;

}

// initialise the bus
void DallasTemperature::begin(void){
```

```
    DeviceAddress deviceAddress;

    _wire->reset_search();
    devices = 0; // Reset the number of devices when we enumerate wire devices

    while (_wire->search(deviceAddress)){

        if (validAddress(deviceAddress)){

            if (!parasite && readPowerSupply(deviceAddress)) parasite = true;

            bitResolution = max(bitResolution, getResolution(deviceAddress));

            devices++;
        }
    }

}

// returns the number of devices found on the bus
uint8_t DallasTemperature::getDeviceCount(void){
    return devices;
}

// returns true if address is valid
bool DallasTemperature::validAddress(const uint8_t* deviceAddress){
    return (_wire->crc8(deviceAddress, 7) == deviceAddress[7]);
}

// finds an address at a given index on the bus
// returns true if the device was found
bool DallasTemperature::getAddress(uint8_t* deviceAddress, uint8_t index){

    uint8_t depth = 0;

    _wire->reset_search();

    while (depth <= index && _wire->search(deviceAddress)) {
        if (depth == index && validAddress(deviceAddress)) return true;
        depth++;
    }

    return false;

}

// attempt to determine if the device at the given address is connected to the
bus
bool DallasTemperature::isConnected(const uint8_t* deviceAddress){

    ScratchPad scratchPad;
    return isConnected(deviceAddress, scratchPad);
```

```
}

// attempt to determine if the device at the given address is connected to the
bus
// also allows for updating the read scratchpad
bool DallasTemperature::isConnected(const uint8_t* deviceAddress, uint8_t*
scratchPad)
{
    bool b = readScratchPad(deviceAddress, scratchPad);
    return b && (_wire->crc8(scratchPad, 8) == scratchPad[SCRATCHPAD_CRC]);
}

bool DallasTemperature::readScratchPad(const uint8_t* deviceAddress, uint8_t*
scratchPad){

    // send the reset command and fail fast
    int b = _wire->reset();
    if (b == 0) return false;

    _wire->select(deviceAddress);
    _wire->write(READSCRATCH);

    // Read all registers in a simple loop
    // byte 0: temperature LSB
    // byte 1: temperature MSB
    // byte 2: high alarm temp
    // byte 3: low alarm temp
    // byte 4: DS18S20: store for crc
    //         DS18B20 & DS1822: configuration register
    // byte 5: internal use & crc
    // byte 6: DS18S20: COUNT_REMAIN
    //         DS18B20 & DS1822: store for crc
    // byte 7: DS18S20: COUNT_PER_C
    //         DS18B20 & DS1822: store for crc
    // byte 8: SCRATCHPAD_CRC
    for(uint8_t i = 0; i < 9; i++){
        scratchPad[i] = _wire->read();
    }

    b = _wire->reset();
    return (b == 1);
}


void DallasTemperature::writeScratchPad(const uint8_t* deviceAddress, const
uint8_t* scratchPad){

    _wire->reset();
    _wire->select(deviceAddress);
    _wire->write(WRITESCRATCH);
    _wire->write(scratchPad[HIGH_ALARM_TEMP]); // high alarm temp
    _wire->write(scratchPad[LOW_ALARM_TEMP]); // low alarm temp

    // DS1820 and DS18S20 have no configuration register
```

```
    if (deviceAddress[0] != DS18S20MODEL) _wire->write(scratchPad
[CONFIGURATION]);

    _wire->reset();

    // save the newly written values to eeprom
    _wire->select(deviceAddress);
    _wire->write(COPYSCRATCH, parasite);
    delay(20);  // <--- added 20ms delay to allow 10ms long EEPROM write
operation (as specified by datasheet)

    if (parasite) delay(10); // 10ms delay
    _wire->reset();

}

bool DallasTemperature::readPowerSupply(const uint8_t* deviceAddress){

    bool ret = false;
    _wire->reset();
    _wire->select(deviceAddress);
    _wire->write(READPOWERSUPPLY);
    if (_wire->read_bit() == 0) ret = true;
    _wire->reset();
    return ret;

}


// set resolution of all devices to 9, 10, 11, or 12 bits
// if new resolution is out of range, it is constrained.
void DallasTemperature::setResolution(uint8_t newResolution){

    bitResolution = constrain(newResolution, 9, 12);
    DeviceAddress deviceAddress;
    for (int i=0; i<devices; i++)
    {
        getAddress(deviceAddress, i);
        setResolution(deviceAddress, bitResolution, true);
    }

}

// set resolution of a device to 9, 10, 11, or 12 bits
// if new resolution is out of range, 9 bits is used.
bool DallasTemperature::setResolution(const uint8_t* deviceAddress, uint8_t
newResolution, bool skipGlobalBitResolutionCalculation){

      // ensure same behavior as setResolution(uint8_t newResolution)
      newResolution = constrain(newResolution, 9, 12);

    // return when stored value == new value
    if(getResolution(deviceAddress) == newResolution) return true;
```

```
    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)){

        // DS1820 and DS18S20 have no resolution configuration register
        if (deviceAddress[0] != DS18S20MODEL){

            switch (newResolution){
            case 12:
                scratchPad[CONFIGURATION] = TEMP_12_BIT;
                break;
            case 11:
                scratchPad[CONFIGURATION] = TEMP_11_BIT;
                break;
            case 10:
                scratchPad[CONFIGURATION] = TEMP_10_BIT;
                break;
            case 9:
            default:
                scratchPad[CONFIGURATION] = TEMP_9_BIT;
                break;
            }
            writeScratchPad(deviceAddress, scratchPad);

            // without calculation we can always set it to max
                  bitResolution = max(bitResolution, newResolution);

                  if(!skipGlobalBitResolutionCalculation && (bitResolution > 
newResolution)){
                        bitResolution = newResolution;
                        DeviceAddress deviceAddr;
                        for (int i=0; i<devices; i++)
                        {
                              getAddress(deviceAddr, i);
                              bitResolution = max(bitResolution, getResolution
(deviceAddr));
                        }
                  }
        }
        return true;  // new value set
    }

    return false;

}

// returns the global resolution
uint8_t DallasTemperature::getResolution(){
    return bitResolution;
}

// returns the current resolution of the device, 9-12
// returns 0 if device not found
uint8_t DallasTemperature::getResolution(const uint8_t* deviceAddress){
```

```
    // DS1820 and DS18S20 have no resolution configuration register
    if (deviceAddress[0] == DS18S20MODEL) return 12;

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad))
    {
        switch (scratchPad[CONFIGURATION])
        {
        case TEMP_12_BIT:
            return 12;

        case TEMP_11_BIT:
            return 11;

        case TEMP_10_BIT:
            return 10;

        case TEMP_9_BIT:
            return 9;
        }
    }
    return 0;

}


// sets the value of the waitForConversion flag
// TRUE : function requestTemperature() etc returns when conversion is ready
// FALSE: function requestTemperature() etc returns immediately (USE WITH
CARE!!)
//        (1) programmer has to check if the needed delay has passed
//        (2) but the application can do meaningful things in that time
void DallasTemperature::setWaitForConversion(bool flag){
    waitForConversion = flag;
}

// gets the value of the waitForConversion flag
bool DallasTemperature::getWaitForConversion(){
    return waitForConversion;
}


// sets the value of the checkForConversion flag
// TRUE : function requestTemperature() etc will 'listen' to an IC to
determine whether a conversion is complete
// FALSE: function requestTemperature() etc will wait a set time (worst case
scenario) for a conversion to complete
void DallasTemperature::setCheckForConversion(bool flag){
    checkForConversion = flag;
}

// gets the value of the waitForConversion flag
bool DallasTemperature::getCheckForConversion(){
    return checkForConversion;
```

```
}

bool DallasTemperature::isConversionComplete()
{
   uint8_t b = _wire->read_bit();
   return (b == 1);
}

// sends command for all devices on the bus to perform a temperature
conversion
void DallasTemperature::requestTemperatures(){

    _wire->reset();
    _wire->skip();
    _wire->write(STARTCONVO, parasite);

    // ASYNC mode?
    if (!waitForConversion) return;
    blockTillConversionComplete(bitResolution);

}

// sends command for one device to perform a temperature by address
// returns FALSE if device is disconnected
// returns TRUE  otherwise
bool DallasTemperature::requestTemperaturesByAddress(const uint8_t*
deviceAddress){

    uint8_t bitResolution = getResolution(deviceAddress);
    if (bitResolution == 0){
     return false; //Device disconnected
    }

    _wire->reset();
    _wire->select(deviceAddress);
    _wire->write(STARTCONVO, parasite);

    // ASYNC mode?
    if (!waitForConversion) return true;

    blockTillConversionComplete(bitResolution);

    return true;

}


// Continue to check if the IC has responded with a temperature
void DallasTemperature::blockTillConversionComplete(uint8_t bitResolution){

    int delms = millisToWaitForConversion(bitResolution);
    if (checkForConversion && !parasite){
        unsigned long now = millis();
        while(!isConversionComplete() && (millis() - delms < now));
```

```
    } else {
        delay(delms);
    }

}

// returns number of milliseconds to wait till conversion is complete (based
on IC datasheet)
int16_t DallasTemperature::millisToWaitForConversion(uint8_t bitResolution){

    switch (bitResolution){
    case 9:
        return 94;
    case 10:
        return 188;
    case 11:
        return 375;
    default:
        return 750;
    }

}


// sends command for one device to perform a temp conversion by index
bool DallasTemperature::requestTemperaturesByIndex(uint8_t deviceIndex){

    DeviceAddress deviceAddress;
    getAddress(deviceAddress, deviceIndex);

    return requestTemperaturesByAddress(deviceAddress);

}

// Fetch temperature for device index
float DallasTemperature::getTempCByIndex(uint8_t deviceIndex){

    DeviceAddress deviceAddress;
    if (!getAddress(deviceAddress, deviceIndex)){
        return DEVICE_DISCONNECTED_C;
    }

    return getTempC((uint8_t*)deviceAddress);

}

// Fetch temperature for device index
float DallasTemperature::getTempFByIndex(uint8_t deviceIndex){

    DeviceAddress deviceAddress;

    if (!getAddress(deviceAddress, deviceIndex)){
        return DEVICE_DISCONNECTED_F;
    }
```

```
    return getTempF((uint8_t*)deviceAddress);

}

// reads scratchpad and returns fixed-point temperature, scaling factor 2^-7
int16_t DallasTemperature::calculateTemperature(const uint8_t* deviceAddress,
uint8_t* scratchPad){

    int16_t fpTemperature =
    (((int16_t) scratchPad[TEMP_MSB]) << 11) |
    (((int16_t) scratchPad[TEMP_LSB]) << 3);

    /*
    DS1820 and DS18S20 have a 9-bit temperature register.

    Resolutions greater than 9-bit can be calculated using the data from
    the temperature, and COUNT REMAIN and COUNT PER °C registers in the
    scratchpad.  The resolution of the calculation depends on the model.

    While the COUNT PER °C register is hard-wired to 16 (10h) in a
    DS18S20, it changes with temperature in DS1820.

    After reading the scratchpad, the TEMP_READ value is obtained by
    truncating the 0.5°C bit (bit 0) from the temperature data. The
    extended resolution temperature can then be calculated using the
    following equation:

                                    COUNT_PER_C - COUNT_REMAIN
    TEMPERATURE = TEMP_READ - 0.25 + --------------------------
                                            COUNT_PER_C

    Hagai Shatz simplified this to integer arithmetic for a 12 bits
    value for a DS18S20, and James Cameron added legacy DS1820 support.

    See - http://myarduinotoy.blogspot.co.uk/2013/02/12bit-result-from-
ds18s20.html
    */

    if (deviceAddress[0] == DS18S20MODEL){
        fpTemperature = ((fpTemperature & 0xfff0) << 3) - 16 +
            (
                ((scratchPad[COUNT_PER_C] - scratchPad[COUNT_REMAIN]) << 7) /
                  scratchPad[COUNT_PER_C]
            );
    }

    return fpTemperature;
}


// returns temperature in 1/128 degrees C or DEVICE_DISCONNECTED_RAW if the
// device's scratch pad cannot be read successfully.
// the numeric value of DEVICE_DISCONNECTED_RAW is defined in
```

```
// DallasTemperature.h. It is a large negative number outside the
// operating range of the device
int16_t DallasTemperature::getTemp(const uint8_t* deviceAddress){

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)) return calculateTemperature
(deviceAddress, scratchPad);
    return DEVICE_DISCONNECTED_RAW;

}

// returns temperature in degrees C or DEVICE_DISCONNECTED_C if the
// device's scratch pad cannot be read successfully.
// the numeric value of DEVICE_DISCONNECTED_C is defined in
// DallasTemperature.h. It is a large negative number outside the
// operating range of the device
float DallasTemperature::getTempC(const uint8_t* deviceAddress){
    return rawToCelsius(getTemp(deviceAddress));
}

// returns temperature in degrees F or DEVICE_DISCONNECTED_F if the
// device's scratch pad cannot be read successfully.
// the numeric value of DEVICE_DISCONNECTED_F is defined in
// DallasTemperature.h. It is a large negative number outside the
// operating range of the device
float DallasTemperature::getTempF(const uint8_t* deviceAddress){
    return rawToFahrenheit(getTemp(deviceAddress));
}

// returns true if the bus requires parasite power
bool DallasTemperature::isParasitePowerMode(void){
    return parasite;
}


// IF alarm is not used one can store a 16 bit int of userdata in the alarm
// registers. E.g. an ID of the sensor.
// See github issue #29

// note if device is not connected it will fail writing the data.
void DallasTemperature::setUserData(const uint8_t* deviceAddress, int16_t
data)
{
    // return when stored value == new value
    if(getUserData(deviceAddress) == data) return;

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad))
    {
        scratchPad[HIGH_ALARM_TEMP] = data >> 8;
        scratchPad[LOW_ALARM_TEMP] = data & 255;
        writeScratchPad(deviceAddress, scratchPad);
    }
}
```

```
int16_t DallasTemperature::getUserData(const uint8_t* deviceAddress)
{
    int16_t data = 0;
    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad))
    {
        data = scratchPad[HIGH_ALARM_TEMP] << 8;
        data += scratchPad[LOW_ALARM_TEMP];
    }
    return data;
}

// note If address cannot be found no error will be reported.
int16_t DallasTemperature::getUserDataByIndex(uint8_t deviceIndex)
{
    DeviceAddress deviceAddress;
    getAddress(deviceAddress, deviceIndex);
    return getUserData((uint8_t*) deviceAddress);
}

void DallasTemperature::setUserDataByIndex(uint8_t deviceIndex, int16_t data)
{
    DeviceAddress deviceAddress;
    getAddress(deviceAddress, deviceIndex);
    setUserData((uint8_t*) deviceAddress, data);
}


// Convert float Celsius to Fahrenheit
float DallasTemperature::toFahrenheit(float celsius){
    return (celsius * 1.8) + 32;
}

// Convert float Fahrenheit to Celsius
float DallasTemperature::toCelsius(float fahrenheit){
    return (fahrenheit - 32) * 0.555555556;
}

// convert from raw to Celsius
float DallasTemperature::rawToCelsius(int16_t raw){

    if (raw <= DEVICE_DISCONNECTED_RAW)
    return DEVICE_DISCONNECTED_C;
    // C = RAW/128
    return (float)raw * 0.0078125;

}

// convert from raw to Fahrenheit
float DallasTemperature::rawToFahrenheit(int16_t raw){

    if (raw <= DEVICE_DISCONNECTED_RAW)
    return DEVICE_DISCONNECTED_F;
```

```
    // C = RAW/128
    // F = (C*1.8)+32 = (RAW/128*1.8)+32 = (RAW*0.0140625)+32
    return ((float)raw * 0.0140625) + 32;

}

#if REQUIRESALARMS

/*

ALARMS:

TH and TL Register Format

BIT 7 BIT 6 BIT 5 BIT 4 BIT 3 BIT 2 BIT 1 BIT 0
S    2^6   2^5   2^4   2^3   2^2   2^1   2^0

Only bits 11 through 4 of the temperature register are used
in the TH and TL comparison since TH and TL are 8-bit
registers. If the measured temperature is lower than or equal
to TL or higher than or equal to TH, an alarm condition exists
and an alarm flag is set inside the DS18B20. This flag is
updated after every temperature measurement; therefore, if the
alarm condition goes away, the flag will be turned off after
the next temperature conversion.

*/

// sets the high alarm temperature for a device in degrees Celsius
// accepts a float, but the alarm resolution will ignore anything
// after a decimal point.  valid range is -55C - 125C
void DallasTemperature::setHighAlarmTemp(const uint8_t* deviceAddress, char
celsius){

    // return when stored value == new value
    if(getHighAlarmTemp(deviceAddress) == celsius) return;

    // make sure the alarm temperature is within the device's range
    if (celsius > 125) celsius = 125;
    else if (celsius < -55) celsius = -55;

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)){
        scratchPad[HIGH_ALARM_TEMP] = (uint8_t)celsius;
        writeScratchPad(deviceAddress, scratchPad);
    }

}

// sets the low alarm temperature for a device in degrees Celsius
// accepts a float, but the alarm resolution will ignore anything
// after a decimal point.  valid range is -55C - 125C
void DallasTemperature::setLowAlarmTemp(const uint8_t* deviceAddress, char
celsius){
```

```
    // return when stored value == new value
    if(getLowAlarmTemp(deviceAddress) == celsius) return;

    // make sure the alarm temperature is within the device's range
    if (celsius > 125) celsius = 125;
    else if (celsius < -55) celsius = -55;

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)){
        scratchPad[LOW_ALARM_TEMP] = (uint8_t)celsius;
        writeScratchPad(deviceAddress, scratchPad);
    }

}

// returns a char with the current high alarm temperature or
// DEVICE_DISCONNECTED for an address
char DallasTemperature::getHighAlarmTemp(const uint8_t* deviceAddress){

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)) return (char)scratchPad
[HIGH_ALARM_TEMP];
    return DEVICE_DISCONNECTED_C;

}

// returns a char with the current low alarm temperature or
// DEVICE_DISCONNECTED for an address
char DallasTemperature::getLowAlarmTemp(const uint8_t* deviceAddress){

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)) return (char)scratchPad
[LOW_ALARM_TEMP];
    return DEVICE_DISCONNECTED_C;

}

// resets internal variables used for the alarm search
void DallasTemperature::resetAlarmSearch(){

    alarmSearchJunction = -1;
    alarmSearchExhausted = 0;
    for(uint8_t i = 0; i < 7; i++){
        alarmSearchAddress[i] = 0;
    }

}

// This is a modified version of the OneWire::search method.
//
// Also added the OneWire search fix documented here:
// http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1238032295
//
```

```
// Perform an alarm search. If this function returns a '1' then it has
// enumerated the next device and you may retrieve the ROM from the
// OneWire::address variable. If there are no devices, no further
// devices, or something horrible happens in the middle of the
// enumeration then a 0 is returned.  If a new device is found then
// its address is copied to newAddr.  Use
// DallasTemperature::resetAlarmSearch() to start over.
bool DallasTemperature::alarmSearch(uint8_t* newAddr){

    uint8_t i;
    char lastJunction = -1;
    uint8_t done = 1;

    if (alarmSearchExhausted) return false;
    if (!_wire->reset()) return false;

    // send the alarm search command
    _wire->write(0xEC, 0);

    for(i = 0; i < 64; i++){

        uint8_t a = _wire->read_bit( );
        uint8_t nota = _wire->read_bit( );
        uint8_t ibyte = i / 8;
        uint8_t ibit = 1 << (i & 7);

        // I don't think this should happen, this means nothing responded, but
maybe if
        // something vanishes during the search it will come up.
        if (a && nota) return false;

        if (!a && !nota){
            if (i == alarmSearchJunction){
                // this is our time to decide differently, we went zero last
time, go one.
                a = 1;
                alarmSearchJunction = lastJunction;
            }else if (i < alarmSearchJunction){

                // take whatever we took last time, look in address
                if (alarmSearchAddress[ibyte] & ibit){
                    a = 1;
                }else{
                    // Only 0s count as pending junctions, we've already
exhausted the 0 side of 1s
                    a = 0;
                    done = 0;
                    lastJunction = i;
                }
            }else{
                // we are blazing new tree, take the 0
                a = 0;
                alarmSearchJunction = i;
                done = 0;
```

```
            }
            // OneWire search fix
            // See: http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1238032295
        }

        if (a) alarmSearchAddress[ibyte] |= ibit;
        else alarmSearchAddress[ibyte] &= ~ibit;

        _wire->write_bit(a);
    }

    if (done) alarmSearchExhausted = 1;
    for (i = 0; i < 8; i++) newAddr[i] = alarmSearchAddress[i];
    return true;

}

// returns true if device address might have an alarm condition
// (only an alarm search can verify this)
bool DallasTemperature::hasAlarm(const uint8_t* deviceAddress){

    ScratchPad scratchPad;
    if (isConnected(deviceAddress, scratchPad)){

        char temp = calculateTemperature(deviceAddress, scratchPad) >> 7;

        // check low alarm
        if (temp <= (char)scratchPad[LOW_ALARM_TEMP]) return true;

        // check high alarm
        if (temp >= (char)scratchPad[HIGH_ALARM_TEMP]) return true;
    }

    // no alarm
    return false;

}

// returns true if any device is reporting an alarm condition on the bus
bool DallasTemperature::hasAlarm(void){

    DeviceAddress deviceAddress;
    resetAlarmSearch();
    return alarmSearch(deviceAddress);

}

// runs the alarm handler for all devices returned by alarmSearch()
void DallasTemperature::processAlarms(void){

    resetAlarmSearch();
    DeviceAddress alarmAddr;

    while (alarmSearch(alarmAddr)){
```

```
        if (validAddress(alarmAddr)){
            _AlarmHandler(alarmAddr);
        }

    }
}

// sets the alarm handler
void DallasTemperature::setAlarmHandler(AlarmHandler *handler){
    _AlarmHandler = handler;
}

// The default alarm handler
void DallasTemperature::defaultAlarmHandler(const uint8_t* deviceAddress){}

#endif

#if REQUIRESNEW

// MnetCS - Allocates memory for DallasTemperature. Allows us to instance a
new object
void* DallasTemperature::operator new(unsigned int size){ // Implicit NSS obj
size

    void * p; // void pointer
    p = malloc(size); // Allocate memory
    memset((DallasTemperature*)p,0,size); // Initialise memory

    //!!! CANT EXPLICITLY CALL CONSTRUCTOR - workaround by using an init()
methodR - workaround by using an init() method
    return (DallasTemperature*) p; // Cast blank region to NSS pointer
}

// MnetCS 2009 -  Free the memory used by this instance
void DallasTemperature::operator delete(void* p){

    DallasTemperature* pNss =  (DallasTemperature*) p; // Cast to NSS pointer
    pNss->~DallasTemperature(); // Destruct the object

    free(p); // Free the memory
}

#endif
```

keywords.txt

```
#######################################
# Syntax Coloring Map For DallasTemperature
#######################################

#######################################
# Datatypes (KEYWORD1)
#######################################
DallasTemperature	KEYWORD1
OneWire			KEYWORD1
AlarmHandler		KEYWORD1
DeviceAddress		KEYWORD1

#######################################
# Methods and Functions (KEYWORD2)
#######################################

setResolution		KEYWORD2
getResolution		KEYWORD2
getTempC		KEYWORD2
toFahrenheit		KEYWORD2
getTempF		KEYWORD2
getTempCByIndex	KEYWORD2
getTempFByIndex		KEYWORD2
setWaitForConversion	KEYWORD2
getWaitForConversion	KEYWORD2
requestTemperatures		KEYWORD2
requestTemperaturesByAddress	KEYWORD2
requestTemperaturesByIndex	KEYWORD2
isParasitePowerMode		KEYWORD2
begin			KEYWORD2
getDeviceCount		KEYWORD2
getAddress		KEYWORD2
validAddress		KEYWORD2
isConnected		KEYWORD2
readScratchPad		KEYWORD2
writeScratchPad		KEYWORD2
readPowerSupply		KEYWORD2
setHighAlarmTemp	KEYWORD2
setLowAlarmTemp		KEYWORD2
getHighAlarmTemp	KEYWORD2
getLowAlarmTemp		KEYWORD2
resetAlarmSearch	KEYWORD2
alarmSearch		KEYWORD2
hasAlarm		KEYWORD2
toCelsius		KEYWORD2
processAlarmss		KEYWORD2
setAlarmHandlers	KEYWORD2
defaultAlarmHandler		KEYWORD2
calculateTemperature	KEYWORD2

#######################################
# Constants (LITERAL1)
```

#####################################

boards.txt

```
##############################################################

atmega168bb.name=ATmega168 on a breadboard (8 MHz internal clock)
atmega168bb.upload.protocol=stk500
atmega168bb.upload.maximum_size=14336
atmega168bb.upload.speed=19200
atmega168bb.bootloader.low_fuses=0xE2
atmega168bb.bootloader.high_fuses=0xDA
atmega168bb.bootloader.extended_fuses=0x00
atmega168bb.bootloader.path=arduino:atmega
atmega168bb.bootloader.file=ATmegaBOOT_168_pro_8MHz.hex
atmega168bb.bootloader.unlock_bits=0x3F
atmega168bb.bootloader.lock_bits=0x0F
atmega168bb.build.mcu=atmega168
atmega168bb.build.f_cpu=8000000L
atmega168bb.build.core=arduino:arduino
atmega168bb.build.variant=arduino:standard
```