The playground is a publicly-editable wiki about Arduino.

- Deutsch
- English
- Español
- Français
- Italiano
- Português
- Български
- Русский

Manuals and Curriculum

Board Setup and Configuration

Development Tools

Interfacing With Hardware

- Output
- Input
- Storage
- Communication
- General

Interfacing with Software

Code Library and Tutorials

Suggestions & Bugs

Electronics Technique

Sources for Electronic Parts

Related Hardware and Initiatives

Arduino People/Groups & Sites

Exhibition

Project Ideas

Languages

PARTICIPATE

- create an account
- suggestions
- formatting suggestions
- all recent changes
- PmWiki
- WikiSandBox training
- Basic Editing
- Cookbook (addons)
- Documentation Index

edit SideBar

Dallas Semiconductor's 1-Wire Protocol

(This page may need splitting into a general guide to 1-Wire, and a separate page with the "using the DS1820" material. I would suggest that the general guide inherit the old links.)

Dallas Semiconductor (now Maxim) produces a family of devices that are controlled through a proprietary **1-wire** protocol. There are no fees for programmers using the Dallas 1-Wire (trademark) drivers.

On a 1-Wire network, which Dallas has dubbed a "MicroLan" (trademark), a single "master" device communicates with one or more 1-Wire "slave" devices over a single data line, which can also be used to provide power to the slave devices. (Devices drawing power from the 1-wire bus are said to be operating in *parasitic power* mode.)

Tom Boyd's guide to 1-Wire may tell you more than you want to know... but it may also answer questions and inspire interest.

The 1-wire temperature sensors have become particularly popular, because they're inexpensive and easy to use, providing calibrated digital temperature readings directly. They are more tolerant of long wires between sensor and Arduino. The sample code below demonstrates how to interface with a 1-wire device using Jim Studt's **OneWire** Arduino library, with the DS18S20 digital thermometer as an example. Many 1-Wire chips can operate in both parasitic and normal power modes.

MicroLans can be accessed directly by an Arduino, using the mature Arduino 1-Wire library. Alternatively, they can be accessed through an interface which costs a little money, but reduces the workload inside the Arduino. The interface cost $8 in kit form at 2/2010. There is a guide to using it from an independent source.

Powering a DS18x20

The chip can be powered two ways. One (the "parasitic" option) means that only two wires need go to the chip. The other may, in some cases, give more reliable operation (parasitic often works well), as an extra wire carrying the power for the chip is involved. For getting started, especially if your chip is within 20 feet of your Arduino, the parasitic option is probably fine. The code below works for either option, anyway.

Parasite power mode
When operating in parasite power mode, only two wires are required: one data wire, and ground. At the master, **a 4.7k pull-up resistor must be connected to the 1-wire bus.** When the line is in a "high" state, the device pulls current to charge an internal capacitor.

This current is usually very small, but may go as high as 1.5 mA when doing a temperature conversion or writing EEPROM. When a slave device is performing one these operations, the bus master must keep the bus pulled high to provide power until the operation completes; a delay of 750ms is required for a DS18S20 temperature conversion. The master can't do anything during this time, like issuing commands to other devices, or polling for the slave's operation to be completed. To support this, the OneWire library makes it possible to have the bus held high after the data is written.

Normal (external supply) mode
With an external supply, three wires are required: the bus wire, ground, and power. **The 4.7k pull-up resistor is still required** on the bus wire. As the bus is free for data transfer, the microcontroller can continually poll the state of a device doing a conversion. This way, a conversion request can finish as soon as the device reports being done, as opposed to having to wait 750ms in "parasite" power mode.

Note on resistors: For larger networks, try something smaller. The ATmega328/168 datasheet indicates starting at 1k6 and a number of users have found smaller to work better on larger networks.

Addressing a DS18x20

Each 1-Wire device contains a unique 64-bit 'rom' code, consisting of an 8-bit family code, a 48-bit serial number, and an 8-bit CRC. The CRC is used to verify the integrity of the data. The sample code, below, ensures that the device being addressed is a DS18S20 by checking for its family code, 0x10. (To use the sample code with the newer DS18B20 sensor, you'd check for a family code of 0x28, instead, and for the DS1822 you'd check for 0x22.)

Before sending a command to a slave device, the master must first select that device using its rom code. You can also address a command to *all* slave devices by issuing a 'skip rom' command, instead. This is only safe for commands that don't elicit a response from the slave devices - data collision will occur if data is requested from more than one slave.

Please see the DS18S20 or DS18B20 datasheets for more detailed information.

Library

Derived from Jim Studts OneWire library (below) you can now control several Dallas ICs through a very simple library by Miles Burton. Dallas Temperature Control Library

For more generic purposes, Jim Studts library can be used to interface with various Dallas ICs.

Jim Studt created a OneWire library that makes it easy to work with 1-Wire devices. The original version only worked with arduino-007 and required a large (256-byte) lookup table to perform CRC calculations. This was later updated to work with arduino-0008 and later releases. The most recent version eliminates the CRC lookup table; it has been tested under arduino-0010.

The OneWire library has a bug causing an infinite loop when using the search function. Fixes to this can be found in this Arduino thread, see posts #3, #17, #24, #27 for a variety of fixes.

Version 2.0 merges Robin James's improved search function and includes Paul Stoffregen's improved I/O routines (fixes occasional communication errors), and also has several small optimizations.

Example code

```
#include <OneWire.h>

// DS18S20 Temperature chip i/o
OneWire ds(10);  // on pin 10

void setup(void) {
  // initialize inputs/outputs
  // start serial port
  Serial.begin(9600);
}

void loop(void) {
  byte i;
  byte present = 0;
  byte data[12];
  byte addr[8];

  if ( !ds.search(addr)) {
      Serial.print("No more addresses.\n");
      ds.reset_search();
      return;
  }

  Serial.print("R=");
  for( i = 0; i < 8; i++) {
    Serial.print(addr[i], HEX);
    Serial.print(" ");
  }

  if ( OneWire::crc8( addr, 7) != addr[7]) {
      Serial.print("CRC is not valid!\n");
      return;
  }

  if ( addr[0] != 0x10) {
      Serial.print("Device is not a DS18S20 family device.\n");
      return;
  }

  ds.reset();
  ds.select(addr);
  ds.write(0x44,1);         // start conversion, with parasite power on at the end

  delay(1000);     // maybe 750ms is enough, maybe not
  // we might do a ds.depower() here, but the reset will take care of it.

  present = ds.reset();
  ds.select(addr);
  ds.write(0xBE);         // Read Scratchpad

  Serial.print("P=");
  Serial.print(present,HEX);
  Serial.print(" ");
  for ( i = 0; i < 9; i++) {           // we need 9 bytes
    data[i] = ds.read();
    Serial.print(data[i], HEX);
    Serial.print(" ");
  }
  Serial.print(" CRC=");
  Serial.print( OneWire::crc8( data, 8), HEX);
  Serial.println();
}
```

[Get Code]

Converting HEX to something meaningful (Temperature)

In order to convert the HEX code to a temperature value, first you need to identify if you are using a DS18S20, or DS18B20 series sensor. The code to read the temperature needs to be slightly different for the DS18B20 (and DS1822), because it returns a 12-bit temperature value (0.0625 deg precision), while the DS18S20 and DS1820 return 9-bit values (0.5 deg precision).

First off, you need to define some variables, (put right under loop() above)

```
int HighByte, LowByte, TReading, SignBit, Tc_100, Whole, Fract;
```

Then for a DS18B20 series you will add the following code below the `Serial.println();` above

```
 LowByte = data[0];
  HighByte = data[1];
  TReading = (HighByte << 8) + LowByte;
  SignBit = TReading & 0x8000;  // test most sig bit
  if (SignBit) // negative
  {
    TReading = (TReading ^ 0xffff) + 1; // 2's comp
  }
  Tc_100 = (6 * TReading) + TReading / 4;    // multiply by (100 * 0.0625) or 6.25

  Whole = Tc_100 / 100;  // separate off the whole and fractional portions
  Fract = Tc_100 % 100;


  if (SignBit) // If its negative
  {
     Serial.print("-");
  }
  Serial.print(Whole);
  Serial.print(".");
  if (Fract < 10)
  {
     Serial.print("0");
  }
  Serial.print(Fract);

  Serial.print("\n");
```

This block of code converts the temperature to deg C and prints it to the Serial output.

A Code Snippet for the DS 1820 with 0.5 Degree Resolution

Above example works only for the B-type of the DS1820. Here is a code example that works with the lower resolution DS1820 and with multiple sensors diplaying their values on a LCD. Example is working with Arduino pin 9. Feel free to change that to an appropriate pin for your use. Pin 1 and 3 of the DS1820 has to be put to ground! In the example a 5k resistor is put from pin 2 of DS1820 to Vcc (+5V). See LiquidCrystal documentation for connecting the LCD to the Arduino.

```
#include <OneWire.h>
#include <LiquidCrystal.h>
// LCD=======================================================
//initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
#define LCD_WIDTH 20
#define LCD_HEIGHT 2

/* DS18S20 Temperature chip i/o */

OneWire  ds(9);  // on pin 9
#define MAX_DS1820_SENSORS 2
byte addr[MAX_DS1820_SENSORS][8];
void setup(void)
{
  lcd.begin(LCD_WIDTH, LCD_HEIGHT,1);
  lcd.setCursor(0,0);
  lcd.print("DS1820 Test");
  if (!ds.search(addr[0]))
  {
    lcd.setCursor(0,0);
    lcd.print("No more addresses.");
    ds.reset_search();
    delay(250);
    return;
  }
  if ( !ds.search(addr[1]))
  {
    lcd.setCursor(0,0);
    lcd.print("No more addresses.");
    ds.reset_search();
    delay(250);
    return;
  }
}
int HighByte, LowByte, TReading, SignBit, Tc_100, Whole, Fract;
char buf[20];

void loop(void)
{
  byte i, sensor;
  byte present = 0;
  byte data[12];

  for (sensor=0;sensor<MAX_DS1820_SENSORS;sensor++)
  {
    if ( OneWire::crc8( addr[sensor], 7) != addr[sensor][7])
    {
      lcd.setCursor(0,0);
      lcd.print("CRC is not valid");
      return;
    }

    if ( addr[sensor][0] != 0x10)
    {
      lcd.setCursor(0,0);
      lcd.print("Device is not a DS18S20 family device.");
      return;
    }

    ds.reset();
    ds.select(addr[sensor]);
    ds.write(0x44,1);         // start conversion, with parasite power on at the end

    delay(1000);     // maybe 750ms is enough, maybe not
    // we might do a ds.depower() here, but the reset will take care of it.

    present = ds.reset();
    ds.select(addr[sensor]);
    ds.write(0xBE);         // Read Scratchpad

    for ( i = 0; i < 9; i++)
    {           // we need 9 bytes
      data[i] = ds.read();
    }

    LowByte = data[0];
    HighByte = data[1];
    TReading = (HighByte << 8) + LowByte;
    SignBit = TReading & 0x8000;  // test most sig bit
    if (SignBit) // negative
    {
      TReading = (TReading ^ 0xffff) + 1; // 2's comp
    }
    Tc_100 = (TReading*100/2);

    Whole = Tc_100 / 100;  // separate off the whole and fractional portions
    Fract = Tc_100 % 100;

    sprintf(buf, "%d:%c%d.%d\337C     ",sensor,SignBit ? '-' : '+', Whole, Fract < 10 ? 0 : Fract);

    lcd.setCursor(0,sensor%LCD_HEIGHT);
    lcd.print(buf);
  }
}
```

Additional 1-Wire libraries

- DS2450 Quad A/D
- DS2409 1-Wire Switch

Share |