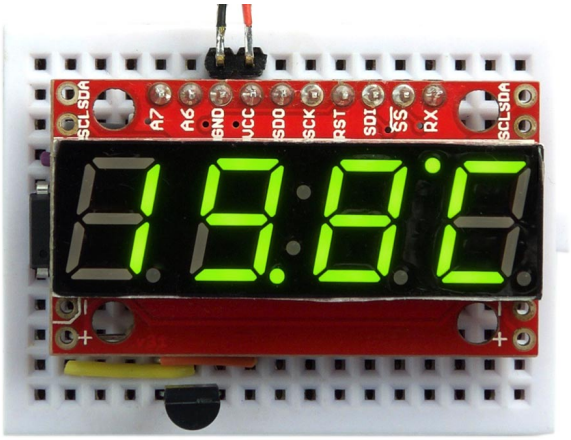
Arduino and AVR projects

Simple 1-Wire Interface

15th March 2017

This article describes a simple 1-Wire interface for the ATtiny85, or other AVR processors. To test it I built a circuit driving a serial 4-digit seven-segment display, displaying temperature using the DS18B20 1-Wire temperature sensor:



A digital thermometer based on the DS18B20 and a simple 1-Wire interface for the ATtiny85.

I also plan to use it in a couple of project that I'll describe in future articles.

Introduction

1-Wire is an ingenious interface that allows a master device to communicate with one or more slave devices over a single wire (plus ground). It's especially useful on processors with a limited number of I/O pins, such as the ATtiny85, as it only uses up one pin.

The 1-Wire interface was originally developed by Dallas Semiconductor, which was acquired by Maxim Integrated in 2001, and Maxim now make a few interesting devices that communicate by 1-Wire:

- The DS18B20 and MAX31820 temperature sensors.
- The DS2417 crystal-controlled real-time clock.
- A series of serial EEPROMs, including the DS28EC20 20Kbit serial EEPROM.

Another feature of 1-Wire is that all devices are programmed with a unique 48-bit serial number in ROM, to allow you to identify multiple devices connected to the same 1-Wire bus.

Implementing the 1-Wire protocol

The 1-Wire interface requires each device to connect to the bus using an open-collector or open-drain output. This allows several devices to be connected to the same bus, and pull the bus low simultaneously without a clash.

You can't explicitly program an ATtiny85 I/O pin as an open-drain output, but fortunately you can achieve the same effect by programming the pin low:

```
PORTB = PORTB & ~(1<<OneWirePin);
```

The two states are then achieved by switching it between input and output. I've provided routines that do this called **PinLow** and **PinRelease**:

```
inline void PinLow () {
  DDRB = DDRB | 1<<OneWirePin;
```

Search

Recent posts

▼ 2017
Four-Channel Thermometer
Flexible GPS Parser
Tiny Face Watch
Driving Four RGB LEDs from an ATtiny85
Big Text for Little Display
ATtiny85 Graphics Display
Tiny Time 2 Watch
10 or 12-bit DAC from the ATtiny85
Simple 1-Wire Interface
Audio Pitch Shifter
Tiny Lisp Computer 2 PCB
GameBone Simple Electronic Game
► 2016
► 2015
► 2014

Topics

► Games
► Sound & Music
► Clocks
► GPS
► Tools
► Tutorials

By processor

► ATtiny85
► ATtiny84
► ATtiny841
► ATtiny2313
► ATtiny861
► ATmega328
► ATmega1284

About me

About me
Contact me

Follow @technoblogy

Donate

Donate

Feeds

RSS feed

```
}

inline void PinRelease () {
  DDRB = DDRB & ~(1<<OneWirePin);
}
```

A third function **PinRead** reads the state of the 1-wire bus:

```
inline uint8_t PinRead () {
  return PINB>>OneWirePin & 1;
}
```

These are all defined as **inline** functions, which makes them effectively like macros without needing to use macro syntax.

Finally, timing is performed using Timer/Counter0. This is set up with a 500kHz clock in **OneWireSetup()**:

```
uint8_t OneWireSetup () {
  TCCR1 = 0<<CTC1 | 0<<PWM1A | 5<<CS10;  // CTC mode, 500kHz clock
  GTCCR = 0<<PWM1B;
 }
```

The routine **DelayMicros()** then allows us to specify a delay of between 0 and 511µsecs, by first setting the counter to zero, and then doing a compare match with OCR1A, waiting until the compare match flag OCF1A is set:

```
void DelayMicros (int micro) {
  TCNT1 = 0; TIFR = 1<<OCF1A;
  OCR1A = (micro>>1) - 1;
  while ((TIFR & 1<<OCF1A) == 0);
}
```

These routines are combined into **LowRelease()**, which pulls the bus low for **low** µsec, and then releases it for **high** µsec:

```
void LowRelease (int low, int high) {
  PinLow();
  DelayMicros(low);
  PinRelease();
  DelayMicros(high);
}
```

The 1-Wire protocol

The 1-Wire protocol has four main components:

- **Reset** resets all the slave devices ready to receive a command: pull bus low for 480µs, release bus for 70µs, read bus, then delay 410µs. If the read returns 0 there are one or more devices present.
- **Write 1** writes a '1' bit to the slaves: pull bus low for 6µs, release bus for 64µs.
- **Write 0** writes a '0' bit to the slaves: pull bus low for 60µs, release bus for 10µs.
- **Read bit** reads a bit from the slaves: pull bus low for 6µs, release bus for 9µs, read bus, delay 55µs.

These are implemented using the above routines. First **OneWireReset()**:

```
uint8_t OneWireReset () {
  uint8_t data = 1;
  LowRelease(480, 70);
  data = PinRead();
  DelayMicros(410);
  return data;                        // 0 = device present
}
```

Then **OneWireWrite()**, which writes one byte of data:

```
void OneWireWrite (uint8_t data) {
  int del;
  for (int i = 0; i<8; i++) {
    if ((data & 1) == 1) del = 6; else del = 60;
    LowRelease(del, 70 - del);
    data = data >> 1;
  }
}
```

Finally, **OneWireRead()** reads a byte:

```
uint8_t OneWireRead () {
  uint8_t data = 0;
  for (int i = 0; i<8; i++) {
    LowRelease(6, 9);
    data = data | PinRead()<<i;
    DelayMicros(55);
  }
  return data;
}
```

To check how tolerant these routines are of variations in clock speed I varied the ATtiny85 internal clock using OSCCAL and checked whether the routines still worked. I was able to vary the clock by about 40% either side of 8MHz before the 1-Wire communication failed. The factory calibration of the internal oscillator is quoted as ±10%, so this should be well within this range.

Reading a block of data

Most 1-Wire devices allow you to read a block of data, such as the serial number or temperature data, and this is provided with a CRC to guard against data errors.

I used an array of 9 bytes to store the data. A **union** allows you to read them as bytes or words:

```
// Buffer to read data or ROM code
static union {
  uint8_t DataBytes[9];
  unsigned int DataWords[4];
};
```

The **OneWireReadBytes()** routine is used to read a specified number of bytes into the buffer:

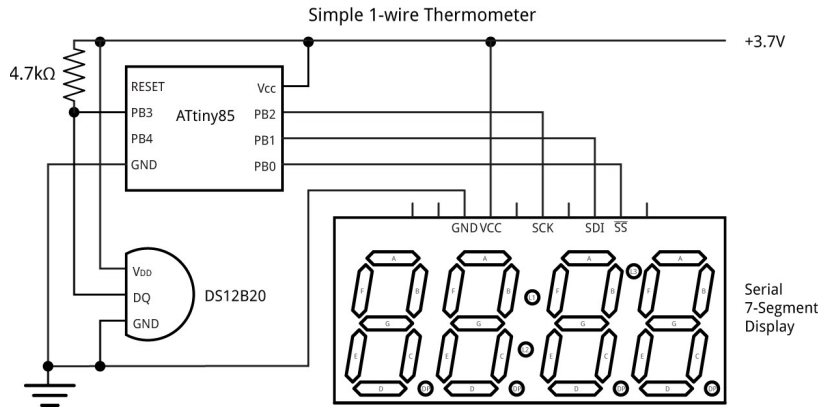
```
void OneWireReadBytes (int bytes) {
  for (int i=0; i<bytes; i++) {
    DataBytes[i] = OneWireRead();
  }
}
```

The **OneWireCRC()** routine calculates a CRC over a specified number of bytes in the buffer:

```
uint8_t OneWireCRC (int bytes) {
  uint8_t crc = 0;
  for (int j=0; j<bytes; j++) {
    crc = crc ^ DataBytes[j];
    for (int i=0; i<8; i++) crc = crc>>1 ^ ((crc & 1) ? 0x8c : 0);
  }
  return crc;
}
```

Simple thermometer

As an example of using the 1-Wire routines with a single 1-Wire device on the bus I built a simple thermometer based on a DS18B20, using a Sparkfun serial seven-segment display. Here's the circuit:



Circuit of the digital thermometer based on a DS18B20 and an ATtiny85.

I used a DS12B20 [1] temperature sensor in a TO-92 transistor package, which has a supply range of 3.0V to 5.5V; the MAX31820 [2] is equivalent and cheaper, but has a maximum supply voltage of 3.7V.

Since there is only one device on the bus you can ignore the serial number, and just send a Skip ROM command which sends subsequent commands to any device.

The **DisplayTemp()** routine shows temperatures below zero with a minus sign, to the nearest degree; for example -10°C. Positive temperatures up to 99°C are displayed with one decimal place; for example, 23.4°C. Temperatures of 100 degrees or more are displayed to the nearest degree; for example, 123°C:

```
void DisplayTemp (int t) {
  SendByte(Cursor_Control);
  SendByte(0);
  int temp = t>>4;
  int points = 0x20;                 // Display degree symbol
  if (temp>=100) SendByte(temp/100); // Hundreds
  if (temp<0) SendByte('-');
  SendByte((temp/10) % 10);          // Tens
  SendByte(temp % 10);               // Units
  if (temp>=0 && temp<100) {
    points = 0x22;                   // Display decimal point
    SendByte(((t & 0x0F)*10)>>4);    // Tenths
  }
  SendByte(Decimal_Control);
  SendByte(points);
  SendByte('C');
}
```

Here's the routine to read the 9-byte scratchpad memory from a single DS18B20 or MAX31820 temperature sensor, check the CRC, and display the temperature in the first two bytes:

```
void Temperature () {
  cli();                           // No interrupts
  if (OneWireReset() != 0) {
    sei();
    DisplayError("Err");
  } else {
    OneWireWrite(SkipROM);
    OneWireWrite(ConvertT);
    while (OneWireRead() != 0xFF);
    OneWireReset();
    OneWireWrite(SkipROM);
    OneWireWrite(ReadScratchpad);
    OneWireReadBytes(9);
    sei();                         // Interrupts
    if (OneWireCRC(9) == 0) {
      DisplayTemp(DataWords[0]);
    } else DisplayError("Crc");
```

```
  }
  delay(1000);
}
```

The **cli()** and **sei()** calls disable interrupts around the 1-Wire routines; they are optional, but without them a temperature reading occasionally gives a CRC error due to the Arduino **millis()** interrupts.

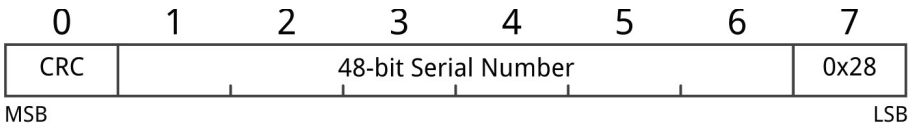
Finding the serial number of each device

All the devices are programmed with a unique 48-bit serial number in ROM, to allow you to identify multiple devices connected to the same 1-Wire bus.

If there are several devices on the bus you need to send a Match ROM command followed by the serial number of the device you want to address. So in this case you need to know the serial numbers.

To find the serial numbers of all the devices on the 1-Wire bus you can use a special routine called Search ROM, which tests the 48 bits a bit at a time for conflicts, and then builds a tree of the device serial numbers. This is a complex procedure, and I haven't implemented it in these routines; if you need it, use one of the 1-wire libraries.

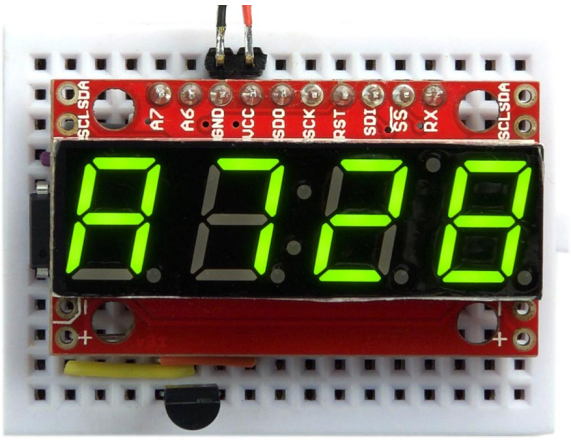
Fortunately, there's a much simpler way of finding the serial numbers: you can connect the devices to the 1-Wire bus one at a time, and use the Read ROM command to read the serial number directly. The serial number is read, together with the product code 0x28 and the CRC, as eight successive bytes LSB first:



Here's a loop that reads the ROM data of a single device into the array **DataBytes[]** and displays it on the 7-segment display, two bytes at a time, highest byte first:

```
void SerialNumber () {
  ClearDisplay();
  cli();
  if (OneWireReset() != 0) {
    sei();
    DisplayError("Err");
  } else {
    OneWireWrite(ReadROM);
    OneWireReadBytes(8);
    sei();
    if (OneWireCRC(8) == 0) {
      for (int i=3; i>=0; i--) {
        Display(DataWords[i]);
        delay(1000);
      }
    } else DisplayError("Crc");
  }
  delay(1000);
}
```

It displays the ROM data as a series of four displays, each consisting of four hexadecimal digits. For example, this is the last display for my DS12B20:



Displaying the ROM data from a DS12B20 temperature sensor using the 1-Wire interface.

The last byte, 0x28, is the product code for the DS12B20.

If there is no device on the bus it displays "Err", and if the CRC doesn't match it displays "Crc".

Compiling the program

I compiled the program using Spence Konde's ATTiny Core, which supercedes the various earlier ATtiny cores [3]. Select the **ATtiny x5 series** option under the **ATtiny Universal** heading on the **Boards** menu. Then choose **Timer 1 Clock: CPU**, **B.O.D. Disabled**, **ATtiny85**, 8 **MHz (internal)** from the subsequent menus. Choose **Burn Bootloader** to set the fuses appropriately. Then upload the program using ISP (in-system programming); I used Sparkfun's Tiny AVR Programmer Board; see ATtiny-Based Beginner's Kit.

Here's the whole 1-Wire program: 1-Wire Program.

---

1. ^ One Wire Digital Temperature Sensor - DS18B20 on Sparkfun
2. ^ One-Wire Ambient Temperature Sensor - MAX31820 on Sparkfun
3. ^ ATTinyCore on GitHub.
