

Arduino Nano Project One: Thermometer

Introduction

This document is designated to be Public Domain

Author: Lewis Balentine, October 2013

This all began as I was searching Amazon for a thermometer that would record the day's highs and low temperature. All the reviews that I read pointed out various problems with the devices from being off by a number of degrees to infantile mortality. As I read those reviews it occurred to me that it would be very advantageous if I could send that temperature to the computer that I have sitting in a closet upstairs. Still that data would have to be accurate to be of any value. Accurate thermometers that can communicate with a computer turn out to be very expensive. Then it occurred to me how hard could it be to build one: a simple microcontroller board, a sensor, some wire and bit of code. An easy and cheap solution well not exactly.

This document goes through all the steps I went through in starting from scratch. I am NOT a programming GURU and I have never professionally written "C" programs. I do have three decades of experience in information technology and some programming background but not in the "C" language. Thus this was a journey into the uncharted wilderness for me. The reason this became an 'expensive' thermometer is because I made a several wrong turnings. I have left those out or explained why they were wrong. This is written from the viewpoint of "assume nothing". Hopefully it may provide a path for others to follow (*that will be somewhat less expensive than the path(s) followed*).

End Product:

What you end up with is an Arduino microcontroller programmed to be a sophisticated thermometer that can report its readings back to a computer via a standard USB connection. These reports may be generated at periodic times from one minute to 24 hours. The device is "user" calibrated and may be programmed to report any combination of raw readings, Celsius readings or Fahrenheit readings. The device may be used in a "self-storage mode" where it is run from an alternate power source (*battery*) and stores from 440 to 7,040 consecutive readings in its internal memory (*estimated real capacity is 3,520*). Both "wear leveling" and "data reduction" are implemented to conserve storage space and extend the life of the internal EEPROM. Over 30 commands have been defined and implemented to allow the connected computer to control the device via a standard terminal program. All software is provided with full source code and fully commented. Source code files may be downloaded from <http://www.keywild.com/arduino>.

Legal Disclaimer:

The author makes no representation or warranty, either expressed or implied, with respect to the data files and/or software, their quality, accuracy, or fitness for any specific application. Therefore the author shall have no liability to any person or any entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by the use of the data files and/or software. This includes, but is not limited to, interruption of service, loss of data, loss of consulting or anticipatory profits or consequential damages from the use of these data files and/or software.

All files unless otherwise noted are the original work product of the author. Unless otherwise noted these files are placed into the Public Domain for the unrestricted use by anyone for any purpose. Placing these files in the public domain shall in no way be construed as an obligation of the author (*or his heirs and/or assigns*) to maintain the web site, web pages, files, data or software. Further it shall in no way limit the author's (*or his heirs and/or assigns*) options to make, produce or use versions of the software, data files, CAD objects or other material posted under the URL in any other commercial or non-commercial venture.

In the event of a legal dispute the court is requested to use a "reasonable person's" interpretation of the "clear intent" of this disclaimer.

The use of these data files and/or software constitutes acceptance of this disclaimer.

Table of contents:

Introduction	2
End Product:.....	2
Legal Disclaimer:	2
Table of contents:	3
Requirements:	10
Access to a working computer:	10
Arduino Compatible Development Board:	10
USB Cable	10
LM34 Temperature Sensor.....	10
Bread Board, 175 Tie points	10
Three Bread Board jumpers, 1-1/2 to 2 inches long.....	10
Thermometer	10
Access to the internet:	10
Optional components.....	11
Small Ceramic Disk Capacitor	11
3.5mm stereo female jack, PCB Mount, 3 contacts	11
3.5mm stereo male plug, solder terminal, 3 contacts	11
3.5mm stereo extension cable	11
22 K Ohm 1/4 watt resistor, solder leads	11
Optional Tool Requirements	11
Wire cutter	11
Small soldering iron.....	11
Small quantity of electrical/electronic solder.....	11
Magnifying Glass or Eye Loop.....	11
Tweezers	11
Microcontroller: Meet the Arduino Nano.....	12
Arduino Development Environment	17
Arduino Programing Environment:.....	17
Setting up the Arduino IDE:	18
Arduino IDE: Compile and Upload.....	22
Enter, Save, Serial Monitor: “Hello Word”	28
Funny Math: Bits, Nibbles and Bytes.....	32
Zeroes and Ones (Decimal, Binary and Hexadecimal)	32
Divide by Zero (yes we can)	39
Special Numbers (true or false?)	49

Memory: FLASH, SRAM, EEPROM	50
SRAM: Hello Word 001/002	52
FLASH: Hello Word 003/004	53
EEPROM: Hello Word 005/006 (Write, Read)	57
EEPROM: EEPROM_Dump, EEPROM_Erase	65
Building a Library: The easy way	69
Functions: Passing Parameters and Return Values	69
Library HexDec: Developing Functions	73
Library HexDec: Overloading	78
Library HexDec: ASCII Table	79
Library HexDec: EEPROMDump	80
Library HexDec: Creating the Library	81
Library HexDec: Testing the Library	83
AVR Internal Temperature Sensor	85
Using the ChipTemp Library	85
Develop Avr Temperature Functions	89
Storing Calibration Constants (EEPROM)	93
Thermometer Program.....	98
Reporting Protocol	98
Thermometer Program, Plan “A”	102
Main File Functions	103
Global Declarations	103
Setup() Function	103
Loop() Function	103
CmdProcessor() Function	104
HelpMe() Function	104
PrintSeperatorLine() Function	104
ReadTwoCharacters ()Function	104
DrainCmdTermiantors() Function	104
DebugPrintCharacters() Function	105
Thermometer Functions File	105
EnableADC() Function.....	105
Read_Calibration_Data() Function	105
Write_Calibration_Data() Function	105
ClearStorage() Function.....	105
EEmodeFlagSet() Function.....	105
EEmodeFlagClear() Function	106

EEmodeFlagTF() Function	106
Check_EEPROM() Function	106
Print_IdString() Function	106
PrintTrueFalse() Function	106
ReportStatus() Function	106
avrRawTemp() Function	106
Convert()Function	106
Report() Function	106
QuickBlink() Function	106
Report2EEPROM() Function	106
DumpStorage() Function	107
PrintOKStr() Function	107
PrintNotRecognized()Function	107
PrintNotImplemented() Function	107
ShutDown() Function.....	107
software_Reset() Function	107
SetRawReadMode() Function	108
SetFahrenheitMode() Function	108
SetCelsiusMode() Function.....	108
SetReportMode() Function.....	108
ToggleDebugMode() Function	108
NewReportTime() Function	108
Report_Reset() Function	108
NewIdString() Function.....	108
NewOffset() Function	108
CelsiusEquals() Function	109
FahrenheitEquals() Function	109
NewCelsius() Function	109
NewFahrenheit() Function.....	109
RestoreFromBackup() Function	109
OverwriteBackup() Function.....	109
TestData1() Function	109
TestData2() Function	109
CalibrationMode() Function	109
void EepromDumpAll() Function	110
Temperature Calibration Theory	110
Temperature Calibration Procedure.....	111

Temperature Calibration Procedure: Observation Point 1	112
Temperature Calibration Procedure: Observation Point 3	112
Temperature Calibration Procedure: Observation Point 2	112
Plan “A”, Evaluation and Summary	113
Thermometer Program, Plan “B”	114
External Temperature Sensor: LM34.....	115
EEPROM Layout.....	118
Global Variables and Constants.....	119
Main Program File Functions.....	121
setup() Function	121
loop()Function	121
cmdProcessor() function	121
HelpMe() function	121
PrintSeperatorLine() function.....	122
ReadTwoCharacters() function.....	122
DrainCmdTermiantors() function	122
DebugPrintCharacters () function.....	122
Thermometer Functions File	123
Read_Calibration_Data() function	123
Write_Calibration_Data() function	123
ClearStorage() function	123
EEmodeFlagSet() function	123
EEmodeFlagClear() function	123
EEmodeFlagTF() function.....	123
Check_EEPROM() function	123
Print_IdString() function.....	123
PrintTrueFalse() function.....	123
ReportStatus() Function	123
AvrTemperature() function	123
ReadRawTempA1() function.....	123
Convert(word RawReading) Function.....	123
nearestquater () function	124
nearesthalf () function.....	124
Report() function.....	124
QuickBlink() function.....	124
Report2EEPROM() function	124
DumpStorage() function.....	124

Response() function	124
PrintOKStr () function	124
PrintNotRecognized() function	124
PrintNotImplemented() function	124
ShutDown() function	124
software_Reset() function	124
SetRawReadMode() function	124
SetCelsiusMode() function	124
SetFahrenheitdMode() function	124
SetReportMode() function	124
ToggleDebugMode() function	124
SetAvrInternalMode() function	124
ToggleRoundMode() function	124
NewReportTime() function	125
Report_Reset() function	125
NewIdString() function	125
PrintDegreeOffsetEffect() function	125
ValueNotAccepted() function	125
NewDegreeOffset() function	125
CalculateDegreeOffset() function	125
FahrenheitEquals() function	125
CelsiusEquals() function	125
NewRefVolt() function	125
RestoreFromBackup() function	125
OverwriteBackup() function	125
TestData1() function	125
TestData2() function	125
CalibrationMode() function	126
EepromDumpAll()function	126
Temperature Sensor Calibration	128
Calibration Theory	128
Calibration Method 1	128
Calibration Method 2	128
Temperature Sensor Extension Cable	128
Plan "B", Evaluation and Summary	130
Thermometer Program, ATMEGA168	131
Arduino Debugging	132

Debugging Methods	132
Common errors to look for:	132
Other Hints:	132
RS232 Serial Monitor	134
FreeBasic Compiler	134
Serial Port Monitor Program	137
PC Alternatives: Microsoft.....	141
PC Alternatives: Non-Microsoft.....	141
Why FreeBasic	142
ArduinoThermometer.exe	143
Strip Semicolon Lines Utility	145
Receiver Modifications:	147
Conclusion	152
Possible Enhancements	152
Temperature Accuracy	152
EEPROM Storage Mode	152
Number of sensors	152
Remote Data Collection.....	152
LCD Display	152
GUI Interface	152
Photo Gallery.....	154
Appendix: Atmel MPU Table	158
Appendix: Arduino Check Speed	160
Appendix: AVR ADC Sensor Registers.....	163
Appendix: ADC Function test	165
Appendix: Disabling Auto Reset.....	170
Appendix: Arduino ElfDump	172
Appendix: Arduino Receiver	177
Appendix: Thermometer.exe.....	180
Main Program Code	180
Global Variables	180
Thermometer Functions.....	182
Ini File for Main Program.....	196
Utility Program	198
Appendix: Thermometer One Program Code (Plan "A").....	205
Thermometer One Main Program File	205
Thermometer One Functions Module.....	210

Appendix: Thermometer One Program Code (Plan "B").....	226
Thermometer One Main Program File	226
Thermometer One Functions Module.....	232
Appendix: Thermometer ATMega168	249
Main Program File (ATMega168).....	249
Thermometer Function ATMega168 File.....	253
Appendix: ASCII Table	268
Appendix: Celsius vs. Fahrenheit Table	273
Appendix: LM34 Data Sheet	277
Appendix: MS takes Bow Shot at console applications	282

Requirements:

No Arduino's were harmed or damaged in the development of this application. That statement actually has more meaning that it may appear. The goal was for there to be no external components, any additional wiring or soldering required for this project. Unfortunately reality stepped in and there is now one external component and 3 wires required.

Access to a working computer:

All the project software is open source and runs under Windows and Linux operating systems. Most of the software will also operate on a MAC computer.

Arduino Compatible Development Board:

These can purchased for as little as \$9 from sources on Amazon.com and Ebay.com. I recommend the Nano but any Arduino with an **ATmega328P** (or **ATmega328**) and a USB port should work. (*The "Pro Mini", "Lilo" and "Lilypad" do NOT have USB ports. The Micro might work but I have had problems with that model*).

USB Cable

Used between the Arduino microcontroller board and the computer. Sometimes this will be provided with the Arduino board.

LM34 Temperature Sensor

This is a Texas Instrument/National Semiconductor series of part for measuring temperature in degrees Fahrenheit. The specific part number recommended is a **LM34DZ**. Alternately any other IC from this line can be used. For those outside the US you may want to consider the LM35 series that is calibrated in Celsius. I paid too much for mine. You should be able to get these for about US\$2.50.

Bread Board, 175 Tie points

This is just to have something so the Nano's pins do not sit directly on the desktop. It also allows the temperature sensor to be connected without any soldering. These can be purchased for about \$5. A 300 or 400 tie point bread board will also work and allow some expansion for somewhat more advanced projects.

Three Bread Board jumpers, 1-1/2 to 2 inches long

You will need three breadboard jumpers between 1-1/2 to 2 inches long. Alternatively you can make your own from insulated 22 to 28 AWG single conductor wire (*Cat 5 or Cat 6 LAN cable is an excellent source*).

Thermometer

This is not an absolute requirement. These are used to calibrate the digital thermometer. A bi-metal dial thermometer, cooking thermometer, wall thermometer or glass tube thermometer may be used. The task of finding an 'accurate' thermometer to use as a reference may be the biggest challenge in this project.

Access to the internet:

Not actually required after the software is downloaded but highly recommended. There are imbedded links throughout the text to references on the internet. These URLs are shown as underlined blue text:

Example: <http://www.keywild.com/arduino>

Optional components

Small Ceramic Disk Capacitor

Atmel recommends placing a capacitor on the Analog reference pin. I used a 22 Pico Farad ceramic disk capacitor but I cannot say that it made any real difference.

3.5mm stereo female jack, PCB Mount, 3 contacts

3.5mm stereo male plug, solder terminal, 3 contacts

3.5mm stereo extension cable

22 K Ohm 1/4 watt resistor, solder leads

These can be used to separate the sensor from the breadboard. Once that is done any 3.5mm stereo cable can be used or the sensor can be plugged directly into the board. A stereo jack with 5 to 11 contacts may be used as an alternative (*the 3 contact jacks are hard to find*). The resistor may be a 1/2 watt or 1/4 watt of any precision and is in fact optional. If it is used then it needs to go inside the 3.5mm stereo plug so the smaller the better.

Optional Tool Requirements

Wire cutter

If you want a really neat breadboard then you will need something to cut the wire and strip insulation from it. Alternatively one may use pre-manufactured breadboard jumpers. At least three of these about 1-1/2 to 2 inches long will be required.

Small soldering iron

Small quantity of electrical/electronic solder

If decide to use an extension cable for the sensor then some very minor soldering will be required. Mounting the sensor in the stereo plug requires soldering. As a starting place, for most small electronics soldering, 1/32 inch (.03) rosin-cored, 60/40 (tin-lead) or 63/37 solder should work fine. Rosin-cored lead-free is fine, too. For a good reference on electronic soldering see the URL:

http://store.curiousinventor.com/guides/how_to_solder/kind_of_solder/

Magnifying Glass or Eye Loop

Tweezers

Some of us with somewhat less than perfect vision may find a magnify glass or eye loop helpful. A set of tweezers are not a bad idea either.

Microcontroller: Meet the Arduino Nano

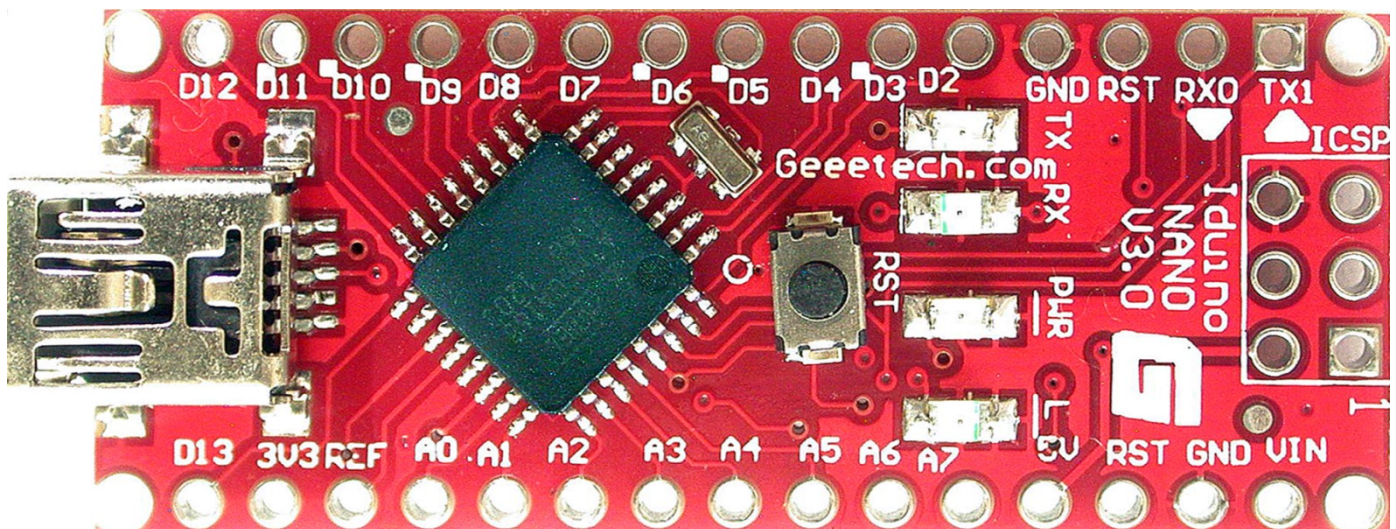
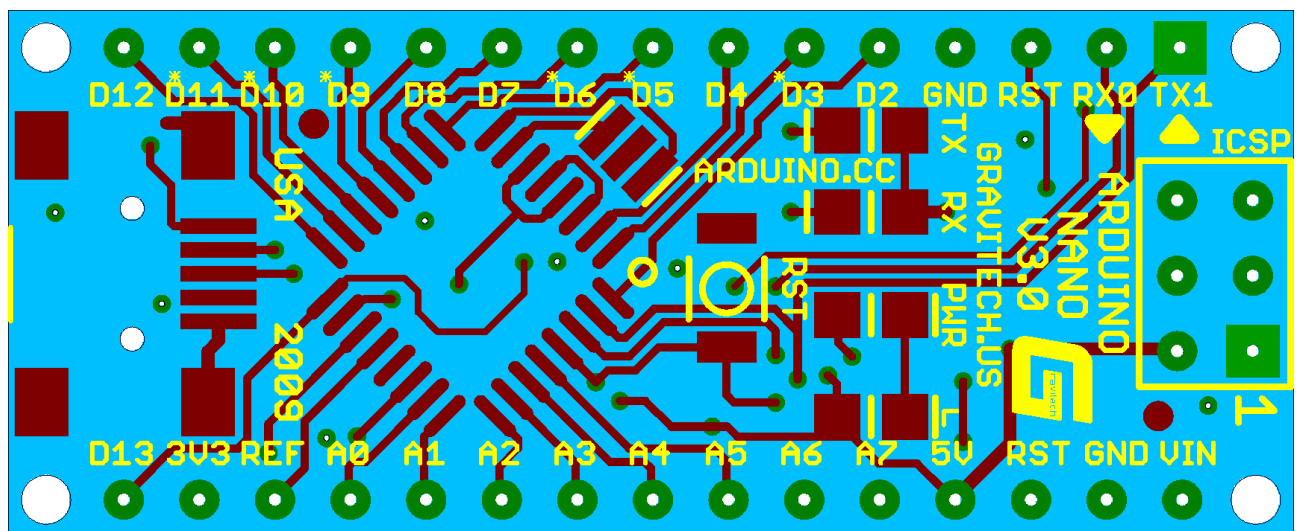
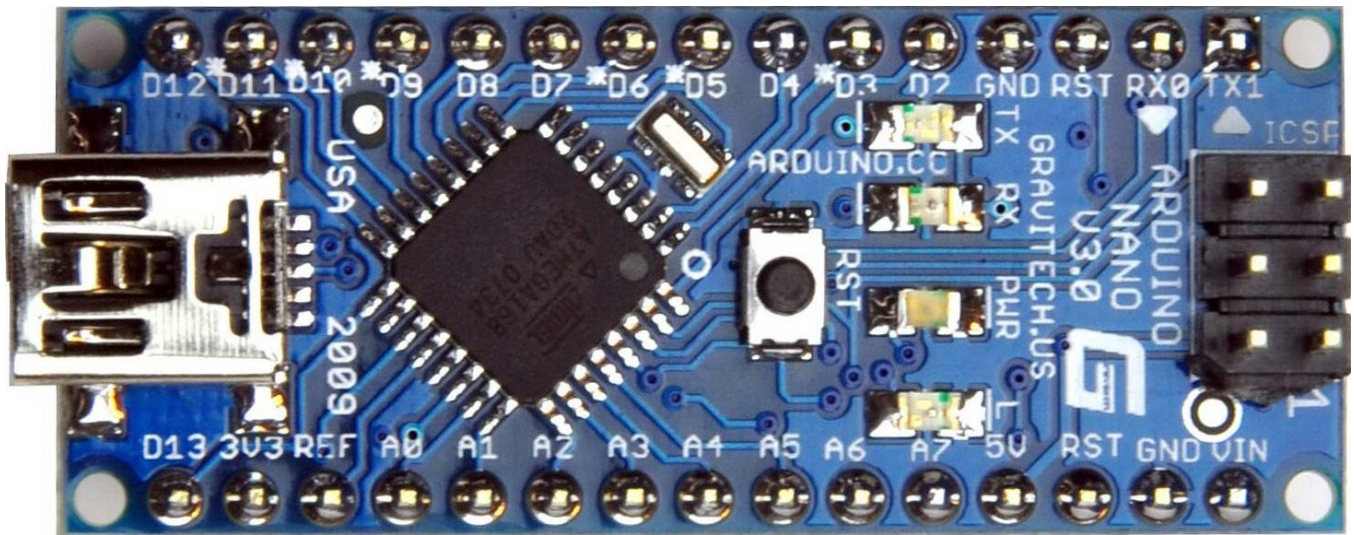
The goal here is to produce a device that reports the current temperature back to a computer over a USB port using the minimum amount of inexpensive hardware. We will in fact only be using a single piece of hardware.

The first question was: “what microcontroller to use?” I had done a little work in the past with PICs and Basic Stamps so I first went looking for a microcontroller board that implemented a form of the basic language. There are several good offerings available. Among these are Parallax Basic Stamp and Basic Micro’s Nano series. I also read about other microcontroller offerings and ran across information about the Arduino line. This is a series of microcontroller boards based on “open design” standards that are not encumbered by proprietary copyrights, licenses (*hardware or software*) and the associated extra cost of these items. An “official” Arduino microcontroller board can be obtained for less than \$20(US). A “clone” Arduino microcontroller board can be obtained for less than \$10(US). One can be “scratch” built by downloading the appropriate files and obtaining the raw hardware though I question that being less expensive or more practical than buying a mass produced board for a “one-of-kind” project.

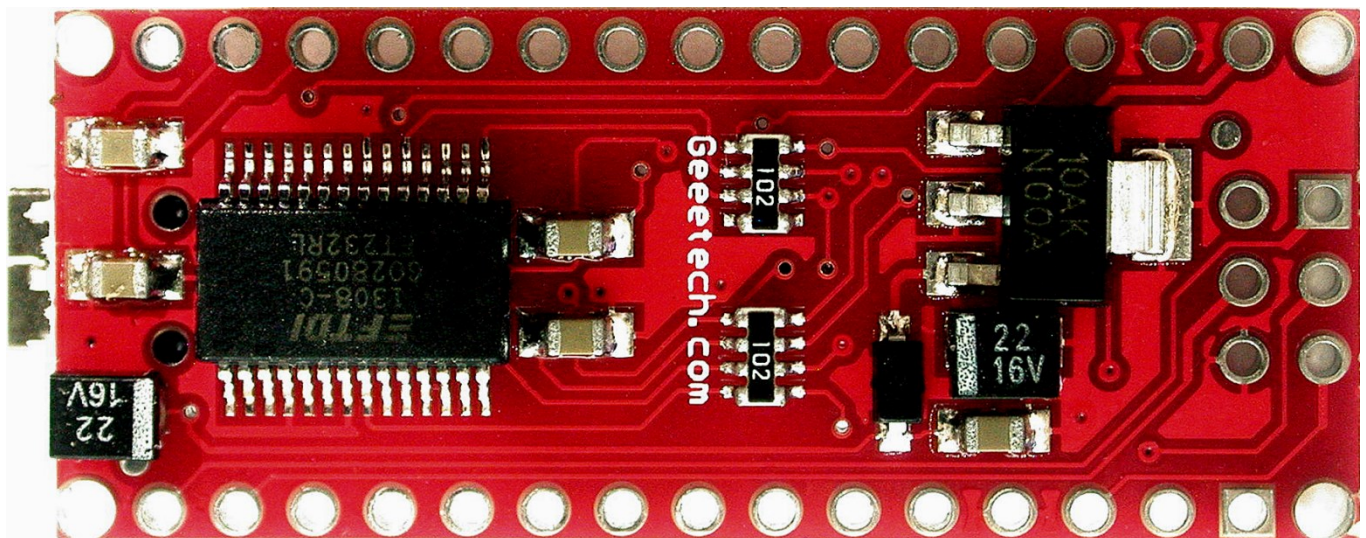
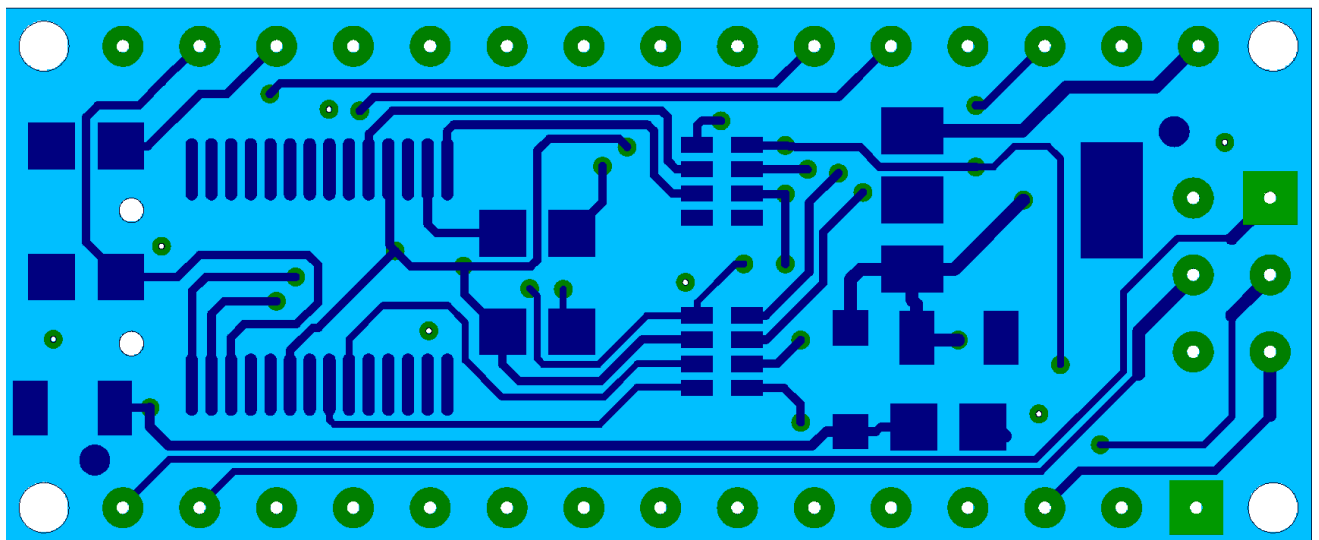
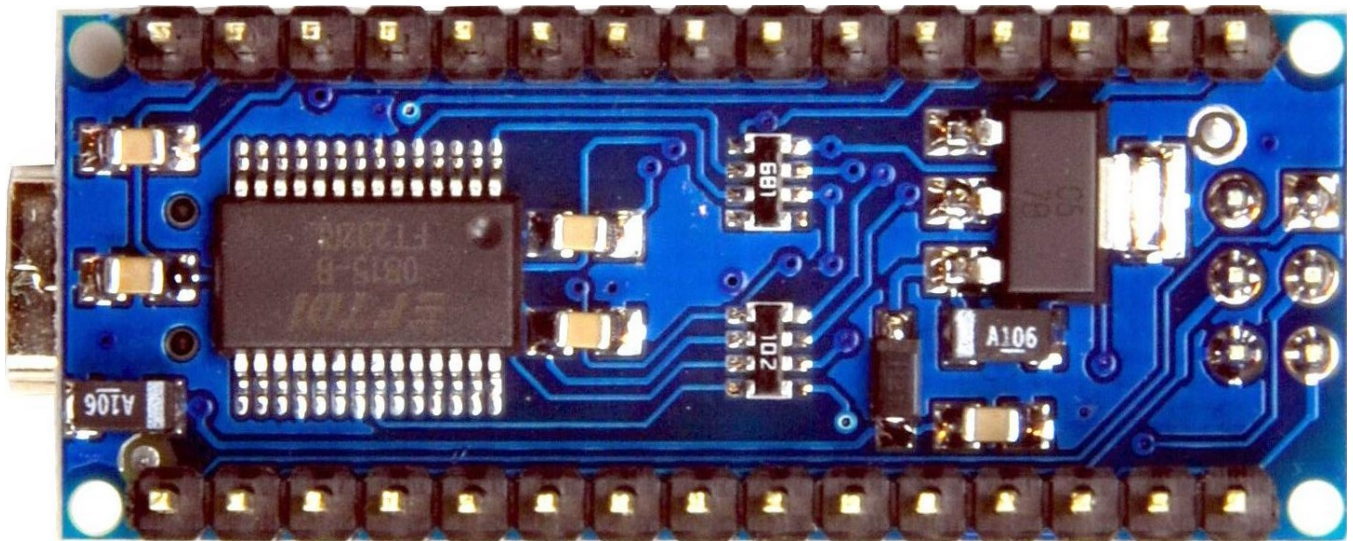
The Arduino line uses a form of the “C” language. I am not a great fan of the “C” language. I find the simplicity of Basic to be much more appealing for non-professional programmers. The complex array of files, rules, structures, pointers and non-intuitive syntax that accompanies “C” and its derivatives may seem friendly to people that work with it on a regular basis but to someone that only uses it occasionally it can be an excruciating experience. However in recent years the popular Visual Basic implementation has been “improved” by Microsoft such that it now conforms to much the same standards as “C”. That “improvement” has also destroyed most of its simplistic advantages. The Arduino team has done an excellent job of isolating the user from some of the complexity in their implementation of the language. One can still delve into the complexities of “C” to use standard tools and make files to produce Arduino programs but most users will not need to go beyond the simple user interface of Arduino IDE (*Integrated Development Environment*). I decided to give up my preference for Basic and use an Arduino based board.

There are 19 “official” Arduino microcontroller boards and derivatives too numerous to count. I chose the “Nano” design (*developed by [Gravitech](#) of Claremont, CA, USA*) as a starting point because:

- 1) It has more or less the same functionality of the larger Arduino Duemilanove
- 2) Small size: slightly smaller than 3/4 x 1-3/4 inches
- 3) 8 Analog Inputs (10 bits = 1024 steps)
- 4) Available with or without header pins
- 5) Uses either the Atmel ATmega168 (v2.3) or ATmega328 (v3.0) AVR microcontroller
- 6) Built in USB port (*USB Mini-B*)
- 7) 5 Volt operation (low power projects can be powered by USB port)
- 8) Availability from numerous suppliers (*many listed on [Amazon.com](#) and [Ebay.com](#)*)
- 9) Cost: \$35(US) from Gravitech down to \$9(US) various clones

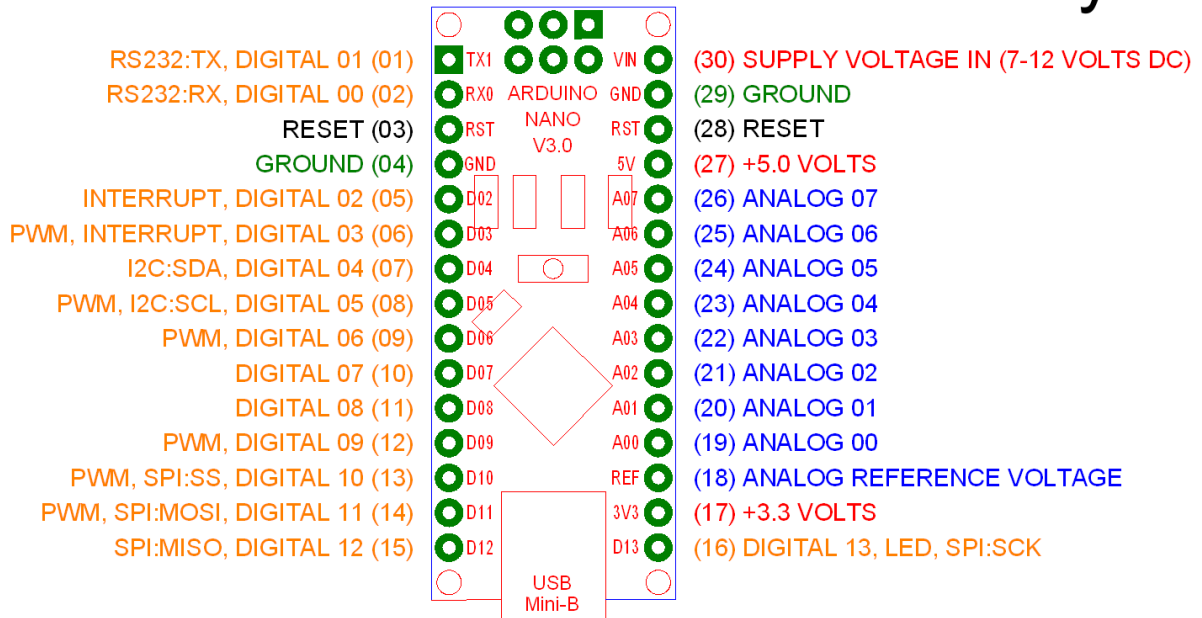


Top Image is a Gravitech Nano board version 3.0 (although it has an ATmega168-20AU).
 Middle image is the board layout showing the top traces
 Bottom Image is a Chinese clone Nano board version 3.0 (although it has an ATmega168-20AU).



Top Image is a Gravitech Nano board version 3.0
 Middle image is the board layout showing the bottom traces
 Bottom Image is a Chinese clone Nano board version 3.0

ARDUINO NANO Version 3.0 Pin Layout



You will probably note a lot of references to the term “AVR”(*Appendix: Atmel MPU Table*). The microcontroller used on Arduino boards is an Atmel AVR chip. Atmel is the manufacturer’s name and AVR is the product name for this series of chips. Per Atmel:

“Atmel® AVR® 8- and 32-bit microcontrollers complement Atmel's ARM® microcontrollers and microprocessors to deliver a unique combination of performance, power efficiency and design flexibility. Optimized to speed time to market, they are based on the industry's most code-efficient architecture for C and assembly programming.”

Just to put things in perspective: In 1982 the [Commodore VIC-20](#) was the best-selling computer of the year, with 800,000 machines sold. One million had been sold by the end of the year and at one point, 9000 units a day were being produced. This little device has more memory and computing power than a Commodore Vic 20.



That is a SainSmart Nano clone in the blue ellipse.

Another interesting alternative is the [Teensy 3.0 boards](#) development board. This has a much more powerful 32 bit ARM Cortex-M4 48 MHz CPU with 128K FLASH, 16K RAM, 2K EEPROM, 14 each 13 bit Analog Inputs and 34 Digital inputs. It also has the advantage of an on board clock (*but requires crystal and battery*). All of this is packed onto a board (*1.4 x 0.7 inches*) that is even smaller than the NANO. Also available from SparkFun: <https://www.sparkfun.com/products/11780>

Arduino Development Environment

The primary goal of this project is to report to the attached computer the temperature over the USB/RS232 communications port. The computer shall provide the user interface and final processing of the data. Thus we actually have to create software for both sides.

Arduino Programing Environment:

For the Arduino we will use the Arduino IDE (*Integrated Development Environment*) and its subset of the “C” language. Although this may seem like an obvious choice there are a number of alternatives that could be used such as:

- 1) the avr-g++ environment used to developed, build and run the Arduino IDE.
- 2) [mikroBasic PRO for AVR](#), mikroPascal PRO for AVR or mikroC PRO for AVR
- 3) [ICCV8 for AVR](#)
- 4) Atmel AVR Studio ([good tutorial here: http://hekilliedmywire.wordpress.com/2010/12/04/22/](http://hekilliedmywire.wordpress.com/2010/12/04/22/))
- 5) [codebender](#) (*online code development for Arduino*)
- 6) [MariaMole](#)
- 7) NextEdge android /iPhone Arduino Compiler (*then you have to get it to your Arduino*)
- 8) Microsoft Visual Studio (configured Arduino with plugins)
- 9) BASCOM-AVR
- 10) SmallC for AVR

Most of these use C or use GCC in the background. The Arduino IDE was chosen because it works, it works across platforms (*Windows, Linux, Mac*) and it is free. There is also implementation of the GDB/INSIGHT debugger for the AVR line (*not covered in this document*).

Arduino Web Site: <http://arduino.cc/en/>
Arduino Download: <http://arduino.cc/en/Main/Software>
Arduino Forum: <http://forum.arduino.cc/>

Another interesting alternative is AttoBasic from:

http://www.cappels.org/dproj/AttoBasic_Home/AttoBasic_Home.html

“AttoBasic is an on-chip resident interactive Basic interpreter loosely based on memories of NASCOM Tiny Basic. AttoBasic interprets single lines from a terminal or an entire program stored in memory without the delay of compiling and loading an entire program.”

One disadvantage is that if you use this in a “commercial” environment then you have to get a license.

Setting up the Arduino IDE:

Download the software from the official ARDUNIO web site:

<http://arduino.cc/en/Main/Software>.

Then follow the direction for installing the software according to your operating system.

There are several 'alternative' IDEs for the Arduino. One that includes a complete rework of the 'standard' Arduino package can be found in this Arduino Forum msg: <http://forum.arduino.cc/index.php?topic=118440.0>

There are a few things that you can do to make the Arduino IDE a bit more efficient and user friendly. When you install Arduino IDE it should create a directory in your documents folder. On Windows this will be:

C:\Users\<User Name>\Documents\Arduino (Windows 7)

or

C:\Documents and Settings\<User Name>\My Documents\Arduino (XP)

It will also create a directory where it installs the actual software. On Windows this will be (*hereafter called <app path>*):

C:\Program Files (x86)\Arduino (64 bit Windows)

or

C:\Program Files\Arduino (32 bit Windows)

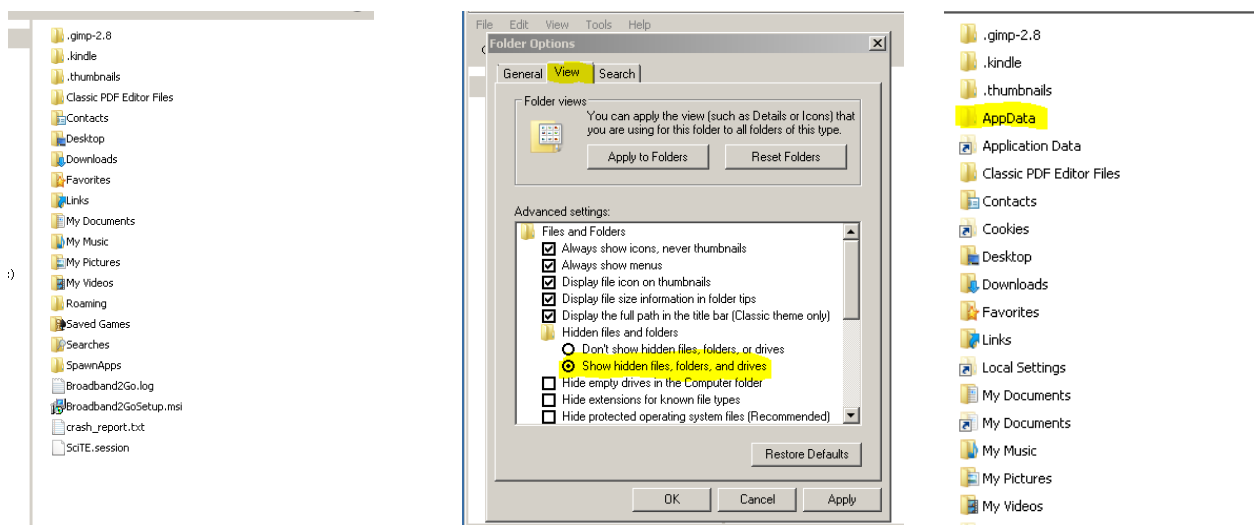
It will also create a directory for its "preferences file". On windows this will be:

<User Home>\AppData\Roaming\Arduino (Windows 7)

or

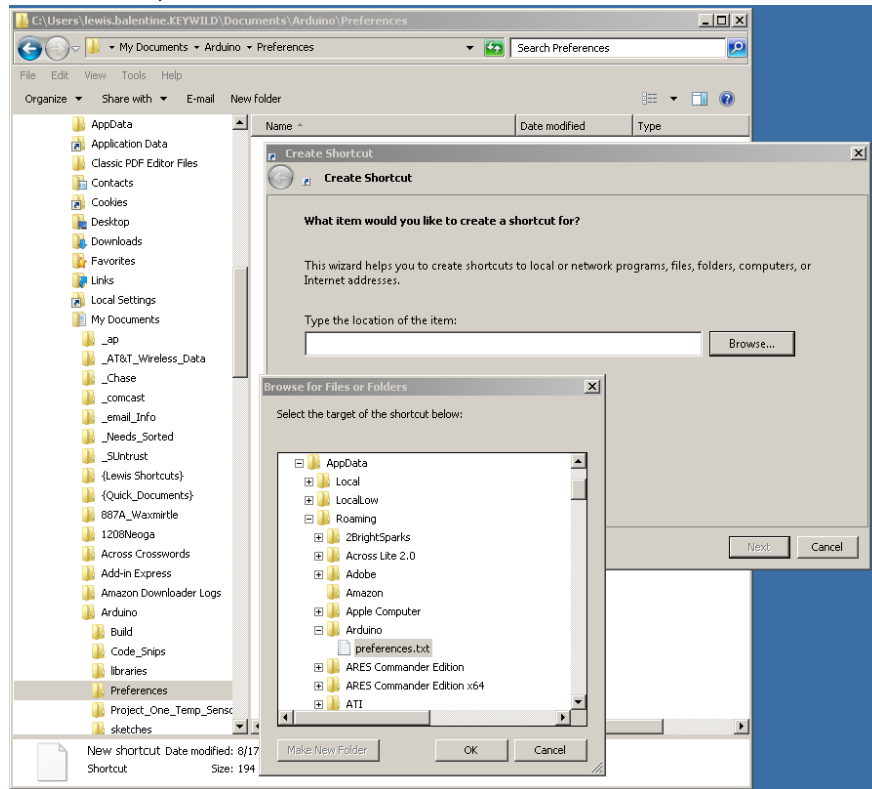
<User Home>\Application Data\Arduino (XP)

This location of this last directory is an unfortunate choice because it is normally hidden from the user. You need to be able to edit the file in this directory. In Windows Explorer Browse to you home directory. It should be something like "**C:\Users\<your name>**" or "**C:\Documents and Settings\<your name>**" (***consistency is not one of Microsoft's strong points***). Then select "**Tools**", "**Folder Options**". Then select the "**View**" tab at the top. Pick the radio button "**Show hidden files, folders, and drives**". Then click on "**OK**" (*you can change it back later ... or not*). You should now be able to see several more folders and files. One of those will be "AppData".



In this directory you will find a folder named "**Roaming**". In that directory will a folder named "**Arduino**" with a single file named "**preferences.txt**". Now that you can see the file let us go back up to our <user home>\My

[Documents\Arduino](#) directory. Create a new directory there named “**preferences**”. In that directory right click you mouse and select “**New**”, “**ShortCut**”. In the dialog box choose the “**Browse**” button and navigate to the file: [<User Home>\AppData\Roaming\Arduino\preferences.txt](#)
Select that file and click on “**OK**”, “**Next**” and “**Finish**”.



Double clicking on this file should open the preferences.txt file in notepad (*or your default system text editor if you have replaced notepad*). You can now go back and change the “Show hidden files ...” if you like.

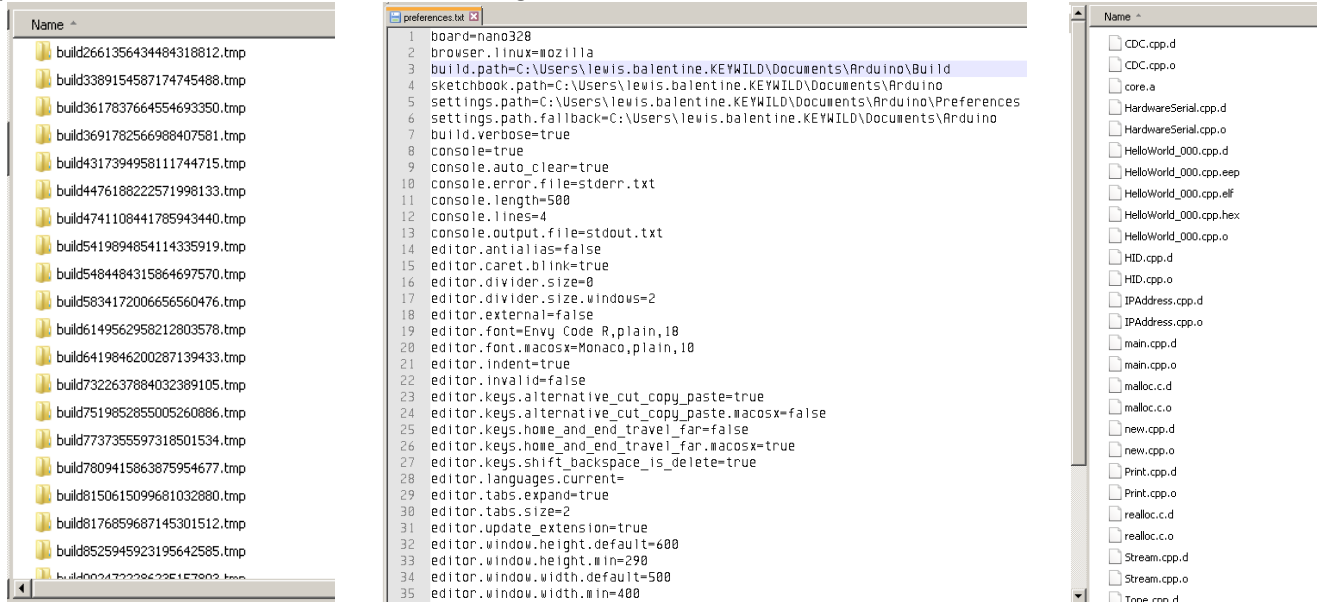
Normally when the Arduino IDE compiles a new program it creates a working directory in your personal temp folder. The path will be something like:

`C:\Users\<user home>\AppData\Local\Temp\build2572074991509959449.tmp`

This is inconvenient for two reasons. First at some point one may want to take a look at the files in “build” directory. Second, and most importantly, the Arduino IDE does a very poor job of cleaning up after itself. You will eventually have dozens of these working directories taking up space and generally making a garbage dump of your machine. We can fix that (*unfortunately it turns out that the IDE creates numerous other temp directories that it does not delete as well*). First create another new folder named [<user home>\My Documents\Arduino\Build](#). Now double click on your new “preferences.txt” short cut. Insert a new line in the file:

`build.path=C:\Users\<your name>\Documents\Arduino\Build`

Be sure that you change <your name> to the actual name your system uses. To answer the more advanced users question: No, the Arduino IDE does not recognize the environmental variable %HOMEPATH% or %HOME%.



You might have noticed the next line in the preferences.txt file is the path to your SketchBook (*where the Arduino IDE stores your projects*). Many (*not all*) of these options are documented in the file:

[<app path>\Arduino\lib\preferences.txt](#)

This file has a number of inaccuracies in it. Note lines 5 and 6 above in preferences file. Those are totally ignored by the Arduino IDE.

There is one option that is helpful and does work. We can change the font used by the editor. The default font is "editor.font=Monospaced,plain,12". First of all there is no font for Windows with the name "monospaced". The IDE appears to be using "Courier New" but in that font there is very little difference between parenthesis and curly braces. The Arduino code uses these a lot and we really need to be able to see the difference easily. Microsoft does not provide any good monospaced fonts. We have to look elsewhere. One font that works well is "Envy Code R". You can download it here:

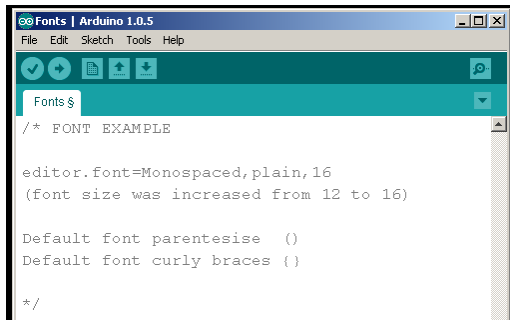
<http://damieng.com/blog/2008/05/26/envy-code-r-preview-7-coding-font-released>

Another font that works well is "Source Code Pro". You can download it here:

<http://sourceforge.net/projects/sourcecodepro.adobe/files/>

Download the font of your choice and double click on the file. It should install automatically. Then change the line in your preferences file to:

`editor.font=Envy Code R,plain,16` or `editor.font=Source Code Pro,plain,16`

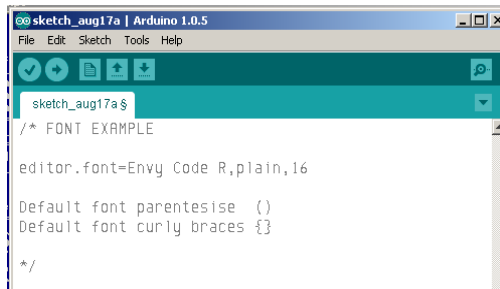


```
/* FONT EXAMPLE

editor.font=Monospaced,plain,16
(font size was increased from 12 to 16)

Default font parentesise ()
Default font curly braces {}

*/
```



```
sketch_aug17a$

/* FONT EXAMPLE

editor.font=Envy Code R,plain,16

Default font parentesise ()
Default font curly braces {}

*/
```

Envy Code R Font:

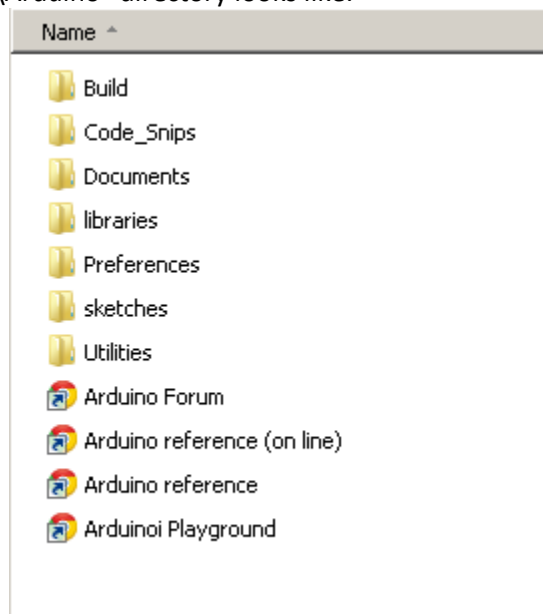
1234567890
ABCDEFGHIJKLMNOPQRSTUVWXYZ
() { } [] < > !@#%\$^&*

1234567890
ABCDEFGHIJKLMNOPQRSTUVWXYZ
() { } [] < > !@#%\$^&*

1234567890
ABCDEFGHIJKLMNOPQRSTUVWXYZ
() { } [] < > !@#%\$^&*

One thing to note: **Do NOT edit the preferences.txt file while you have the Arduino IDE running.** It will overwrite your changes when it exits.

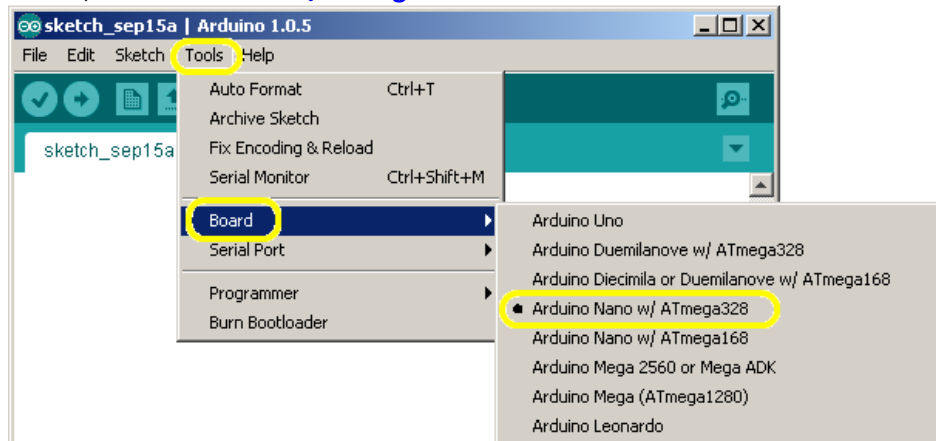
This is what my “My Documents\Arduino” directory looks like:



You already know what the folders: “Build” and “Sketches” are for. “Code_Snips” is for storing bits of code that I might want to use later or something that I have found on the internet that I want examine closer at some point when I have time. “Documents” is where I store related documentation ... like this document that I am writing in Word. “Libraries” is created by the Arduino IDE install program for storing users supplied/produced libraries (*code shared among several sketches*). “Utilities” holds small programs to do other things relative to Arduino environment. So far the only thing in it is a program to produce an assembly code listing from the from the ELF file produced by the Arduino IDE. The actual work is done by the utilities that come with the IDE but this program simplifies using them as they require some really long command lines (*about 260 characters*). The four shortcuts at the bottom should be self-explanatory. You may note that I have two shortcuts for the “Reference Guide”. There are sometimes differences between the on-line version and the one that comes with the IDE.

Arduino IDE: Compile and Upload

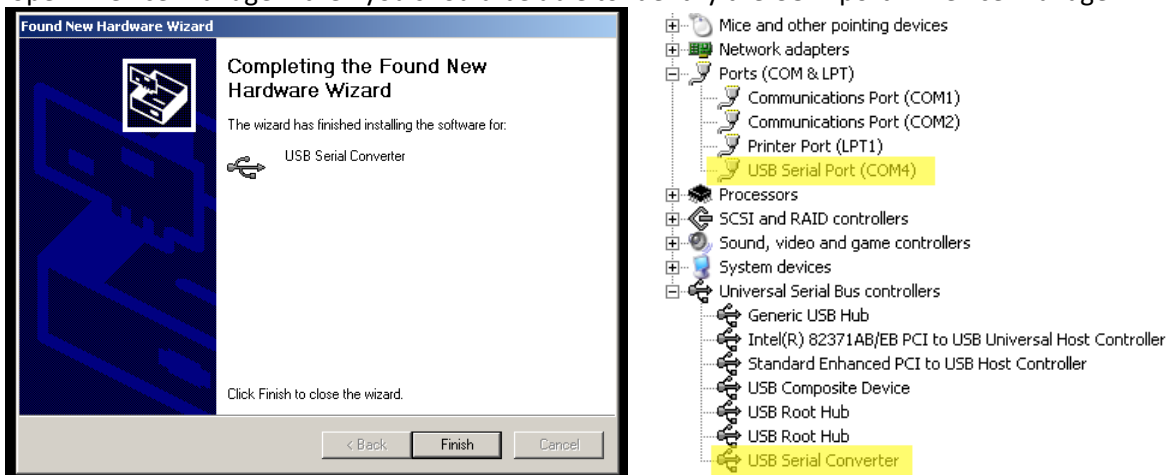
First thing that we need to do is learn how to enter, compile and upload a program in the in the Arduino IDE. Once the software is installed it must be configured for the ARDUNIO that is being used. From the top menu select **“Tools”**, **“Board”**, **“Arduino Nano w/ ATmega328”**.



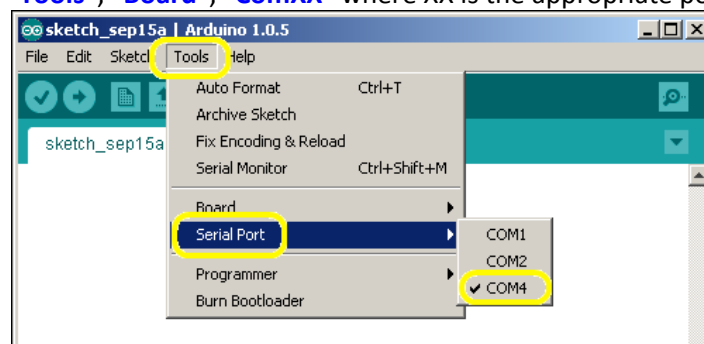
Next we have to tell the software where to find the Nano. Because we are using a USB hosted device that creates a “virtual” com port we can be certain that the port will not be COM1 or COM2. When the Nano is plugged into the computer Windows should automatically search for and find the device driver. If not then you can download the driver from:

<http://www.ftdichip.com/FTDrivers.htm>.

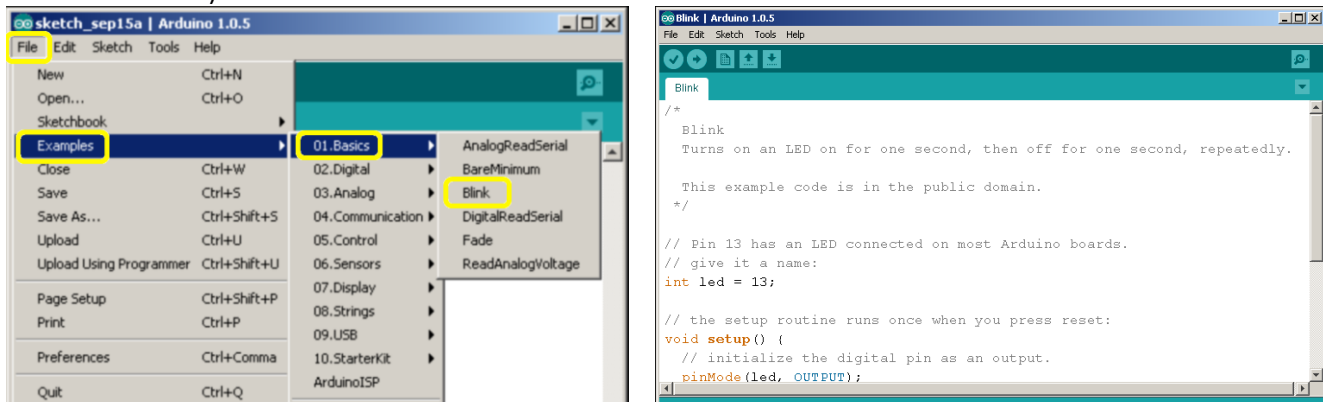
If you open “Device Manager” then you should be able to Identify the COM port in Device Manager.



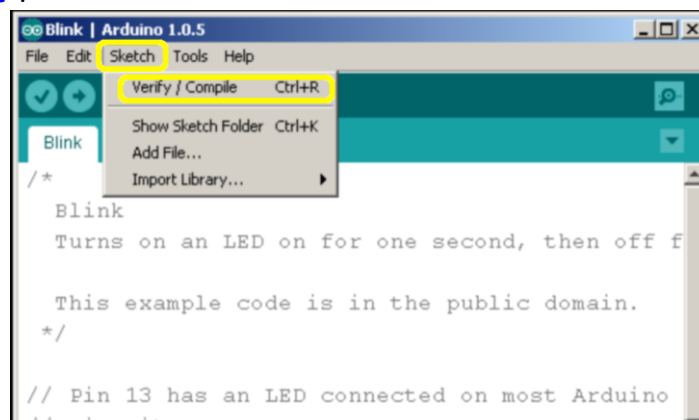
From the top menu select **“Tools”**, **“Board”**, **“ComXX”** where XX is the appropriate port number.



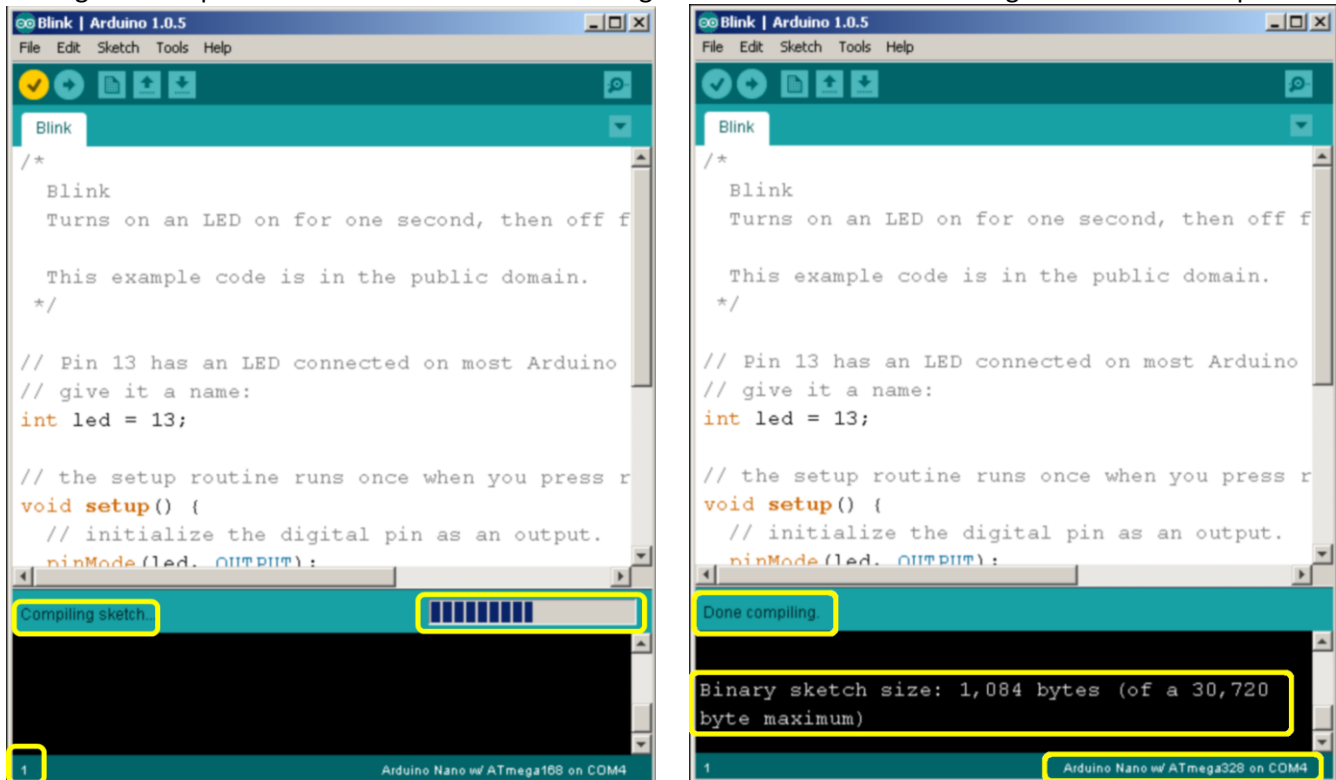
Most (*if not all*) Arduino boards have an LED ([Light Emitting Diode](#)) attached to pin 13. There is a small program to make this LED turn on and off. This program (*it is often called “Blinky”*) is used to verify the configuration and that you have a working Arduino board. From the top menu select “File”, “Examples”, “01. Basics”, “Blink”. This will open up a new Arduino window. You can close the previous blank one (*this is one of the things that I do **NOT** like about this IDE*).



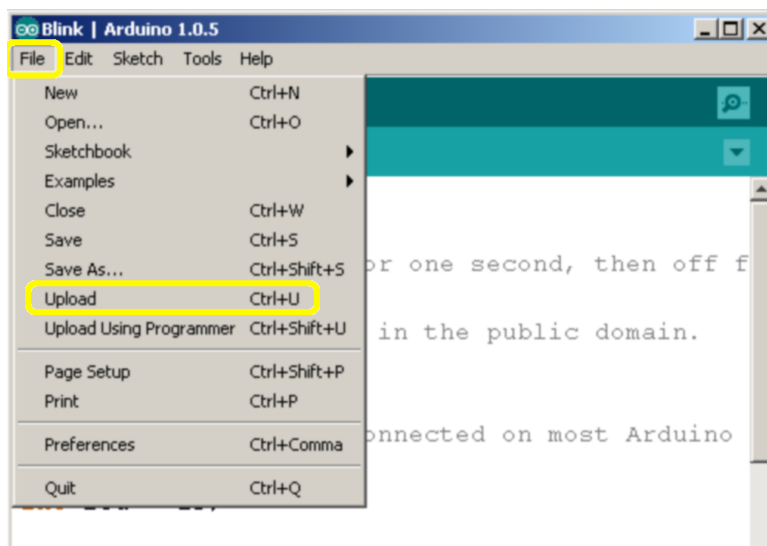
We need to compile the source code. Arduino programs are called “sketches”. From the top menu select “Sketch”, “Verify/Compile”.



In the area between the top and bottom panes of the program you will see a message telling you “Compiling Sketch”. To the right will be a progress bar. That itty-bitty number down in the lower left corner is the current line number. It is very useful when you are trying to debug a program. After the compile is completed the message will change to “Done Compiling”. In the bottom pane will be the results of the compile. In this case it tells us how big our program is. If there had been a problem like bad “syntax” then there would be error messages in this pane. Note also that at the bottom right the software shows the target board and com port.



It is time to upload the program. From the top menu select “File”, “Upload”. The Arduino software will compile the sketch again and automatically upload it to the Nano over the USB connection. After a moment the LED should begin blinking.



When you select “compile” or “upload” the Arduino IDE actually uses several industry standard open source programs to turn the source code into an executable file and upload it to the Nano (*avr-gcc, winavr, avdude*). For a detailed explanation see one of these URLs:

<http://arduino.cc/en/Hacking/BuildProcess>

<https://code.google.com/p/arduino/wiki/BuildProcess>

<http://openhardwareplatform.blogspot.com/2011/03/inside-arduino-build-process.html>

Now let us take a look at the Actual source code for this sketch.

```
/*  
  Blink  
  Turns on an LED on for one second, then off for one second, repeatedly.  
  
  This example code is in the public domain.  
*/  
  
// Pin 13 has an LED connected on most Arduino boards.  
// give it a name:  
int led = 13;  
  
// the setup routine runs once when you press reset:  
void setup() {  
  // initialize the digital pin as an output.  
  pinMode(led, OUTPUT);  
}  
  
// the loop routine runs over and over again forever:  
void loop() {  
  digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)  
  delay(1000);               // wait for a second  
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW  
  delay(1000);               // wait for a second  
}
```

Notice that “key words” are color coded in the code window. In the “C” language and the Arduino environment there are two ways to put in comments. One way is to place matching delimiters “/*” and “*/” around a section. We see this method used at the beginning of the sketch where the program is described. The second method is to use “//”. Anything on the line after two forward slashes is ignored by the compiler. We see that method used where the code explains Pin 13. Comments may appear anywhere in the program listing. Next we have our first line of code.

```
int led = 13;
```

This is a declaration of an “integer” variable with the name “led” that is initialized to be equal to “13”. All this does is to make the program code easier to read later on. Statements in “C” code are terminated with a semicolon “;” (*think of it like a period that ends a complete sentence*). If we had another LED on pin 14 we could change the variable in the program so as to blink either LED. As there is only the one then a constant (*a value that does not change during the execution of the program*) could have been used. That declaration would look like this:

```
const int led = 13;
```

Next we have our first function.

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```

All functions take the form of:

<pre>Type Name (Parameters) { Program Code; }</pre>

- "Type"** The data type of the returned value from the function. The keyword **"void"** indicates that **"setup"** will not be returning a value.
- "Name"** The name of the function that is used to call it from other sections of the program code. The **"setup"** function is the first of two functions required by all Arduino sketches. This function runs once when the program first begins.
- "Parameters"** A coma delimited list of values that are passed to the function when it is called. This list is always enclosed in parentheses **"()"**. The **"setup"** function is called without parameters.
- "Program Code"** The program statements that are to be executed by the function. These are always enclosed in braces **"{ }"**. In this case the keyword **"pinMode"** is used to set Pin 13 to a digital output (*"led" was set to equal 13 in the declare statement above*). We only need to do this once so setup is the appropriate place.

The **"setup"** function is followed by the second required function named **"loop"**.

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);             // wait for a second
  digitalWrite(led, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);             // wait for a second
}
```

Note that this function, like the **"setup"** function, returns nothing and gets no calling parameters. This function is what programmer's generally try to avoid ---- an endless loop, but here it serves a purpose. It will run until the end of time, the Nano is reset or power is removed. The comments following each statement explain what the statement does.

Just for completeness here is a sample of the disassembled machine code for this program.

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
100:      80 91 00 01    lds    r24, 0x0100
104:      61 e0        ldi     r22, 0x01        ; 1
106:      0e 94 b8 01    call   0x370          ; 0x370 <digitalWrite>
  delay(1000);           // wait for a second
10a:      68 ee        ldi     r22, 0xE8        ; 232
10c:      73 e0        ldi     r23, 0x03        ; 3
10e:      80 e0        ldi     r24, 0x00        ; 0
110:      90 e0        ldi     r25, 0x00        ; 0
112:      0e 94 e5 00    call   0x1ca          ; 0x1ca <delay>
  digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
116:      80 91 00 01    lds    r24, 0x0100
11a:      60 e0        ldi     r22, 0x00        ; 0
11c:      0e 94 b8 01    call   0x370          ; 0x370 <digitalWrite>
  delay(1000);           // wait for a second
120:      68 ee        ldi     r22, 0xE8        ; 232
122:      73 e0        ldi     r23, 0x03        ; 3
124:      80 e0        ldi     r24, 0x00        ; 0
126:      90 e0        ldi     r25, 0x00        ; 0
128:      0e 94 e5 00    call   0x1ca          ; 0x1ca <delay>
}
12c:      08 95        ret

0000012e <setup>:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
12e:      80 91 00 01    lds    r24, 0x0100
132:      61 e0        ldi     r22, 0x01        ; 1
134:      0e 94 79 01    call   0x2f2          ; 0x2f2 <pinMode>
}
138:      08 95        ret
```

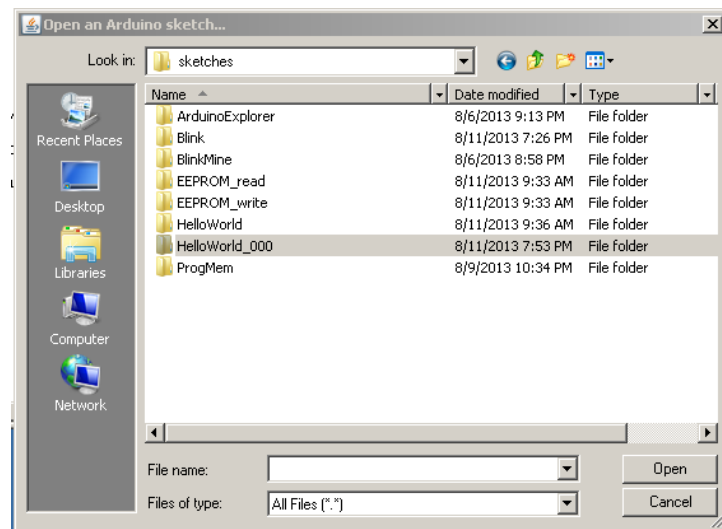
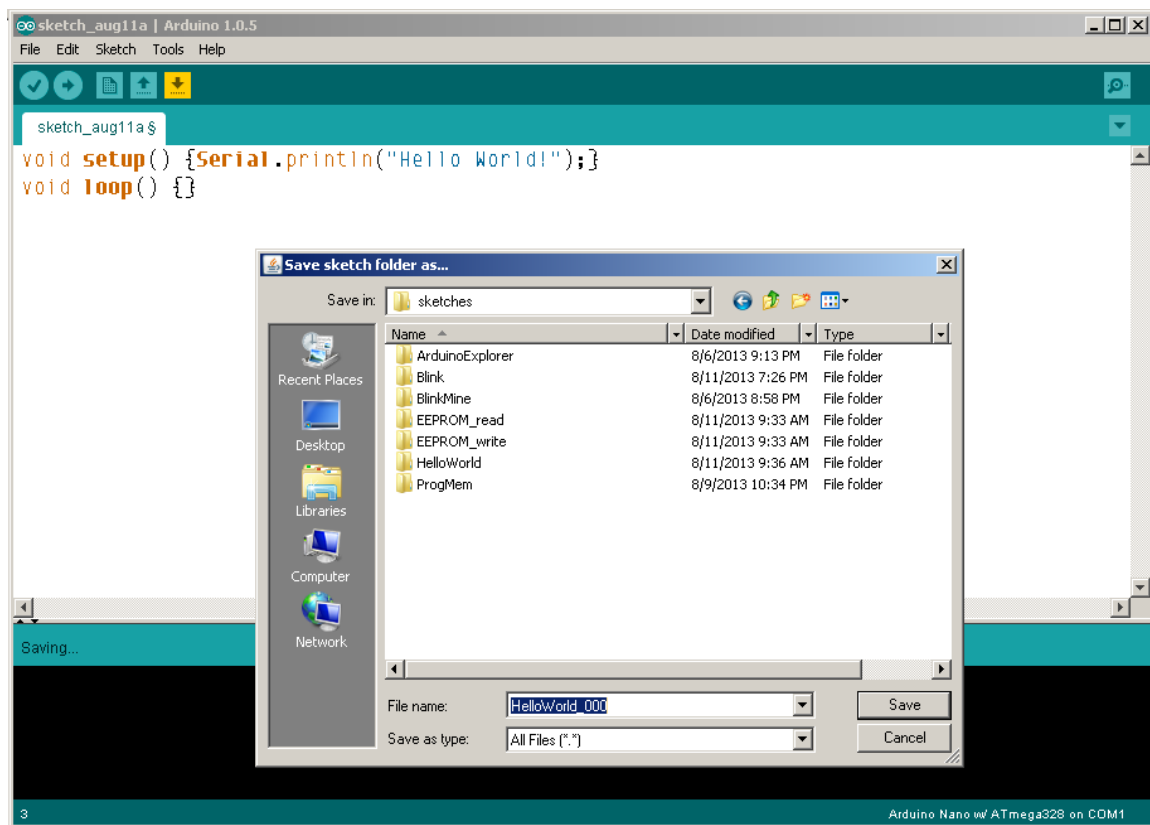
Enter, Save, Serial Monitor: “Hello Word”

The classic first program for any system is called “Hello World”. Open the Arduino IDE and type the following seven lines into the editor window (*this is about as simple as it gets* - **caution: “C” is case sensitive**):

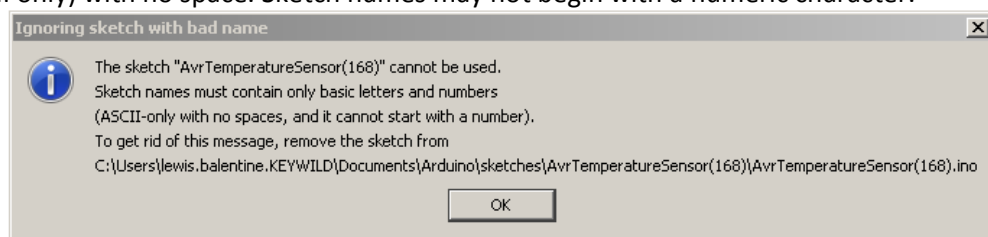
```
void setup()
{ Serial.begin(9600);
}
void loop()
{ Serial.println("Hello World!");
  delay (1000);
}
```

Note the two functions [Serial.begin](#) and [Serial.println](#). These two functions belong to a special “class” library that is part of the Arduino base system. The class member names (*begin and println*) are separated from the name of the library class (*Serial*) by a dot “.”. This library is only included in the final program when one of its functions is used. It adds about 1,750 bytes to the size of final program file. The “[Serial.begin](#)” function is called a “constructor” function. It constructs a new “instance” of the serial class that the other members can use. It must be called before any of the other class members can be called. In this case it is initializing the default TTL ([transistor to transistor level](#)) serial port at a speed of 9,600 [baud](#) . [Serial.println](#) sends a line of text to the serial port with a terminating character. The default serial port use [RS-232](#) serial protocol (*this is one of several protocols the Arduino can use*). This allows the Arduino to communicate with the computer via a serial terminal program.

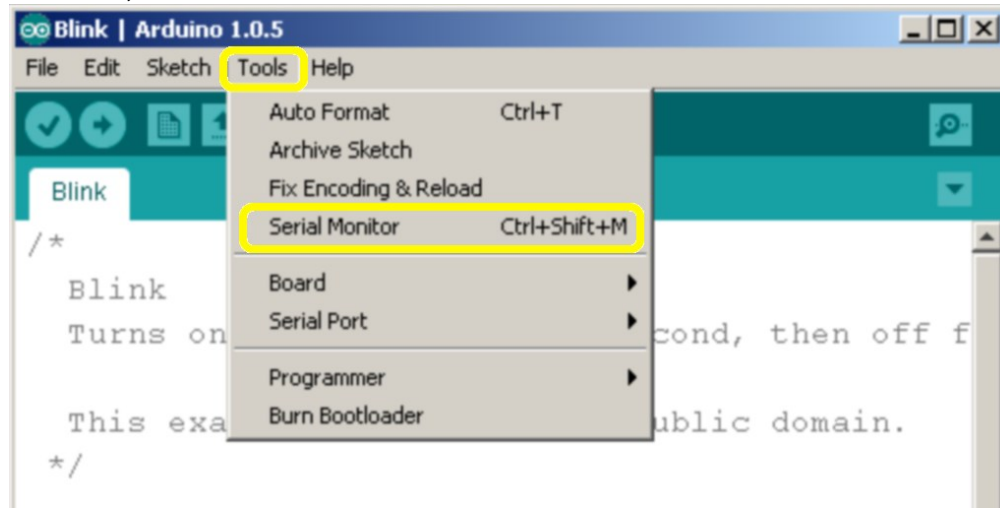
Now from the top menu select “[File](#)”, “[Saves As](#)”. When prompted for a name use “HelloWorld_000”. Notice that the Arduino IDE automatically creates a new folder for the sketch with the same name that you gave the file. This is your “project” folder. In that folder you should find a file with the name “HelloWorld_000.ino”. This is a plain text file that can be opened with any text editor except for Microsoft Windows Notepad (*consider replacing notepad with something like [Notepad++](#)*). The Microsoft convention is that all lines end with a combination of carriage return and line feed (*ASCII 13, 10*). The Arduino editor uses the Linux convention of terminating lines with only a line feed (*ASCII 10*). ASCII stands for American Standard Code for Information Interchange. See <http://en.wikipedia.org/wiki/Ascii> for more information on ASCII character codes.



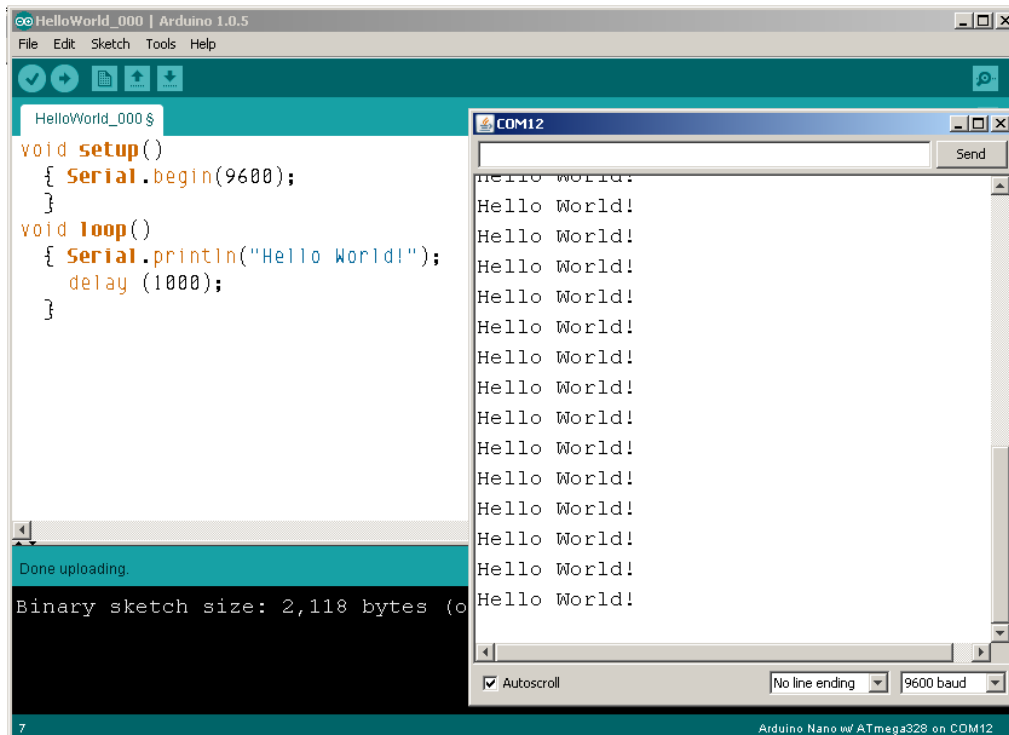
Warning: The Arduino IDE is very picky about file names: Sketch names may only include basic letters and numbers (ASCII only) with no space. Sketch names may not begin with a numeric character.



Compile and upload the program to the Nano. So you are saying to yourself “Nothing is happening”. From the top menu select “**Tools**”, “**Serial Monitor**”.



Now you should see the Nano printing Hello World once a second until the end of time (or when you disconnect it).



Whenever the computer opens the serial connection to Arduino the device is automatically “reset”. It has a special program called a “boot loader” that is run when the device is reset. That program waits for a few moments listening to the serial port. If it receives a special set of codes from the computer then it begins the process to upload a new program. Otherwise it starts the program that was previous uploaded. That is how the Arduino IDE is able to load new programs and communicate with the device over the same serial connection.

A brief word about my personal programing style (*or lack thereof*):

I tend to try to make program code as easy as possible for someone who is not familiar it with to read. Thus there will be lots of verbose comments. I also have a tendency to try to get a single concept into a space where it can be seen and comprehended without paging back and forth or digging through multiple files. This happens to be in direct conflict with conventional "C" programing style. There will also be examples of structures and functions that are supported by the language but are frowned upon by the many professional level programmers. Same experienced and/or professional "C" programmers are going to be offended by some of the code presented here. So be it.

The concept is for a person with absolute no knowledge or experience with the "C" language to build enough of a basic understanding of the language to get the task accomplished. It is NOT a dissertation aimed at producing the optimum "C" language program.

One must learn to walk years before they can enter the trials for Olympic Sprinters.

Some of the code shown will be in the form of screen captures because formatting code segments in Microsoft Word is a time consuming pain in the southern most region.

Funny Math: Bits, Nibbles and Bytes

Zeroes and Ones (Decimal, Binary and Hexadecimal)

Have heard that computing is all about “zeros and ones”? More likely you heard it was all about “ones and zeros” but zero should always come before one. Regardless of order the concept is true and that leads to some different ways of doing math. Input the following code into a new sketch called “Bits_and_Bytes01”.

```
void setup () // note this is shown in properly formatted “C” style
{
  Serial.begin (9600);
  word w = 0x000F;

  Serial.print ("w: ");
  Serial.print (w, DEC);
  Serial.print (" ", );
  Serial.print (w, HEX);
  Serial.print (" ", );
  Serial.print (w, BIN);
}

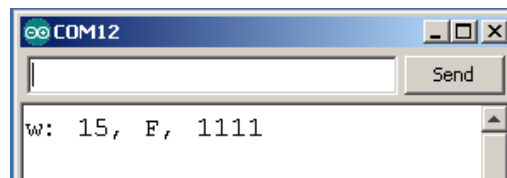
void loop() {
} // do nothing
```

We use the HEX input format “0x000F” for the initializing the word variable “w”. The “0x” (that is a zero not an upper case O) indicates the HEX format. The characters that follow it must be in the set “0123456789ABCDEF” (lower case alpha characters can be used but generally not). A word is a 16 bit unsigned number. Bytes are eight bits and nibbles are four bits. Our variable “w” is two bytes or four nibbles or sixteen bytes long. Each HEX character represents one nibble. Because the first three nibbles are zero we could have written “0xF”.

Notice that we have used a different member of the Serial class. `Serial.print` is like `Serial.println` but it does not send a newline termination. The `DEC`, `HEX`, `OCT` and `BIN` keywords tell the `Serial.print` function how to format the variable “w” when it is printed. It should be obvious but we “try” to never assume:

- DEC = format as decimal (*number base 10*) format
- HEX = format as hexadecimal (*number base 16*) format
- OCT = format as octal (*number base 8*) format ... (rarely used)
- BIN = format as binary (*number base 2*) format

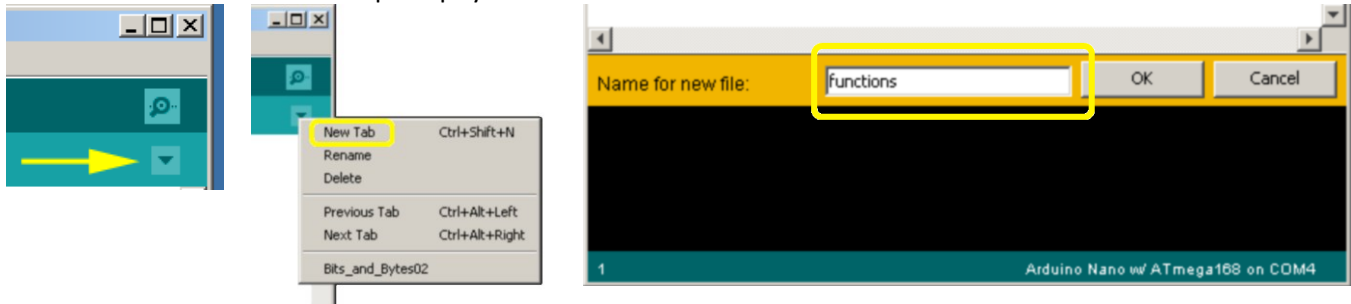
The Arduino IDE has a special function “Auto Format” feature under the tools menu to format your code in proper “C” style. I find it inconsistent. Notice in the setup function that it drops the open parenthesis to the first characters of the next line. The loop function has the open parenthesis appended to the end of the line where the function is declared. I find such inconsistencies irritating (*consistency in next to godliness*). That is another reason you will find that my code is often NOT properly formatted.



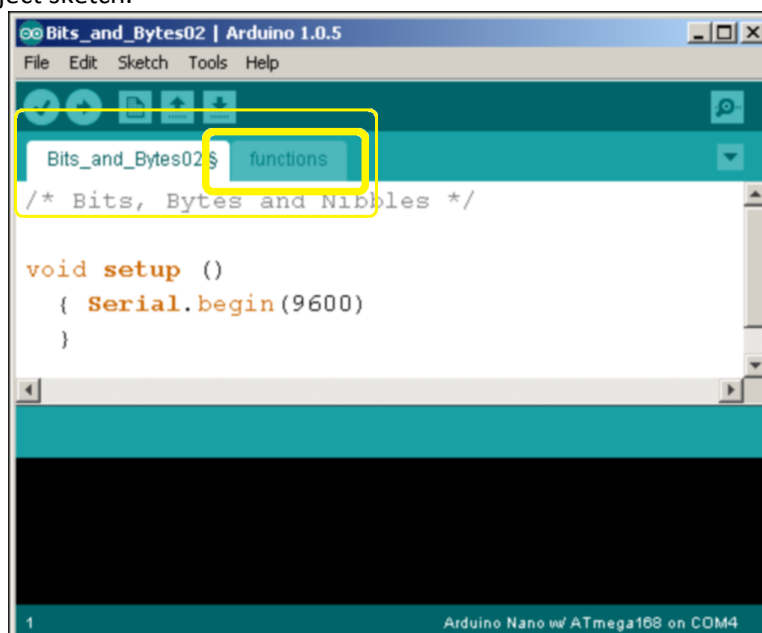
That is not all very impressive, but we are going to make it more interesting ... after we create a function to print the value of a word variable in all three formats on a single line. Open the Arduino IDE and start a new project named “Bits_and_Bytes01”. We are going to create an additional file for this project.

We want to add a file to the project. That option is not part of the IDE’s menu system. We must use an idiotcon

for that. It is in the top left corner and looks like an upside down triangle. Click on that you will have a menu. Select “**New Tab**”. The IDE will prompt you for a file name: use “functions”. Then click on the “**OK**” button.



Now you should now have two tabs for the IDE editor window including one named “functions”. The file is saved in the same directory of your sketch. The IDE will open any addition “.ino” files it finds in the sketch directory when you open the project sketch.



Open the new tab and enter the following code (*hint ... you can copy and paste*).

```
void printdnhb (char c, word n)
{
  Serial.print (c);
  Serial.print (": ");
  Serial.print (w, DEC);
  Serial.print (" ");
  Serial.print (w, HEX);
  Serial.print (" ");
  Serial.print (w, BIN);
}
```

The keyword “**void**” tells the compiler that our user defined function named “**printhdc**” will not be returning a value. In the parameter section we tell the compiler to expect two variables: a single character variable “**c**” and a two byte word variable “**n**”. These variables will ONLY be visible inside the function. Note that we had to add one line to print the incoming character. Now go back to first tab and enter a line to call our new function.

```
void setup ()
{
  Serial.begin (9600);
  word w = 0x000F;
```

```

    printdhhb ('w', w);
}

void loop() {
} // do nothing

```

When you run this program the output should look the same as the previous one. Now add some more values.

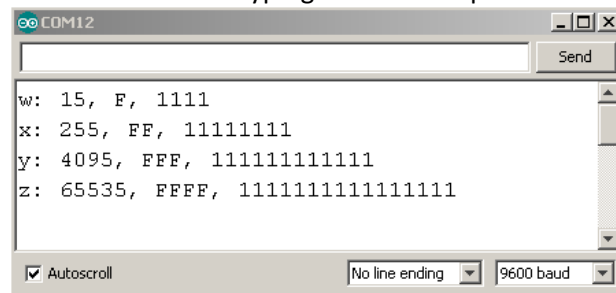
```

void setup ()
{
    Serial.begin (9600);
    word v = 0x0000;
    word w = 0x000F;
    word x = 0x00FF;
    word y = 0x0FFF;
    word z = 0xFFFF;
    printdhhb ('v', w);
    printdhhb ('w', w);
    printdhhb ('x', x);
    printdhhb ('y', y);
    printdhhb ('z', z);
}

void loop() {
} // do nothing

```

By creating the function we avoided all that extra typing to enter the print statements for the new values.



This would be much easier to comprehend if the numbers were “right justified”. Even better would be to have them zero padded. We can do that! Go back to the functions tab and enter this code (*hint ... read ahead a bit*).

```

void printdhhb (char c, word n)    // properly formatted “C” style code
{
    byte i,s;
    Serial.print (c);
    Serial.print (": ");
    if (n<10)                        // begin if then for decimal
    {
        s=4;
    }
    else if (n<100)
    {
        s=3;
    }
    else if (n<1000)
    {
        s=2;
    }
    else if (n<10000)
    {
        s=1;
    }
    else {
        s=0;
    }
}

```

```

}                                     // end of if then for decimal
i=0;
while (i < s)                         // begin while loop for decimal
{
    Serial.print (' ');
    i= i+1;
}                                     // end of while loop for decimal
Serial.print (n, DEC);

Serial.print (" ");
if (n<0x10)                          // begin if then for hex
{
    s=3;
}
else if (n<0x100)
{
    s=2;
}
else if (n<0x1000)
{
    s=1;
}
else
{
    s=0;
}                                     // end of if then for hex
i=0;
while (i < s)                         // begin while loop for hex
{
    Serial.print ('0');
    i= i+1;
}                                     // end of while loop for hex
Serial.print (n, HEX);

Serial.print (" ");
Serial.println (n, BIN);
}

```

Obviously whoever determined the proper format for “C” code was getting paid by the number of lines of code that they produced. Here is the (*almost*) human readable form with the binary formatting added as well.

```

void printdwb (char c, word n)       // improperly formatted “C” style code
{ byte i,s,v;
  Serial.print (c);
  Serial.print (" ");

  if      (n< 10) { s=4;}             // begin if then for decimal
  else if (n< 100) { s=3;}
  else if (n< 1000) { s=2;}
  else if (n<10000) { s=1;}
  else      { s=0;}                 // end of if then for decimal
  i=0;
  while (i < s)                     // begin while loop for decimal
  { Serial.print (' ');
    i= i+1;
  }                                     // end of while loop for decimal
  Serial.print (n, DEC);

  Serial.print (" ");
  if      (n< 0x0010) { s=3;}         // begin if then for hex
  else if (n< 0x0100) { s=2;}
  else if (n< 0x1000) { s=1;}
  else      { s=0;}                 // end of if then for hex
  i=0;
  while (i < s)                     // begin while loop for hex

```

```

    { Serial.print ('0');
      i= i+1;
    }
    Serial.print (n, HEX);

    Serial.print (" ");
    /* the Arduino IDE/compiler has a problem with the binary format values larger than a byte
       so we have to do this a byte at a time */
    v = highByte(n);
    if (v< B00000010) { s=7;} // begin if then for binary high byte
    else if (v< B00000100) { s=6;}
    else if (v< B000001000) { s=5;}
    else if (v< B000010000) { s=4;}
    else if (v< B000100000) { s=3;}
    else if (v< B001000000) { s=2;}
    else if (v< B010000000) { s=1;}
    else { s=0;} // end of if then for binary high byte
    i=0;
    while (i < s) // begin while loop for binary high byte
    { Serial.print ('0');
      i++;
    } // end of while loop for binary high byte
    Serial.print (v, BIN);
    Serial.print (" ");
    v = lowByte(n);
    if (v< B00000010) { s=7;} // begin if then for binary low byte
    else if (v< B00000100) { s=6;}
    else if (v< B000001000) { s=5;}
    else if (v< B000010000) { s=4;}
    else if (v< B000100000) { s=3;}
    else if (v< B001000000) { s=2;}
    else if (v< B010000000) { s=1;}
    else { s=0;} // end of if then for binary low byte
    i=0;
    while (i < s) // begin while loop for binary low byte
    { Serial.print ('0');
      i= i+1;
    } // end of while loop for binary low byte
    Serial.println (v, BIN);
}

```

Obviously there are better ways to do this, but first we need to learn some funny math and other things. One of those is “control structures”. A control structure is used where the program has to make a decision: “**What do I do next?**” In this program we have our first control structure the “**if then else**” structure.

```

if (n< 10) { s=4;} // begin if then for decimal
else if (n< 100) { s=3;}
else if (n< 1000) { s=2;}
else if (n<10000) { s=1;}
else { s=0;} // end of if then for decimal

```

Usually a control structure has a criteria or condition to use in making its decision. The “if” keyword is followed by that criteria enclosed within parentheses. In this case we testing the value of the variable “n” to see if it is less than 10 by using the less than comparison operator “<”. If the condition within the parentheses is **true** then the program executes the statements within the curly braces that follow. In this case we have only one statement so I have written the entire sentence on one line to make it easier to read. If the condition within the parentheses is **false** then the program skips those steps and goes to the next line. The “else” keyword ties our next line to the previous “if” control. We increase the test by a factor of ten for each digit. The program executes the first line that it finds to be true and skips the rest. We end to the structure with a simple “else” statement that is to be executed if none of the test conditions preceding it were found to be true. In this case we are dividing our decimal number by decimal ten to see how many zeroes we need to pad to the left of the number. If the

number is between 0 and 9 (inclusive) then we need 4 zeroes. We follow this same logic until we get to the point where the number is greater than (or equal to 10000). In that case we do not need any extra zeroes. This is a form of “fall through” logic. We fall through each test until we find a match. Then we come to our next control structure:

```
i=0;
while (i < s)                // begin while loop for decimal
{ Serial.print ( ' ');
  i= i+1;
}                             // end of while loop for decimal
```

This is called a “while” loop. It will execute the statements between the curly braces for as long as the test condition is true. There needs to be some sort of statement within those braces to affect the test condition. Otherwise the loop will execute forever (like the main loop required by the Arduino IDE). Here we use the statement “i = i+1;” to increase the value of our variable “i”. Note that we set the value of “i=0;” before we execute the while statement. If “s” is equal to zero then the while loop will not execute at all. We use this loop to print the required number of space in front of the decimal value. Then we print the decimal value.

Note: Conditional statements and/or control structures may be followed by a group of executable statements but that is not always the case. If there is only a single statement then the curly braces are NOT required. These two sections of code are both valid and equivalent:

```
if (n< 10) { s=4;}           // begin if then for decimal
else if (n< 100) { s=3;}
else if (n< 1000) { s=2;}
else if (n<10000) { s=1;}
else { s=0;}                 // end of if then for decimal
//-----
if (n< 10) s=4;              // begin if then for decimal
else if (n< 100) s=3;
else if (n< 1000) s=2;
else if (n<10000) s=1;
else s=0;                   // end of if then for decimal
```

Note: A while loop does NOT require executable statements.

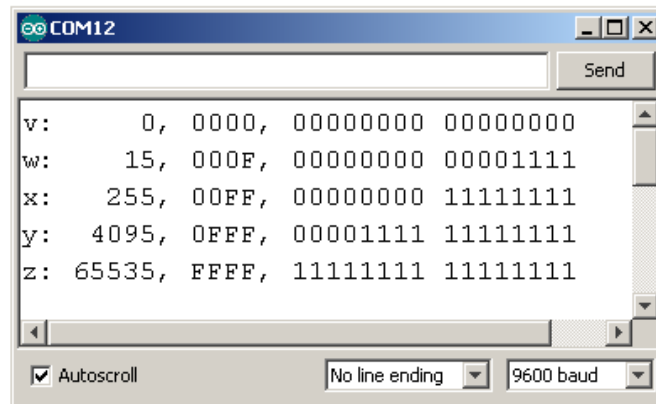
```
n=0;
while (n<10) {n = n+1;}      // all three of these statements
while (n<10) n = n+1;        // are valid
while (n++<10);              // and produce the same result
. . .
n=0;
while (n==0);                // valid but generates an endless loop
while (n!=0);                // valid, does nothing (may be ignored by the compiler)
```

Note: A while may use multiple conditions separated by commas.

```
byte a=13, b=0, c=1;
Serial.println ();
while (a>0, b<5, c!=7 )     // multiple conditions specified for while loop
{ Serial.print ("a=");
  Serial.print (a--);
  Serial.print (" , b=");
  Serial.print (b++);
  c=a-b;
  Serial.print (" , c=(a-b)=");
  Serial.print (c);
  Serial.println ();
  // the next statement waits for serial input
  while (Serial.available()==0); // note that the executable statement is the parameter
}
```

The same type of logic is used for the hex and binary values. The difference is how we specify the value limits we are testing. We use the hex notation for the hex values and binary notation for the binary values. The “B” in the front of the number “B00000010” tells the compiler to interpret the following digits as a binary number. Each digit must be in the set “01”. Some programming languages have used a similar notation for HEX numbers (i.e. “H00FF”) and some assembly code system used postfix notation (i.e. “00FFh”). You may see this form of hex notation in notes, comments and documentation. Unfortunately consistency is not one of the “C” language’s strong points. Because the IDE/compiler cannot handle the Binary format of more than one byte we must introduce our first bit of funny math. The two keywords “lowByte” and highByte” (remember “C” is case sensitive) are used to split our two byte word value into single bytes in the temporary byte variable “v”.

If you run the program with the new form of the function then it should be much easier to understand the results.



Notice now all our highest values are odd numbers. In the everyday life zero has no value so you were probably taught in kindergarten to count starting with one. Digital computers only have two numbers: zero and one. Thus zero is very important. In “C” and most other programming languages we start counting at zero.

Side note:
Traditional Chinese weighing units was a hexadecimal system (like ounces in the English system). Early Chinese suanpan abacuses with two ‘heaven’ beads and five ‘earth’ beads could be used for either decimal or hexadecimal calculations.

The “C” language allows us to pass variables of the same size but of a different type to a function. It is up to the programmer to be certain that the function will handle the variable passed properly. In the setup function comment out the variable declarations and declare them to be integers. Then add the additional code at the bottom.

```
void setup ()
{
  Serial.begin (9600);
  // word v = 0x0000;
  // word w = 0x000F;
  // word x = 0x00FF;
  // word y = 0x0FFF;
  // word z = 0xFFFF;

  int v = 0x0000;
  int w = 0x000F;
  int x = 0x00FF;
  int y = 0x0FFF;
  int z = 0xFFFF;
  printdwb ('v', w);
  printdwb ('w', w);
  printdwb ('x', x);
}
```

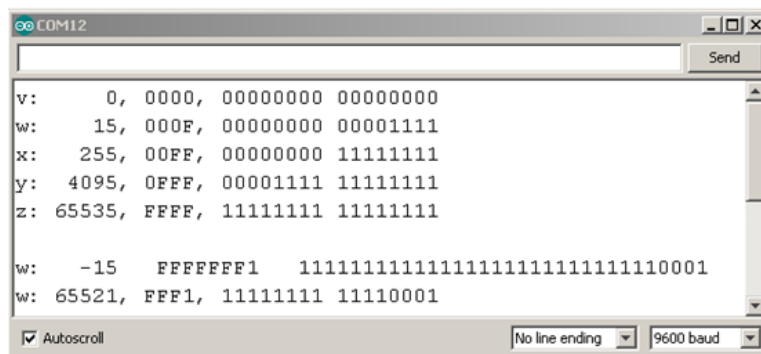
```

    printdhhb ('y', y);
    printdhhb ('z', z);

    Serial.println("");
    w=-15;
    Serial.print ("w:  ");
    Serial.print (w, DEC);
    Serial.print (" ");
    Serial.print (w, HEX);
    Serial.print (" ");
    Serial.println (w, BIN);
    printdhhb ('w', w);
}

void loop() {
} // do nothing

```



WOW! Look at all those ones. Our Arduino handles negative numbers as the “two's complement” (see http://en.wikipedia.org/wiki/Two's_complement). When the “C” compiler passed the parameters to our function it checked the number of parameters and size but did not place limits on the type. Our function was not designed to handle negative numbers and had no way of knowing that the passed parameter was supposed to be a signed integer. The systems Serial.print function did a bit better in that it recognized that the number was negative but it handled the number as signed long (*four bytes*). This is something to be very conscious of. If you deal with negative numbers be sure that the functions that you use are provisioned for negative numbers. The same thing applies for using unsigned numbers.

Side note:

The official ASCII character table is only 7 bits. However most ASCII character charts lists character codes from 0 to 255 (*8 bits*). Note the definition of the char type in the Arduino reference:

The char datatype is a signed type, meaning that it encodes numbers from -128 to 127.

The second page (128-255) of the character chart should actually be listed as negative numbers, however the way the two's complement works out ... it is the same thing (hint: $-255 = 128 \sim \sim \sim$ *funny math* $\sim \sim \sim$).

Save your work.

Divide by Zero (yes we can)

Copy the previous files to a new folder. Be sure that you rename the folder and the file “Bits_and_Bytes03” (*and do not forget to include the functions file*). We are going to do some funny math. Before we do that let us take a quick look at some simple normal math.

16/2=8 8/2=4 4/2=2 2/2=1	2*2*2*2=16 4*4=16	16/16=1	16/0= ??? NOT ALLOWED!
-----------------------------------	----------------------	---------	----------------------------------

You were probably taught that you cannot divide by zero. Well we can and by doing it different ways we can get different answers.

```

/* Bits, Bytes and Nibbles */

void setup ()
{
  Serial.begin (9600);
  word a,b,c;

  Serial.println ("a=0xFFFF");
  a=0xFFFF;
  printdwb ('a', a);
  Serial.println ("");

  Serial.print ("16=");
  Serial.println (16, HEX);
  Serial.println ("a/16=");
  a=a/16;
  printdwb ('a', a);
  Serial.println ("");

  Serial.print ("16*16=");
  Serial.println (16*16, HEX);
  Serial.println ("a/(16*16)=");
  a=a/16;
  printdwb ('a', a);
  Serial.println ("");

  Serial.print ("16*16*16=");
  Serial.println (16*16*16, HEX);
  Serial.println ("a/(16*16*16)=");
  a=a/16;
  printdwb ('a', a);
  Serial.println ("");

  Serial.print ("16*16*16*16=");
  Serial.println (16*16*16*16, HEX);
  Serial.println ("a/(16*16*16*16)=");
  a=a/16;
  printdwb ('a', a);
  Serial.println ("");

  a=0xFFFF;
  b=16*16*16*16;
  printdwb ('b', b);
  a=a/b;
  printdwb ('a', a);
  Serial.println ("");
}

void loop() {
} // do nothing

```

```

COM12
a=0xFFFF
a: 65535, FFFF, 11111111 11111111

16=10
a/16=
a: 4095, 0FFF, 00001111 11111111

16*16=100
a/(16*16)=
a: 255, 00FF, 00000000 11111111

16*16*16=1000
a/(16*16*16)=
a: 15, 000F, 00000000 00001111

16*16*16*16=0
a/(16*16*16*16)=
a: 0, 0000, 00000000 00000000

b: 0, 0000, 00000000 00000000
a: 65535, FFFF, 11111111 11111111

```

There are no comments in this code because the print statements make it “self-documenting”. We start off with word value set to its maximum value 0xFFFF (*all ones*). We divide it four times by the number 16. According to normal math that should be the same as dividing by (4*16) but such is not the case. Lastly we take the number 0xFFFF and divide it directly by zero. The last value returned is zero. The compiler does not object and returns the original value (*same as dividing by one*). *~~~funny math~~~*

What is actually happening we are hitting the upper and lower limits? When we divide by 16 we are shifting the number to the right by one hexadecimal position. The ones on the right are discarded and the left is filled with zeroes. When we do this four times all of the ones have been discarded. Thus the answer is zero.

Did you notice 16*16*16*16=0? We hit the upper limit and it rolled over to the beginning which is of course zero. So why did directly dividing by zero return the original number. I do not have a good answer for that one. I believe it is because integer division is actually done using the shift operator and subtraction. Shift left or right zero positions does nothing.

The whole point to this exercise is that dividing a hex number by decimal sixteen is similar to dividing a decimal number by decimal ten. In hexadecimal notation the decimal value 16 = hexadecimal 10. All it is doing is moving the number left or right one position. Computer processors have a special instruction for this that is much faster than integer division. It is called the “shift” instruction. In “C” the operation is noted by bitwise shift operator “>>”.

XXXX >> 0	is the same as XXXX / 1	(1)
XXXX >> 1	is the same as XXXX / 2	(2)
XXXX >> 2	is the same as XXXX / 4	(2*2)
XXXX >> 3	is the same as XXXX / 8	(2*2*2)
XXXX >> 4	is the same as XXXX / 16	(2*2*2*2)
XXXX >> 5	is the same as XXXX / 32	(2*2*2*2*2)
XXXX >> 6	is the same as XXXX / 64	(2*2*2*2*2*2)
XXXX >> 7	is the same as XXXX / 128	(2*2*2*2*2*2*2)
XXXX >> 8	is the same as XXXX / 256	(2*2*2*2*2*2*2*2)
XXXX >> 9	is the same as XXXX / 512	(2*2*2*2*2*2*2*2*2)
XXXX >> 10	is the same as XXXX / 1,024	(2*2*2*2*2*2*2*2*2*2)

XXXX >> 11 is the same as XXXX / 2,048	(2*2*2*2*2*2*2*2*2*2*2)
XXXX >> 12 is the same as XXXX / 4,096	(2*2*2*2*2*2*2*2*2*2*2*2)
XXXX >> 13 is the same as XXXX / 8,192	(2*2*2*2*2*2*2*2*2*2*2*2*2)
XXXX >> 14 is the same as XXXX / 16,384	(2*2*2*2*2*2*2*2*2*2*2*2*2*2)
XXXX >> 14 is the same as XXXX / 32,768	(2*2*2*2*2*2*2*2*2*2*2*2*2*2*2)
XXXX >> 16 is the same as XXXX / 65,536	(2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2)

Hopefully you will have noticed an important pattern. This is handy for isolating a portion of a piece of data ... just like the lowByte and highByte keywords we used earlier. Comment out the earlier code (*I hope that you have found the “Edit, Comment/Uncomment” feature*) so we can try something else. First let us modify our print function a slightly.

```
// void printdwb (char c, word n)
void printdwb (char c[], word n)
{ byte i,s,v;
  Serial.print (c);
  // Serial.print (": ");
  ...   /// those three dots means the code is continued on as it was previously
```

All we are doing changing the function such that it can print a char array rather than a single character by adding the array notation “[]”. The compiler now passes pointer to the array rather than trying to pass the entire array itself. Serial.print knows the difference between a pointer and char value. Now try this in setup.

```
/* Bits, Bytes and Nibbles */

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  printdwb ("      DHB values 0xFFFF: ", 0xFFFF);
  printdwb ("      DHB values 0xFFFF/16: ", (0xFFFF / 16));
  printdwb (" DHB values 0xFFFF >> 4: ", (0xFFFF >> 4));
  Serial.println ("");
  printdwb ("      DHB values 0xFFFF: ", 0xFFFF);
  printdwb ("      DHB values 0xFFFF*16: ", (0xFFFF * 16));
  printdwb (" DHB values 0xFFFF << 4: ", (0xFFFF << 4));
  Serial.println ("");
  printdwb ("      DHB values 0xFFFF: ", 0xFFFF);
  printdwb ("      DHB values 0xFFFF/256: ", (0xFFFF / 256));
  printdwb (" DHB values 0xFFFF >> 8: ", (0xFFFF >> 8));
  Serial.println ("");
  printdwb ("      DHB values 0xFFFF: ", 0xFFFF);
  printdwb ("      DHB values 0xFFFF*256: ", (0xFFFF * 256));
  printdwb (" DHB values 0xFFFF << 8: ", (0xFFFF << 8));
  Serial.println ("");

  // Serial.println ("a=0xFFFF");
  // a=0xFFFF;
  . . .
```

```

DHB values 0xFFFF: 65535, FFFF, 11111111 11111111
DHB values 0xFFFF/16: 4095, 0FFF, 00001111 11111111
DHB values 0xFFFF >> 4: 4095, 0FFF, 00001111 11111111

DHB values 0xFFFF: 65535, FFFF, 11111111 11111111
DHB values 0xFFFF*16: 65520, FFF0, 11111111 11110000
DHB values 0xFFFF << 4: 65520, FFF0, 11111111 11110000

DHB values 0xFFFF: 65535, FFFF, 11111111 11111111
DHB values 0xFFFF/256: 255, 00FF, 00000000 11111111
DHB values 0xFFFF >> 8: 255, 00FF, 00000000 11111111

DHB values 0xFFFF: 65535, FFFF, 11111111 11111111
DHB values 0xFFFF*256: 65280, FF00, 11111111 00000000
DHB values 0xFFFF << 8: 65280, FF00, 11111111 00000000

```

Sometime you want a specific part of the whole that is not conveniently located on the end. In that case you simple use a combination. Let us put the center 8 bits into the lower byte. We are going to use a number that is a bit more distinctive.

```

/* Bits, Bytes and Nibbles */

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  a=0x4321;
  printdnhb ("DHB values 'a' = 0x4321: ", a);
  a=a << 4;
  printdnhb ("      DHB values 'a' << 4: ", a);
  a=a >> 8;
  printdnhb ("      DHB values 'a' >> 8: ", a);

  // printdnhb ("      Hex value 0xFFFF: ", 0xFFFF);
  . . .

```

```

DHB values 'a' = 0x4321: 17185, 4321, 01000011 00100001
DHB values 'a' << 4: 12816, 3210, 00110010 00010000
DHB values 'a' >> 8: 50, 0032, 00000000 00110010

```

See how the digits “32” moved from the center to the left end and then they were moved to the right end. The “C” language lets us do things inside the function calls as well. This version works the same as the code above (*but is a bit less obvious*).

```

/* Bits, Bytes and Nibbles */

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  printdnhb (" DHB values 'a' = 0x4321: ", (a=0x4321));
  printdnhb ("      DHB values 'a' << 4: ", (a=a << 4));
  printdnhb ("      DHB values 'a' >> 8: ", (a=a >> 8));

  // printdnhb ("      Hex value 0xFFFF: ", 0xFFFF);
  . . .

```

There are “many ways to skin a cat” in the “C” language. An even shorter form uses the compound operator notation and no inside parenthesis. In the case of the shift operator the operator must precede the equals sign.

```
/* Bits, Bytes and Nibbles */

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  printdnhb (" DHB values 'a' = 0x4321: ", a=0x4321);
  printdnhb ("      DHB values 'a' << 4: ", a <<= 4);
  printdnhb ("      Hex value 'a' >> 8: ", a >>= 8);

  // printdnhb ("      Hex value 0xFFFF: ", 0xFFFF);
  . . .
}
```

There are some other compound operators that are very useful but look a little funny when you first see them. Go back and look at the loop our printdnhb function.

```
i=0;
while (i < s)                // begin while loop for decimal
{ Serial.print (' ');
  i= i+1;
}                             // end of while loop for decimal
```

We can use the “C” compound increment function “++” to shorten this code a bit.

```
i=0;
while (i++ < s) { Serial.print (' ');} // while loop for decimal
```

The “++” operator increments the variable “i” by one: (i=i+1). This is “postfix” notation. That means it returns the value of “i” to the function before it does the “i=i+1” operation. The first time into the loop the value zero is returned. The next time the value one is returned and so on until the value reaches “s”. There is also a “prefix” notation form:

```
i=0;
while (++i < s) { Serial.print (' ');}
```

In this case the value is incremented before it is returned. Thus the first time in to the loop the value of “i” is one. There is a similar function to decrement the value of a variable: “--”. To make the program even shorter we can initialize the value for the variable inside the test criteria.

```
while (i=0, ++i < s) { Serial.print (' ');}
```

Here are some more compound operators *~~~funny math~~~*:

Operation	Compound Operator	Sample	Same as
Addition	+=	i += 5	(i = i + 5)
Subtraction	-=	i -= 5	(i = i - 5)
Multiplication	*=	i *= 5	(i = i * 5)
Division	/=	i /= 5	(i = i / 5)

We are going to add another control structure. It is the “for” control structure. Here is a simple example.

```
for (i=0; i<10; i= i+1)
{
  Serial.println (i, DEC);
}
```

The “for” keyword needs three sets of parameters. The parameters are delimited by semicolons. The first is for the initialization. This statement is executed one time. The second is the test condition. This is executed before the before executing the code block. If the result is false then the code block is not executed. The third will be executed after the code block. The “for” control structure repeatedly executes the block of code {statements

between the curly braces} until the condition returns false. The example above would print the digits “0” to “9” to the serial terminal. You will frequently see the compound operator used in the third parameter set of the “for” control structure. I have been using the word “set” because you can actually have multiple comma delimited statements in each parameter set as in this example:

```
for (i=0, c=65; i<26, c<100; i++, c++)
{
    Serial.println (char(c));
}
```

This example would print the characters “A” to “Z” to the serial terminal.

Consider the “if then else” structure that we used in the printdnhb function.

```
if (n< 10) { s=4;} // begin if then for decimal
else if (n< 100) { s=3;}
else if (n< 1000) { s=2;}
else if (n<10000) { s=1;}
else { s=0;} // end of if then for decimal
```

This was not bad for decimal and hex but it got rather long for binary. It uses a value that is repeatable multiplied by 10. We can take advantage of some of our funny math and new control structures to shorten the function.

```
void printdnhb (char c[], word n) // function to print value as
{ byte s; // Decimal, Hexidecimal and binary
  long i;

  Serial.print (c); // print incoming string
  for (i=10, s=0; i <= 10000; i *= 10) // pad decimal value with spaces
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print (' ');}
  Serial.print (n, DEC); // print decimal value

  Serial.print (" "); // print seperator
  for (i=0x10, s=0; i <= 0x1000; i *= 0x10) // pad hex value with zeroes
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print ('0');}
  Serial.print (n, HEX); // print hex value

  Serial.print (" "); // print seperator
  /* the Arduino IDE/compiler has a problem with the binary format values larger than a byte
  so we have to do this a byte at a time */
  printbin(highByte(n)); // call function to print high byte
  Serial.print (' '); // print seperator
  printbin(lowByte(n)); // call function to print low byte
  Serial.println (); // print a new line
}

void printbin (byte v) // function to pad and print one binary byte
{ word i;
  byte s;
  byte n; // used for a nibble
  n= v >> 4; // move high nibble into low nibble position
  for (i= B10, s=0; i <= B1000; i *= B10) // pad binary value with zeroes
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print ('0');}
  Serial.print (n, BIN); // print binary value
```

```

Serial.print (' ');

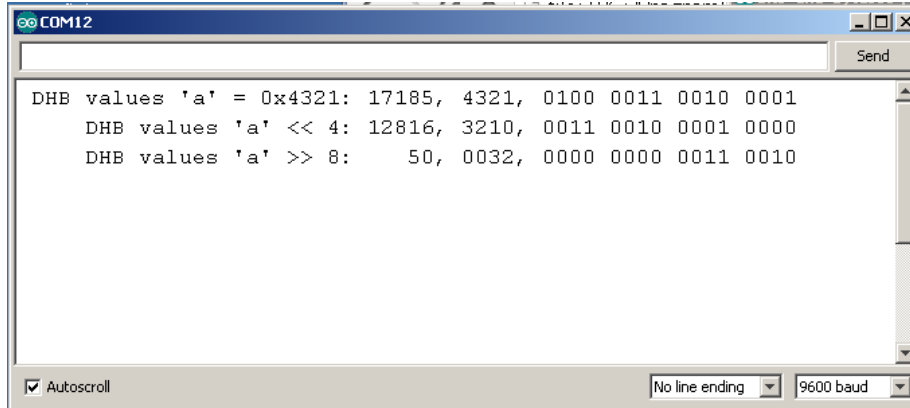
n= v << 4;                                // roll the high bits off to the left
n= n >> 4;                                // move the left over bits back to the right
for (i= B10, s=0; i <= B1000; i *= B10)  // pad binary value with zeroes
{ if (n<i) { s++;}
}
i=0;
while (i++ < s) { Serial.print ('0');}
Serial.print (n, BIN);                    // print binary value
}

```

Oops ... another function slipped in there. Our function calls are now three levels deep. Every level deeper we go the code becomes more difficult to read. The upper levels begin to look nothing at all like the original language. Take a look at the setup function in our program. It is comprised principally of calls to our user defined functions. In order to understand what is happening one must start at the top and work their way down to the bottom level while learning the meanings of the user defined “keywords” we have added to the language. Two or three levels are not too bad. Unfortunately the larger and more complex the program becomes then the tendency is for the number of function call levels to increase. This is true for all programming languages.

In the new function we have used the shift operators “>>” and “<<” to isolate each nibble. That way we can print each nibble separately making the binary format much easier to read.

The other thing that you notice is that the variable “i” was changed to a “long”. A long is a 4 byte variable. This was done in order for “i” to be able to hold the larger values required in the test condition for the “if” control structures. If you change it to a “word” (two bytes) then the value will overflow. The program will crash and burn. Note that we used the compound operator “*=” to increase the value of “i” by a factor of 10.



Now that we have our nifty new print function we want to take a look at two more operators. They are the bitwise “or” and the bitwise “and” operators. The first thing that happens is we run into the Arduino’s binary input limit again but this time we know how to deal with that.

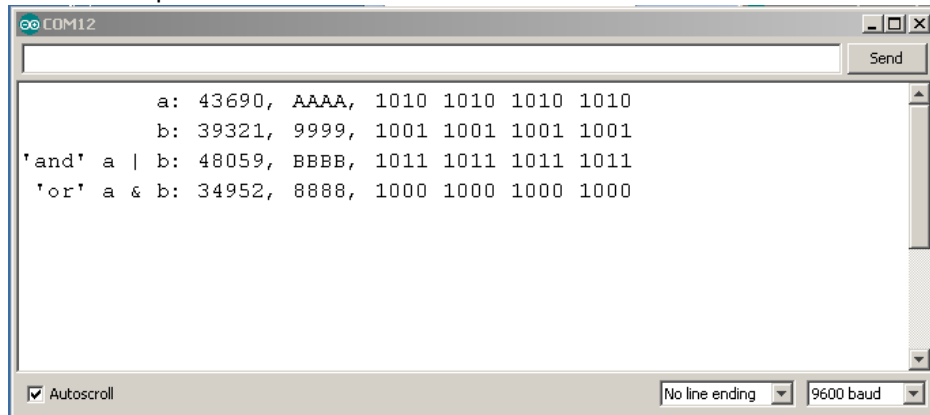
```

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  a= (B10101010 << 8) + B10101010;
  b= (B10011001 << 8) + B10011001;

  printdwb ("      a: ", a );
  printdwb ("      b: ", b );
  printdwb (" 'and' a | b: ", a | b);
  printdwb (" 'or' a & b: ", a & b);
}

```

The first thing to note is the use of the “<<” operator to get around the limitations of the Arduino’s IDE/compiler. The last two lines have our new operators in them. The “or” operator “|” (*that is a vertical bar not a capital I*) returns a one for each bit position where either number has a one. The “and” operator “&” returns a one for each bit position where both number have a one.



```

a: 43690, AAAA, 1010 1010 1010 1010
b: 39321, 9999, 1001 1001 1001 1001
'and' a | b: 48059, BBBB, 1011 1011 1011 1011
'or' a & b: 34952, 8888, 1000 1000 1000 1000
  
```

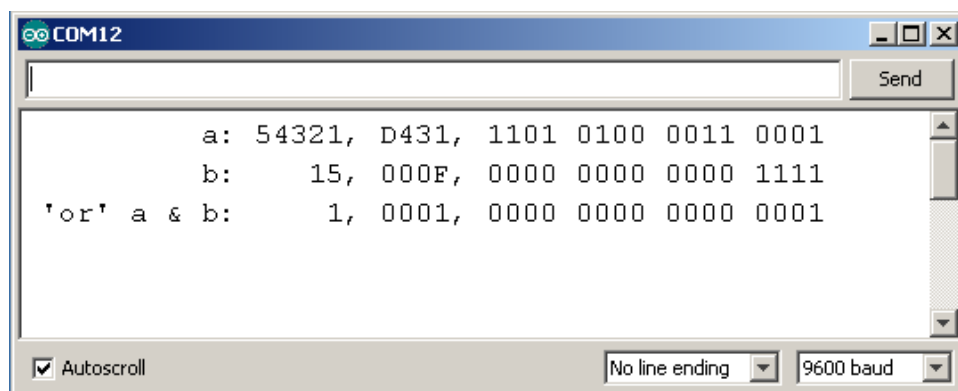
The numbers that we have used represent every possible combination of one and zero between the two numbers. The “and” operator is frequently used to isolate a portion of a number. For example a place where we only want the lower nibble.

```

void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  a= 54321;           // we want the lowest nibble of the word
  b= B1111;          // this is our bit mask

  printdwb ("      a: ", a );
  printdwb ("      b: ", b );

  printdwb (" 'or' a & b: ", a & b);
}
  
```



```

a: 54321, D431, 1101 0100 0011 0001
b:      15, 000F, 0000 0000 0000 1111
'or' a & b:      1, 0001, 0000 0000 0000 0001
  
```

This could be used to simplify our “printdwb” function but I will leave that up to you. The last thing we are going to look at is the actual “.cpp” file generated by the Arduino IDE to send to the compiler. That is shown below. Note that the two file have been combined into one with the main file first. The lines highlighted in yellow tell the compiler what line of what file the included lines come from. The compiler uses this information to return the correct location of any errors it encounters. The lines highlighted in green are called “forward declarations”. These tell the compiler what are functions are defined in the rest of the file.

```

#line 1 "Bits_and_Bytes03.ino"
/* Bits, Bytes and Nibbles */

#include "Arduino.h"
void setup ();
  
```

```

void loop();
void printdhhb (char c[], word n);
void printbin (byte v);
#line 3
void setup ()
{
  Serial.begin (9600);
  word a,b,c;
  a= (B10101010 << 8) + B10101010;
  b= (B10011001 << 8) + B10011001;

  printdhhb ("      a: ", a );
  printdhhb ("      b: ", b );
  printdhhb ("and' a | b: ", a | b);
  printdhhb (" 'or' a & b: ", a & b);
}

void loop() {
} // do nothing

#line 1 "functions.ino"
void printdhhb (char c[], word n)           // function to print value as
{ byte s;                                   // Decimal, Hexidecimal and binary
  long i;

  Serial.print (c);                         // print incoming string
  for (i=10, s=0; i<=10000; i *= 10)        // pad decimal value with spaces
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print (' ');}
  Serial.print (n, DEC);                     // print decimal value

  Serial.print (" ");                       // print seperator
  for (i=0x10, s=0; i<=0x1000; i *= 0x10)   // pad hex value with zeroes
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print ('0');}
  Serial.print (n, HEX);                     // print hex value

  Serial.print (" ");                       // print seperator
  /* the Arduino IDE/compiler has a problem with the binary format values larger than a byte
     so we have to do this a byte at a time */
  printbin(highByte(n));                    // call function to print high byte
  Serial.print (' ');                       // print seperator
  printbin(lowByte(n));                     // call function to print low byte
  Serial.println ();                        // print a new line
}

void printbin (byte v)                      // function to pad and print one binary byte
{ word i;
  byte s;
  byte n;                                  // used for a nibble
  n= v >> 4;                               // move high nibble into low nibble position
  for (i= B10, s=0; i <= B1000; i *= B10) // pad binary value with zeroes
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print ('0');}
  Serial.print (n, BIN);                    // print binary value

  Serial.print (' ');

  n= v << 4;                               // roll the high bits off to the left
  n= n >> 4;                               // move the left over bits back to the right
  for (i= B10, s=0; i <= B1000; i *= B10) // pad binary value with zeroes
  { if (n<i) { s++;}
  }
  i=0;
  while (i++ < s) { Serial.print ('0');}
  Serial.print (n, BIN);                    // print binary value
}

```

Special Numbers (true or false?)

There are constants that are defined that are not really numbers but they always have a value. The keyword **"false"** is defined to be equal to zero. Most often the keyword **"true"** will be set equal to one or minus one but it may be any number other than zero. Many times these two special numbers are used with an associated data type named "boolean".

If you set a variable equal to **"false"** then that variable will be evaluated as zero. However if you set a variable equal to **"true"** then the value returned from evaluating the variable should not be considered predictable (*actually for a specific system it will be ... but that is not something to depend on*).

These conditional phrases are equivalent:

```
boolean grape;
...
if (grape == false)
if (grape == 0)
```

As are these:

```
boolean berry;
...
if (berry == true)
if (berry != 0)
```

```
/* sample code */
boolean grape = true;
boolean berry = false;
...

if (grape)          makeJuice ();           // will execute make grape juice
if (berry)          makeJelly ();           // will not execute make berry jelly

if (grape == true)  makeJuice ();           // will execute make grape juice
if (berry == true)  makeJelly ();           // will not execute make berry jelly

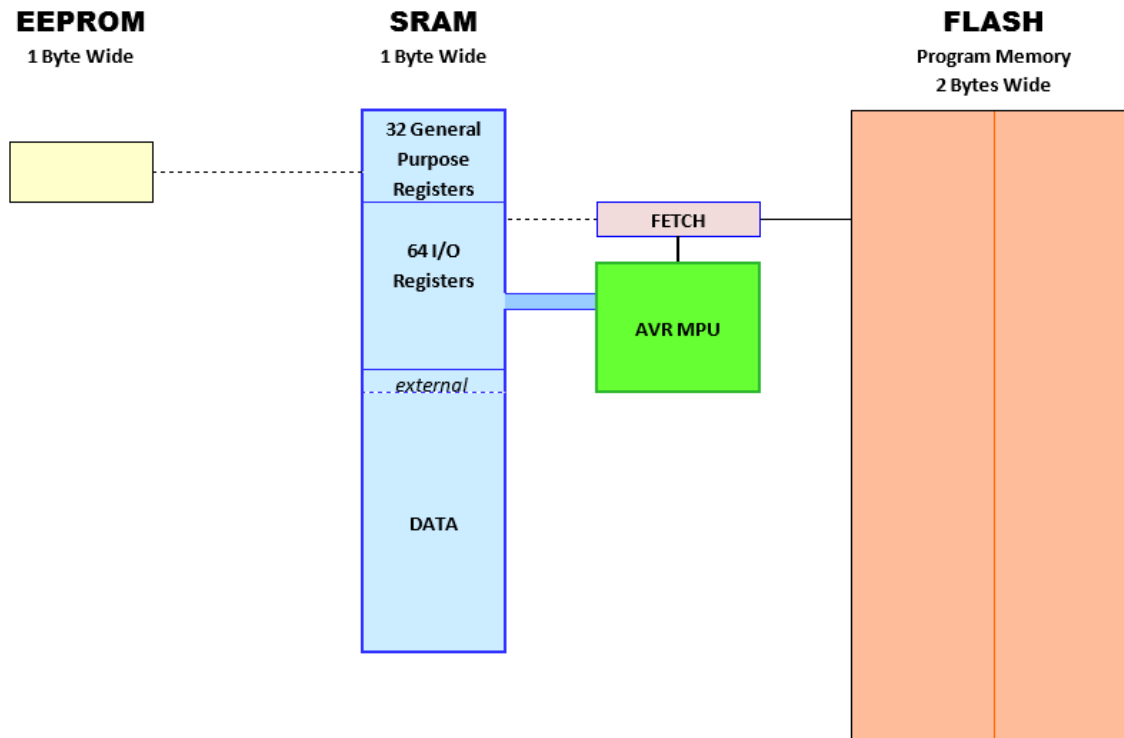
if (grape != false) makeJuice ();           // will execute make grape juice
if (berry != false) makeJelly ();           // will not execute make berry jelly

//-----
if (grape == false) makeJuice ();           // will not execute make grape juice
if (berry == false) makeJelly ();           // will execute make berry jelly

if (grape != true)  makeJuice ();           // will not execute make grape juice
if (berry != true)  makeJelly ();           // will execute make berry jelly
```

Memory: FLASH, SRAM, EEPROM

The Nano comes with three type of memory that have different properties and functions:



EEPROM:	1 K for the ATmega328	(1/2 K for the ATmega168)
SRAM:	2 K for the ATmega328	(1 K for the ATmega168)
FLASH:	32 K for the ATmega328	(16 K for the ATmega168)

EEPROM retains its values when power is shut off and even when a new sketch is uploaded but access is slow. The AVR core actually has to be shut down to access EEPROM (*2 clock cycles for write, 4 clock cycles for read*). Thus a special function is used in order to read or write the data in the EEPROM. It is good for about 100,000 write cycles. Well written software will compare the value to be written to value that is already stored there. If the two are the same then it does not overwrite the value. This is the location to store values that may change but need to be retained when the device is rebooted. An example of such data would be a calibration factor for the internal temperature sensor, the location of a remote sensor or a data log.

FLASH retains its data until it is overwritten by a special piece of software (*boot loader*) or hardware (*programmer*). It is good for about 10,000 write cycles. Our sketches can read from the FLASH memory but they cannot write to it. This is where the code for the sketches is stored when they are uploaded. A special “**FETCH**” mechanism moves program code from the FLASH memory to the microprocessor core. One of the registers in the SRAM address space is used as a “program pointer” to tell the **FETCH** mechanism which bytes to transfer.

SRAM loses its values whenever power is removed but it is very fast. There is not practical limit to the number of times that it can read and written. The AVR core has direct access to all the SRAM address space. This is where our variables and the programs stack are stored. What is a stack? It is a special section of memory used by the software to pass parameters between functions including the return address. It is also used for local variables defined within a function. It is possible to write a function that repeatedly calls itself (*that is called a recursive function*) and overfill the stack. That is called stack overflow and will most likely crash your Nano. Some of the AVR mpu’s have a provision for external RAM. In that case it is “mapped” into a reserved area of the SRAM space one section at a time. None of the Arduino boards (*that I have seen*) have this option.

All we have to do to read or write to the **SRAM** is declare a variable. Examples:

```
byte      MyByte   = 8;          \\ 8 bit unsigned number initialized to eight
word      MyWord   = 16;         \\ 16 bit unsigned number initialized to sixteen

char      Alpha    = 'A';        \\ 8 bit signed number initialized to sixty-five (A = ASCII 65)
char      MyChar   = 0;          \\ 8 bit signed number initialized to zero
unsigned char MyUnChar = MyByte;  \\ same as byte
int        MyInt    = 1;         \\ 16 bit signed number (-32,768 to 32,767)
unsigned int MyUnInt = 1;        \\ same as word
long       MyLong   = 2;         \\ 32 bit signed number (-2,147,483,648 to 2,147,483,647)
unsigned long MyUnLong = 2;      \\ 32 bit unsigned number (0 to 4,294,967,295)

float MyFloat = 10.01;           \\ 32 bit signed floating point number (-3.4028235E+38 to 3.4028235E+38)
/* double */                     \\ see float (just in case you were looking for double)

char      MyString[] = "Hello"; \\ string are arrays of the type character
```

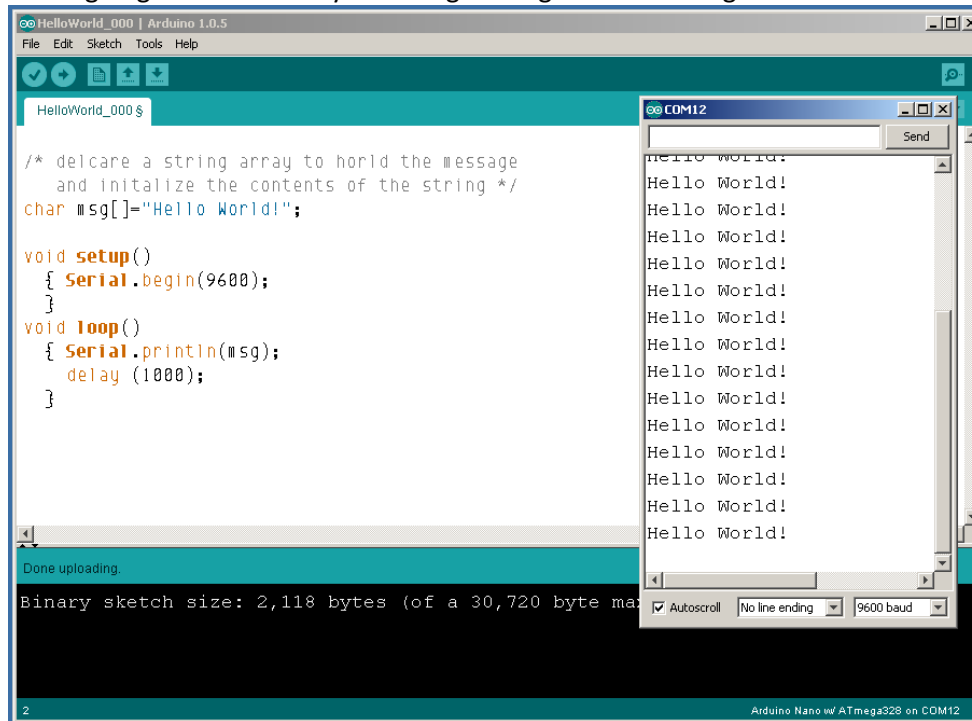
FLASH is a bit more of a challenge. In theory if you declare a constant then the preprocessor/compiler should be able to calculate and embed the value in line with the machine code. I am however told that the value gets copied to RAM. We will ignore that for the moment as it really does not affect us at the level we are working. There is however another method of storing and retrieving data in FLASH using a set of special functions. This is best utilized for such things as a data table or group of strings where you are going to look up a specific value or string. That saves you from having to place the entire table in the valuable and scarce SRAM space.

EEPROM is much the same as FLASH in that we need to use a special set of functions to read and write it. If we use multiple sketches with the same Arduino board then we also need a bit of upfront planning so that we do not overwrite valuable data that another sketch needs. We may also want several sketches to be able to access the same data. This can be done by including a file in our sketch that holds the declarations for the locations in EEPROM.

We are going to expand the “Hello World” program with versions that use each form of memory.

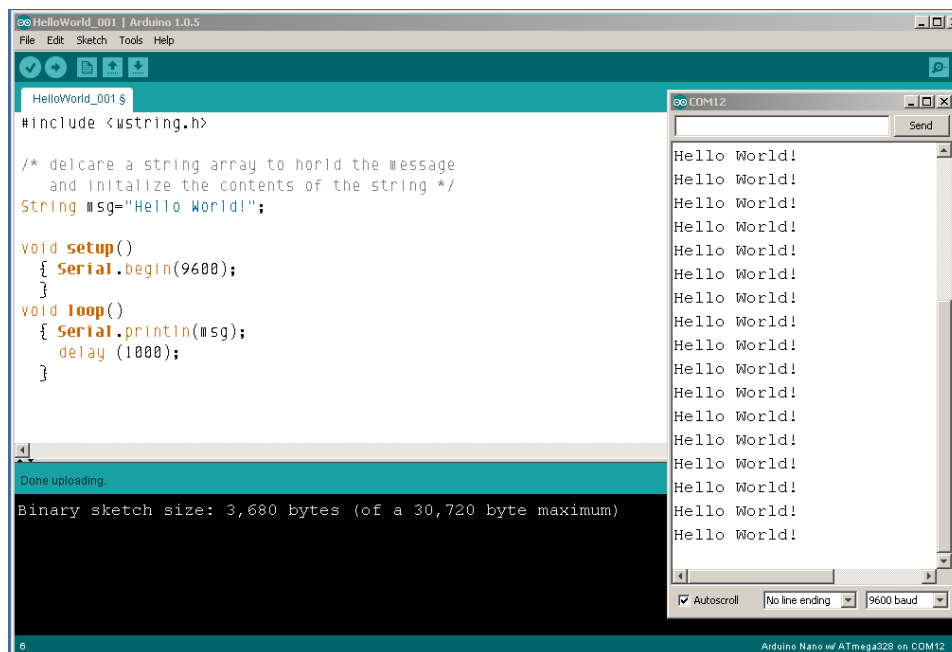
SRAM: Hello Word 001/002

In this version we are going to use SRAM by declaring a string for our message.



Notice that we did not specify the size of the array. The compiler will figure that out for us. The other thing to notice is that this version and the previous version came out the very same size: 2,118 bytes.

Another way to do this is to use the string library that comes with Arduino IDE. We use the “#include” statement to do that.



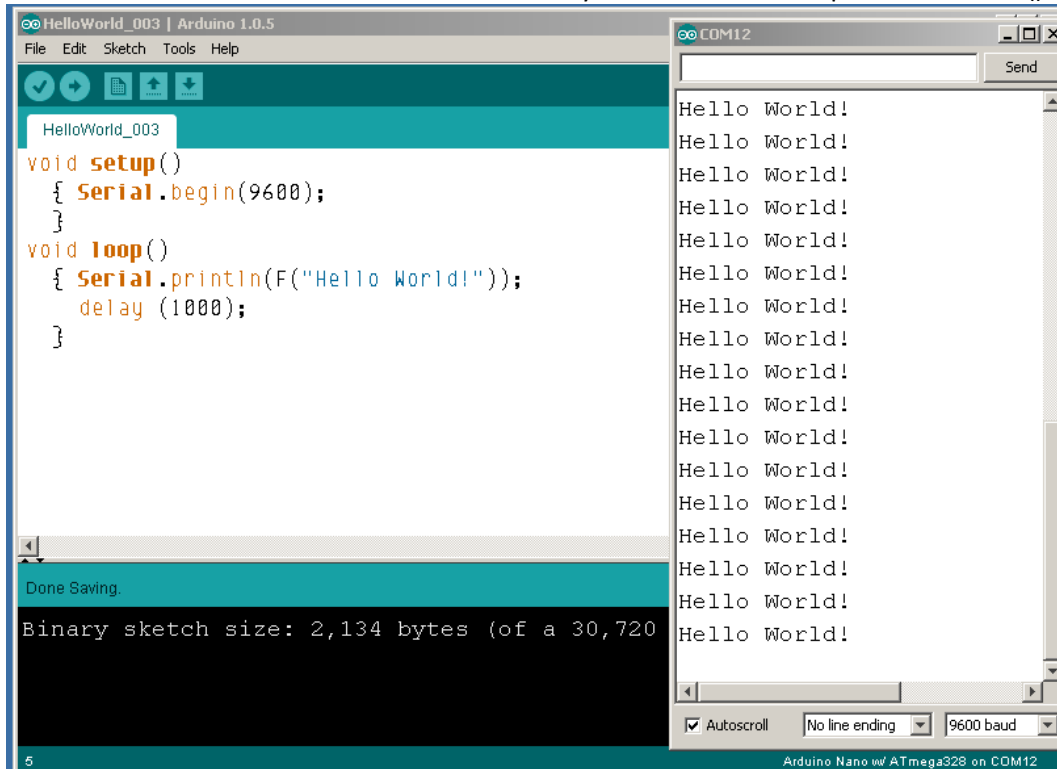
Wstring.h is found in the directory:

<your program path>\Arduino\hardware\arduino\cores\arduino

This provides a new variable type "String". There are additional functions added to manipulate the string such concatenation and compare but note what that costs us. The size of the program is now 3,680 bytes. The Arduino is probably not the optimum location to be manipulating strings but it is good to know we have that capability if needed.

FLASH: Hello Word 003/004

Flash memory (*or program memory*) is where we want to store most of our data because it is the largest area we have to work with. In the case of Hello World the easiest way to that is with the special function F().



There seems to be some controversy about the value of using this function because the Arduino has to copy the string variable from FLASH to SRAM in order for the serial print routine to print it. This is however considered the 'proper' method of printing inline literals. Quote from Mathew Ford of Forward Computing and Control Pty. Ltd.: <http://www.forward.com.au/pfod/ArduinoProgramming/index.html>

"The **F()** syntax just puts the string in Program memory (*Flash*) and casts the resulting point to a unique class. This unique class insures that the correct **print()** method is called to read the bytes from the program memory and **write()** them. Any class that inherits from **Print** can use this approach.

That includes **Server**, **Stream**, **Client**, **HardwareSerial**, **SoftwareSerial** and **UDP**."

Well that is cool but it is not much help if we want to manipulate the string or if we want to use the same string several places in the program. In order to do that we have to use functions from "PROGMEM" that are found in "<your program path>\avr\pgmspace.h". This is a collection of functions to store and recall data from flash memory. These are defined in `avr/pgmspace.h`. If you look at some of the examples that are on line then be careful that they use the latest version: the proper syntax has changed. The current syntax is documented at the URL:

http://www.nongnu.org/avr-libc/user-manual/group_avr_pgmspace.html

Before we go on we want to make two small additions to our program. First we will add a description of the program at the top. Then we will make the Nano wait for us to talk to it.

```

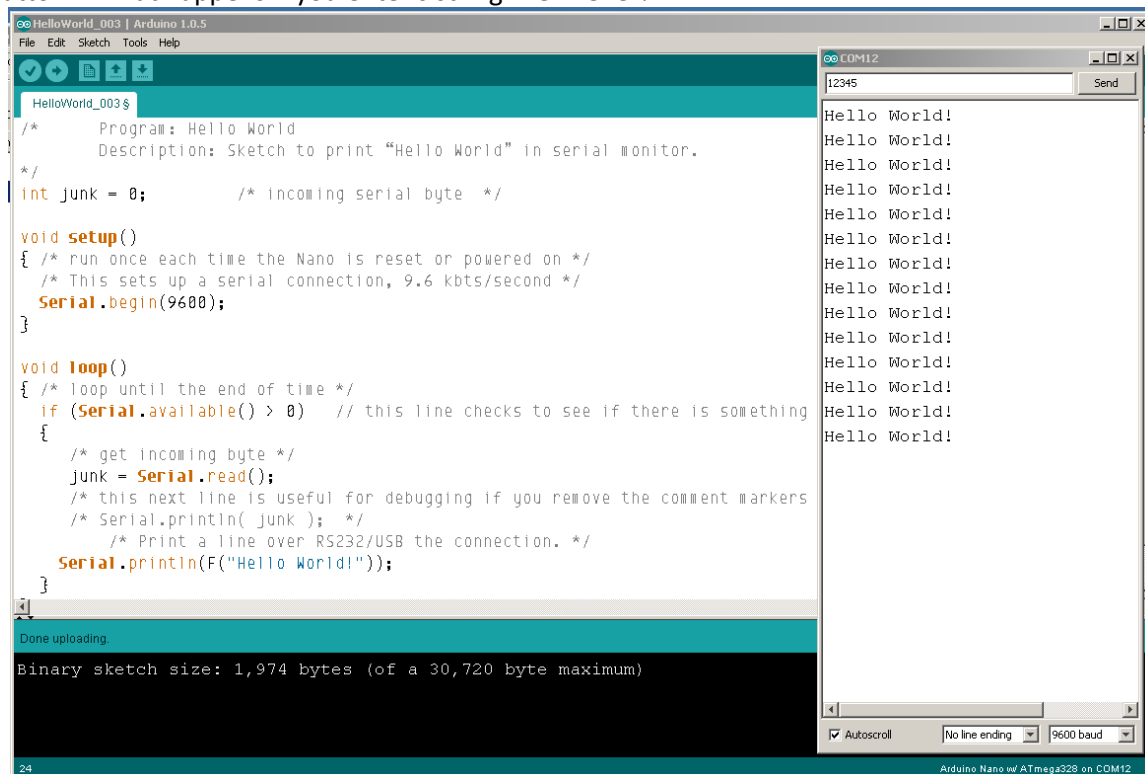
/* Program: Hello World
   Description: Sketch to print "Hello World" in serial monitor.
*/
int junk = 0;          /* incoming serial byte */

void setup()
{ /* run once each time the Nano is reset or powered on */
  /* This sets up a serial connection, 9.6 kbps/second */
  Serial.begin(9600);
}

void loop()
{ /* loop until the end of time */
  if (Serial.available() > 0) // this line checks to see if there is something to read
  {
    /* get incoming byte */
    junk = Serial.read();
    /* this next line is useful for debugging if you remove the comment markers */
    /* Serial.println( junk ); */
    /* Print a line over RS232/USB the connection. */
    Serial.println(F("Hello World!"));
  }
}

```

Upload the new code. Then switch to the monitor window and type a single character followed by selecting the send "button". What happens if you enter a string like 12345 ?



PROGMEM is really most useful for tables. So our program is going to build a table in four languages. It will then inquire which language to use. Please read the comments in the following code for details.

```

/* Program: Hello World in four languages using PROGMEM */

//---declarations-----

```

```

/* include library to use flash memory (PROGMEM) for fixed data storage */
#include <avr/pgmspace.h>

/* this is where we define our PROGMEM string */
char HelpStr[] PROGMEM = "E=English, F=French, S=Spanish, I=Italian, A=All";
char EnglishGreetingStr[] PROGMEM = "Hello World";
char FrenchGreetingStr[] PROGMEM = "Bonjour tout le monde";
char SpanishGreetingStr[] PROGMEM = "Hola mundo";
char ItalianGreetingStr[] PROGMEM = "Ciao mondo";

/* Here we set up an enumerated array of constants.
   This is how all the example code is shown. I was
   unable to find an example anywhere that put strings
   in PROMEM did not use an array to access them. */
const char *Language[] PROGMEM =
{ EnglishGreetingStr,           // Language [0]
  FrenchGreetingStr,           // Language [1]
  SpanishGreetingStr,          // Language [2]
  ItalianGreetingStr };        // Language [3]

/* Here we set up a separate constant name for each string.
   This may be the only example of this type syntax/usage. */
const char *English PROGMEM = {EnglishGreetingStr};
const char *French PROGMEM = {FrenchGreetingStr};
const char *Spanish PROGMEM = {SpanishGreetingStr};
const char *Italian PROGMEM = {ItalianGreetingStr};
const char *HelpMe PROGMEM = {HelpStr};

/* Unfortunately the software cannot use the strings in
   PROGMEM directly. We have to use a special function to
   transfer the strings from PROGMEM to SRAM so that they
   are in the same memory where the program runs. This
   sets an area in SRAM to work with the strings. */
char OutBuffer[60];

/* this byte is used to read from the RS232/USB port */
char junk;

//---setup-----
void setup()
{ /* run once */
  Serial.begin(9600); // This sets up a serial connection, 9.6 kbps/s.
  /* wait for serial port to connect. Needed for Leonardo only */
  while (!Serial) { ; }
  /* We are going to prompt the user for input.
     Note that we are using the F() function here to wrap the
     "string literal" */
  Serial.println(F("Enter: E, F, S, I or A"));
  Serial.println("");
}

//---main-----
void loop()
{ /* loop until the end of time */
  if (Serial.available() > 0)
  {
    /* get incoming byte */
    junk = Serial.read();
    /* this next line is useful for debugging if you remove the comment markers */
    /* Serial.println( junk ); */
    /* now that we have a character from the user we have to decide what
       to do. First off we are really particular. We will only accept
       1 of the 5 characters we asked for to begin with. If the character
       is not in that group we are going to send the user another message
       to explain the situation. That is the string tha we defined as
    */
  }
}

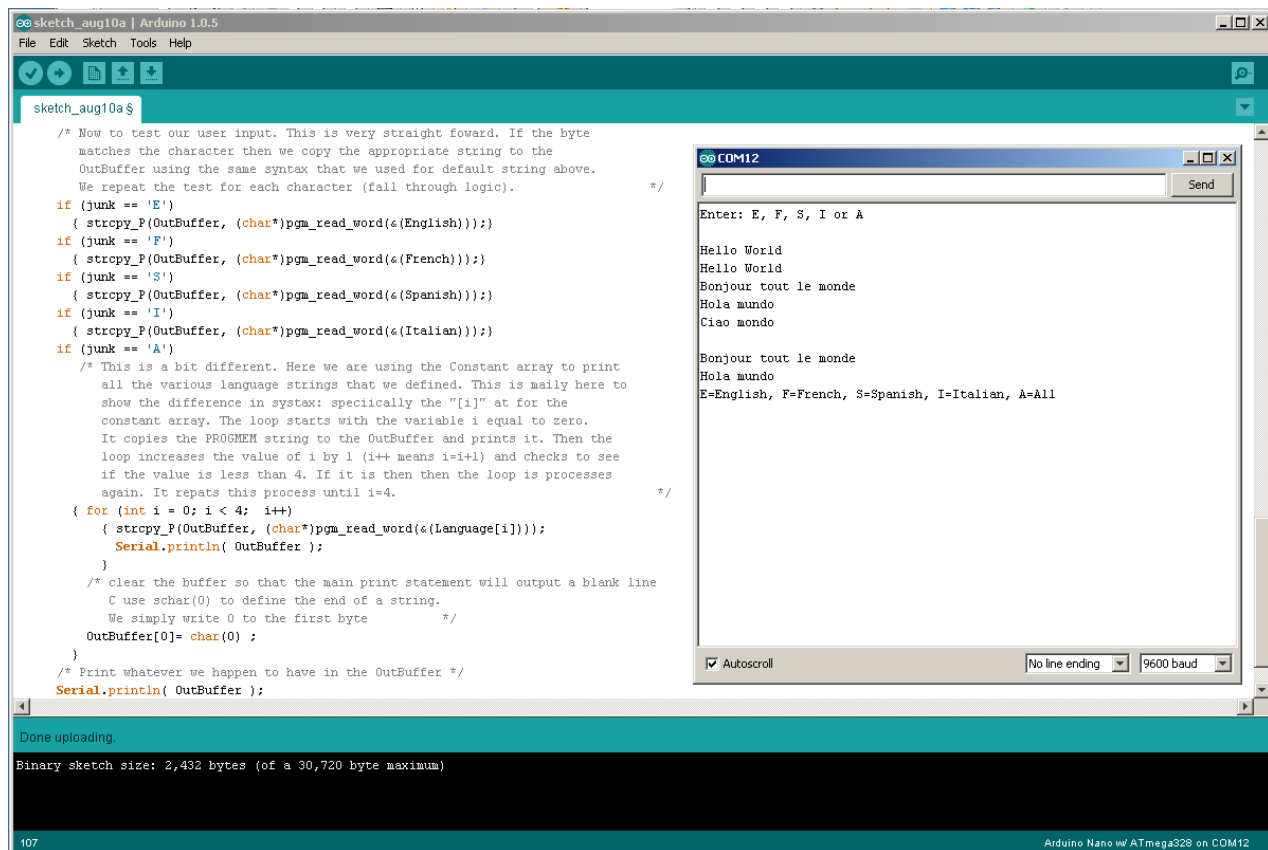
```

```

    "HelpMe" in our declarations. So the first thing that we do is copy
    that string to the OutBuffer using the special PROGMEM string copy
    function. I am not going to try to explain the syntax because I
    do not understand it myself. The important parts to know are:
    1) strcpy_P() is the special function to copy a string from PROGMEM
    2) the first parameter (in this case "OutBuffer") is the SRAM array
       to copy the string to.
    3) the last parameter (in this case "HelpMe") is the constant that
       points to the string that we wish to retrieve.
    */
    strcpy_P(OutBuffer, (char*)pgm_read_word(&(HelpMe)));

    /* Now to test our user input. This is very straight forward. If the byte
       matches the character then we copy the appropriate string to the
       OutBuffer using the same syntax that we used for default string above.
       We repeat the test for each character (fall through logic).
    */
    if (junk == 'E')
    { strcpy_P(OutBuffer, (char*)pgm_read_word(&(English)));}
    if (junk == 'F')
    { strcpy_P(OutBuffer, (char*)pgm_read_word(&(French)));}
    if (junk == 'S')
    { strcpy_P(OutBuffer, (char*)pgm_read_word(&(Spanish)));}
    if (junk == 'I')
    { strcpy_P(OutBuffer, (char*)pgm_read_word(&(Italian)));}
    if (junk == 'A')
    /* This is a bit different. Here we are using the Constant array to print
       all the various language strings that we defined. This is mainly here to
       show the difference in syntax: specifically the "[i]" at for the
       constant array. The loop starts with the variable i equal to zero.
       It copies the PROGMEM string to the OutBuffer and prints it. Then the
       loop increases the value of i by 1 (i++ means i=i+1) and checks to see
       if the value is less than 4. If it is then then the loop is processes
       again. It repeats this process until i=4.
    */
    { for (int i = 0; i < 4; i++)
        { strcpy_P(OutBuffer, (char*)pgm_read_word(&(Language[i])));
          Serial.println (OutBuffer);
        }
        /* clear the buffer so that the main print statement will output a blank line
           C use char(0) to define the end of a string.
           We simply write 0 to the first byte
        */
        OutBuffer[0]= char(0) ;
    }
    /* Print whatever we happen to have in the OutBuffer */
    Serial.println (OutBuffer);
}
}

```



EEPROM: Hello Word 005/006 (Write, Read)

First of all we need to do a bit of planning. We need to define the addresses of the data that we are going to store in the EEPROM. Think of it as a hotel with a lot of rooms. The hotel guests are data bytes. We need to be sure that we send parties of guests to the right suite of rooms and that the rooms are large enough to accommodate the number of guests in each party. So the first question is “How many guests can we accommodate?” The Arduino headers include ‘<your program path>\Arduino\hardware\tools\avr\avr\include\avr\io.h’ that defines symbols for each board. This file is actually more like an index that points to the appropriate file for the specific microprocessor.

```

191 #elif defined (__AVR_ATmega3250__)
192 # include <avr/iom3250.h>
193 #elif defined (__AVR_ATmega3250P__)
194 # include <avr/iom3250.h>
195 #elif defined (__AVR_ATmega328P__)
196 # include <avr/iom328p.h>
197 #elif defined (__AVR_ATmega329__)
198 # include <avr/iom329.h>
199 #elif defined (__AVR_ATmega329P__)
200 # include <avr/iom329.h>
201 #elif defined (__AVR_ATmega3290__)

```

For our CPU in the Nano is says to include the file ‘io328p.h’. If we look at that file then we find the symbol ‘E2END’.

```

/* Constants */
#define SPM_PAGESIZE 128
#define RAMEND      0x8FFF /* Last On-Chip SRAM Location */
#define XRAMSIZE     0
#define XRAMEND      (RAMEND + XRAMSIZE)
#define E2END        0x3FFF
#define E2PAGESIZE   4
#define FLASHEND     0x7FFF

```

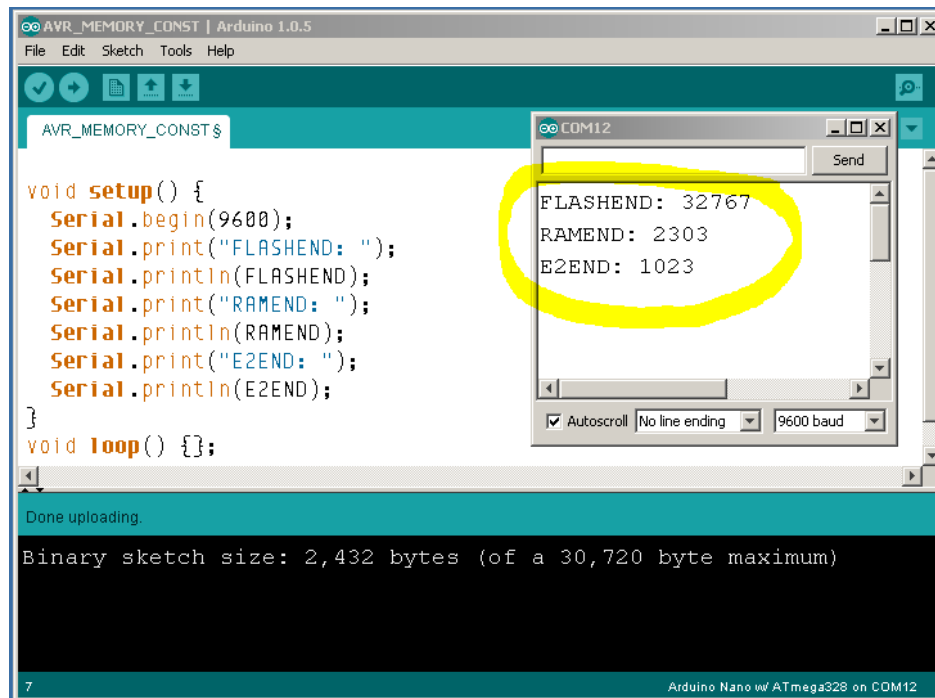
FLASHEND: The last byte address in the Flash program space.
RAMEND: The last on-chip RAM address.
E2END: The last EEPROM address.

Here is a simple program to try but the results may surprise you.

```

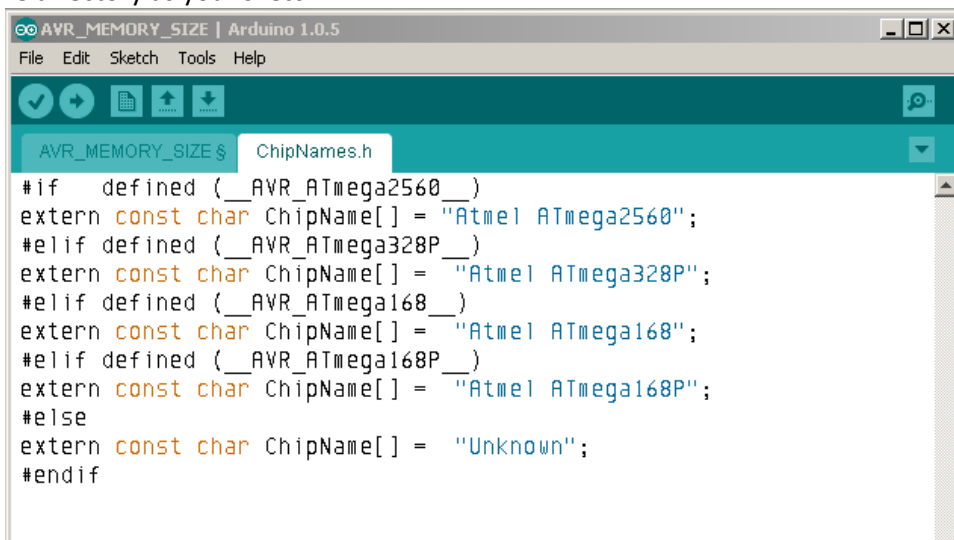
void setup() {
  Serial.begin(9600);
  Serial.print("FLASHEND: ");
  Serial.println(FLASHEND);
  Serial.print("RAMEND: ");
  Serial.println(RAMEND);
  Serial.print("E2END: ");
  Serial.println(E2END);
}
void loop() {} // do nothing

```



If you are sharp then you are asking “Hey! Should not the number for RAMEND be 2047 or 2048 ???”

Well yes and no. The address space that the SRAM lives in is also used for CPU registers and IO mapping. So the first 255 addresses are ‘reserved’: $2303 - 255 = 2047$. Let’s try something a little different. This program will use two files. Open up the Arduino IDE and created new sketch named “AVR_MEMORY_SIZE” and save it. Then create a second file named “ChipNames.h”. Enter this code in the new window and then save the file. It will be saved in the same directory as your sketch.



What we are doing here is to take advantage of some more of the information that is stored in the system header files. In this case we are setting a constant for a proper name for our processor. Statements beginning with the pound sign “#” (*I am American. If I had been British I would have used the word “hash”*) are directives to a special program called the pre-compiler. It runs before the compiler and sets our constant value at compile-time. We could have done this in the main code using if or case statements. In that case the proper name would be determined at “runtime” (*when the program executes*). However each compile is targeted for a particular

processor so we can determine the proper value at compile time. This is more efficient because in theory the program code runs many times but the final compile only has to run once. We will take advantage of a few more defines below. Now go back to the first windows and enter this program code.

```
/* get the size of the AVR RAM areas */
#include "ChipNames.h" // include our other file.

float TempVal = 0;

void setup() {
  Serial.begin (9600);

  /* print chip name */
  Serial.print("Microprocessor type: "); //---- Put proper designation in here ----
  Serial.println(ChipName);

  /* speed code taken from http://playground.arduino.cc/Main/ShowInfo */
  Serial.print(F("Speed = "));
  Serial.print(F_CPU / 1000000, DEC);
  Serial.println(F(" MHz"));

  /* get Flash RAM size */
  TempVal= FLASHEND;
  TempVal= ((TempVal+1)/1024);
  Serial.print("Size of FLASH(using FLASHEND): ");
  Serial.print(TempVal);
  Serial.println("K");

  /* get SRAM size */
  TempVal= (RAMEND -255); // sram begin at 256
  TempVal= ((TempVal+1)/1024);
  Serial.print("Size of SRAM (using RAMEND): ");
  Serial.print(TempVal);
  Serial.println("K");

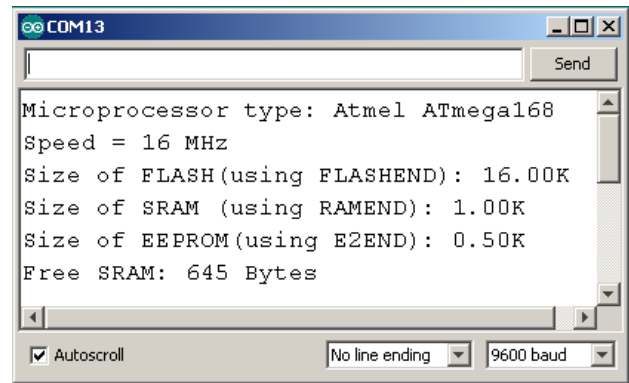
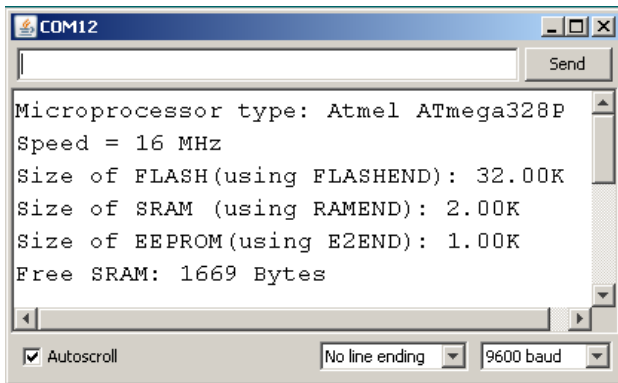
  /* get EEPROM size */
  TempVal= (E2END);
  TempVal= ((TempVal+1)/1024);
  Serial.print("Size of EEPROM(using E2END): ");
  Serial.print(TempVal);
  Serial.println("K");

  /* get free (unallocated) SRAM */
  Serial.print("Free SRAM: ");
  Serial.print(freeRam());
  Serial.println(" Bytes");
}

void loop() {}

/* http://stackoverflow.com/questions/960389/how-can-i-visualise-the-memory-sram-usage-of-an-avr-program */
int freeRam () {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

I ran this program on two different Arduino boards to show the what happens when one has a different AVR chip.



Now that we know how many rooms we have we need to check our reservations:

```

      1      2      3      4      5
12345678901234567890123456789012345678901234567890
HelpStr, "E=English, F=French, S=Spanish, I=Italian, A=All", party of 48
English, "Hello World", party of 11
French, "Bonjour tout le monde", party of 21
Spanish, "Hola mundo", party of 10
Italian, "Ciao mondo", party of 10

```

Each of parties will also have a hotel employee to assist them: null (*ASCII 0 to terminate the string*). Addressing begins with zero (0) and our rooms just happen to be sized in increments of four (4, *an arbitrary choice by the accounting department*). The HelpStr is going to get the deluxe suite on the ground floor that will accommodate 52. The rest will get each get rooms less than half that size: 24. This will however be adequate room to allow the parties to change their guest list a bit without inconveniencing the other guests. To be done properly we should put this in a separate file that will be shared between our two programs. But for the moment we shall just duplicate it in each sketch. So our first EEPROM program is going to be named "Hello Word 005". All it is going to do is check to the guests into their rooms (*write the data to the EEPROM*).

```
HelloWorld_005 $
/*
 * Program: Hello World 005
 * This program stores our Greeting strings in EEPROM.
 * These values will remain when the board is turned off.  */

#include <EEPROM.h>
/* Addresses in EEPROM */
const word HelpStr = 00; /* EEPROM location 0000 through 0051 */
const word English = 52; /* EEPROM location 0052 through 0075 */
const word French = 76; /* EEPROM location 0076 through 0099 */
const word Spanish = 100; /* EEPROM location 0100 through 0023 */
const word Italian = 124; /* EEPROM location 0124 through 0147 */
const word TheEnd = 148; /* the end of our used EEPROM area */

void setup()
{
  Serial.begin(9600);
  storestring (HelpStr, English, "E=English, F=French, S=Spanish, I=Italian, A=All\0");
  storestring (English, French, "Hello World\0");
  storestring (French, Spanish, "Bonjour tout le monde\0");
  storestring (Spanish, Italian, "Hola mundo\0");
  storestring (Italian, TheEnd, "Ciao mondo\0");
  // the "\0" is an "escape sequence" that tells the compiler we mean char(0)
  // which we could not normally type in a string. The compiler should terminate
  // the string for us ... I am just being extra careful (belts and suspenders).
}

void loop() { } // do nothing

void storestring (word addrstart, word addrend, char greeting[])
{
  byte chrcount = 0; // this for the index into our char array
  byte singlechar=0x01; // holds a single character from the array
  word addr=addrstart; // this is the address that will be written
  // continue until come to char(0)
  while ((singlechar != 0) && (addr != addrend)) // or we reach the end of the allotted space
  {
    singlechar= greeting[chrcount++]; // get one character
    EEPROM.write(addr++, singlechar); // write it to the EEPROM
    if (singlechar !=0) Serial.print (char(singlechar)); // this is for debugging
    if (singlechar ==0) Serial.print ("\0"); // "\0" will print "\0"
    delay(10); // allow time for EEPROM write to complete
  }
  Serial.println(""); // print a line feed
}
```

There are a lot of things to notice here. The `#include` line uses the `<>` delimiters (less than, greater than). This is the “C” convention for a system supplied library. The convention for user files/libraries is for `"` delimiters (double quotes). The Arduino IDE may look for the files in different locations according to the delimiters used.

In the `setup` function we have put a `“\0”` at the end of each of our strings. This is an “escape sequence” that tells the compiler we want an [ASCII](#) zero there. We cannot normally type that character because it is a non-printing control character. You are likely to see some used in “C” code:

<code>\0</code>	=	Null	ASCII zero (0)
<code>\n</code>	=	New Line	ASCII ten (10)
<code>\r</code>	=	Carriage Return	ASCII thirteen (13)
<code>\t</code>	=	Tab	ASCII nine (9)
<code>\e</code>	=	Escape	ASCII twenty-seven (27)
<code>\\</code>	=	Backslash	ASCII ninety-two (92)

We created our first function in this program. Note that we are also passing it parameters and how the parameters are declared. We also declared three variables within the function: `chrcount`, `singlechar` and `addr`. These variables only exist when the function is executing. That saves us valuable SRAM space. The special function that we used was `EEPROM.write`. This is part of a class library. The class is “EEPROM” and the function is “write”. You should notice the period between the class and the function (*remember that “C” is case sensitive*). This function writes a single byte at a time.

Also notice the “`while() {}`” loop structure. We repeat this section of code while the conditions are true. The two conditions are that our current character is not ASCII 0 and we have not reached the end of the allocated address space. The “`!=`” (*exclamation point and equal sign*) is the “C” comparison operator for “not equal”. The “`&&`” is the “C” logical operator for “and” (*both conditions must be true*). In order to enter the loop we initialize the “singlecharacter” variable to a number other than zero. Remember that “`0x01`” is the “C” hex notation for one (1). “`0x0F`” would be decimal fifteen (15). A bit further down “`==`” is the “C” comparison operator for “equal”. We also used the escape sequence “`\\`” to print a backslash.

Our variables “chrcount” and “addr” are incremented by the “C” compound operator “`++`”. The placement of this operator can be very important if you use it inside a function call such as “`greeting(chrcount++)`”. In this case the “`++`” operator follows the variable so that it returns the current value before it increments the value of the variable. When the “`++`” operator proceeds the variable then it increments the value before returning the value to the function.

Thus:

```
singlecharacter = greeting(chrcount++);
```

is the same as

```
singlecharacter = greeting(chrcount);  
chrcount = chrcount +1;
```

Conversely

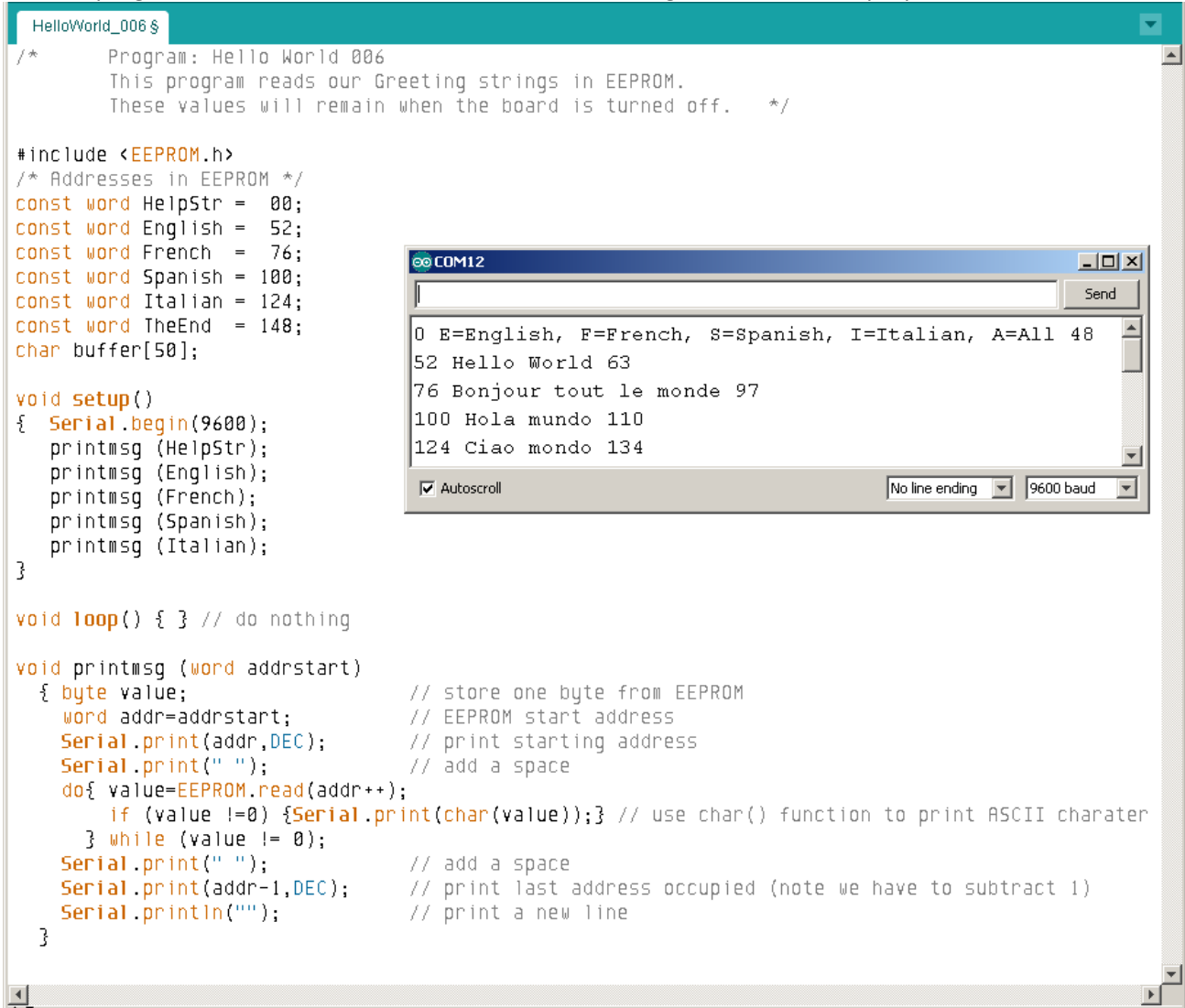
```
singlecharacter = greeting(++chrcount);
```

is the same as

```
chrcount = chrcount +1;  
singlecharacter = greeting(chrcount);
```

The biggest problem with this program is we will not know if the guests made it to their room until we write something to read the data back from the EEPROM.

This is the program HelloWorld006 that checks to see if all our guests are in their proper rooms.



The screenshot shows an IDE window titled 'HelloWorld_006 \$' containing the following C++ code:

```
/* Program: Hello World 006
   This program reads our Greeting strings in EEPROM.
   These values will remain when the board is turned off. */

#include <EEPROM.h>
/* Addresses in EEPROM */
const word HelpStr = 00;
const word English = 52;
const word French = 76;
const word Spanish = 100;
const word Italian = 124;
const word TheEnd = 148;
char buffer[50];

void setup()
{ Serial.begin(9600);
  printmsg (HelpStr);
  printmsg (English);
  printmsg (French);
  printmsg (Spanish);
  printmsg (Italian);
}

void loop() { } // do nothing

void printmsg (word addrstart)
{ byte value; // store one byte from EEPROM
  word addr=addrstart; // EEPROM start address
  Serial.print(addr,DEC); // print starting address
  Serial.print(" "); // add a space
  do{ value=EEPROM.read(addr++);
    if (value !=0) {Serial.print(char(value));} // use char() function to print ASCII character
  } while (value != 0);
  Serial.print(" "); // add a space
  Serial.print(addr-1,DEC); // print last address occupied (note we have to subtract 1)
  Serial.println(""); // print a new line
}
```

Overlaid on the code is a 'COM12' serial monitor window. It displays the following output:

```
0 E=English, F=French, S=Spanish, I=Italian, A=All 48
52 Hello World 63
76 Bonjour tout le monde 97
100 Hola mundo 110
124 Ciao mondo 134
```

The serial monitor window has 'Autoscroll' checked, 'No line ending' selected, and '9600 baud' set.

The special function that we use in this program is “[EEPROM.read](#)”. Just like its counterpart it reads a single byte at a time. For that reason we used the “[Serial.print](#)” function rather than the “[Serial.println](#)” we have been using. “[Serial.print](#)” does not print a new line so we can print a whole bunch of single bytes this way. Note also that we used an additional parameter in “[Serial.print\(addr, DEC\)](#)”. This tells the function to convert the value of address to the string representation of a decimal number before it prints it. In “[Serial.print\(char\(value\)\)](#)” we use [char](#) to convert the value that we retrieved to an ASCII character before printing it. That is called “casting”. Then we print the ending address but we have to subtract 1 because we have just added 1 to the address in our loop. Lastly we use [Serial.print\(“”\)](#) to send a new line.

Notice the structure of the loop that does the printing. This is the “other form” of the “[while](#)” loop (*the one we are not supposed to use*): “[do {} while\(\)](#)”. In this case the loop will always execute at least once before it gets to its condition check. It is thus possible to enter the loop when an invalid condition exists. Of course it also avoids the necessity of the variable “value” to something other than zero. From my perspective if you are certain that you want to execute the loop at least once then this form is at least acceptable (*not to mention that it also has a definitive end*).

Now that you have this program (*you did save it, yes?*) go back and upload the “blinky” program and run it. Then upload this program again and run it. You will see that all our guests are still in their rooms.

EEPROM: EEPROM_Dump, EEPROM_Erase

It would be nice to have a program to dump the contents of the EEPROM. We might be using an Arduino that someone had used on another project and we want to see if there is anything there that should be saved before we overwrite or erase it. For that matter we might even want to know if we need to erase it. First we need to do a little planning about what our print out is going to look.

0	1	2	3	4	5	6
12345678901234567890123456789012345678901234567801234567890						
XXXX	HLHLHLHL	HLHLHLHL	HLHLHLHL	HLHLHLHL	AAAAAAAAAAAAAAAA	

We know that our maximum address is decimal 1024 so we reserve the first four characters for the address represented by XXXX. Then we add two spaces. We are going to print 16 hex values (0-FF) in groups of four separated by a space (*HL stands for High nibble, Low nibble*). Follow that with two spaces and ASCII character for each of the bytes. That is a total of 46 characters but we need a terminating null as well. So make the buffer 48 characters long. A bit of quick arithmetic should tell you that will be 64 lines. That is less than one page if you capture and print it out. Another thing that we have to be careful of is special characters. Anything under decimal 32 is considered to be a non-printing character. There are also the special combinations of characters like “\t” or “\n”. We are going to take the simple approach and print a period for any value under 32 or equal to 92 or over 126 (*otherwise we have to get very creative*).

```

EEPROM_Dump $
/*      Program: EEPROM Dump
        This program prints the entire contents of the EEPROM
        in HEX and ASCII (when applicable) to the USB/RS232 terminal */
#include <EEPROM.h>
unsigned long addr;                // used for address in EEPROM
byte i,j,c;                        // i=1 to 16 bytes, j=HEX index, c= single character
char buffer[22];                  // char array for ASCII string

void setup()
{
  Serial.begin(9600);              // initialize serial port
  buffer[0] = ' ';                // put space at beginning of char array
  buffer[1] = ' ';                // put second space at beginning of char array
  buffer[18] = 0;                  // string termination for buffer
  buffer[19] = 0;                  // belts and suspenders
  Serial.println(F("EEPROM DUMP:")); // information, notice the F() wrapper
  do
  {
    /* Read 16 characters from the EEprom into our buffer */
    /* ----- Print the address padded with spaces ----- */
    if (addr<1000) {Serial.print(' ');} // space padding, char(32) is a space
    if (addr<100) {Serial.print(' ');} // space padding
    if (addr<10) {Serial.print(' ');} // space padding
    Serial.print(addr, DEC);          // starting address for line
    Serial.print(" ");              // seperator, could be replaced with a tab ("\t")
    for (i=0; i<16; ++i)             // 16 bytes at a time
    {
      c = EEPROM.read(addr + i);      // read one byte
      buffer[i+2]=char(c);            // stuff the character in our buffer
      /*--- print HEX value ---*/
      if (c < 0x10) {Serial.print('0');} // pad HEX value with 0, 0x10 = F0 = 16
      Serial.print(c, HEX);           // using HEX tells print how to format the number
      ++j;                            // increment index for seperator
      if (j == 4)                     // if we have printed 4 values
      {
        Serial.print(' ');          // print a seperator
        j=0;                         // set our index back to zero to start over
      }
      /*--- Evaluate character ---*/
      if (c < 32) {buffer[i+2]='.';} // do not print control codes
      if (c ==92) {buffer[i+2]='.';} // do not print "\"
      if (c >126) {buffer[i+2]='.';} // do not print 'extended' characters
    }
    Serial.println(buffer);           // begin wiht a new line
    addr=addr +16;
  } while (addr < E2END);             // E2END is the end of the EEPROM
}

void loop() { } // do nothing

```

Not much new here. Run the program and see what the output looks like.

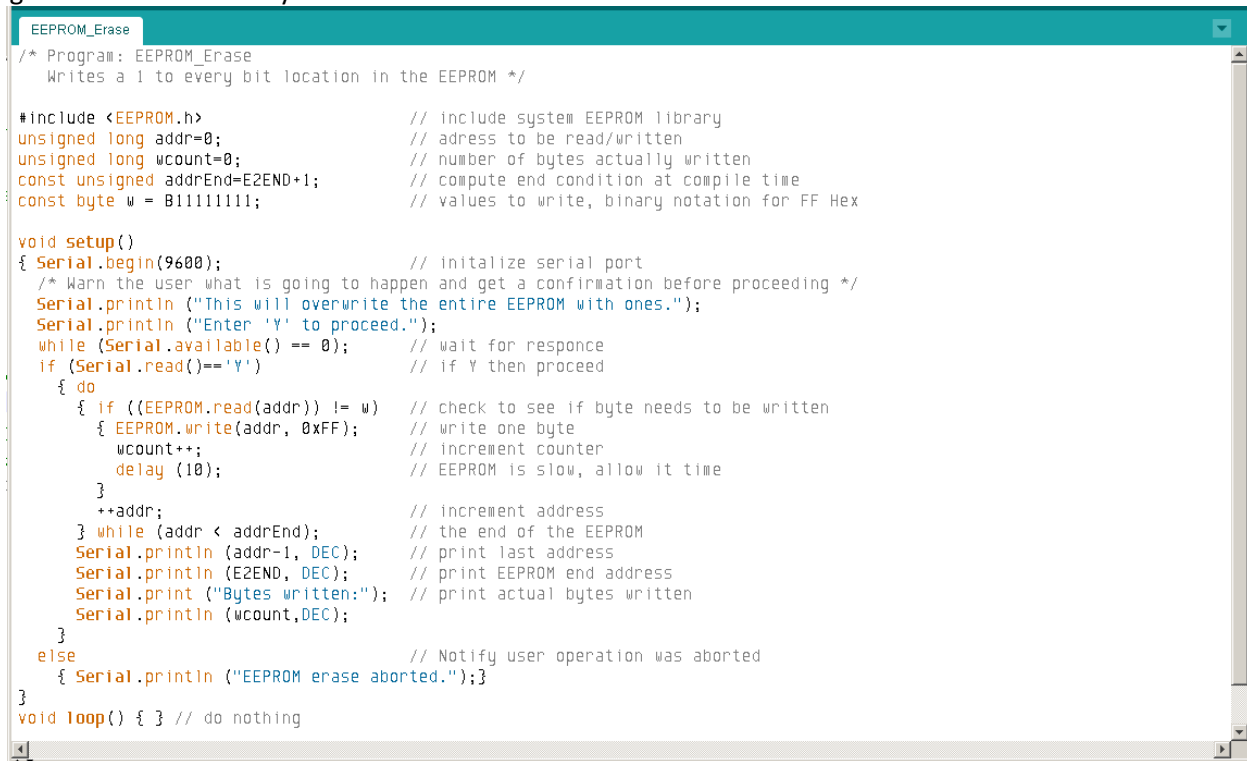
```

COM12
EEPROM DUMP:
 0 453D456E 676C6973 682C2046 3D467265 E=English, F=Fre
16 6E63682C 20533D53 70616E69 73682C20 nch, S=Spanish,
32 493D4974 616C6961 6E2C2041 3D416C6C I=Italian, A=All
48 00FFFFFF 48656C6C 6F20576F 726C6400 ....Hello World.
64 FFFFFFFF FFFFFFFF FFFFFFFF 426F6E6A .....Bonj
80 6F757220 746F7574 206C6520 6D6F6E64 our tout le mond
96 6500FFFF 486F6C61 206D756E 646F00FF e...Hola mundo..
112 FFFFFFFF FFFFFFFF FFFFFFFF 4369616F .....Ciao
128 206D6F6E 646F00FF FFFFFFFF FFFFFFFF mondo.....
144 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
160 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
176 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
192 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
208 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....

```

That is pretty much what was expected. The thing to note is that the rest of the EEPROM is filled with ones rather than zeroes (*as I had been lead to believe*): FF Hex = 11111111 binary. This has been consistent across Arduino boards that I have seen.

So now let us erase the EEPROM. This will be simple because we are not really going to erase it. We are just going to write ones to every bit location.



```
EEPROM_Erase
/* Program: EEPROM_Erase
   Writes a 1 to every bit location in the EEPROM */

#include <EEPROM.h>                // include system EEPROM library
unsigned long addr=0;              // address to be read/written
unsigned long wcount=0;            // number of bytes actually written
const unsigned addrEnd=E2END+1;    // compute end condition at compile time
const byte w = 0xFF;              // values to write, binary notation for FF Hex

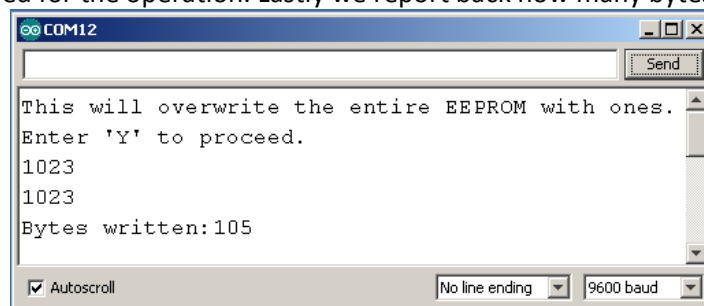
void setup()
{
  Serial.begin(9600);              // initialize serial port
  /* Warn the user what is going to happen and get a confirmation before proceeding */
  Serial.println ("This will overwrite the entire EEPROM with ones.");
  Serial.println ("Enter 'Y' to proceed.");
  while (Serial.available() == 0); // wait for response
  if (Serial.read()=='Y')          // if Y then proceed
  {
    do
    {
      if ((EEPROM.read(addr)) != w) // check to see if byte needs to be written
      {
        EEPROM.write(addr, 0xFF); // write one byte
        wcount++;                 // increment counter
        delay (10);               // EEPROM is slow, allow it time
      }
      ++addr;                     // increment address
    } while (addr < addrEnd);      // the end of the EEPROM
    Serial.println (addr-1, DEC);  // print last address
    Serial.println (E2END, DEC);  // print EEPROM end address
    Serial.print ("Bytes written:"); // print actual bytes written
    Serial.println (wcount, DEC);
  }
  else
  {
    Serial.println ("EEPROM erase aborted.");
  }
}

void loop() { } // do nothing
```

First thing to note is that we require the user to confirm that they want to overwrite the data. You may recall that earlier I said well written program check the EEPROM before overwriting it. I have looked at the low-level assembly code in:

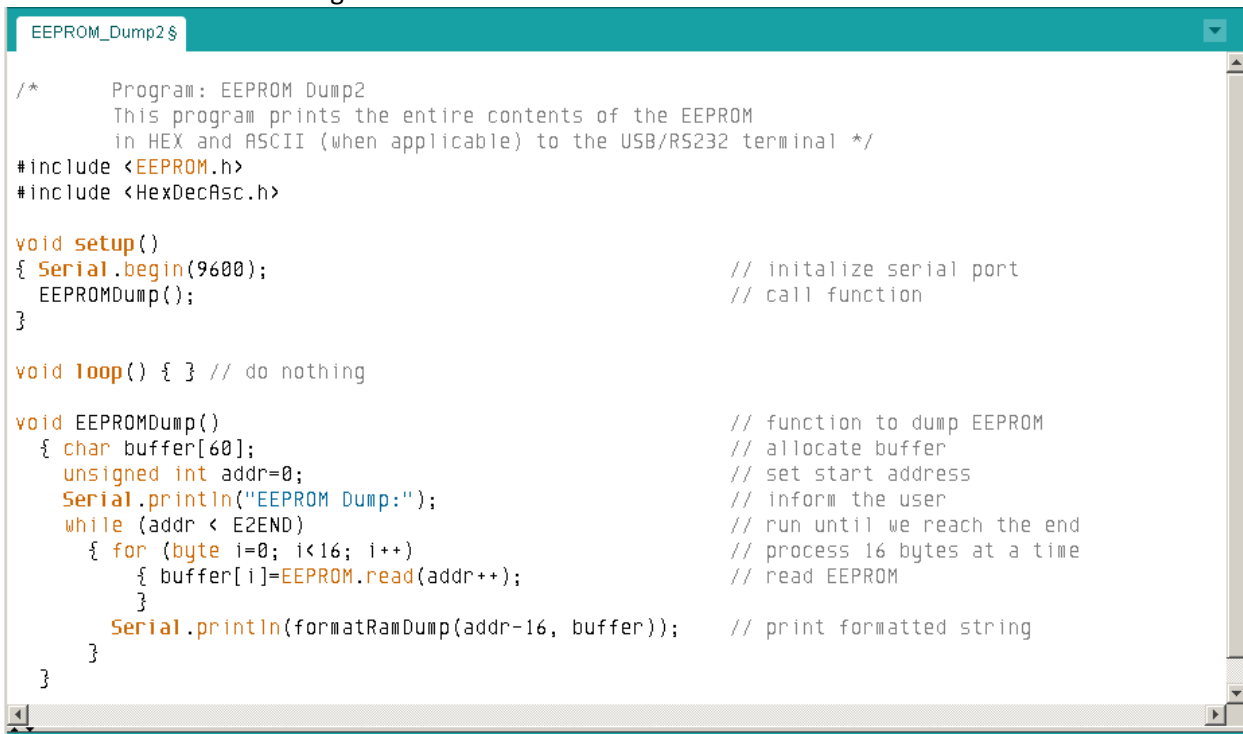
```
<app path>\Arduino\hardware\tools\avr\avr\include\avr\EEPROM.h
```

It does not appear to perform this check. So our code does check before it overwrites location. This also reduces the amount of time required for the operation. Lastly we report back how many bytes were actually written.



```
COM12
This will overwrite the entire EEPROM with ones.
Enter 'Y' to proceed.
1023
1023
Bytes written:105
Autoscroll No line ending 9600 baud
```

Going back and review the code in EEPROM_Dump. It would be convenient if we just had a function that would format the data of 16 byte array. With a function like that our main loop would be somewhat simplified. And our sketch would look something like this:

A screenshot of an IDE window titled "EEPROM_Dump2\$". The code is as follows:

```
/*      Program: EEPROM Dump2
   This program prints the entire contents of the EEPROM
   in HEX and ASCII (when applicable) to the USB/RS232 terminal */
#include <EEPROM.h>
#include <HexDecAsc.h>

void setup()
{ Serial.begin(9600);          // initialize serial port
  EEPROMDump();                // call function
}

void loop() { } // do nothing

void EEPROMDump()              // function to dump EEPROM
{ char buffer[60];              // allocate buffer
  unsigned int addr=0;          // set start address
  Serial.println("EEPROM Dump:"); // inform the user
  while (addr < E2END)          // run until we reach the end
  { for (byte i=0; i<16; i++)    // process 16 bytes at a time
    { buffer[i]=EEPROM.read(addr++); // read EEPROM
    }
    Serial.println(formatRamDump(addr-16, buffer)); // print formatted string
  }
}
```


Building a Library: The easy way

For version two of our EEPROM Dump program we are going to build a library that formats various values as the representation in Hexadecimal, Decimal or ASCII characters. The first step is to build a prototype program where we will develop each and test each routine that we need. Along the way we are going to find out a little about:

- passing parameters to a function
- returning a value from a function
- pointers (*reference and dereference*)
- returning a char array from a function
- multiple references (*names*) to a single char array
- overloading a function (*optional arguments*)

Then we will convert those routines into a library file. Lastly we will rewrite the EEPROM Dump program.

Functions: Passing Parameters and Return Values

Open the Arduino IDE and enter the following code.

```
void setup ()
{
  Serial.begin (9600);
  Serial.println(demoreturn(1234,10));
}

int demoreturn (int a, int b)
{
  int c;
  c = a / b;
  return c;
}

void loop() {} // do nothing
```

Compile, upload and run this program in the IDE's serial monitor. It should output "123". Notice that we did not get a decimal value "123.4" or remainder. That is because we are using integer math. All results are in whole numbers. We will take advantage of this later on.

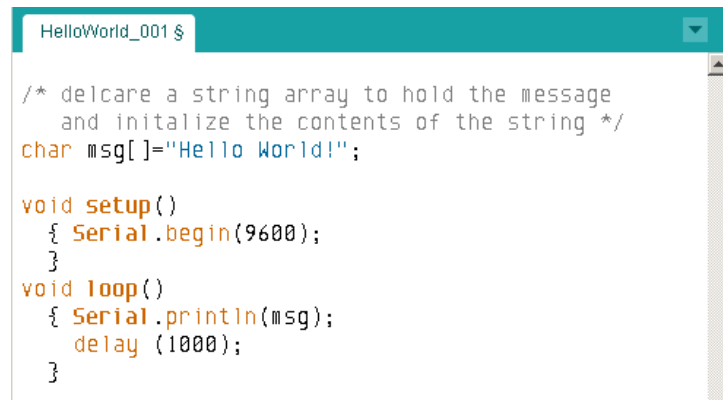
The first line of the function is the declaration "`int demoreturn (int a, int b)`". We have been using "`void`" as our type for the previous functions. This time we use "`int`" to tell the compiler that we are going to return an integer value. We also told the compiler that the function receives two integer values in the parameters section (*between the parentheses*). The variables called "`a`" and "`b`" are only visible inside the function. The actual program instructions are between the curly braces. The first thing we do is allocate space for the variable "`c`" in order to have something to hold result. Then we define "`c`" to be equal to "`a`" divided by "`b`". Lastly we use the keyword "`return`" to tell the compiler that the value of "`c`" is the value to be returned by the function. We did not really need the variable "`c`". The function could have been written to return the result of calculation directly. As I prefer to see single sentences on a single line I would probably write it like this:

```
int demoreturn (int a, int b) { return (a/b);} // divide a by b and return the result
```

Inside the setup function we have the call to our function "`demoreturn(1234,10)`". The earlier description of SRAM mentions that part of it was used as the "stack". What actually happens when a function is called is that the mpu "pushes" the values for the parameters on the stack. The function can then "pop" these values off the stack and use them. For an "integer, word or address" value two bytes are pushed and popped. For a "byte or char" value only one byte needs to be pushed or popped so when convenient you really want to use the smaller

data type. When the function has completed its program steps it can return a value in the same manner. It pops the return address off the stack and pushes the returned value on the stack.

Using the stack in this manner imposes some limitations. The largest value that can be passed is a long or unsigned long that is four bytes. A complex data structure is one that cannot be passed via normal stack operations. The one that “C” supports natively is the char array or “C string” structure (*other complex structures can be defined but that is beyond the scope of this document*). Complex data structures must be passed by a different method. The simplest method is to declare these data type globally at the top of our program. We did that in HelloWorld_001:



```
/* declare a string array to hold the message
and initialize the contents of the string */
char msg[]="Hello World!";

void setup()
{ Serial.begin(9600);
}

void loop()
{ Serial.println(msg);
  delay (1000);
}
```

This method quickly use up our limited RAM because these variables “live” for the entire life of the program. A better method is to declare these complex data structures inside of a function. That way they begin life when the function is called and end life when the function exits. To do that we need a way to pass these complex data structures between parts of our program. The method used in the “C” language is called “pointers”. A pointer is like a street address that tells the program where to find the data structure: “[That string lives at number 0x001A on SRAM bus route.](#)” The AVR is a very small computer so it only has one bus route: SRAM (*EEPROM and FLASH may be considered the suburbs --- you must take a special transfer bus to get there*). Thus we only need to pass the address which is always two bytes long. Earlier we demonstrated that “C” will accept any kind of two byte value when it is expecting two byte value. That is NOT true in the case of pointers. This is a case where “C” checks to see if the calling statement is sending a pointer and if the receiving function is expecting a function. That requires a few subtle differences.

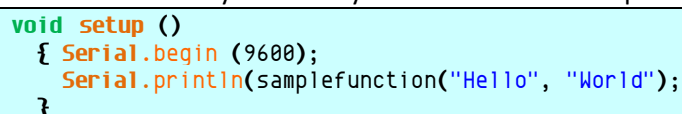
Passing a pointer such as a char array to a function is fairly simple. When you create an array “C” actually constructs a pointer for it. So all you need to do is use the name of the char array. Creating a function that expects a pointer is very similar. You simply add the array constructor in you declarations. We did both in the function “`printdnhb`” of “Bits and Bytes”.



```
void printdnhb (char c[], word n)           // function to print value as
{ byte s;                                   // Decimal, Hexidecimal and binary
  long i;

  Serial.print (c);
  . . .
```

Returning a data structure such as a char array or string from a function is where it gets a bit more complicated. You have to tell the compiler that you are returning a pointer and what kind of data structure that pointer is for. At this point we only have one kind of data structure: the char array. The following code will NOT compile for a number of reasons. By this time you should be able to spot some of them.



```
void setup ()
{ Serial.begin (9600);
  Serial.println(samplefunction("Hello", "World"));
}
```

```
// join two char arrays with a blank space between them, return the new array
char samplefunction(char a, char b)
{ char c[30];
  byte i,k,t;
  for (I=0, k=0, t=1; (t!=0 & k < sizeof c);) // declare internal variables
  { c[k++] = a[i]; // initialize variables and set test condition
    t=a[++i]; // get one character, increment array index
  } // set test variable
  c[k++] = ' '; // add a space and increment return array index

  for (i=0, t=1; (t!=0 & k < sizeof c);) // repeat with second array
  { c[k++] = b[i];
    t=b[++i];
  }
  c[k] = 0; // add a null
  return c; // return array
}

void loop() {} // do nothing
```

Let us start with the obvious typo bugs. In the “setup” function there is a missing close parentheses at the end of the second statement. The letter “I” in the “for” statement should not be capitalized (*that was compliments of user hostile Microsoft Word automatically correcting my text for me*). At the end of the last statement in the “sample function” there is a missing semicolon.

The function returns that char array but the function is declared as a simple char type. We need to fix that. Insert an asterisk character “*” (shift 8) before that function name to tell “C” that we are returning a pointer. This is called the “[dereference operator](#)” operator in “C”. We have a similar problem with the parameters. We are passing two char arrays to the function but the variables “a” and “b” are declared as simple char types. We need to add the array designator “[]” after each one. The dereference operator could also be used. There is one more very BIG problem. The array that we are returning will cease to exist (*actually be deallocated*) when the function exits. The simple method to resolve this is to insert the keyword “static” in front of the declaration for the char array “c[]”. This is much the same as declaring a global variable at the top of the program. The difference is that the variable is only visible inside the function in which it is declared (*unless you pass a pointer to the variable*).

The keyword “sizeof” is new but it is correctly used to do exactly what it says. It will return the size of the char array “c”. In this case that would be the size we declared it to be: 30 bytes. This operator normally returns the size of the variable or type of the operand. There is however an exception ([from Wikipedia](#)):

" To use sizeof, the keyword "sizeof" is followed by a type name or an expression (*which may be merely a variable name*). If a type name is used, it must always be enclosed in parentheses, whereas expressions can be specified with or without parentheses. When sizeof is applied to the name of a **static array** (*not allocated through malloc*), the result is the size in bytes of the whole array. This is one of the few exceptions to the rule that the name of an array is converted to a pointer to the first element of the array, and is possible just because the actual array size is fixed and known at compile time, when sizeof operator is evaluated. "

In my narrow minded opinion these exceptions make the operation of “sizeof” inconsistent but most others consider it perfectly normal. In this function we are using the “sizeof” operator to avoid the possibility of writing beyond the end of our char array (*a condition that is also known as buffer overflow*). Here is the rewritten debugged code. Save it as “HelloWorld_007”.

```
void setup ()
{ Serial.begin (9600);
  Serial.println(samplefunction("Hello", "World"));
  Serial.println(samplefunction("Hello", "World beyond your imagination"));
}
```

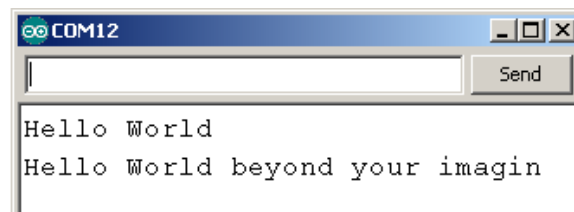
```

// join two char arrays with a blank space between them, return the new array
char * samplefunction(char a[], char * b)
{ static char c[30];
  byte i,k,t;
  for (i=0, k=0, t=1; (t!=0 & k < sizeof c);) // declare internal variables // initialize variables and set test condition
  { c[k++] = a[i]; // get one character, increment array index
    t=a[++i]; // set test variable
  }
  c[k++] = ' '; // add a space and increment return array index

  for (i=0, t=1; (t!=0 & k < sizeof (c);) // repeat with second array
  { c[k++]=b[i];
    t=b[++i];
  }
  c[k] = 0; // add a null
  return c; // return array
}

void loop() {} // do nothing

```



The problem with this program is it uses just as much SRAM as declaring the array globally in the first place. The common method used to circumvent this problem is to pass the target array to the function from the calling function. In that manner the array only lives as long as the calling function(s). “HelloWorld_008” demonstrates that method. Note that the “[sizeof](#)” function MUST be used in the function where the char array is declared.

```

void setup ()
{ Serial.begin (9600);
  demofunction("Hello", "World");
  demofunction("Hello", "World beyond your imagination");
}

void demofunction(char a[], char * b)
{ static char c[30]; // this array is discarded when the function exits
  Serial.println (samplefunction(a,b,c, sizeof c));
}

// join two char arrays with a blank space between them, return the result
char * samplefunction(char a[], char * b, char * c, byte n)
{ byte i,k,t; // declare internal variables
  for (i=0, k=0, t=1; (t!=0 & k<n-1);) // initialize variables and set test condition
  { c[k++] = a[i]; // get one character, increment array index
    t=a[++i]; // set test variable
  }
  c[k++] = ' '; // add a space and increment return array index

  for (i=0, t=1; (t!=0 & k<n-1);) // repeat with second array
  { c[k++]=b[i];
    t=b[++i];
  }
  c[k] = 0; // add a null
  return c; // return array
}

void loop() {} // do nothing

```

Library HexDec: Developing Functions

Open the Arduino IDE and start a new project named “Develope_HexDec”. Put a comment at the top the file for the name and enter the standard template code along with a line to initialize the serial port.

```
/* Program to develop HEX/DEC/ASC conversion routines for a library
demonstrates returning a string from a function
demonstrates multiple references to the same char array
demonstrates overloading function to make function parameter optional
prints a ASII table
prints a EEPROM dump
This code is placed in the public domain: August 2013, Lewis Balentine, Houston, Texas, USA
*/
#include <EEPROM.h>

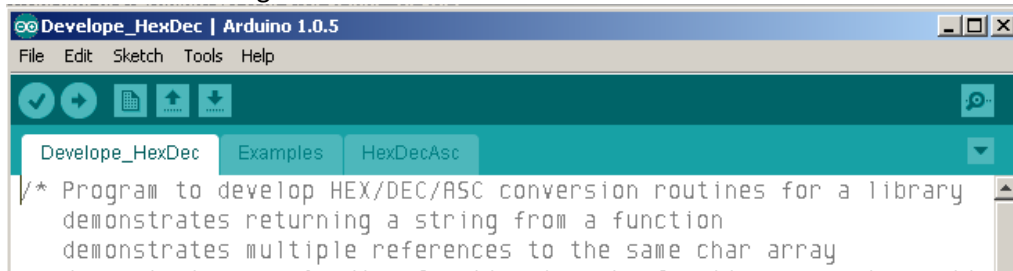
void setup ()
{ Serial.begin (9600);
}

void loop() {} // do nothing
```

Save the file and then use the “upside down triangle icon” to create two additional files for this project:

Examples.ino	(this will hold our demo/test code)
HexDecAsc.ino	(this will hold our library functions)

The IDE should now look something like this:



The first function that we are going to define is one to return a digit from the set “01234567890ABCDFE” according to its location within the set. Rather than using a char array (*as is commonly done*) our function will be based on adding an appropriate value to return the character code. This avoids taking up SRAM space for the char array. Place the following code in the “HexDecAsc” tab.

```
/* --- This returns the character for a Hex Digit --- */
char hexDigit(byte n) // works for oct, decimal, bcd or hex digits
{ if (n>15) // we will sell no wine before its time
  { return (63);} // or accept any digit above 15: '?'
  else if (n>9) // if greater than nine need an ALPHA charater
  { return (n + 0x37);} // Hex(10) + Hex(37)=Hex(41), Hex(41) = Decimal(65) = 'A'
  else // else we need a digit
  { return (n + 0x30);} // Hex(00) + Hex(30)=Hex(30), Hex(30) = Decimal(48) = '0'
}
```

This is fall through logic. We first test that the value provided is within the appropriate range for our function. If it is not then we return a question mark “?”. Next we test to see if the value is greater than nine. If it is then we return an alpha character from the set “ABCDEF”. Otherwise we return a numeric character from the set “0123456789”. Now enter the following code in the “Examples”.

```
// ----- examples -----
void demoHexDigit()
{ Serial.println ("Demo/test digit conversion: ");
  for (byte i=0; i<17; i++)
  { Serial.print(hexDigit(i));
```

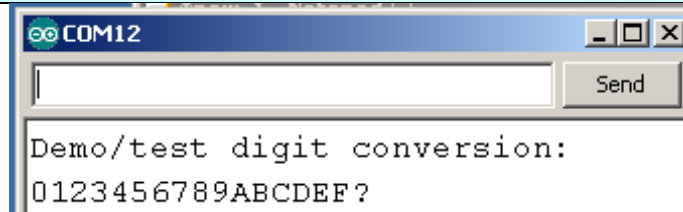
```

    }
    Serial.print();
}

```

This goes into the main tab under setup:

```
demoHexDigit();
```



That will be the pattern for all the functions:

- 1) Write the function (in HexDecAsc.ino)
- 2) Create a demo/test function (in Examples.ino)
- 3) Call the demo/test function (in Deveope_HexDec.ino)

The next function is a simple filter for our ASCII characters (goes in the "HexDecAsc" tab).

```

/* --- Filter characters codes. This allows only defined ASCII characters ----- */
char rtnASCIIcode(byte n)
{ if (n==92) {return '.';} // avoid chance of escape sequence by blocking "\"
  else if (n< 32) {return '.';} // non-printing control characters
  else if (n>126) {return '.';} // 8 bit characters are undefined by ASCII
  else { return char(n);} // that leaves the all the rest
}

```

This is the example routine.

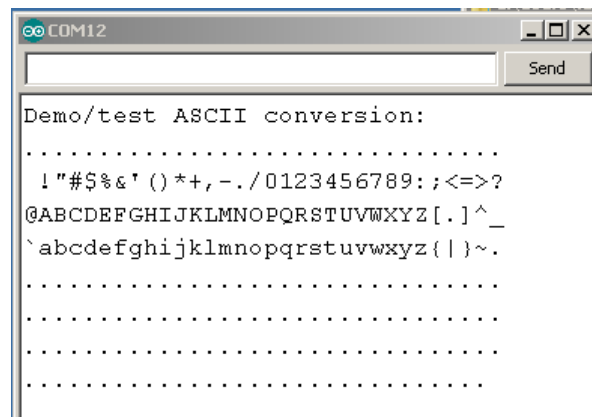
```

void demoRtnASCIIcode ()
{ Serial.println ("Demo/test ASCII conversion: ");
  byte j=0;
  for (byte i=0; i<255; i++)
  { Serial.print(rtnASCIIcode(i));
    if (j++ == 31)
    { Serial.println();
      j=0;
    }
  }
  Serial.println();
}

```

Add one more line for the setup function.

```
demoRtnASCIIcode ();
```



The next function converts a byte to two hex digits using our `formatBytesAsHex` function. It returns a pointer to a char array with the ASCII characters. Note how we are splitting the byte into nibbles.

```
char *formatByteAsHex(byte n, char tmp[]) // this function formats 'n' as two hex digits
{ tmp[1] = hexDigit(n & 0x00001111); // in the char array 'tmp' that is passed to it.
  tmp[0] = hexDigit(n >> 4); // it returns that the pointer to that char array
  return tmp; // temp MUST BE at least two bytes long
}
```

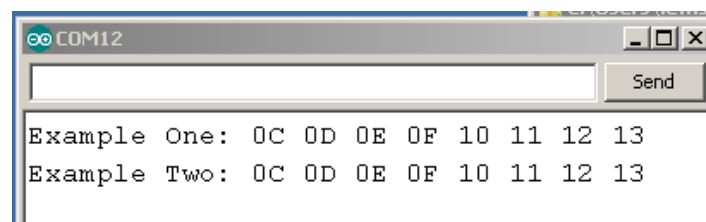
The example code is a bit more complex this time. The concept here is to demonstrate that we can use the return value of the function or use the function just to format the variable `"tmp"`. That makes the use of the function very flexible.

```
// --- example one --- (demo use of formatByteAsHex --- returned string)
void printHexByteOne() // example using returned string with print
{ char tmp[]=" "; // in this case we send a number and string
  Serial.print("Example One: "); // the string is returned to Serial.Print
  for (byte i=12; i<20; i++) // with hex digits
  { Serial.print(formatByteAsHex (i, tmp));
  }
  Serial.println(""); // output: "0C 0D 0E 0F 10 11 12 13"
}

// --- example two --- (demo use of formatByteAsHex --- passes string)
void printHexByteTwo() // example using returned string with print
{ char tmp[]=" "; // this is nearly the same but in this case
  Serial.print("Example Two: "); // we call the format function and then
  for (byte i=12; i<20; i++) // we call the print function. The idea is to
  { formatByteAsHex (i, tmp); // show it can be used either way.
    Serial.print (tmp);
  }
  Serial.println(""); // output: "0C 0D 0E 0F 10 11 12 13"
}
```

Here is the code to run the two examples.

```
/* --- demo/test use of formatByteAsHex --- */
printHexByteOne(); // uses returned string, see examples
printHexByteTwo(); // uses returned string, see examples
```



Now we need to do the same thing with a word size variable but we add decimal to our options as well.

```
char *formatWordAsHex(word w, char tmp[]) // this function formats 'n' as four hex digits
{ formatByteAsHex(lowByte(w), & tmp[2]); // temp MUST BE at least four bytes long
  return formatByteAsHex(highByte(w), tmp);
}

char *formatWordAsDec(word n, char tmp[]) // this function formats 'n' as five Dec digits
{ tmp[4] = hexDigit(n % 10); // Use integer math to get each digit: 54321%10=1
  tmp[3] = hexDigit((n % 100)/10); // 54321 % 100 = 21, 21/10=2
  tmp[2] = hexDigit((n % 1000)/100); // 54321 % 1000 = 321, 321/100=3
  tmp[1] = hexDigit((n % 10000)/1000); // 54321 % 10000= 4321, 4321/1000=4
  tmp[0] = hexDigit(n / 10000); // 54321/10000=5
  return tmp; // temp MUST BE at least five bytes long
}
```

Notice the first statement of “formatWordAsHex” function. We use the “&” reference operator to get the address of the third byte of our char array. This operator returns the address of the operand. Now we need test routines.

```
// --- example three --- (demo use of formatWordAsHex --- returns string)
void printHexWordOne() // example using returned string with print
{ char tmp[]=" "; // in this case we send a number and string
  word ww = 0xD431; // sample word, this could be address
  Serial.print("Example Three (D431): "); // the string is returned to Serial.Print
  Serial.println(formatWordAsHex(ww, tmp)); // range of word is 65K
}

// --- example four --- (demo use of formatWordAsHex --- passes string)
void printHexWordTwo() // example using passed string with print
{ char tmp[]=" "; // in this case we send a number and string
  word ww = 54321; // sample word, this could be address
  Serial.print("Example Four (54321): "); // the string will be passed Serial.Print
  formatWordAsHex(ww, tmp);
  Serial.println(tmp);
}

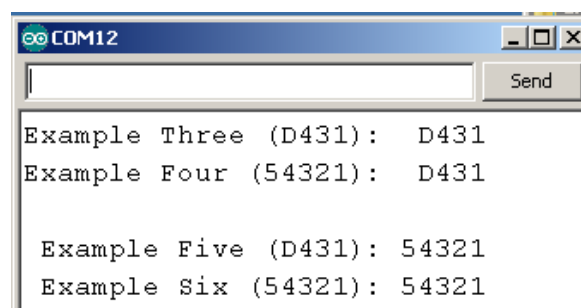
// --- example five --- (demo use of formatWordAsDec --- returns string)
void printDecWordOne() // example using returned string with print
{ char tmp[]=" "; // in this case we send a number and string
  word ww = 0xD431; // sample word, this could be address
  Serial.print(" Example Five (D431): "); // the string is returned to Serial.Print
  Serial.println(formatWordAsDec(ww, tmp));
}

// --- example six --- (demo use of formatWordAsDec --- passes string)
void printDecWordTwo() // example using returned string with print
{ char tmp[]=" "; // in this case we send a number and string
  word ww = 54321; // sample word, this could be address
  Serial.print(" Example Six (54321): "); // the string will be passed Serial.Print
  formatWordAsDec(ww, tmp);
  Serial.println(tmp);
}
```

WOW! There was a lot of button pushing and clicking to format that piece of code. There is a lot more code in the test routines than the functions because a lot of time was previously spent optimizing the code for the functions (*the test routines were used to test those*). Here is the bit to go into the main tab.

```
/* --- demo/test use of formatWordAsHex --- */
printHexWordOne(); // uses returned string, see examples
printHexWordTwo(); // uses passed string, see examples
Serial.println();

/* --- demo/test use of formatWordAsDec --- */
printDecWordOne(); // uses returned string, see examples
printDecWordTwo(); // uses passed string, see examples
Serial.println();
```



That brings us down to the really big function that was the target to begin with.

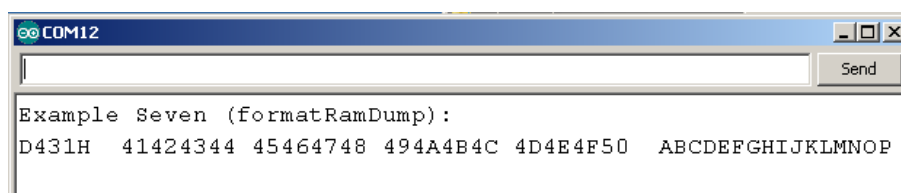
```
char *formatRamDump(char hd, word addr, char data[], char buffer[])
// this function formats the address and data into the buffer with the ASCII
// representation of:
//     address in Hex (hd='h' or 'H') or default as Decimal
//     data in HEX (MUST BE 16 bytes)
//     data in ASCII
// buffer array must be at least 60 bytes
// 0      1      2      3      4      5      6
// 0123456789012345678901234567890123456789012345678901
// 00000 HLHLHLHL HLHLHLHL HLHLHLHL HLHLHLHL AAAAAAAAAAAAAA\n
{ byte i,p,s;
  buffer[60]=0; // stuff a null terminator
  for (i=0; i<16; i++) // do ASCII first
    { buffer[i+44]=rtuASCIIcode(data[i]); } // get the ASCII code
  buffer[42]=' '; // two spaces
  buffer[43]=' ';
  i=15; // do HEX next and do it backwards
  s=0; // counter for seperator
  for (p=40; p>6; p=p-2) // four bytes at a time
    { formatByteAsHex(data[i--],& buffer[p]); // get Hig/low hex characters
      if (++s == 4) // if we have done 4 bytes
        { buffer[--p]=' '; // add a space
          s=0; // and reset the counter
        } // do it again sam
    }
  buffer[5]=' '; // two spaces
  buffer[6]=' ';
  if (hd == 'h' || hd == 'H') // if H or h was sent then use HEX for address
    { formatWordAsHex(addr, buffer); // Flag as HEX format
      buffer[4]='H';
    }
  else // otherwise use decimal
    { formatWordAsDec(addr, buffer); }
  return buffer; // done, return the char array pointer
}
```

Notice that we are working from the back (*buffer[60]*) to the front (*buffer[0]*). You will see the reason for that later. Thus it is time for the test routine.

```
// --- example seven --- (demo/test use of demoFormatRamDump )
void demoFormatRamDump()
{ word addr= 54321;
  char buffer[64];
  char data[16];
  for(byte i=0; i<16; i++) { data[i]= i + 0x41; } // stuff the data "ABCDEFGHIJKLMNOP"
  Serial.println("Example Seven (formatRamDump):");
  Serial.println(formatRamDump('H', addr, data, buffer));
}
```

The main tab gets this.

```
/* --- demo/test use of formatRamDump --- */
demoFormatRamDump();
```



Library HexDec: Overloading

We have been talking about the “C” language. The Arduino IDE uses the C++ implementation of the “gcc” compiler. “C++” is an extension or enhancement to the “C” language. One of the things it add is something called “overloading”. This allows one to add to the definition or complete replace the definition of an existing operator or function. It is primarily intended to be used to extend the functionality of defined operators and function. For example the “+” operator can be extended to be used to concatenate two strings.

Some program languages such as Basic allow for functions to be defined with optional arguments. Overloading provides this capability in C++. We are going to define some alternates versions of our functions to reduce the number of parameters that they require. The idea is we are going to stuff the bytes to be converted into the char arrays before the function is called. Each of the following function should be added to the “HexDecAsc” after the previous function definition of the same name.

```
char *formatByteAsHex(char tmp[]) // this function 'overloads' the previous one
{ return(formatByteAsHex(byte(tmp[0]), tmp));}

char *formatWordAsHex(char tmp[]) // this function 'overloads' the previous one
{ return(formatWordAsHex(word(tmp[0],tmp[1]), tmp));}

char *formatWordAsDec(char tmp[]) // this function 'overloads' the previous one
{ return(formatWordAsDec(word(tmp[0],tmp[1]), tmp));}

char *formatRamDump(word addr, char data[], char buffer[]) // overload for default decimal
{ return formatRamDump('D', addr, data, buffer);}
char *formatRamDump(word addr, char buffer[]) // overload decimal, data=buffer
{ return formatRamDump('D', addr, & buffer[0], buffer);}
```

In the first three functions we are simply extracting the byte or the word to be converted from the char array and passing it to the previously defined function. In the last two we are proving for a default format of Decimal rather than Hex. In the last function we also use the buffer as the data array. This is why we designed out function to work from back to front. By the time the conversion process gets to the address all of the conversion has taken place. That is fortunate because the original data is overwritten by the conversion.

Add these three lines at the end of the “demoFormatRamDump” function.

```
Serial.println("Example Seven (formatRamDump/overloaded):");
Serial.println(formatRamDump(addr, buffer));
Serial.println();
```

Add this new function to the Example code.

```
// --- example eight --- (demo/test use of overloading )
void demoOverLoading()
{ Serial.println("Example Eight (overloading):");
  //      012345678900123456789 // cheat: to figure byte locations
  char tmp[]="A B C D E F G H "; // Sample Data
  Serial.println(tmp); // print sample data
  for (byte i=7; i>0; i--) // with hex digits
  { formatByteAsHex (& tmp[(2*i)]);} // format each byte except the first
  Serial.println(formatByteAsHex(tmp)); // format and print first byte
  tmp[4]=0; // stuff with null terminator
  tmp[5]=0; // stuff with null terminator
  tmp[0]=0xD4; // stuff high byte
  tmp[1]=0x31; // stuff low byte
  Serial.print("HEX 0xD431 as HEX: ");
  Serial.println(formatWordAsHex(tmp));
  tmp[0]=0xD4; // stuff high byte
```

```

tmp[1]=0x31; // stuff low byte
Serial.print("HEX 0xD431 as DEC: ");
Serial.println(formatWordAsDec(tmp));
Serial.println();
}

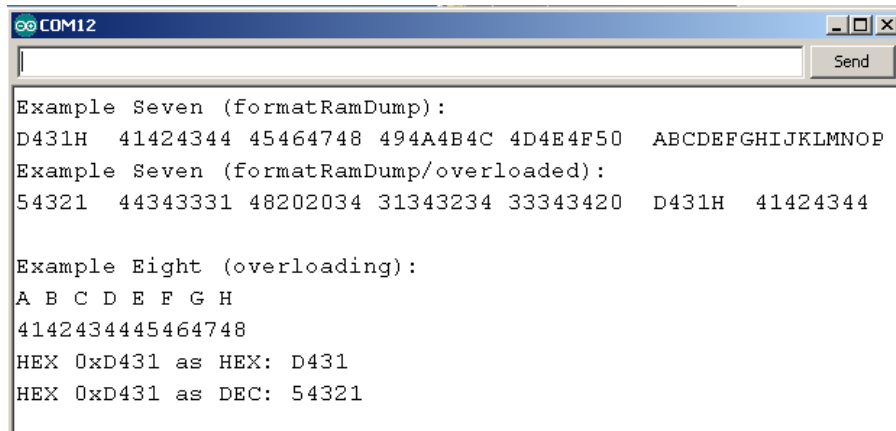
```

Add this line to the setup function.

```

demoOverLoading(); // uses returned string, see examples

```



```

COM12
Example Seven (formatRamDump):
D431H 41424344 45464748 494A4B4C 4D4E4F50 ABCDEFGHIJKLMNOP
Example Seven (formatRamDump/overloaded):
54321 44343331 48202034 31343234 33343420 D431H 41424344

Example Eight (overloading):
A B C D E F G H
4142434445464748
HEX 0xD431 as HEX: D431
HEX 0xD431 as DEC: 54321

```

Library HexDec: ASCII Table

We are going to test our ASCII function by printing a table of ASCII characters. We are doing this to show how to make multiple references to the same char array using different identifiers (*variable names*).

```

/ --- ASCII Table --- (demo multiple references)
/* Of special note in this first function is there are multiple references (names) tied to
   the same char array. This was done so as to pass the array to the hex conversion function
   without copying the bytes back and forth. The array is declared, allocated and printed
   using the name "tmp". Pieces of the array are operated on using the name "OutHex".
   Two points:

   1) The second (or more) name is declared as a "pointer" using the "*"
      dereference pointer operator. Basically this tells the compiler to reserve space for
      an address.

   2) That address must be assigned using "&" reference pointer operator. Basically this
      returns the physical memory address of an object. In this case it returns the address
      of the char array member. Place the "&" in front of the array member reference:
      OutHex= &tmp[3];
      "OutHex" is now considered a char array starting with the third member of the
      character array "tmp".

   Also note that in this function the values to be converted are actually stored in the
   char array that holds the result. The values are in fact overwritten by their ASCII
   Hex characters. This is the reason that the conversion routine was written in a manner
   that starts with the Low nibble. After the High nibble is converted the original
   source data is destroyed.
*/

void printASCIITable() // print out an ASCII table with Hex codes
// this is included because it is simple to do with the functions in hand
{ char tmp[]="0 1 2 3 4 5 6 7 8 9 A B C D E F "; // two spaces after
  byte i,j,n,l; // each character
  char *OutHex;

  Serial.println();

```

```

Serial.println("ASCII Table:");

for (j=2; j<8; j++)
{
  for (i=0; i<16 ; i++)
  {
    n=i*3;
    l= (j * 16) +i;
    tmp[n]=char(l);
    tmp[n+1]=' ';
  }
  Serial.print ("ASCII: ");
  Serial.println(tmp);
  for ( i=0; i<16 ; i++)
  {
    n=tmp[i*3];
    OutHex= & tmp[i*3];
    formatByteAsHex (n, OutHex);
  }
  Serial.print("  HEX: ");
  Serial.println(tmp);
}
}

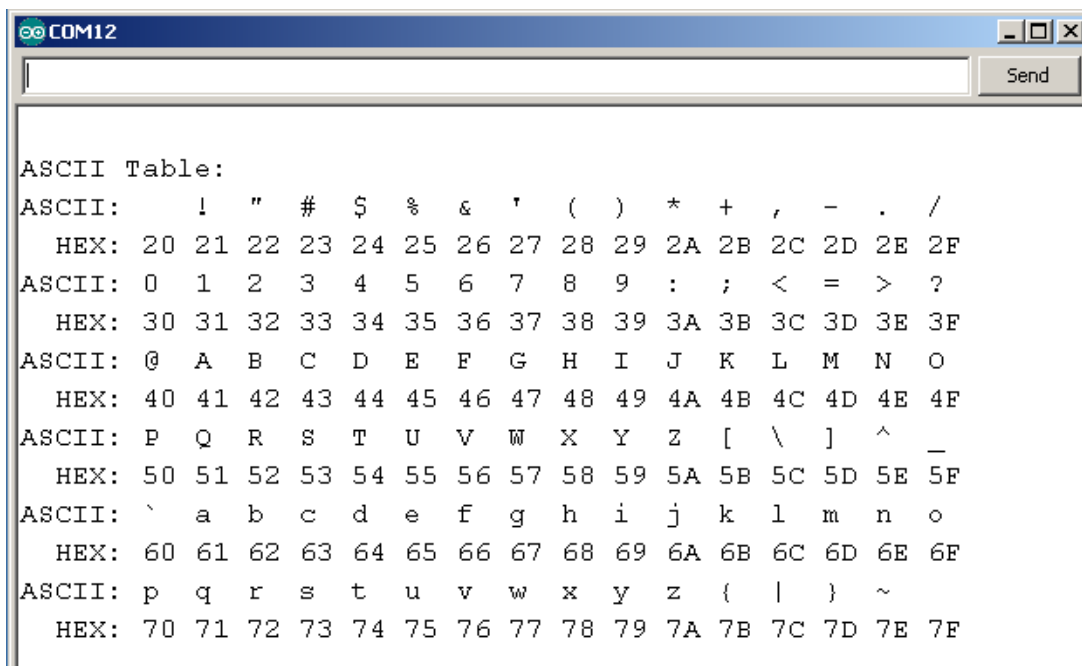
```

Of course the setup function gets another line as well.

```

//  /* --- print ASCII Table --- */
printASCIITable();

```



```

COM12
Send

ASCII Table:
ASCII:  ! " # $ % & ' ( ) * + , - . /
HEX:  20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
ASCII:  0 1 2 3 4 5 6 7 8 9 : ; < = > ?
HEX:  30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
ASCII:  @ A B C D E F G H I J K L M N O
HEX:  40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
ASCII:  P Q R S T U V W X Y Z [ \ ] ^ _
HEX:  50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
ASCII:  ` a b c d e f g h i j k l m n o
HEX:  60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
ASCII:  p q r s t u v w x y z { | } ~
HEX:  70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F

```

Library HexDec: EEPROMDump

When you're up to your neck in alligators,
it is easy to forget the objective was to drain the swamp.

(unknown origin)

The original objective was to simply the process of doing an EEPROM dump via a function that could be used for other similar programs (possibly a flash or sram dump). Thus the last step before we create our library is to verify that our function "formatRamDump" can be used in that manner.

```

// --- example nine --- (EEPROMDump)
void EEPROMDump()
{
  char buffer[60];
  word addr=0;

```

```

Serial.println("EEPROM Dump:");
while(addr < E2END)
{ for (byte i=0; i<16; i++)
  { buffer[i]=EEPROM.read(addr++);
  }
  Serial.println(formatRamDump(addr-16, buffer));
}
}

```

Well that is certainly simpler ... but does it work? Guess where the following goes.

```

//      /* --- EEPROM Dump --- */
EEPROMDump();

```

```

COM12
EEPROM Dump:
00000  453D456E 676C6973 682C2046 3D467265 E=English, F=Fre
00016  6E63682C 20533D53 70616E69 73682C20 nch, S=Spanish,
00032  493D4974 616C6961 6E2C2041 3D416C6C I=Italian, A=All
00048  00FFFFFF 48656C6C 6F20576F 726C6400 ....Hello World.
00064  FFFFFFFF FFFFFFFF FFFFFFFF 426F6E6A .....Bonj
00080  6F757220 746F7574 206C6520 6D6F6E64 our tout le mond
00096  6500FFFF 486F6C61 206D756E 646F00FF e...Hola mundo..
00112  FFFFFFFF FFFFFFFF FFFFFFFF 4369616F .....Ciao
00128  206D6F6E 646F00FF FFFFFFFF FFFFFFFF mondo.....
00144  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00160  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
-----

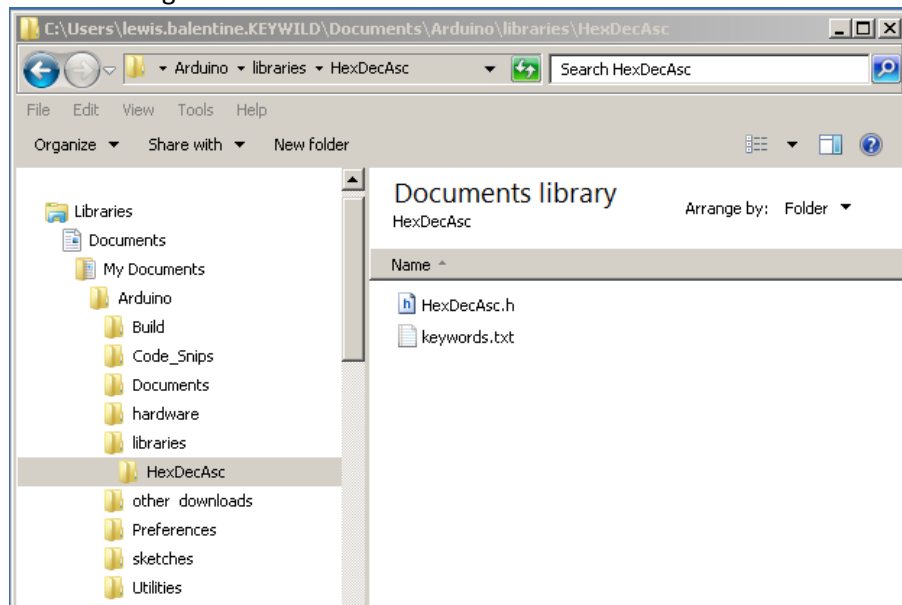
```

Library HexDec: Creating the Library

The proper method of creating a library is to create class such as was done for the `Serial` class that you are so familiar with by now. This involved several files, defining constructors and destructors, complex naming conventions and getting the thing to compile without errors. The whole concept of the Arduino hardware and software was to create an open platform that was simple and inexpensive. Creating a proper class library is anything but simple. Once the library is created then using it is also more complex as one must prefix each function call with the class name as is done in `Serial.print()`. In the next section we will compile a proper class library, but in this section we want something less complicated.

There is a far simpler and more reliable approach although it violates several concepts espoused to produce proper "C" code/programs. That is to place everything, including the functions, in a single header file. I will admit that this is not the correct method to be used in a large or complex project but it works amazing well in a simple small environment where one wishes to share a few functions between several programs.

Close the Arduino IDE. Now create a new folder named “HexDecAsc” in your library folder. Copy the file “HexDecAsc.ino” from you project directory to the new library directory. Rename it “HexDecAsc.h”. The directory should look something like this:



Now open the file “HexDecAsc.h” with any text editor (You can even use *Microsoft Notepad.exe*). We want to add the following lines at the top of the file.

```
#ifndef HEXDECASC
#define HEXDECASC
#include "Arduino.h"

char hexDigit();
char rtnASCIIcode();
char *formatByteAsHex();
char *formatWordAsHex();
char *formatWordAsDec();
char *formatRamDump();
```

The first three lines that begin with “#” are directives to the pre-processor/compiler/linker. The cryptic one at the beginning means “if HEXDECASC is not defined then do the following” . The next one defines “HEXDECASC”. These two lines keep us from accidentally using the library twice in the same program. The third line says to include the header file “Arduino.h”. We need this for all libraries used with the Arduino. It defines basic things such as the various data types and the root operators.

The next six lines are forward declarations for our functions. Note that we have NOT include the parameter information.

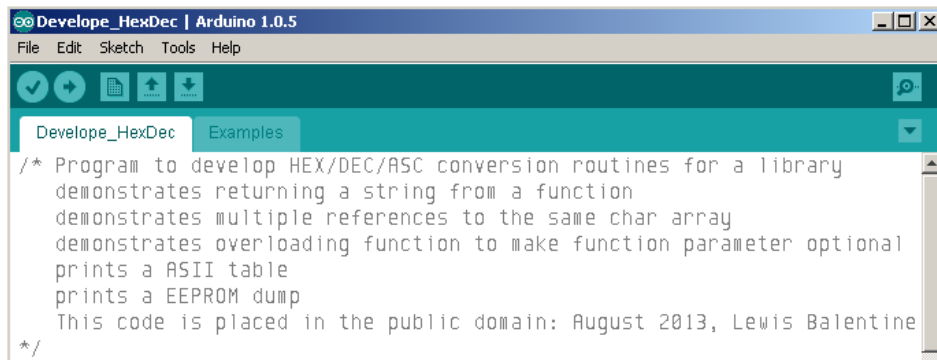
Now add this line to the end of the file.

```
#endif
```

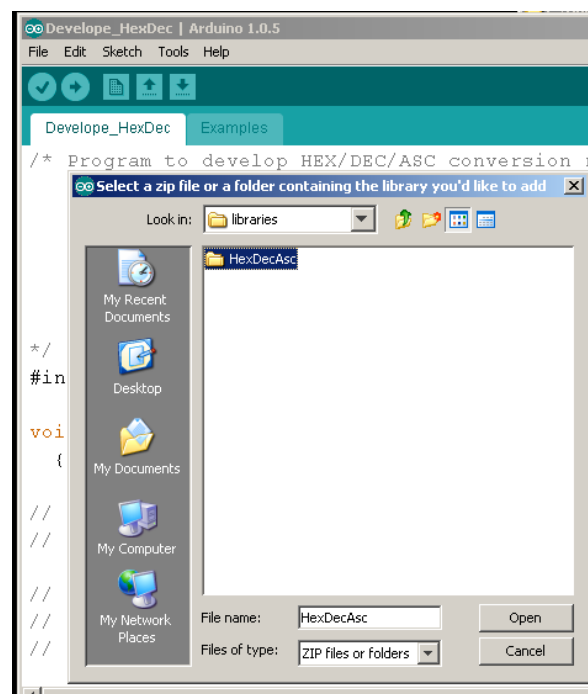
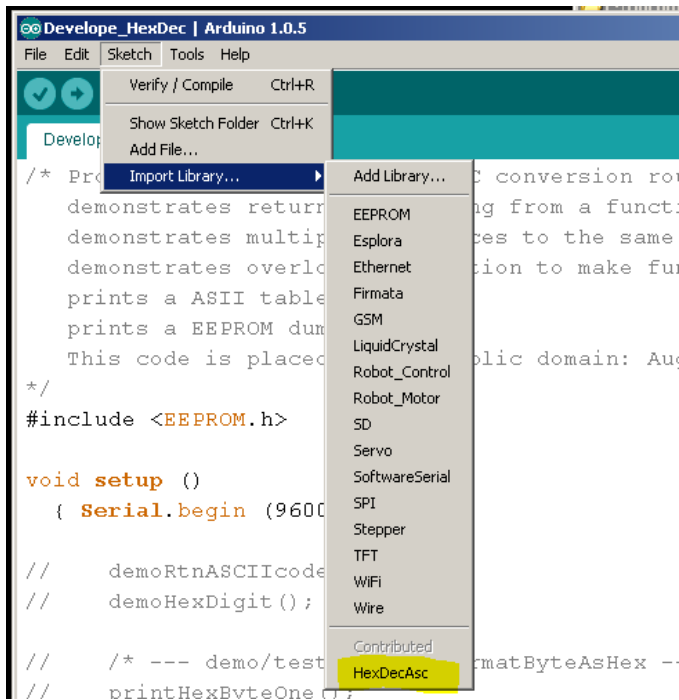
This is the end of our “ifndef” directive. Now save the file and you are done creating the library.

Library HexDec: Testing the Library

Now go back to the sketch directory. Rename the function file “HexDecConvert.ino” to something funky like “HexDecConvert.xyz”. Open the main file with the Arduino IDE (*you should be able to double click on it*). Note that you only have two tabs now.



Use the top menu and select “Sketch”, “Import Library”, “HexDecAsc”. If the library does not show up then use “Sketch”, “Import Library”, “Add Library” and browse to the directory where the library is located.

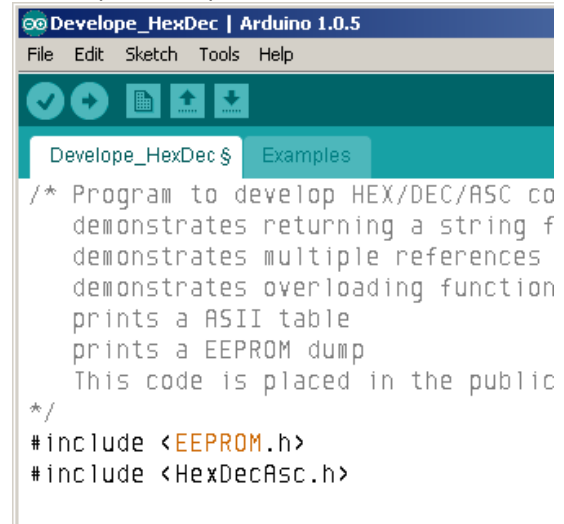


The Arduino IDE adds a single line to the top of your main sketch file. Personally I like to move that line down below the line for the EEPROM library but that is purely a matter of personal preference.



```
Develop_HexDec | Arduino 1.0.5
File Edit Sketch Tools Help
Develop_HexDec $ Examples
#include <HexDecAsc.h>

/* Program to develop HEX/DEC/ASC co
demonstrates returning a string fr
demonstrates multiple references t
demonstrates overloading function
prints a ASII table
prints a EEPROM dump
This code is placed in the public
*/
#include <EEPROM.h>
```



```
Develop_HexDec | Arduino 1.0.5
File Edit Sketch Tools Help
Develop_HexDec $ Examples
/* Program to develop HEX/DEC/ASC co
demonstrates returning a string f
demonstrates multiple references
demonstrates overloading function
prints a ASII table
prints a EEPROM dump
This code is placed in the public
*/
#include <EEPROM.h>
#include <HexDecAsc.h>
```

All the demo/test code should now work as before but you can use these functions in other programs as well by simply importing the library. You should also be able to go back and rewrite the EEPROMDump program using the library (*hint the code was included at the end of the previous section*).

AVR Internal Temperature Sensor

The AVR MPUs have built in temperature sensors. The bad news is for our purposes there are a few problems with them:

- 1) The readings are in degrees Kelvin
- 2) They are not very accurate (+/- 10°C)
- 3) They measure the internal temperature of the MPU rather than the surrounding atmosphere
- 4) They are difficult to read

The good news is most of these problems can be diminished. They also have the distinct advantages that they are already paid for and wired into our circuit (*thus requiring no additional hardware*). There are several on line references that are useful:

Most people with experienced in using microcontrollers and the AVR line in particular are of the opinion that the only practical use for the internal temperature sensor is to determine if there is a problem that is causing your system to overheat. Unfortunately I am a complete novice and assume nothing. Thus before I accept that the internal sensor cannot be used for a practical application I will travel down that path until I find a point of no return.

References:

[Arduino Playground, Internal Temperature Sensor](#) (arduino.cc)
[AVR122: Calibration of the AVR's internal temperature reference](#) (Atmel)
[AVR121: Enhancing ADC resolution by oversampling](#) (Atmel)
[Arduino / AVR internal temperature sensor interface](#) (avdweb)
[ANALOG INPUTS \(ANALOG TO DIGITAL CONVERTER\)](#) (QEEWiki)
[Analogue to Digital Conversion on an ATmega168](#) (protostack)
[Advanced Arduino ADC – Faster analogRead](#) (Marulaberry Projects)

Using the ChipTemp Library

The internal temperature sensor cannot be read via the standard “analogread” function of the Arduino language. To read the internal AVR sensor one must go a bit deeper and deal directly on the AVR hardware. I started with the sample code from Arduino playground. The problem with this code is it makes the assumption that whoever is looking at is familiar with a bunch a strange looking terms like “ADMUX, REFS1, REFS0, MUX3, ADCSRA, ADEN, ADCSRA, ADSC” thus it is pretty much unreadable to the novice. As it turns out these are references to the internal registers of the MPU as defined in the Arduino header files. There is a full description of the applicable registers in the Appendix: AVR ADC Sensor Registers.

The code by Albert van Dalen at avdweb is a bit less confusing because it encapsulates the temperature reads in the form of a library:

<http://www.avdweb.nl/arduino/hardware-interfacing/temperature-measurement.html>

Mr. van Dalen references the code provided by “SpikedCola” and “marcello.romani”. That code was presented in the Arduino forum thread “Using the Internal Temperature Sensor”. Here is a copy of the code that he posted (*with the comments moved to make it more readable*).

```
// http://forum.arduino.cc/index.php/topic,8140.0.html

void setup()
{
  Serial.begin(9600);
  ADMUX = 0xC8;           // turn on internal reference,
                          // right-shift ADC buffer,
```

```

        delay(10); // ADC channel = internal temp sensor
    } // wait a sec for the analog reference to stabilize

void loop()
{
    Serial.println(averageTemperature()); // so we can debug
    delay(500); // just to slow things down a bit
}

int readTemperature()
{
    ADCSRA |= _BV(ADSC); // start the conversion
    while (bit_is_set(ADCSRA, ADSC)); // ADSC is cleared when the conversion finishes
    return (ADCL | (ADCH << 8)) - 342; // combine bytes & correct for temp offset
    (approximate)
}

float averageTemperature()
{
    readTemperature(); // discard first sample (never hurts to be safe)

    float averageTemp; // create a float to hold running average
    for (int i = 1; i < 1000; i++) // start at 1 so we dont divide by 0
    { // get next sample, calculate running average
        averageTemp += (readTemperature() - averageTemp)/(float)i;
    }
    return averageTemp; // return average temperature reading
}

```

To keep things simple we are going to start with use Mr. van Dalen's code but we have to create a library (a proper C++ class library) to do that. I have made some minor changes (*shown below in bold*) to Mr. van Dalen's code. Save these two files in a new library folder named "ChipTemp":

```

// <user home>\Arduino\libraries\ChipTemp\ChipTemp.h
// http://www.avdweb.nl/arduino/hardware-interfacing/temperature-measurement.html
//-----

#ifndef ChipTemp_H
#define ChipTemp_H
//-----
// #include <WProgram.h>
// this is a small change to Mr. van Dalen's code.
// "WProgram.h" was used by the earlier versions of the Arduino IDE
// "Arduino.h" is used by the current version (1.0.5)
//-----
#include "Arduino.h"

// ATmega328 temperature sensor interface
// Rev 1.0 Albert van Dalen
// Based on "InternalTemp"
// Requires 166 ... 204 bytes program memory
// Resolution 0.1 degree
// Calibration values, set in decimals

//static const float offset = 335.2; // change this!
static const float offset = 329.0; // sainsmart Nano number one
static const float gain = 1.06154;

static const int samples = 1000; // must be >= 1000, else the gain setting has no effect

// Compile time calculations
static const long offsetFactor = offset * samples;
static const int divideFactor = gain * samples/10; // deci = 1/10

class ChipTemp
{
public:
    ChipTemp();
    int deciCelsius();
    int celsius();
    int deciFahrenheit();
    int fahrenheit();

private:

```

```

    inline void initialize();
    inline int readAdc();
};

#endif

```

```

// <user home>\Arduino\libraries\ChipTemp\ChipTemp.cpp
// http://www.avdweb.nl/arduino/hardware-interfacing/temperature-measurement.html
//-----

#include "ChipTemp.h"
// #include <WProgram.h>

ChipTemp::ChipTemp()
{
}

inline void ChipTemp::initialize()
{ ADMUX = 0xC8; // select reference, select temp sensor
  delay(10); // wait for the analog reference to stabilize
  readAdc(); // discard first sample (never hurts to be safe)
}

inline int ChipTemp::readAdc()
{ ADCSRA |= _BV(ADSC); // start the conversion
  while (bit_is_set(ADCSRA, ADSC)); // ADSC is cleared when the conversion finishes
  return (ADCL | (ADCH << 8)); // combine bytes
}

int ChipTemp::deciCelsius()
{ long averageTemp=0;
  initialize(); // must be done everytime
  for (int i=0; i<samples; i++) averageTemp += readAdc();
  averageTemp -= offsetFactor;
  return averageTemp / divideFactor; // return deci degree Celsius
}

int ChipTemp::celsius()
{ return deciCelsius()/10;
}

int ChipTemp::deciFahrenheit()
{ return (9 * deciCelsius()+1600) / 5;
}

int ChipTemp::fahrenheit()
{ return (9 * deciCelsius()+1600) / 50; // do not use deciFahrenheit()/10;
}

```

Mr. van Dalen also provides some code to test the library functions. Create a new sketch directory named “ChipTempDemo” and create this file in it.

(Special note: The Arduino IDE is VERY particular. The extension “.ino” MUST be lower case!)

```

// <user home>\Documents\Arduino\sketches\ChipTempDemo\ChipTempDemo.ino
// http://www.avdweb.nl/arduino/hardware-interfacing/temperature-measurement.html
//-----

// #include <avr/pgmspace.h> // Not needed in current version

#include <ChipTemp.h>

ChipTemp chipTemp;

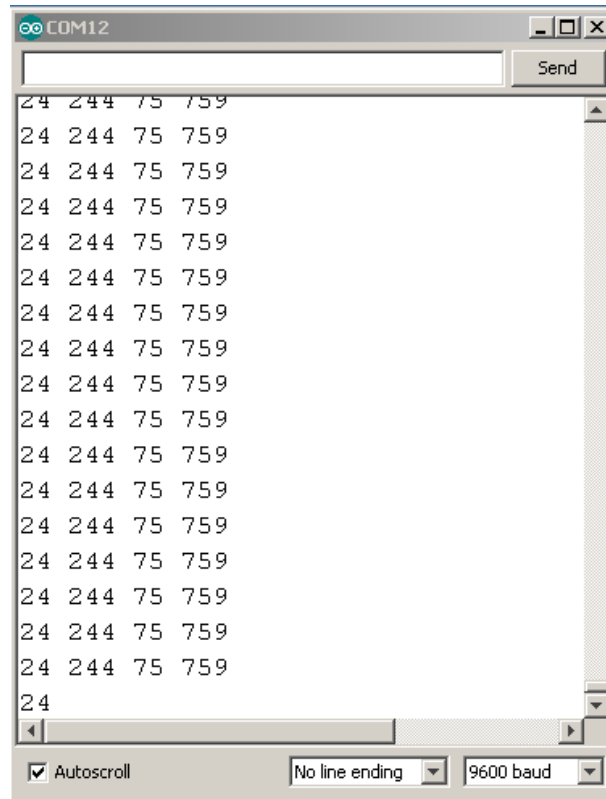
void setup()
{ Serial.begin(9600);
  Serial.println("Celsius decimalCelsius Fahrenheit decimalFahrenheit");
}

void loop()
{ delay(500);
  Serial.print(chipTemp.celsius());
  Serial.print(" ");
  Serial.print(chipTemp.deciCelsius());
}

```

```
Serial.print(" ");  
Serial.print(chipTemp.fahrenheit());  
Serial.print(" ");  
Serial.println(chipTemp.deciFahrenheit());  
}
```

Now open the file in the Arduino IDE by double clicking on it. Compile and upload the file. Then open the serial monitor. You should see something like this. The first two numbers represent degrees Celsius. The second two numbers represent degrees Fahrenheit.



To be honest I was a bit surprised that the library compiled after one minor problem was fixed (*I have not had much joy compiling class libraries*). My Nano had been unplugged for some time before I upload the program to it. It took about 30 minutes for the output to stabilize at: "24 244 75 759". This is the effect of the chip heating up and dissipating power as it runs.

The problem is the thermometer that I had sitting beside it was reading just over 88 degrees Fahrenheit. Looking at the library code there are two constants that are used to adjust the returned values: gain and offset. I added the following line (*and commented out the one it replaced*) in the header file. Then I get a much closer temperature reading.

```
static const float offset = 329.0; // sainsmart Nano number one
```

The "offset" is an adjustment between what the temperature values inside the chip are and the temperature value for the outside real world. The other adjustment "gain" is a linear correction factor applied to the temperature scale. These two factors are different for each Arduino board due to variations in the manufacturing, chip enclosure, chip mounting, voltage regulation, current power use and high noon position of the large asteroid known as Pluto. Well maybe Pluto is not involved. The point is I would prefer to have these factors stored in the EEPROM for each board rather than having to change the library for each board (*not mention having to keep track of which board gets which set of factors*).

Develop Avr Temperature Functions

We are going to build our own library (*the simple kind*) that has a function that returns the raw data from the ARV. That will allow us to apply an appropriate “offset” and “gain” factor without having to change any of the library code. We will also make provisions for the function accept a parameter to determine the number of samples to be read. Let us create a new Arduino project and call it “AvrTemperatureSensor”. Add an extra file to hold our functions “AvrTemperatureSensorFunctions.ino”. Put the standard setup and loop functions in the main window. The only parts we are going to use from the previous code examples are those to initialize and read the AVR temperature sensor.

The techniques of “Oversampling” and “Decimation” are documented in the Atmel AVR121.pdf document.

“Decimation” is used to increase the resolution of the ADC. This technique requires that there be noise on the input signal with a mean (*average*) value of zero. As we are dealing with an internal sensor and an internal voltage reference it is extremely difficult to add any other circuitry to provide the required noise ... unless that noise just happens to exist au naturel. Much to my surprise some testing has revealed there is probably sufficient evenly dispersed noise to increase the resolution of the ADC via “Decimation”.

“The extra samples, *m*, achieved by oversampling the signal are added, just as in normal averaging, but the results are not divided by *m* as in normal averaging. Instead the result is right shifted by *n*, where *n* is the desired extra bit of resolution, to scale the answer correctly. Right shifting a binary number once is equal to dividing the binary number by a factor of 2. As seen from Equation 3-1, increasing the resolution from 10-bits to 12-bits requires the summation of 16 10-bit values. A sum of 16 10-bit values generates a 14-bit result where the last two bits are not expected to hold valuable information. To get ‘back’ to 12-bit it is necessary to scale the result.”

Translation:

Add 16 consecutive 10 bit ADC reading together and right shift the result 2 places.
This yields a virtual 12 bit ADC with a range of 0 to 4096 (rather than 0-1024)

The temperature sensor is calibrated in degrees Kelvin. As we are adding two virtual bits to the ADC this changes the virtual calibration from 1 degree to ¼ degree Kelvin.

Both of the previous mentioned routines take 1000 sequential readings and average. In the case of ChipTemp it is done with a “normal average”. SpikedCola uses a “moving average”. Averaging data from an ADC measurement is equivalent to a low-pass filter and has the advantage of attenuating signal fluctuation or noise, and flatten out peaks in the input signal. I tried number alternative filtering algorithms. In the end I concluded that oversampling was far simpler and in most cases more reliable. Here is the test code that I came up with.

```
/* Project to develop functions for AVR internal temperature sensor */

unsigned long Time;
float save;
void setup()
{ Serial.begin(9600);
}

void loop()
{ word raw;
  float temp;
  byte i;
  raw = avrRawTemp(512); // 512 is the numbers of samples
  temp= (((float(raw))/4)-331.5)*1.8)+32;
  // conversion:
  // float to convert to floating point number
  // divide to scale from 1/4 degree increnments to 1 degree increnments
  // subtract 273 to convert from kelvin to celcius
```

```

// subtract 58.5 (offset varies by board) for external temperature
// multiply by 1.8, add 32 for degrees fahrenheit
// --- accuracy can be increased by using two point calibration ---
Serial.print(raw); // raw reading returned by function
Serial.print(", ");
Serial.print(temp); // converted to degrees Fahrenheit
Serial.print(", ");
Serial.print(Time); // benchmark time to do reading
if (raw<save) Serial.print(" <<<"); // temperature decrease
if (raw>save) Serial.print(" >>>"); // temperature increase
Serial.println();
save=raw; // save current temperature
}

```

There is one thing to note in the code. The level of parenthesis in the conversion to Fahrenheit is important. The conversion from a word value to a floating point value must be done at the very inside of the conversion. Here is the code for function that returns the averaged raw values of the AVR temperature sensor.

```

word avrRawTemp(word samples)
{
  /* samples: the number of samples to average */
  /* this number will be reduced to a power of 2! */
  /* return: degrees Kelvin * 1/4, range 0 to 4096 */
  /* each sample has 16 ADC reads for the 12 bit virtual ADC */
  /* REF: Atmel document number AVR121.pdf */
  //-----//
  /* on 16Mhz ATmega328 512 samples requires just under 1 second */
  /* 16 samples (16*16=256) gives fairly consistent results */
  /* on a steady-state system in under 40K microseconds */
  //-----//

  unsigned long RawSum=0; // used to sum samples for averaging
  word RawTemp=0; // used to accumulate 10 bit ADC readings
  word test=0; // used to count samples
  byte exp=0; // samples = 2 to the exp power,
  // used as shift operand

  byte k=0; // counter for ADC reads
  unsigned long Start=micros(); // this was used for benchmark timing
  // turn on internal reference,
  // right-shift ADC buffer,
  ADMUX = 0xC8; // ADC channel = internal temp sensor
  delay(10); // wait for the analog reference to stabilize

  // as "C" and "C++" lack an basic exponential or power function (or operator)
  // we must resort to loops to calculate the binary exponent
  while (samples>1) { samples /=2; exp++;} // calculate exponent for power of 2
  samples=1; // make sure samples = 1 (not 0)
  while (test++ < exp) { samples *=2;} // set samples value to power of two

  test=0; // reset test because we have abused it
  while (test++ < samples) // oversampling loop (for averaging)
  {
    for (k=0; k<16; k++) // virtual ADC loop, 16 readings
    {
      ADCSRA |= _BV(ADSC); // start the conversion
      while (bit_is_set(ADCSRA, ADSC)); // ADSC cleared when the conversion finishes
      RawTemp += (ADCL | (ADCH << 8)); // accumulate the reading (low byte first)
    }
    RawSum += (RawTemp >>2); // accumulate virtual 12 bit ADC value
    RawTemp=0; // zero ADC accumulator for the next sequence
  }
  Time=micros()-Start; // record benchmark time
  return ((RawSum)>>exp); // average by shifting bit position, LSBs lost
}

```

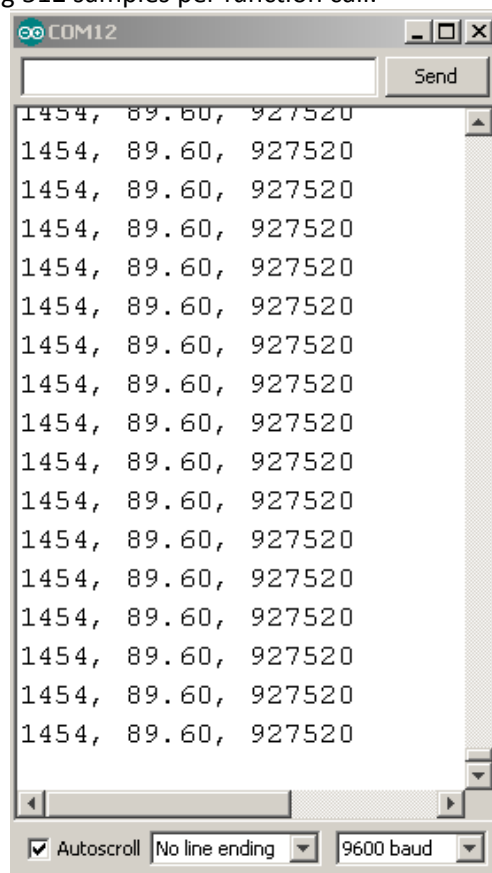
The portions highlighted in yellow were shamelessly copied from the code by “SpikedCola”. I have formatted the terms “ADMUX, ADCSRA, ADSC, ADCSRA, ADCL and ADCH” as keywords but I believe that they are actually constants

that are defined the Arduino header files. I believe that “`_BV`” and “`bit_is_set`” are low level macros defined within the back end tool chain used by the Arduino IDE.

The function “`micros()`” returns the number of microseconds since the Arduino board began running the current program. This function is called at the beginning and end to determine how long it takes to gather the requested number of samples. The global variable “`Time`” is used to return this information to the main loop. The two lines that have these calls as well as the declaration for the variable “`Start`” should be commented out of the final version of the function. The variable “`RawSum`” is 32 bit (four bytes) unsigned long integer. It can hold over a million samples from our virtual 12 bit ADC. If we had used word variable (2 Bytes) we would have been limited to sixteen samples.

I was surprised to find out that “C” and “C++” have not a function or operator for exponential operations (*well there probably is one in a math library somewhere but not in the basic language itself---* **EDIT NOTE: see Arduino reference for the function `pow()`**). I restricted all the variables in this function to integers because integer math is inherently faster and more accurate than floating point. For that reason I want to be able to use a shift instruction to do the averaging. This also has the advantage of discarding the remainder that is beyond the significant digits of our calculations. This is a fairly simple calculation. We simply keep dividing the variable “`samples`” by two until it is less than two. We increment the variable “`exp`” with each division. Then we have set the value to “`samples`” by going through a multiplication loop “`exp`” times. This also insure that whatever requested number of was given the function it will always use the next lowest power of two. Thus if you call for five samples you are only going to get four.

Here is a sample of the output using 512 samples per function call.



Storing Calibration Constants (EEPROM)

Now that we have gone to all the trouble to get our calibration data we need to put the results in the EEPROM. Then we can create that uses that data to report the temperature. First we must determine exactly what we are going to store and where we are going to store it. We want the program to report the actual raw reading that it gets from the sensor as well as the temperature in degrees Fahrenheit Celsius and degrees Fahrenheit. By definition the reference point is going to be 72 degrees Fahrenheit (22.22 degrees Celsius) so the first piece of data we want is raw sensor data reading for that point. That will be stored as our offset point in a two byte word.

Assume for a moment that the correct sensor reading for 72 degrees Fahrenheit is 1485 and that we have a sensor reading of 1500. That will give us a positive difference of 15 units. Those units are actually one quarter of one degree Kelvin. Thus the nominal conversion factor for Celsius is 0.2500. The ratio of degrees Fahrenheit to Degrees Celsius is 1.8. Thus the nominal conversion factor for Fahrenheit is $0.2500 * 1.8000 = 0.4500$. Thus the formulas to report the correct temperatures will be:

$$\begin{aligned}\text{Temperature in Celsius} &= 22.22 + ((\text{Current Reading}-\text{Offset}) * (\text{Conversion Factor for Celsius})) \\ \text{Temperature in Fahrenheit} &= 72.00 + ((\text{Current Reading}-\text{Offset}) * (\text{Conversion Factor for Fahrenheit}))\end{aligned}$$

We are going to store the Celsius number as a two byte word. To convert that number to a word value we multiply it by 65532. To convert the word value back floating point we divide it by 65532. We are also going to store an ID string to identify the specific sensor if we happen to have more than one connected to our computer. That will be 16 bytes long without a null terminator. We also need to store flag that controls how the report data is written and how often it written.

We will call this program "EEPROM_TempSensor_Calibration_Constants". It is rather long because we are going to validate our data after it is stored.

```
/* EEPROM_TempSensor_Calibration_Constants */
/* Stores internal temperature sensor Calibration Constants in EEPROM
   Uses first 32 bytes of EEPROM for working storage
   Uses last 32 bytes of EEPROM for backup storage */

#include <EEPROM.h>
//=====
//== replace these with the actual values to be used for this specific Arduino board ==//
//=====
word      CovrtOffsetR    = 1387;          // Observed Reading for 72 degrees Fahrenheit
float     Covrt2Celsius   = 0.25;         // calibrated conversion factor for celsius
char      IdString[17]="Your string here"; // fill in your ID string here
//.....1234567890123456 // 16 charateres max
//=====
// EEPROM addresses constants
const word EEmask      = 0;               // 1 byte location of EEPROM storage mode mask
const word EEflag      = 1;               // 1 byte location of EEPROM storage mode flag
const word EEOffsetR    = 2;               // 2 byte location of CovrtOffset
const word EECelsius    = 4;               // 2 byte location of Covrt2Celsius
const word EEminutes    = 6;               // 2 byte location of Report Target Minutes
const word EEunused0    = 8;               // 2 byte location -- unused --
const word EEunused1    = 10;              // 2 byte location -- unused --
const word EEunused2    = 12;              // 2 byte location -- unused --
const word EEunused3    = 14;              // 2 byte location -- unused --
const word EEidtring     = 16;              // ID string w/o termiantion size (16)
const word EEidsize      = 16;              // 24 byte location of IdString
const word EEwdszsize    = EEidtring+EEidsize; // Working data storage size (32)
//-----
// you may notice some extra space allocated ... these may be used in a later program
//-----
const word StorageWorking=EEmask;          // EEPROM start for working copy of constants
```

```

// EEPROM start for backup copy of constants
// note we have to add 1 to the value
// Becasue addresses begin with zero not one
const word StorageBackup = ((E2END-(EEwdsz)+1));
const word CovrtFactor = 65532; // The higher this number is the better

//-----
void setup()
{ char WorkStr[EEwdsz+4]; // used to build array to write
  // this array is a bit long so that we can stuff an extra zero byte at the end
  float tempflt=0; // used to convert celsius & fahrenheit factors
  word tempwr=0; // used to save celsius & fahrenheit factors
  float saveflt; // used to save conversion factor
  word i=0; // index into work array
  word j=0; // counter
  word k=0; // marker
  char c; // one character

  Serial.begin(9600); // so we can report progress
  // build array to store -----
  // EEmask and EEflag -----
  WorkStr[i++]=0xFF; // stuff hex FF in array
  WorkStr[i++]=0xFF; // stuff hex FF in array
  // CovrtOffset -----
  WorkStr[i++] = highByte(CovrtOffsetR); // stuff CovrtOffset in array
  WorkStr[i++] = lowByte (CovrtOffsetR);
  // Celsius -----
  tempwr = word(Covrt2Celsius*CovrtFactor); // mutiply and drop decimals
  Serial.print (F("Celcius factor = "));
  Serial.println (Covrt2Celsius, 5);
  Serial.print (F("multiplied by "));
  Serial.print (CovrtFactor);
  Serial.print (F(" = "));
  Serial.println (Covrt2Celsius * CovrtFactor, 5);
  Serial.print (F("Celcius factor stored as Word= "));
  Serial.print (tempwr, DEC);
  Serial.print (F("(HEX: "));
  Serial.print (tempwr, HEX);
  Serial.println (F(")"));
  WorkStr[i++] = highByte(tempwr); // stuff Celcius factor in array
  WorkStr[i++] = lowByte(tempwr);
  Serial.println ();
  Serial.flush ();

  // Minutes between report lines -----
  WorkStr[i++] = 0x00; // stuff hex 00 in array
  WorkStr[i++] = 0x01; // stuff hex 01 in array

  // Extra space we may use later -----
  while (i < EEidsz) {WorkStr[i++] = 0xFF;} // stuff hex 00 in array

  // ID String -----
  IdString[17] = 0; // make sure we have a null terminator
  Serial.print (F("ID String: "));
  // Serial.println (IdString); // we are not going to print the
  // ID string until we filter it
  j = 0; c = 1; k = i; // stuff id string in array
  while (c != 0)
  { c = IdString[j++];
    if (c < 32) c = 0; // filter non-printing characters
    WorkStr[i++] = c;
  }
  while (j++ < EEidsz) { WorkStr[i++] = 0xFF;} // fill in the rest with ones
  Serial.println (& WorkStr[k]); // now print it
  Serial.flush();
}

```

```

// EEPROM Write -----
// erase the the entire the EEPROM
for (i=0 ;i<EEEND; i++)
{ if (EEPROM.read (i) != 0xFF)
    EEPROM.write(i, 0xFF);
}

// copy the array to EEPROM (working copy)
for (i=0 ;i<EEwdsize; i++)
{ if (EEPROM.read (StorageWorking+i) != WorkStr[i])
    EEPROM.write(StorageWorking+i, WorkStr[i]);
}
// copy the array to EEPROM (backup copy)
for (i=0 ;i<EEwdsize; i++)
{ if (EEPROM.read (StorageBackup +i) != WorkStr[i])
    EEPROM.write(StorageBackup +i, WorkStr[i]);
}
Serial.println ();
Serial.println (F("====="));
Serial.println ();
Serial.flush();

//-----
// now read it all back
//-----
i=EEidtring;j=0; c=1;
while (c!=0,j< EEidsize)
{c=EEPROM.read( i++);          // read the ID string
 WorkStr[j++]=c;
}
WorkStr[EEidsize]=0;          // just in case
Serial.print (F("ID String read from EEPROM: "));
Serial.println (WorkStr);    // print it
Serial.flush();

// Get the Offset; -----
tempwrd= EEPROM.read(EEoffsetR)<<8;
tempwrd= tempwrd + EEPROM.read(EEoffsetR + 1);
Serial.print (F("Offset read from EEPROM: "));
Serial.println (tempwrd, DEC);    // print it
Serial.flush();

// Get the celsius factor; -----
tempwrd= EEPROM.read(EEcelsius)<<8;
tempwrd= tempwrd + EEPROM.read(EEcelsius + 1);
Serial.print (F("Celsius factor read from EEPROM: "));
Serial.println (tempwrd, DEC);    // print it
// now we need to convert it
tempflt= float(tempwrd)/ CovrtFactor;
Serial.print (F("Celsius factor converted back to floating point: "));
Serial.println (tempflt, 5);    // print it
Serial.flush();

Serial.println ();
Serial.println (F("Th-th-th-th-th- ... that's all folks!"));
Serial.flush();
// http://www.youtube.com/watch?v=-_kwXNVCaxY
// http://en.wikipedia.org/wiki/Porky_Pig

// //---//---//---//---
// Serial.print (F("--Got Here, i= "));
// Serial.println (i, DEC);
// Serial.println (j, DEC);
// Serial.flush();
// while(true);          // debugging, stops program

```

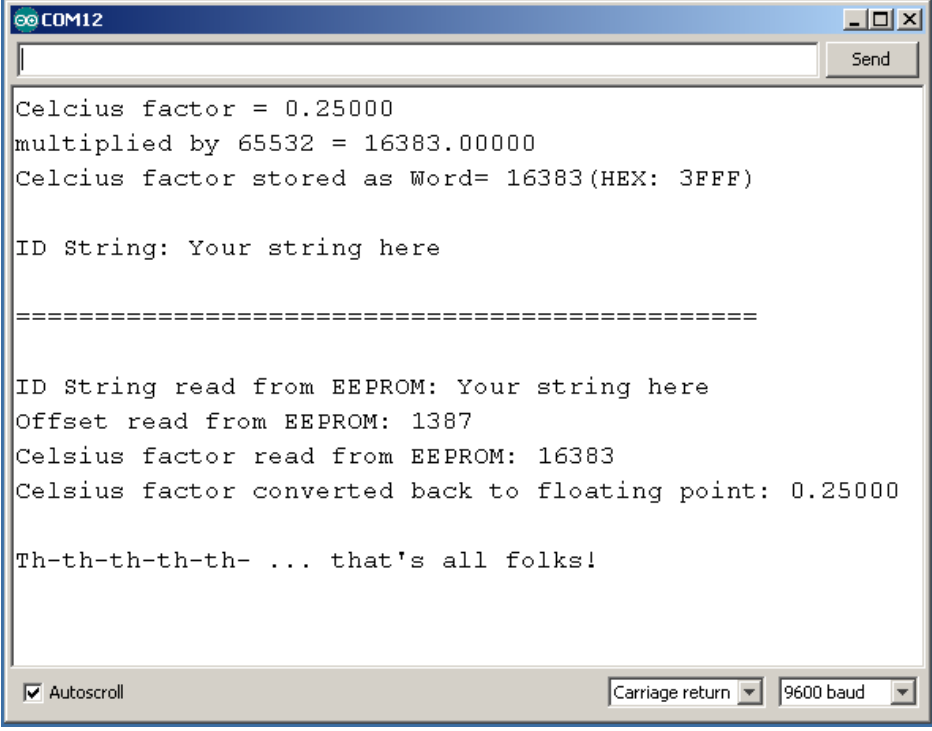
```
//  //---//---//---//---
}

void loop() {} // do nothing
```

There is nothing that you have not already seen in this program except for `Serial.flush`. That is a member of the serial library that sends all the waiting characters out the serial port before returning to the program. Normally the serial library operates in the background but in this case I had some bugs where the program would lockup before the previous `Serial.print` statement got all of its characters to my terminal. That made it difficult to identify the line where the problem occurred. So I added a bunch of flush statements.

Note that we are only storing the Celsius conversion factor. The Fahrenheit conversion factor will be derived directly from the Celsius conversion factor by multiplying by 1.8. So of the extra allocated space comes from having completely rewritten the program several times during the development. The 36 byte number worked out well for so the extra bytes are left in there for future use.

No effort was made to optimize or reduce the size of the program because it is well under the Flash size and in theory it should only be needed to be run once. The output should look like this (*be sure you replace the variables with your own values*).



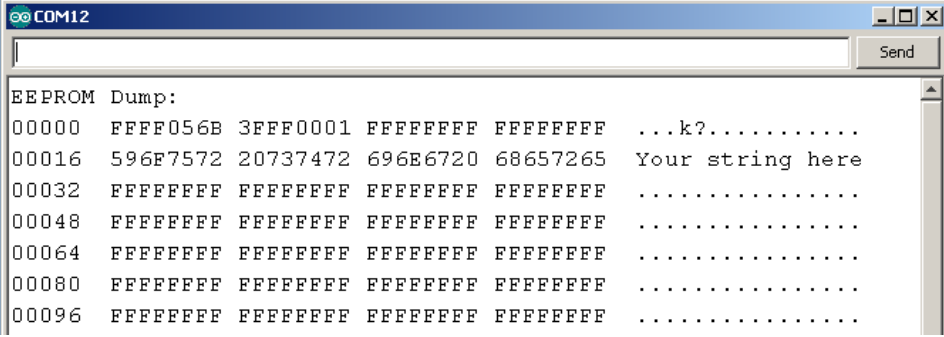
```
COM12
Celcius factor = 0.25000
multiplied by 65532 = 16383.00000
Celcius factor stored as Word= 16383 (HEX: 3FFF)

ID String: Your string here

=====

ID String read from EEPROM: Your string here
Offset read from EEPROM: 1387
Celsius factor read from EEPROM: 16383
Celsius factor converted back to floating point: 0.25000

Th-th-th-th-th- ... that's all folks!
```



```
COM12
EEPROM Dump:
00000 FFFF056B 3FFF0001 FFFFFFFF FFFFFFFF ...k?.....
00016 596F7572 20737472 696E6720 68657265 Your string here
00032 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00048 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00064 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00080 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00096 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
```

```
00960  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00976  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF .....
00992  FFFF056B 3FFF0001 FFFFFFFF FFFFFFFF ...k?.....
01008  596F7572 20737472 696E6720 68657265 Your string here
```

☒ Autoscroll Carriage return 9600 baud

Note how handily this puts the ID string into the last 16 bytes of the EEPROM.

Thermometer Program

Reporting Protocol

Now that we know how to read the internal temperature sensor and we have our calibrated constants we are almost ready to create the actual program to report the temperature from the Arduino back to the computer. We must first do a little planning. We need to define sort of temperature reporting protocol. It would probably be useful if that protocol provided for two way communications as well.

Linear Calibrated Temperature Sensor(s) Reporting Protocol

Established: September 2013 by Lewis Balentine

This Protocol is designated to be Public Domain

- A. Default communications will be via RS232 protocol at 9600 Baud
- B. All communications will be done in ASCII 7 bit characters
- C. The device will monitor the serial port for commands as specified below
- D. All commands will be Two Characters of which the first must be an Alpha character
- E. Command terminations/separators may be either a carriage return (ASCII 13) character or a new line (ASCII 10) character or null character (ASCII 0) or a tab character (ASCII 9) or a space character (ASCII 32) or any combination of the these characters
- F. Only one command is accepted at a time but additional data may be sent as required by the command. This data shall be delimited from the command by a command terminations/separator. When that data is an ASCII string then the space character (ASCII 32) is excluded from the list of valid terminations/separators within the length of the string.
- G. The following two character commands will be considered valid
 - 1. **ID** Output sensor/location ID string(s)
For multiple sensors each ID string will be proceeded by a designation digit/character, a colon and a space
 - 2. **ST** Output Status (*as applicable to implementation*)
 - a. Reporting mode, true or false
 - b. Debug mode active, true or false
 - c. Report Raw reading, true or false
 - d. Report Fahrenheit temperature, true or false
 - e. Report Celsius temperature, true or false
 - f. Internal Temperature, true or false
 - g. Minutes between readings
 - h. Reference voltage
 - i. Sensor Parameters
Repeat the ID, Offset, Fahrenheit, Celsius constants as applicable for each sensor.
For multiple sensors each ID string will be preceded by a designation numeral, a colon and a space.
 - l. If any current in memory constants have not been written to storage then include line that to that effect.
 - m. Report storage Mode flag set if it is set
 - 3. **RT** Raw True = include raw reading from temperature sensor
 - 4. **RF** Raw False = do not include raw reading from temperature sensor
 - 5. **RV** New reference voltage
 - 6. **CT** Celsius True = include degrees Celsius
 - 7. **CF** Celsius False = do not include degrees Celsius
 - 8. **C=** Celsius input. Recalculate raw reading offset based on input temperature.

- Device with multiple sensors must be placed in "Single Sensor" mode.
9. **FT** Fahrenheit True = include degrees Fahrenheit
 10. **FF** Fahrenheit True = do not include degrees Fahrenheit
 11. **F=** Fahrenheit input. Recalculate raw reading offset based on input temperature.
Device with multiple sensors must be placed in "Single Sensor" mode.
 12. **T#** Sets time between report lines where # is one of the following
 - a. **1** Report reading every 01 minute
 - b. **2** Report reading every 02 minutes
 - c. **3** Report reading every 03 minutes
 - d. **4** Report reading every 04 minutes
 - e. **5** Report reading every 05 minutes
 - f. **6** Report reading every 10 minutes
 - g. **7** Report reading every 15 minutes
 - h. **8** Report reading every 20 minutes
 - i. **9** Report reading every 30 minutes
 - j. **0** Report reading every 60 minutes
 - k. **A** Report reading every 2 hours
 - l. **B** Report reading every 4 hours
 - m. **C** Report reading every 6 hours
 - n. **D** Report reading every 8 hours
 - o. **E** Report reading every 12 hours
 - p. **F** Report reading every 24 hours

(it is intended that the data is the average temperature for the given period)
 13. **TT** Followed by data (*units, number*) for other reporting period (*not implemented*)
 14. **PF** Stop printing report lines and accept commands only (report printing mode/state)
 15. **PT** Resume printing report lines (report printing mode/state)
 16. **DB** Toggle debug mode for extended reporting
 - a. Average time for each read cycle
 - b. Number of reads cycles for each report line
 - c. Actual time for each report Line
 - d. Other information according to implementation
 17. **DO** New Degree offset for minor adjustment to temperature scale
 18. **DF** New Degree offset for minor adjustment to temperature scale (*Fahrenheit*)
 19. **DC** New Degree offset for minor adjustment to temperature scale (*Celsius*)
 20. **S:** plus command separator plus sensor designator. This command is ONLY used for devices with multiple sensors. The designator shall be a single alpha or digit character as determined by the implementation. This command selects the sensor for all following sensor specific commands until another "S:" command is received. This command essentially places the device in "single sensor" mode. To exit this mode enter the "S:" command without a designator.
 21. **L:** plus command separator plus new ID/Location string
 22. **O:** plus command separator plus new Raw reading offset (*capital O colon*)
 23. **C:** plus command separator plus new Celsius scale factor
 24. **F:** plus command separator plus new Fahrenheit scale factor
 25. **A:** Extended protocol (*zero, colon*).
These commands are specific to a given specific implementation.
 26. **WW** Write new constants to device storage
"WW" MUST be upper case!
This command (**WW:**) shall only update the working copy of the constant data
 27. **W+** Overwrite backup constant data with working constant data. "W" MUST be upper case!

28. **W**– Overwrite working constant data with backup constant data. “W” MUST be upper case!
29. **E+** This is a special mode that writes the readings to EEPROM rather than to the serial port
(*This will of course require an alternate power source*)
 - a. Set device storage flag
 - b. On the next “Reset” or “Startup”
 1. Clear device storage flag
 2. Read and store RawReading to device storage until space is exhausted
 3. Shutdown, Sleep or Resume normal operation as available in implementation
30. **E**– Clears Flag for EEPROM mode
31. **ED** Dumps data from device storage according to current conversion constants
(*debug mode ignored and all three values are output*)
32. **EC** Clears device storage area (if EEPROM writes 0xFF to all locations)
33. **AA** Extended protocol .
These commands are specific to a given specific implementation.
34. **M?** Undefined, reserved for future use by this protocol specification.
35. **N?** Undefined, reserved for future use by this protocol specification.
36. **U?** Undefined, reserved for future use by this protocol specification.
37. **X?** Undefined, reserved for future use by this protocol specification.
38. **Y?** Device specific command(s) (*implementation specific*).
39. **Z?** Device specific command(s) (*implementation specific*).
40. **LL** Output list of device implemented commands
Each line shall be prefixed with semicolon and space
The first line shall include device Identification and/or serial number
The required output is the 2 character commands
Optionally each line may include a short description
41. **??** Same as **LL**
42. **SS** Shutdown or Sleep (*implementation specific*).
This command MUST have two consecutive calls.
The device will respond with “; **SHUTDOWN**” or “; **SLEEPING**” as applicable.
43. **00** Turn rounding on or off (*implementation dependent*)
44. **!!** Reset or reboot device (*that is two exclamation marks*)
This command MUST have two consecutive calls.
The device will respond with “; **RESETTING**” (*implementation limited*)
- H.** The device/application may implement any set or subset of the commands that include the following commands: **ST, CT, CF, FF, FT, T1, T2, T3, T4, T5, ??**
- I.** Any response line from the device that is NOT a report line shall be prefixed with a semicolon “;” and a space.
- J.** Valid commands that do not otherwise generate responses shall respond with the two character command plus space plus “**OK**”.
- K.** If the device receives a command it does not recognize then it will respond with “??”.
- L.** If the device receives a command it recognizes but is not implemented then it may respond with either “**XX**” or “??” but “**XX**” is preferred.
- M.** Commands with a terminating colon may be used for multiple sensors by replacing the colon with a numeral to identify the sensor number.
- N.** Report lines from the device shall consist of the designated data fields separated by a tab character (ASCII 09) in the following order:
 1. Raw reading
 2. Calibration corrected Celsius temperature
 3. Calibration corrected Fahrenheit temperature
 4. Extended debugging data as defined above

That should be enough to confuse the issue. Our device currently only has one sensor but the protocol makes provisions for multiple sensors (*Engineering is the art of "Planning and Forethought"*). Tab characters (ASCII 09) are one of the commonly used delimiters for text files. This makes it easy to import the data file into a spreadsheet program or database for charting and/or analysis. The semicolons prefixed to the devices responses may it easy to strip out those lines from the data file or signal the receiver application that this is NOT a normal reporting line. If the PC application includes device commands in its output stream and/or log then it should prefix these with a semicolon and a space as well. Although the protocol specifies all upper case characters for command characters it is recommended that the device application accepts either upper or lower case or a combination of both with the exception "W" commands. The reset command "!!" (*that is two exclamation marks*) is intended to be used for "If all else fails then abort and start over". The reset command may also be used as an entry point to update the device software (*depending on the reset characteristics of the device*).

The commands "A:", "A?", "X?" and "Z?" (*the "?" is wild card that is to be interpreted as any character*) are intended to be used to extend the protocol as may be required for a specific sensor(s) while keeping the functionality of the basic protocol in place. This allows for a standard reporting application to use sensor(s) with extended capabilities. However an extended reporting application specific to the implementation may be created to take advantage of the additional features (*for example "wet" and "dry" bulbs or a humidity sensor*). Any command beginning with the letter "M", "N", "U" or "X" is defined to be undefined and reserved for future use by this specification.

Thermometer Program, Plan “A”

READ BEFORE YOU PROCEED !!!

Plan “A” to use the internal temperature was a dismal failure.

The program code does however provide the Basis for Plan “B” which is a total success.

This section we will discuss the structure of the program, the variable usage and specific details of the Implementation. It will also list all the functions and describe their use. The program code is divided into two files. The first “ThermometerOne.ino” has the declarations, setup function, loop function, command processor function and those functions directly related to parsing commands codes. The second file named “ThermometerOneFunctions.ino” contains the code to implement the various sections of the protocol. The two source files together are a bit over 50 Kbytes. They are fully included in the

Appendix: Thermometer One Program Code. Note that much of the “debugging code” has been left in place but commented out to provide examples of the debugging methods used (*these lines should be obvious*).

Main File Functions

Global Declarations

Includes

The first items listed are the “#include”s. In order to implement the sleep/shutdown function we need the resources provide by avr/sleep.h. EEPROM.h is included so that we can read and write to the EEPROM. Our own library “HexDecAsc.h” is included to implement a device specific function that dumps the EEPROM contents to the serial port. This was needed for debugging purposes.

EEPROM address

The first group of these is the constants that identify specific locations where conversion constants and operational parameters are to be stored. There are two complete 36 byte copies of this data: one at the start of the EEPROM and one at the end. The first is designated as the working copy. The last is designated as the backup copy. If the first section should ever “wear out” (*as in more than 10,000 writes*) then the two sections can be swapped by redefining their locations.

The second group is a set of four word variables used by the application to delineate the beginning and ends of the sections of the EEPROM where reading are to be written to or read from during storage mode operation. The implementation uses these in such a way as to spread the writes out across the entire range. This is referred to as “wear leveling”. With an ATmega329P this results in 952 bytes of data storage $[1024 - 2 * (36) = 952]$. In a best case scenario this is adequate for 7,600 readings or to put it another way: one reading every 5 minutes for 26 days. A more reasonable estimate would be somewhere around 15 days.

Conversion Factors/Calibraton Data

These variables are used to store the factors needed to convert from raw readings of the virtual 12 bit ADC to actual temperatures in degrees Celsius and Fahrenheit. It also includes a variable used to determine if any of these have been changed in the current session. Two constants are also defined for the temperatures at the raw reading offset.

Global operational mode Variables

A number of modes (*or machine states*) are defined. These Boolean variables are used to indicate if a given mode is active. In some cases multiple modes may be active at the same time.

Global work Variables

These are global static working variables that are available to all functions. For the most part they deal with timings and command handling. The variables “LastRead”, “Consecutive” and “gap” are defined in this section in order to preserve their values between calls to the functions that use them.

Setup() Function

The setup function does exactly what it is intended to do: configure the hardware and operational parameters. The parameters are read from the EEPROM and default timing numbers and operation modes are established. Then if the device is NOT in EESTORAGE mode the operational status is sent to the serial port before the main loop begins.

Loop() Function

The loop function continually cycles through three tasks. The first task is to determine if there is a possibility of a command waiting in the serial buffers. If there is then it calls a function to read the command and sends any valid result to the command processor.

The second task is to collect temperature data. The variable “gap” is used in this task to regulate how often this task is performed.

The last task is to determine if it is time to report or record data. A delay of up to 120 milliseconds has been incorporated in this task in order to make the reports as periodic as possible. In most cases there is less than plus or minus 1 millisecond error however if there has been any command processing then all bets are off. Things like changing the report timing in mid-stream completely negate the precision of the report trigger for the current report cycle.

CmdProcessor() Function

This is the main control function to implement the protocol. It accepts a two character command and redirects the application to the appropriate function. If the “debug mode” is active then the two letter command is reported to the serial port as well. This feature is intended to be used in the development process of an application for the receiving end. Examples of non-implemented commands are also included.

At the bottom of this function there are four application specific commands defined. “Z1” and “Z2” insert sample data into the EEPROM. Although these were written and used for debugging purposes they can also be used to initialize a virgin device. The “ZZ” command changes the number of seconds per minute to 10 in order to reduce the reporting period. It also sets all the modes so that only raw readings are reported. This is intended to be used to collect data for a two point calibration.

The “ZD” command uses the “HexDecAsc” library to dump the entire contents of the EEPROM to the serial port. This was used extensively in debugging the EEPROM writes and reads.

HelpMe() Function

This is the companion to the CmdProcessor function. It dumps a list of all implemented commands to the serial port. It was placed directly below the CmdProcessor function in order to make it easier to keep the two in sync. Notice the extensive use of the F() macro in this function. Without it the program had a tendency to lock up and crash during the development. The use of this macro has definite benefits.

PrintSeperatorLine() Function

Some of the list functions such as the HelpMe function above print a line of dashes at the beginning and end of their output. The function provides a way to accomplish that without having to insert a line of dashes in those functions thus conserving program space.

ReadTwoCharacters ()Function

This is the main input function. It retrieves characters from the serial port, strips off command terminators and capitalizes the alpha characters (*with the exception of ‘w’*). It also saves the previous command and incorporates a time limit. If a complete command is not received within 250 milliseconds then it aborts and program control returns to the loop() function. Under more ideal condition the function only requires on the order of 15-20 milliseconds. If it does detect a valid command then it drains the serial stream of any remaining terminator characters. In order to provide for the maximum input flexibility the definition of terminating characters is extremely liberal. In either case it returns a Boolean value to indicate if a valid command has been received.

Note: There is a separate input handler for the ID string that does not use a space as a command terminator.

DrainCmdTermiantors() Function

This function is used to drain the serial stream of any remaining command termination characters.

DebugPrintCharacters() Function

This was the principle debugging method used for the ReadTwoCharacters function. The function has been overloaded so that it could be used to print CMDs received by other functions such as the main reporting function. The CmdProcessor used this function in debugmode.

Thermometer Functions File

EnableADC() Function

This function is called by the setup routine to configure the ADC unit. It is probably not required because it essentially replicates the default settings. This insures that something (*i.e. an unusual boot loader or previous program*) has not redefined those parameters. The required defines are include just above the function definition.

Read_Calibration_Data() Function

This function is called by the setup routine to retrieve operational parameters and calibration data from the EEPROM. It is also called when some of those parameters are written to the EEPROM (*the exception being the EEMODE flag*).

Write_Calibration_Data() Function

This writes operational parameters and calibration data from memory back to the EEPROM working data storage. It only writes those parameters that are flagged as being new. Note that it also clears the EEMODE flag. This command must be called manually using the “WW” command and the “WW” must be upper case (*most commands may be upper or lower case*).

ClearStorage() Function

This function is used to clear the EEPROM storage area between the working and backup copies of the parameter data. It does a read before write to avoid excessive wear on the EEPROM. It is called by the Check_EEPROM setup function but it may also be called using the “EC” command. Each read requires 4 machine cycles. Each write requires 2 machine cycles. Thus a single byte may require 6-8 cycles to process.

EEmodeFlagSet() Function

This function is used to set the EEMODE Flag so that at the next reset or restart the application will write to the EEPROM rather than report its data to the serial port. This was originally written as simple set to non-zero value. That could have caused to excessive to the low order bit if the device is frequently used in EEPROM mode. The nature of way the EEPROM work dictate that it is “erasing a zero bit” that cause wear.

<http://electronics.stackexchange.com/questions/21232/100k-eeeprom-writes-per-bit-or-as-a-whole>

It's not just write cycles that's specified, but erase/write cycles. On the AVR EEPROM can be erased by byte. Erasing sets all bits to 1, writing selectively clears bits. You can't program a 1, just 0s. If you want to set at least one bit to 1 you have to erase that byte.

Erasing removes the charges from the FET's floating gate, but on each erase cycle some of the charge remains on the floating gate, which won't be removed through the quantum tunneling. This charge accumulates and after a number of cycles there's so much charge left on the floating gate that the bit still will read 0 after erasure. That's what determines EEPROM life, it's erasure rather than writing. So you can safely write additional 0s, as long as you don't erase.

So the function was rewritten to compare two bytes a mask and a flag byte. A zero bit is rotated through the bit positions of the flag byte to set the EEPROM flag. To clear the EEPROM flag the mask is set even to the flag. Thus there is only one write to each byte for each EEPROM mode cycle and zero erase are evenly spread across both bytes. In theory this may increase the life of the EEPROM flag (*100,000 E/W cycles*) by a factor of 16 . It will at the very least double the life. This function must be manually called by the “E+” command.

EEmodeFlagClear() Function

This is the complement to the EEmodeSetFlag function. It is automatically called in the Check_EEPROM setup function and may be manually called by the “E-” command.

EEmodeFlagTF() Function

This function is used to check the EEMODE flag and returns true or false.

Check_EEPROM() Function

This function is called by the main Setup function to check if the current run is designated to go to the EEPROM. If the EEMODE flag is set then the EEPROM mode is set true and the Report and Debug modes are set to false. The function then determines the End of the last EEPROM session. It then uses ClearStorage to erase the entire storage area. A marker of two 0 bytes is written to the EEPROM to indicate the start of the current session. The storage variables are set and the function exits.

Print_IdString() Function

This function Outputs the current “ID/Location” string to the serial port. This function is called by the Status function or it may be called manually by the “ID” command.

PrintTrueFalse() Function

This function Outputs the “True” or “False” to the serial port. It is only used by the Status function.

ReportStatus() Function

This function Outputs the current operation status and parameters to the serial port. It is called by the main setup function and may be manually called using the “ST” command.

avrRawTemp() Function

This is the function called by the main loop to read the internal temperature sensor. It is a clone of the similar function presented in the previous sections that uses 16 consecutive 10 bit ADC reads to produce a virtual 12 bit reading. The main difference is a fixed number of 64 reads using in line constants. These changes help reduce the cycle time for the function.

This is the function that would need to be rewritten if this application is used for a different temperature sensor or for multiple sensors.

Convert()Function

This function is called by the reporting functions to convert Raw Readings the Celsius and Fahrenheit. It rounds these numbers to the nearest ¼ degree for Celsius and ½ degree for Fahrenheit.

Report() Function

This is the function called by the main loop to send output to the serial port or the EEPROM. It first calculates the average temperature for the report period. Then if the EEPROM mode is active then it calls the secondary function Report2EEPROM otherwise it write to the serial port. If debug mode is active it also reports the cycle time and number of readings. This feature was used to derive some of the timing constants.

QuickBlink() Function

This function blinks the LED on digital pin 13 for 2 milliseconds. It is used in EEPROM mode to indicate report cycles, writes and shutdown.

Report2EEPROM() Function

This is the function that is used to store data in the EEPROM. It implements both wear leveling and data reduction. The wear leveling is done by beginning each new session where the old session ended. This is controlled by the Check_EEPROM function. The data reduction is based on the concept that there are typically a

number of consecutive readings that are the equal. The function counts up to 16 consecutive readings and stores the count in the high nibble of the word that it writes to the EEPROM. This increases the amount of data that can be stored and reduces the number of writes thus increasing the life of the EEPROM as well. It is important to note that the range of readings recorded will never exceed 2047. This insures that there will always be a high order zero in the reading and thus no word written to the EEPROM storage will ever be all ones. That is important because areas with all ones are considered to be unused. When the function reaches the end of the available storage it wraps around to the beginning (*this has NOT been extensively tested due to the extended time periods required*). When it reaches the beginning mark the sleep/shutdown function is called.

DumpStorage() Function

This is the compliment to the Report2EEPROM function. It locates the beginning of the last recorded session by searching for the two zero bytes marker. It then decodes the data and dumps it to the serial port using the current reporting constants. It dumps all three temperatures: Raw, Celsius and Fahrenheit separated by tab characters. Lastly it reports the number of readings and the amount of storage used. This function must be called manually using the “ED” command. There are a lot of example debugging lines in this function that are commented out.

PrintOKStr() Function

This function prints the current two letter command followed by “OK” to the serial port. Various command function use this function to acknowledge the command was accepted and executed.

PrintNotRecognized()Function

This function prints the current two letter command followed by “??” to the serial port. Various command function use this function to indicate that the command was not recognized.

PrintNotImplemented() Function

This function prints the current two letter command followed by “XX” to the serial port. Various command function use this function to indicate that the command is not implemented.

ShutDown() Function

This command is used to cease execution of the program. It disables interrupts, disables the ADC and places the ATmega328 into “sleep power down” mode. This is as close to shut down as we can come. This mode still uses nearly 10 milliamps of power due to the inefficient 5 volt voltage regulator.

<http://playground.arduino.cc/Learning/arduinoSleepCode>

Sleep is commonly used to save power on Arduino boards. For some Arduino variants, however, there is not much benefit. For example, the Arduino serial and USB boards use a 7805 type of power regulator, which needs 10mA when the Atmega IC is in idle mode. Putting these boards to sleep will cut a few mA off the total power consumption however it will still be high.

An application specific designed device could greatly reduce the power usage but in that case one might also want to revisit the data collection and command processor routines to reduce power between those as well. Power reduction was not considered in the software design as this application is primarily intended to be used with the device connected to computer. This function may be manually called using the “SS” command.

software_Reset() Function

This function may be called to restart the application from the beginning. The implementation method uses a simple assembly call to the zero vector. This does NOT reboot the device. A better implementation would be to use the vector to the boot loader however the boot loader would need to be checked for compatibility. It is expected that the “OptiBoot” would work well in this manner. However for this application the failsafe approach was chosen. This function may be manually called using the “!!” command.

This function is intended to be used in the case "If all else fails". As such it should be rewritten using some kind of interrupt handling so that one may recover from an infinite loop. This may require modifying the Serial library.

SetRawReadMode() Function

This function sets the raw reading report mode to true or false. This function may be manually called using the "RT" or "RF" commands. Default is true.

SetFahrenheitMode() Function

This function sets the Fahrenheit report mode to true or false. This function may be manually called using the "FT" or "FF" commands. Default is true.

SetCelsiusMode() Function

This function sets the Celsius report mode to true or false. This function may be manually called using the "CT" or "CF" commands. Default is true.

SetReportMode() Function

This function sets the report mode to true or false. This function may be manually called using the "RT" or "RF" commands. Default is true.

ToggleDebugMode() Function

This function sets the toggles the debug reporting mode between true and false. This function must be manually called using the "DB" command. Default is false.

NewReportTime() Function

This function is used to change the time between report lines. Note that only the times specified in the protocol are implemented (*i.e. TT is not implemented*). This function may be manually called using the "T#" command where "#" is in the set "1,2,3,4,5,6,7,8,9,0,A,B,C,D,E,F". Default is "1".

Report_Reset() Function

This function reset the report line variables and report timing variables. It is normally called by the NewReportTime function but may be called by other functions as well (*i.e. when new parameters are written to EEPROM or restore from backup*).

NewIdString() Function

This function reads an ID/Location string from the serial port and stores in active memory. The function incorporates a 5 second timeout to read the value. Failure to provide the data in a timely manner will abort the command. This command includes internal code to read data from the Serial port. It does not capitalize the string or terminate when a space is received. It will terminate if more than 24 printable ASCII characters are received. If a value is received and recorded then the new data flag is set as well. Command terminators are drained from the serial stream in either case. This function must be called manually using the "L:" command.

As I am reviewing the code for this function it seems that I neglected to include code to drain any extra printable characters from the serial stream. That oversight will be corrected.

NewOffset() Function

This function reads a new raw reading offset from the serial port and stores in active memory. The function waits 2 seconds for the data to enter the serial stream before attempting to read the value. Failure to provide the data in a timely manner will abort the command. If a value is received and recorded then the new data flag is set as well. Command terminators are drained from the serial stream in either case. This function must be called manually using the "O:" command.

CelsiusEquals() Function

FahrenheitEquals() Function

These two functions are the primary way a user will calibrate the device. They take the user's input to recalculate the raw reading offset for 20 degrees Celsius (68 degrees Fahrenheit). This is accomplished by multiplying the difference between the input temperature and the offset temperature by the temperature conversion factor. The result is subtracted from the current raw reading to produce a new raw reading offset.

NewCelsius() Function

This function reads a Celsius conversion factor from the serial port and stores in active memory. The function waits 2 seconds for the data to enter the serial stream before attempting to read the value. Failure to provide the data in a timely manner will abort the command. If a value is received and recorded then the new data flag is set as well. Command terminators are drained from the serial stream in either case. This function must be called manually using the "C:" command.

NewFahrenheit() Function

This function reads a Fahrenheit conversion factor from the serial port and stores in active memory. The function waits 2 seconds for the data to enter the serial stream before attempting to read the value. Failure to provide the data in a timely manner will abort the command. If a value is received and recorded then the new data flag is set as well. Command terminators are drained from the serial stream in either case. This function must be called manually using the "F:" command.

The three functions above use commands included in the Serial library to read floating point or integer values. At some point that code should be replaced to incorporate a timeout and insure that any extra characters are properly dealt with.

RestoreFromBackup() Function

This overwrites the working parameter storage with data that has been saved in the backup copy. This function must be called manually using the "W-" command and the "W" must be upper case (*most commands may be upper or lower case*).

OverwriteBackup() Function

This is the compliment to the RestoreFromBackup function. This overwrites the backup copy with data from the working parameter storage. This function must be called manually using the "W+" command and the "W" must be upper case (*most commands may be upper or lower case*).

TestData1() Function

TestData2() Function

These two functions replace data in the working EEPROM storage a set defined within the function. These two functions were originally written to test the Backup and Restore functions. They can however be used to initialize a virgin device as well. These functions must be called manually using the "Z1" or "Z2" commands.

CalibrationMode() Function

This function is used to gather information for new calibration constants using the three point calibration method. It disables all reporting except for raw readings and set the time period to 1 minute. Then it lies to the AVR by telling it the each minute is only ten seconds long. The result is a raw reading every 10 seconds. The "newflg" is set to zero so that none of these parameters are accidentally written to the EEPROM. This function must be called manually using the "ZZ" command.

void EepromDumpAll() Function

This function dumps the entire EEPROM to the serial port 16 bytes at a time in Hex and ASCII. It uses a set of library functions that was presented previously in this document. The only difference is that it prefixes each line with a semicolon and a space"; ". This function must be called manually using the "ZD" command.

-----and that ladies and gentlemen concludes our presentation for this evening-----

Temperature Calibration Theory

The internal temperature sensor is designed to generate a linear output. That is to say that for each 1 degree increase in temperature the sensor outputs an equal increase in voltage regardless of the temperature. That ratio is approximately 1 mV (*millivolt = 1volt/1000*) per degree Kelvin. However, due to the process variation in manufacturing the temperature sensor output voltage varies from one chip to another. The sensor was intended to be used to determine if there was a problem that caused the MPU to overheat. Thus lack of accuracy and the variation between devices is acceptable for its intended use.

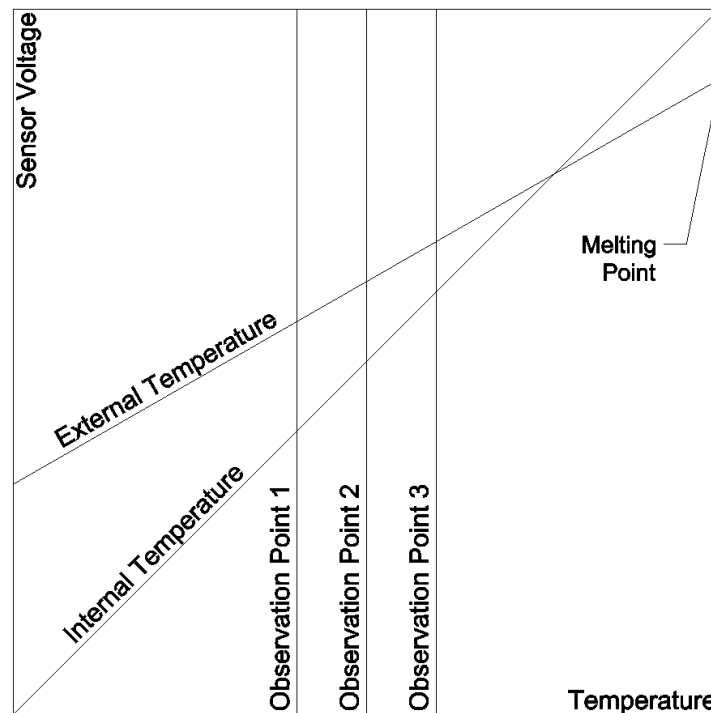
There are several additional factors that may affect the reading that is returned by the internal sensor. The three principle ones are:

- 1) Variations in the power supply
- 2) Variation in the amount of current the MPU is using
- 3) The ability of the MPU to dissipate the internally produced heat to the environment around it

In this application the power supply will always be the USB port from the computer that the Nano is attached to. Thus the power supply should be well regulated and constant (*unless the computer goes to sleep*). The task of the MPU will be limited to reading temperature sensors and reporting back to the computer at regular intervals. Thus the amount work the MPU is doing and the amount of current that it is using should be constant as well. The case would be much different if the same device was being used to do something such as control a group stepper motors via its PWM facility.

That leaves the ability of the MPU to dissipate heat. The MPU's heat is absorbed by the environment around it via radiation and conduction. An increase in the environment temperature makes it more difficult to dissipate that heat and in turn causes an increase in the internal temperature of the device. It is this cause and effect relationship that we are attempting to exploit. Assuming there are no other outside factors (*i.e such as being in a location that alternates between bright sunlight and shadow*) then for any given air temperature there should be a given MPU temperature. The difference between these two temperatures what we are calling "offset".

The question becomes is this offset equal for all temperatures. Clearly the answer is no. Consider the extreme case if the Arduino was placed inside a high temperature furnace. Rather than dissipating heat it would absorb heat and eventually melt (*the melting point of silicon is 2,577 degrees Fahrenheit or 1,414 degrees Celsius*). We are hoping that over the relative small range of interest that the two lines are approximately linear and closely parallel. The only way to establish the exact offset at any temperature is to make a physical observation. It is impractical to do this for every possible temperature. Thus we will choose three convenient observation points.



The observation points must be within the fairly wide operating temperature range of the MPU: -55 degrees Celsius to 125 degrees Celsius (*-67 degrees to 257 degrees Fahrenheit*). Observation point 1 should be as close to 0 degrees Celsius (*32 degrees Fahrenheit*) as possible. The second observation be close to 20 degrees Celsius (*68 degrees Fahrenheit*). The third observation needs to be over 38 degrees Celsius (*100.4 degrees Fahrenheit*). A “gain” factor can then be produced from the readings at observation points 1 and 3:

$$\text{Gain} = (T3 - T1) / (V3 - V1)$$

Where:

T3 = the temperature at Observation point 3

T2 = the temperature at Observation point 2

T1 = the temperature at Observation point 1

V3 = the sensor voltage reading at Observation point 3 (0 to 4096) – raw reading

V2 = the sensor voltage reading at Observation point 2 (0 to 4096) – raw reading

V1 = the sensor voltage reading at Observation point 1 (0 to 4096) – raw reading

From those new observations we will come up with a new simple formula for calculating the temperature scale factor:

$$\text{Scale Factor} = (T3 - T1) / (V3 - V1)$$

Temperature Calibration Procedure

I have six addition thermometers at my disposal:

- 1) Honeywell Electronic HVAC thermostat (unknown)
- 2) Six inch Taylor “Confortmeter” spirit bulb wall thermometer (-35 to 55 Celsius)
- 3) Taylor pocket bi-metal stem one inch dial thermometer (0-220 Fahrenheit)
- 4) Omega SST armored twelve inch fractional calibrated partial immersion photographic spirit bulb thermometer (5 to 55 Celsius)
- 5) BCR six inch full immersion mercury bulb thermometer (-50 to 50 Celsius)
- 6) Mastech MS826T Multimeter with Type K sensor (-20 to 400 Celsius)

No two of the above register closer than two degrees across their effective range. The Fractional Omega (*the most expensive of the lot*) is the worst in that it consistently reads at least ten degrees lower than any of the rest. The Mastech, with its probe frozen in a block ice, reports between 5 and 6 degrees Celsius. It was this inconsistency that instigated this project.

Ultimately I had to choose one as a point of reference. The BCR was only one that I had any confidence in. In addition I determined that the Mastech about a 2.5 degree error at 25 degrees Celsius but otherwise was consistent with the BCR mercury thermometer at ambient temperatures.

Temperature Calibration Procedure: Observation Point 1

["This porridge is too cold," she said.](#)

Ideally one would use shaved ice made from distilled water to pack both the Nano and the thermometer. I just be used a bag of ICE that was acquired at a local grocery store. I placed this in a plastic dishpan of water along with a full immersion 6 inch BCR mercury bulb thermometer. The Nano was placed inside two plastic bags and immersed in the water along with the thermometer. Both were isolated from the bag of ice. After the reading has stabilized both the temperature and the raw reading was recorded. I note there was a lot of condensation inside of the first plastic bag. The inner bag was an anti-static bag and did not show any condensation problems (*much smaller volume as well*).

Raw reading = 1343

Temperature = 6.75 degrees Celsius

Temperature Calibration Procedure: Observation Point 3

["This porridge is too hot!" she exclaimed.](#)

This was a little more difficult. For this I used a small cardboard box (6.5 x 4 x 1 inches). The Nano and the 6 inch BCR mercury bulb thermometer were mounted in the box. The box was place in an Oven (*standard electric cooking range*) that had been preheated to above 50 degrees Celsius and allowed to cool down to 50 degrees Celsius (*because that is the top of the range of the BCR thermometer*). The Thermometer One program was placed in "Calibration mode". The oven door was closed and the system was allowed to reach equilibrium. A reading was recorded from the program then the door was opened and the thermometer read very quickly. This was done several times to insure consistency.

Raw reading = 1511

Temperature = 44 degrees Celsius

Now we can calculate the scale factor:

Scale Factor = $(T3-T1) / (V3-V1)$

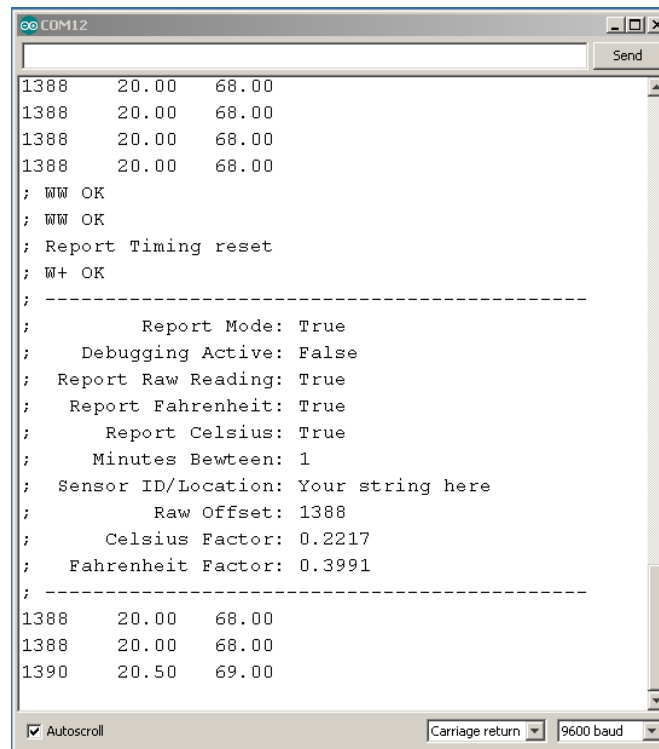
Scale Factor = $(44-6.75) / (1511-1343) = 37.25 / 168 = 0.2217$

That is slightly less than the theoretical 0.2500 scale factor.

Temperature Calibration Procedure: Observation Point 2

["Ahhh, this porridge is just right," she said happily and she ate it all up.](#)

For this point the HVAC system was set to bring the temperature down to 20 degrees Celsius. The "C:" command was used to plug in the new scale factor and the "C=" was used to set the offset.



The screenshot shows a serial terminal window titled 'COM12'. It displays a series of data rows and configuration parameters. The data rows show raw readings (1388, 1388, 1388, 1388, 1390) and corresponding Celsius (20.00, 20.00, 20.00, 20.00, 20.50) and Fahrenheit (68.00, 68.00, 68.00, 68.00, 69.00) temperatures. Configuration parameters include Report Mode: True, Debugging Active: False, Report Raw Reading: True, Report Fahrenheit: True, Report Celsius: True, Minutes Between: 1, Sensor ID/Location: Your string here, Raw Offset: 1388, Celsius Factor: 0.2217, and Fahrenheit Factor: 0.3991. The window also has a 'Send' button, an 'Autoscroll' checkbox, and dropdown menus for 'Carriage return' and '9600 baud'.

```
COM12
1388 20.00 68.00
1388 20.00 68.00
1388 20.00 68.00
1388 20.00 68.00
; WW OK
; WW OK
; Report Timing reset
; W+ OK
; -----
;       Report Mode: True
;       Debugging Active: False
;       Report Raw Reading: True
;       Report Fahrenheit: True
;       Report Celsius: True
;       Minutes Between: 1
;       Sensor ID/Location: Your string here
;       Raw Offset: 1388
;       Celsius Factor: 0.2217
;       Fahrenheit Factor: 0.3991
; -----
1388 20.00 68.00
1388 20.00 68.00
1390 20.50 69.00
[Autoscroll] [Carriage return] [9600 baud]
```

So the final question is: “How accurate is it?” Not off by at least 5 degrees at 30 degrees Celsius.

Plan “A”, Evaluation and Summary

- 1) A number of Arduino boards were successful programmed with an application to report the temperature back to the computer at periodic intervals. That item is rated as a “success”.
- 2) The Arduino application(s) that were developed have the desired level of capabilities for two way communications, adjustments, storage and reporting. That is rated as a “success”.
- 3) The applications has been designed and implemented without any modifications or additions to any of the Arduino board. That is item rated as a “success”.
- 4) The design of the application and protocol are such that modification of the application to use a different temperature sensor should only require changing one function and the calibration factors. That item is rated as a “success”.
- 5) The design of the application and protocol are such that modification of the application to support multiple temperature sensors is possible. This would require implementing “Single Sensor” selection protocol as well as changes to the reporting functions. Until these actions have been attempted and completed successfully that is rated as “questionable”.
- 6) The ability to extend the range of the ADC from 10 bits to 12 bits via “Oversampling” and “Decimation” have been demonstrated and confirmed. While the limitations of having “appropriate” noise available is a limiting factor it does provide an interesting alternative. That item is rated as a “success”.
- 7) While the linearity of the internal temperature sensor may be without question what we are actually attempting in this application is to quantify the Arduino’s ability to dissipate its internally generated heat. The linearity of that function and its relationship to the function of the internal temperature sensor remain questionable. The methods used in the application were not able to exploit this relationship. Accuracy could not be obtained. That item is rated as a “failure”.

To say plan “A” is a dismal failure is being kind. Plan “A” is a loser!

Thermometer Program, Plan “B”

In order to proceed with this project I had to give up the concept that no external parts or hardware modifications would be required. If the internal temperature sensor cannot be utilized for this purpose then an external one must be used. Several alternatives exist:

- K Type thermocouple
- SMBus/I2C temperature sensor
- Analog devices AMD590 2 wire variable current output temperature sensor
- Maxim DS18S20 1-Wire Digital Thermometer
- 10K Ohm at 25° C 1% thermistor and 10K 1% resistor (*look for these on Ebay.com*)
- National Semiconductor LM34 Analog Temperature Sensor

The last two of these are probably the more practical approaches. The thermistor approach is probably the least expensive but it would require a table driven conversion routine. The Maxim or SMBus is probably the simplest if one can obtain the part in a suitable package. I chose the last one.

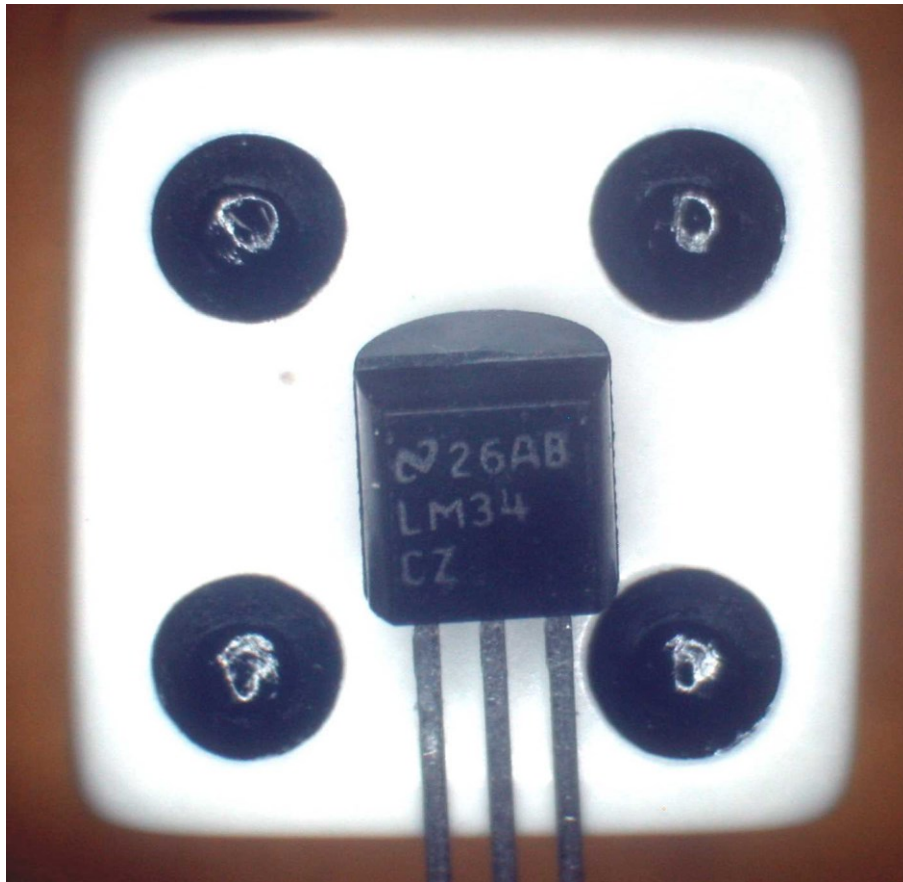
We will need to dump all the scale and offset stuff that was done previously and replace it with a more traditional ADC voltage conversion. That will entail changing some of Global variables as well. As we will still be using the internal 1.1 Voltage references we can still reference the internal Temperature sensor as well. An additional 40 bytes of the EEPROM will also be set aside for a future table based conversion routine. That could be useful for using a thermistor as a temperature sensor. At the same time we are going to do a little work on reducing the size of the program. Most of the program remains the same so in this section we are only going to cover the additional hardware, EEPROM locations, Global Variables and functions changes. The full program code is in the Appendix: Thermometer One Program Code (Plan “B”).

External Temperature Sensor: LM34

I had several of these on hand for another future project acquired via Amazon.com for US\$6.00 each. The specific part number I have is:

National Semiconductor LM34CZ Analog Temperature Sensor

The picture below shows the part sitting on top of a common 16mm dice like one might find in a board game or on the dice tables in at casino. As it is shown in the picture the lead on the left is for power (5 Volts DC). The lead on the right is the ground connection. The center lead is for the analog output signal.



Power << Signal >> Ground

A LM34DZ would work also (*and it is the least expensive part in the series*). Those may also be acquired from Avnet, Arrow, DigiKey, Newark Electronics, Jaemco, other electronic supply houses as well as EBAY. At electronic supplies they typically cost between US\$2.00 to US\$2.50. I have seen them on EBAY for less than US\$2.00. You can also pay more than US\$15 for parts in this series (LM34CAH). The difference is in the Temperature scale, accuracy and the packaging. The datasheet (*see appendix*) covers the various options but here is the simplified version:

Part Number	Scale	Low	High	Accuracy	Typical
LM34	Fahrenheit	-50	300	2 Degree	0.8
LM34A	Fahrenheit	-50	300	1 Degree	0.4
LM34C	Fahrenheit	-40	230	3 Degree	1.6
LM34CA	Fahrenheit	-40	230	1 Degree	0.4
LM34D	Fahrenheit	32	212	3 Degree	1.2

Add a one letter suffix for the packaging:

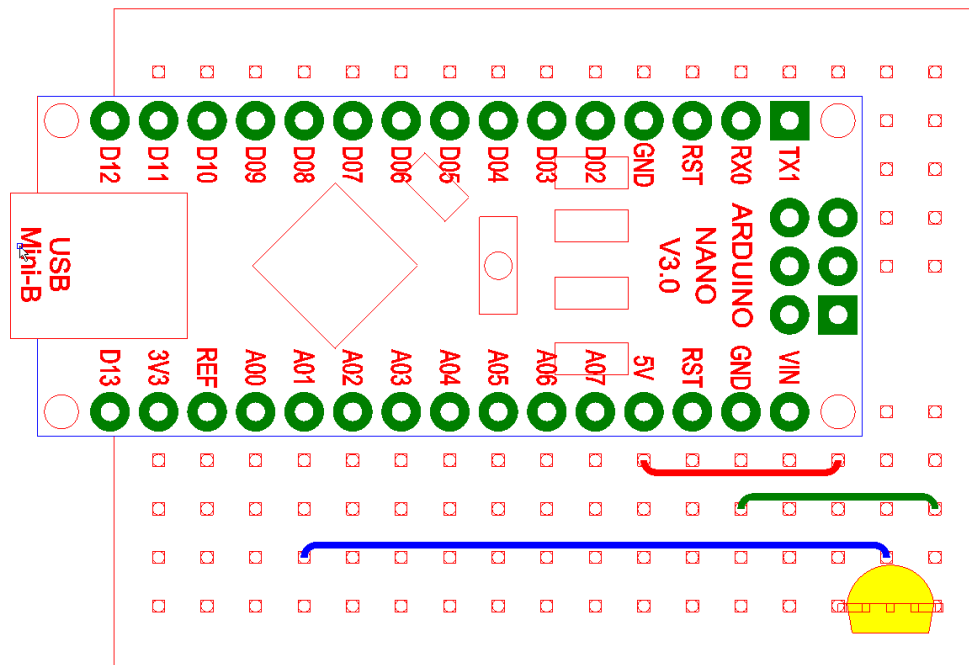
Suffix	Package
H	TO-46 Hermetic Sealed Metal Can, 3 through hole leads
M	SO-8 Small Outline Surface Mount, 8 legs
Z	TO-92 Plastic Transistor Package, 3 through hole leads

The nice thing about these devices is that they are Linear Analog Temperature very similar to the AVR internal sensor. The big difference is that they report 10 millivolts per degree Fahrenheit and they are guaranteed to be within 1 degree of accuracy at 77 degrees Fahrenheit. They typically expect better than 1/2 degree accuracy. As they are so similar to the internal sensor only minor differences will be needed in the software to switch between the internal temperature sensor and the external temperature sensor.

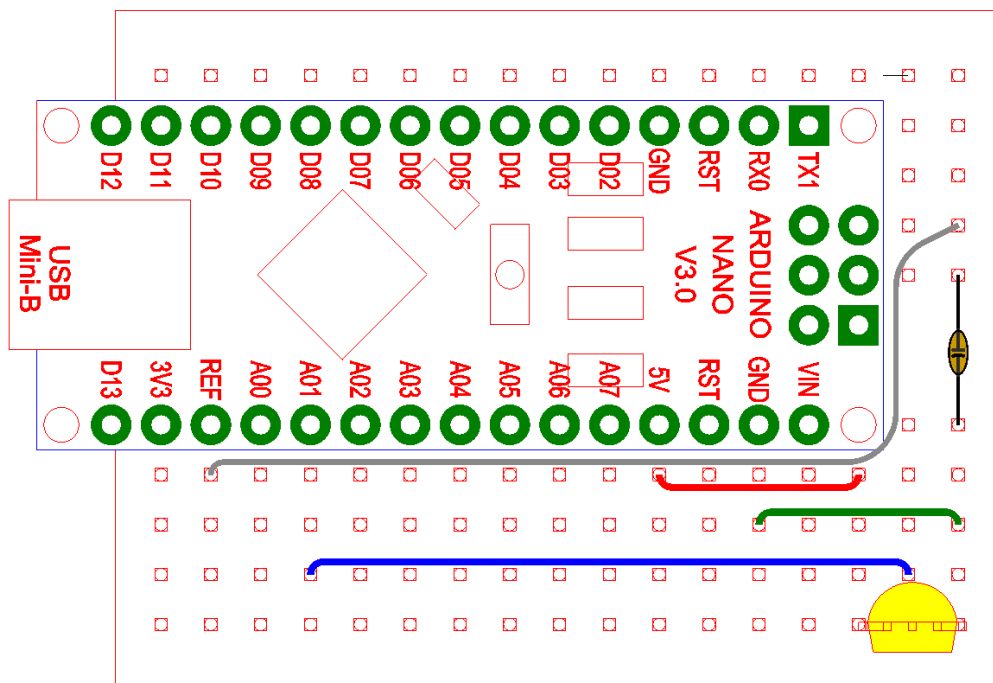
The most practical package for our use is the plastic TO-92. The most limiting factor is that the main conduit of heat to the sensor is the three wires it uses for leads. With the TO-46 metal can the main conduit is the can itself. It also offers the advantage that it can be soldered or cemented to a heat sink or other heat conducting medium. It is however the most expensive packaging.

Note: A similar series of parts exist for the Celsius scale. That is the LM35 series.

I used a 175 tie point breadboard. I cheated a bit by mounting the Nano such that one set of pins are not in the breadboard. That is not a problem in this case because we are not going to use those pins in this application. That left three rows of tie points at the other end which all we need for the sensor. Here is the layout.



Atmel also recommends putting a capacitor on the Analog Reference pin (REF) to reduce noise. As an option you can run a wire from the REF pin around to the other side and stick a small disk capacitor in the middle. I used a 22 Pico farad ceramic disk capacitor (*because that is what I had available*). I cannot tell that it has made any difference.



EEPROM Layout

Starting from the top lets first look at the new EEPROM layout:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
mask	flag	ref volts		offset		minutes		unused		unused		unused		unused	
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ID String 16 Bytes long w/o termination															
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Reserved space for future Table															
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Reserved space for future Table															
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Reserved space for future Table															
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
Reserved space for future Table															
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
Reserved space for future Table															
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Begin		EEPROM		Mode		Data		Storage		area					
976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991
		EEPROM		Mode		Data		Storage		area		End			
992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007
mask	flag	offset		ref volts		minutes		unused		unused		unused		unused	
1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023
ID String 16 Bytes long w/o termination															

This device will support a special “EEPROM mode” that writes the raw reading data to the EEPROM rather than the computer. This is for use with an external power supply such as a battery in a remote location (*i.e an outbuilding such as a green house or an enclosed area such as a refrigerator*). The two bytes “EEmask” and “EEflag” (bytes 0&1) are used together to control this mode.

With our new sensor we are measuring voltage directly as compared to our reference voltage. The quantity that is unknown is the reference voltage. While that value is supposed to be 1.1 volts that is not entirely accurate. A reading of the reference pin on my Nano showed 1.067 volts ... but how close is my VOM meter to being accurate? A global variable named “RefVoltage” is used so that value may be adjusted as needed. It is stored as a two byte word at the location “EErefvolt” (*bytes 2&3*). Once that has been calibrated it should not need to be reset. However you might want the Arduino to reflect the temperature relative to your favorite thermometer or

the electronic thermostat on you HVAC system. For that purpose the global variable “DegreeOffset” has been established to move the scale up or down. It is stored as a two byte word at the location “EEoffset” (bytes 4&5). The global variable “MinuteTarget” is used to control how often that report data is generated. It is stored as a two byte word at the location “EEoffset” (bytes 6&7).

After the report parameters we have space for 4 words of data that have not been used (bytes 8 through 15). That is followed by our 16 character ID/Location string (bytes 16 through 31). All of the proceeding is called our “working data”. That 32 byte segment has a duplicate backup copy at the end of EEPROM which conveniently places our ID/Location string in the last 16 bytes. Following the working data are 40 words reserved for use by a table based conversion routine (bytes 32 through 111). The rest of the EEPROM is used for data storage during the EEMPROM mode (440 words: bytes 112 through 991 on an ATmega328).

Global Variables and Constants

Now let’s take a closer look at the global variables and constants.

```
const word EEwdsz = EEidtring+EEidsz;
const word StorageWorking=EEmask;
const word StorageBackup=((E2END-(EEwdsz))+1);
```

The constant “EEwdsz” defines the size of our EEPROM working data block. The two constants “StorageWorking” and “StorageBackup” are used to define the start address of working data storage and its back up copy. These three values are used routines that backup and restore the data blocks.

```
word StorageBegin =StorageWorking+EEwdsz+EEtbsz;
word StorageMark =StorageBegin;
word StorageEnd =StorageBackup;
word StorageIndex =StorageBegin;
```

These variables are used by the routines that write, read and erase the EEPROM mode data storage. The “StorageMark” and “StorageIndex” were added so that wear leveling could be implemented.

```
float RefVoltage;
const float CovrtFactorV= 8192;
float DegreeOffset;
const float CovrtFactor0= 1024;
word MinuteTarget = 1;
char IdString[EEidsz+1];
byte newflg=0;
```

The first two variables were previous discussed. The two constants are used to convert them from floating point variables to words so that they can be written to and recalled from the EEPROM. The variable “MinuteTarget” controls the time between generated reports. The variable “IDString” is used to hold a copy of the ID/Location string in ram. The variable “newflg” is used as a bit level flag to indicate that when any of the above variables have been changed and not written to EEPROM.

```
float Celsius;
float Fahrenheit;
float AvrCelsius;
float AVRFahrenheit;
```

These variables hold the current temperature values in their respective scales.

```
boolean ReportMode = true;
boolean RtnRawRead = true;
boolean RtnCelsius = true;
boolean RtnFahrenh = true;
boolean DeBug = false;
boolean RtnAvrRead = false;
boolean EepromMode = false;
boolean RoundMode = true;
```

These variables control the operational modes of the device and establish the default operation.

ReportMode	Turns printing of report lines on or off
RtnRawRead	When true sensor reading is included as the first value of a report line
RtnCelsius	When true sensor Celsius is included as the next value of a report line

RtnFahrenh	When true sensor Fahrenheit is included as the next value of a report line
DeBug	When true debug data is included as last values of a report line
RtnAvrRead	When true AVR temperature data is included as an appended report line
EepromMode	When true all other modes are set to false and sensor readings are written to EEPROM
RoundMode	When true Celsius is rounded to nearest quarter and Fahrenheit is rounded to nearest half.

```
word      SecondsMinute = 60000;
unsigned long SecondsTarget = 0;
unsigned long RptTrigger   = 0;
byte      gap              = 0;
unsigned long RptStartTime = 0;
unsigned long CycleStart   = 0;
unsigned long CycleTime    = 0;
```

This group of variable is used to control the operational timing. The variable “SecondsMinute” is used to convert “MinuteTarget” to “SecondsTarget” which is the number of seconds between report lines. The variable “RptTrigger” holds the time that the next report is to be generated. The variable “gap” is used to control the ratio of data reads to serial port reads or to put it another way the gap between data reads. The last three variables are used in Debug mode to calculate actual times between report lines and the time required for a data read cycle.

```
unsigned long Accumalator = 0;           // Accumalate temperature reads
unsigned long CycleCount  = 0;           // Cycles per Report line
```

The variable “Accumalator” hold the sum of all data reads for the current report cycle (*Note that each data read is actually the truncated average of 1024 sensor reads*). The variable “CycleCount” holds the number data reads in the current report cycle.

```
char      cmd[]           = {0,0,0};
char      prevcmd[]       = {0,0,0};
```

These two variables hold the current and previous two character command strings read from the serial port. The zero terminator is only used by the Debug mode to print the current command to the serial port.

```
word      LastRead        = 0;
byte      Consecutive     = 0;
```

These two variables are used to implement a simple form of data reduction for the EEPROM recording mode. They are defined as global variables in order to provide with a pre-allocated static RAM memory location so that the values are retained between calls to EEPROM recording function.

Main Program File Functions

setup() Function

There are not really any changes here. This function configures the hardware, reads the parameters from the EEPROM and set the initial operational modes and timings factors.

loop()Function

This function still has the same three basic tasks:

- 1) Check the serial port for commands
- 2) Collect data
- 3) Report/Record data

The only change is the name of the function to collect the data is now "ReadRawTempA1()".

cmdProcessor() function

Functionally the same but the commands were put in alphabetical order. Several commands deleted. New commands added for Degree Offset and Reference Voltage.

HelpMe() function

Tabs were added between the command and description. The list was edited to match the command processor. That made it much easier to extract the Table shown below.

Arduino AtMega328 Temperature Sensor 1.0

ID	Output ID string
ST	Output Status
RT	Raw=True
RF	Raw=False
FT	Fahrenheit=True
FF	Fahrenheit=False
F=	Enter Current Fahrenheit
CT	Celsius=True
CF	Celsius=False
IT	AVR Internal Temperature=True
IF	AVR Internal Temperature=False
DO	New Degree Offset (Fahrenheit)
DF	Same as DO
RV	New Reference Voltage
T1	Report time = 01 minutes
T2	Report time = 02 minutes
T3	Report time = 03 minutes
T4	Report time = 04 minutes
T5	Report time = 05 minutes
T6	Report time = 10 minutes
T7	Report time = 15 minutes
T8	Report time = 20 minutes
T9	Report time = 30 minutes
T0	Report time = 60 minutes
TA	Report time = 02 hours
TB	Report time = 04 hours

TC Report time = 06 hours
TD Report time = 08 hours
TE Report time = 12 hours
TF Report time = 24 hours
PF Print mode = False
PT Print mode = True
DB Debug mode toggle
L: New Location
WW Write Calibration data to EEPROM
W+ Overwrite Backup Calibration data
W- Restore from Backup Calibration data
E+ Set Flag to send next run to EEPROM
E- Clear Flag to send next run to EEPROM
EC Clear EEPROM Storage
ED Dump data stored in EEPROM
LL List implemented commands
?? List implemented commands
SS Shutdown (send twice)
!! Reset (send twice)
Z1 Write test data 1
Z2 Write test data 2
ZZ 5 Second reporting for calibration
ZD Dump ALLL EEPROM to serial
 Response 'XX' = not implemented
 Response '??' = not recognized

PrintSeperatorLine() function

ReadTwoCharacters() function

This was rewritten to save a few bytes. The long string of dashes was replaced with “for” loop that prints the dashes. It also has the advantage that the length of the line can be adjusted easily.

DrainCmdTermiantors() function

DebugPrintCharacters () function

These functions had no changes.

Thermometer Functions File

The following functions were deleted or replaced in their entirety:

CelsiusEquals()	deleted
FahrenheitEquals()	replaced in its entirety
NewOffsetR()	deleted
NewCelsius()	deleted
NewFahrenheit()	deleted
avrRawTemp()	replaced in its entirety
Convert()	replaced in its entirety

Read_Calibration_Data() function

Write_Calibration_Data() function

These functions were rewritten to read and write the new parameters. Note that provisions have been made for the degree offset to positive or negative. If the word stored in EEPROM has the high bit set then the value is negative. A separate divisor is used for the degree offset and voltage reference to allow for different ranges.

ClearStorage() function

EEmodeFlagSet() function

EEmodeFlagClear() function

EEmodeFlagTF() function

These functions had no changes.

Check_EEPROM() function

A line was added to blink the LED 30 times when entering EEPROM recording mode. This is intended to give the user a visual indication that the device is in the proper mode.

Print_IdString() function

PrintTrueFalse() function

These functions had no changes.

ReportStatus() Function

This function was rewritten to delete the old parameters and add the new ones. Tab characters were added between the description and the parameter. The ID string and degree offset are now the only parameters that are specific to given sensor. The strings labels were edited to shorten the lengths saving a few more bytes.

AvrTemperature() function

This function replaces the avrRawTemp() function for the internal sensor. A 12 bit synthetic reading is no longer generated. It now uses the reference voltage to convert the ADC reading to a raw voltage. Conversion of the voltage to Celsius and Fahrenheit is now done with in the function as well as printing a separate report line.

ReadRawTempA1() function

This function replaces the avrRawTemp() function for the external sensor. The first part is essentially the same as the AvrTemperature() function except for the ADMUX setting. Note that we also initialize the pin (A1) at every pass. That is because I am paranoid.

Convert(word RawReading) Function

This function was totally replaced. It is now much simpler consisting of only three lines. Rounding to nearest half or quarter is optional was made optional and calls subroutines for that pupose.

nearestquater () function

This function was removed from the Convert() function. It rounds the incoming floating point value to the nearest quarter (0.25) and returns that value. Provisions have been added to accommodate negative numbers as well.

nearesthalf () function

This function was removed from the Convert() function. It rounds the incoming floating point value to the nearest quarter (0.50) and returns that value. Provisions have been added to accommodate negative numbers as well.

Report() function

This function was modified to add a call to the AvrTemperature() function if the AVR Temperature mode is active. Otherwise there is no change.

QuickBlink() function**Report2EEPROM() function**

These functions had no changes.

DumpStorage() function

The strings that were used to print the header line were combined into a single string with imbedded tab characters. A similar was made to the bottom of the report for the quantity labels. The wording was also changed to delete a few characters. Deleting the extra Serial.print() statements saved a few additional bytes.

Response() function

This function was added to handle the output of the next three functions: PrintOKStr(), PrintNotReconized() and PrintNotImplemented(). This saved several more bytes.

PrintOKStr () function

Rewritten to call Response() function.

PrintNotRecognized() function

Rewritten to call Response() function.

PrintNotImplemented() function

Rewritten to call Response() function.

ShutDown() function**software_Reset() function****SetRawReadMode() function****SetCelsiusMode() function****SetFahrenheitdMode() function****SetReportMode() function****ToggleDebugMode() function**

These functions had no changes.

SetAvrInternalMode() function

This function was added. It is used to change the AVR temperature reporting mode by changing the values if the global variable "RtnAvrRead" to true or false. Default is "false". It must be called using the command "IT" or "IF".

ToggleRoundMode() function

This function was added to toggle the rounding mode of the Convert() function.

NewReportTime() function

Report_Reset() function

These functions had no changes.

NewIdString() function

One line was added to this function to fix a bug. If a shorter string was entered then the end characters from the previous string were not deleted. The line `"while (n<EEidsize) IdString[n++]=0;"` was added to resolved this problem.

PrintDegreeOffsetEffect() function

This function was added to allow show the user the effect of adjusting the degree offset. It prints the temperature as read and then as adjusted by the offset (*in both Celsius and Fahrenheit*). This function is called by NewDegreeOffset() and CalculateDegreeOffset().

ValueNotAccepted() function

This function was added to handle response with an input parameter was not accepted either due to timeout or invalid characters. It can be called by: NewDegreeOffset() , FahrenheitEquals(),CelsiusEquals() or NewRefVolt(). This saved a few more bytes.

NewDegreeOffset() function

This function was added to allow the user to manually adjust the degree offset. This function must be manually called with the command `"F="`.

CalculateDegreeOffset() function

This function was added. Used by FahrenheitEquals() and CelsiusEquals() to adjust the current degree offset. Print output was added to inform the user of the effect of the change.

FahrenheitEquals() function

This function was completely rewritten. It now calls the function CalculateDegreeOffset(). This function must be manually called with the command `"F="`.

CelsiusEquals() function

This function was completely rewritten. It now calls the function CalculateDegreeOffset(). This function must be manually called with the command `"C="`.

NewRefVolt() function

This function was added to allow the user to manually adjust the value for the reference voltage. This function must be manually called with the command `"RV"`.

RestoreFromBackup() function

OverwriteBackup() function

These functions had no changes.

TestData1() function

TestData2() function

These two functions were rewritten to change the parameters. These function load default data into the EEPROM and then call Read_Calibration_Data() to read that data. These functions must be manually called with the command `"Z1"` or `"Z2"`.

CalibrationMode() function

This function was modified such that it now toggles the 5 second calibration mode on or off. This function must be manually called with the command **"ZZ"**.

EepromDumpAll()function

This function had no changes

```
COM12
T 00 DB ST Send

; -----
; Report: True
; Debug: False
; Raw: True
; Fahrenheit: True
; Celsius: True
; Avr: False
; Round: True
; Minutes: 1
; Voltage: 1.0750
; Sensor ID: SainSmartNano328
; Offset: 0.0000
; -----
0709 23.50 74.50
0709 23.50 74.50
0709 23.50 74.50
; IT OK
; 00 OK
; DB OK
; Command Processor Received: ST
; -----
; Report: True
; Debug: True
; Raw: True
; Fahrenheit: True
; Celsius: True
; Avr: True
; Round: False
; Minutes: 1
; Voltage: 1.0750
; Sensor ID: SainSmartNano328
; Offset: 0.0000
; -----
0709 23.57 74.43 98 477 240001
;353 97.56 207.62 ARV
0709 23.57 74.43 124 479 60000
;353 97.56 207.62 ARV
0709 23.57 74.43 124 479 60000
;353 97.56 207.62 ARV
```

Temperature Sensor Calibration

Calibration Theory

The external temperature sensor is designed to generate a linear output in Degrees Fahrenheit. That is to say that for each 1 degree increase in temperature the sensor outputs an equal increase in voltage regardless of the temperature. That ratio is 10 mV (*millivolt = 1volt/1000*) per degree Fahrenheit. There are several additional factors that may affect the reading that is returned by the external sensor:

- 1) Resistance in the path between the sensor and Analog pin
- 2) Capacitance in the path between the sensor and Analog pin
- 3) The accuracy of the reference voltage

For items 1 & 2 TI recommends a 2K Ohm decoupling resistor in line with the output of the LM34 for long distances. The major variable in this implementation is the 1.1 reference voltage produced internally by the AVR . One obvious solution is to use a very accurate high impedance Voltmeter to read the reference voltage at the AREF pin. This may not always be possible (*i.e. the user may not have such a meter available*). The practical alternative is to place the temperature sensor in an environment with a known temperature and adjust the voltage reference as need to obtain the correct output. This also resolves the first problem.

Calibration Method 1

Obtain the use of a known good thermometer. Place the thermometer and the device in a container of some sort that allows the user to view the thermometer but protects both items from hot or cold airflows or direct sunlight. A common corrugated cardboard box is simple and effective container. Wait for the temperature to stabilize on both the sensor and the thermometer. Adjust the value for the reference voltage until both devices show the same temperature.

Calibration Method 2

This requires a bit more work but makes for a very accurate calibration. Remove the sensor from the bread board. Attach the sensor to one end of cable of suitable length with at least 3 conductors (*common Ethernet LAN cable is appropriate*). Enclose the sensor end of the cable in a small plastic bag. Wrap the bag around the sensor and the cable to form a tight bundle. Use short pieces of tape to secure the bundle and seal the top. Place the bundled end of the sensor end of the cable into a small container (*i.e. a one pint plastic milk bottle*) leaving the top of the bundle above the top of the container. Fill the container with water but leave some space for expansion. Place the container and the cable in a plastic bag (*i.e. a grocery bag*). Place this package in a freezer and allow the water to freeze into a solid block.

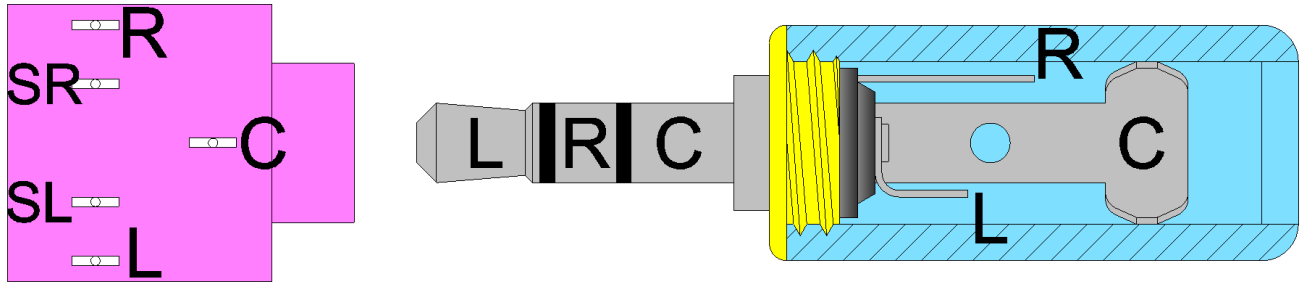
Remove the package from the freezer and properly discard the outside plastic bag. Unbundle the cable and attach the end where the sensor was mounted to the bread board (*be careful to insure that you maintain the correct polarity*). Wait for the ice in the container to begin to melt. Adjust the value for the reference voltage until both device reports 32 degrees Fahrenheit. You are done.

Temperature Sensor Extension Cable

It would be helpful if we had a simple way to move the sensor from the board to a remote location. For that we need a common connector and some kind of standard easily obtained extension cable. Consider the use of a 3.5mm Stereo cable (*i.e. like those used for the iPhone, iPod, Ipad, Android and various mp3 players*). All you need to do is replace the sensor on the breadboard with a 3.5mm stereo female jack and then mount the sensor in a 3.5mm male plug (*consider including the 22K ohm resistor*). Then you can use any 3.5mm stereo cable that is available and those are CHEAP! (*hint: look at <http://www.monoprice.com>*).

SL=Switched Left
SR=Switched Right

L=Left
R=Right
C=Common



The pink thing on the left of this illustration is the bottom of a 5 pin 3.5mm PC board stereo jack. On the right we have a standard 3.5mm phone plug. Connect the ground to “Common”, power (5 volts) to “Right” and signal to “Left”. Ignore the switched pins on the jack (*they are switched “out” when there is a plug in place*).

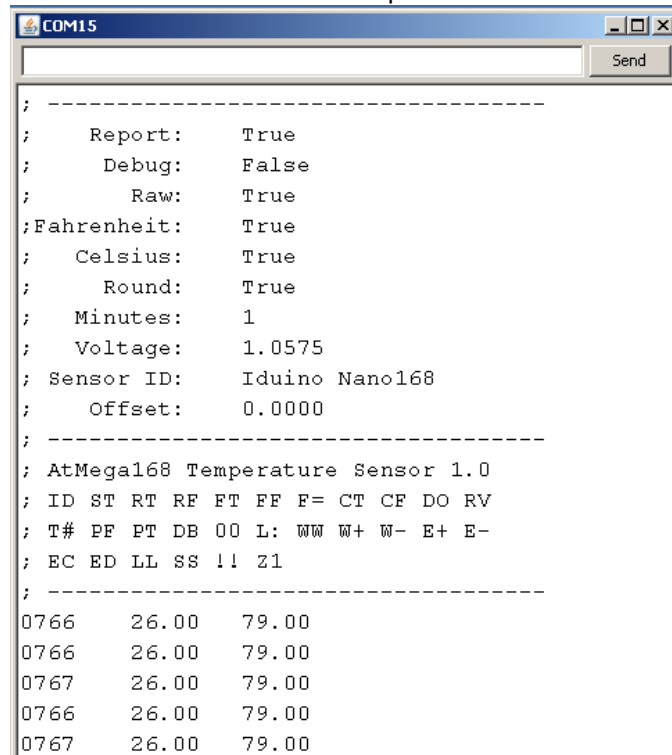
Plan “B”, Evaluation and Summary

- 1) A number of Arduino boards were successful programmed with an application to report the temperature back to the computer at periodic intervals. That item is rated as a “success”.
- 2) The Arduino application(s) that were developed have the desired level of capabilities for two way communications, adjustments, storage and reporting. That is rated as a “success”.
- 3) The applications has been designed and implemented without any modifications or additions to any of the Arduino board. That is item rated as a “success”.
- 4) The design of the application and protocol are such that modification of the application to use a different temperature sensor should only require changing one function and the calibration factors. In plan “B” these changes were implemented and the code further modified to make additional changes in the future even simpler. That item is rated as a “success”.
- 5) The design of the application and protocol are such that modification of the application to support multiple temperature sensors is possible. This would require implementing “Single Sensor” selection protocol as well as changes to the reporting functions. Until these actions have been attempted and completed successfully that is rated as “questionable”.
- 6) Calibration, testing and comparison have verified the repeatability of the readings from the device as well as tracking well with the reference instruments. That item is rated as a “success” (*finally!*).
- 7) The one detractor at this point is that the size of the binary hex code that is downloaded to the Arduino is over 17 K bytes. It would be desirable to get this down to a size that could be loaded into a ATmega168. That is likely to require deleting some of the functionality.

Plan “B” is a winner !

Thermometer Program, ATMEGA168

The reason for having gone to the additional trouble of reducing the size of the program was because it had grown to 18KB for the binary image. I really wanted a version that would be able to run on the ATMEGA168 because I happen to have a couple of these that I purchased before I found out that they do not have internal temperature sensors. However in order to reduce it sufficiently some functionality must be sacrificed. The first thing to go of course is the Internal Temperature Sensor function. The next major change was to reduce the Help system to a very short list of commands without descriptions.



```
COM15
Send

; -----
;   Report:      True
;   Debug:       False
;   Raw:         True
; Fahrenheit:    True
;   Celsius:     True
;   Round:       True
;   Minutes:     1
;   Voltage:     1.0575
; Sensor ID:     Iduino Nano168
;   Offset:      0.0000
; -----
; AtMega168 Temperature Sensor 1.0
; ID ST RT RF FT FF F= CT CF DO RV
; T# PF PT DB OO L: WW W+ W- E+ E-
; EC ED LL SS !! Z1
; -----
0766    26.00    79.00
0766    26.00    79.00
0767    26.00    79.00
0766    26.00    79.00
0767    26.00    79.00
```

Next to go was the function that dumped a list of the EEPROM in Hex and ASCII. It was still a bit too plump so one set of test data was removed as well as the report times over one hour. The debugging code was also removed (*even though it was commented out*). Then it compiled with just over 200 bytes to spare. Full EEPROM recording mode and the EEPROM data dump feature is include all be it a somewhat smaller storage area. The full program code is included in the Appendix: Thermometer ATMEGA168.

... now the question is what to do with that extra 200 bytes of flash memory

Arduino Debugging

The most glaring weakness of the Arduino system is the lack of a good run time debugger or any debugger for that matter. Atmel does support a hardware level debugger in their development package but that is not commonly available and the Arduino “reset” design interferes with it. There are a few simulators but nothing that seems to be well recommended. Thus we are reduced to the oldest, most basic debugging methods.

Debugging Methods:

- 1) Review your code for syntax errors.
- 2) Put in lots of print statements.
Use these to isolate the area with the problem.
Until you have a finished project you may want to comment these out so you can reuse them.
- 3) Reduce you code to the most basic pieces.

Common errors to look for:

- 1) Each “C” stamen must end with a semicolon --- look for missing semicolons.
- 2) “C” is case sensitive --- look for improperly capitalized keywords.
This applies to user defined functions, variables and constants as well.
- 3) Improperly nested delimiters `{([])}` --- look for miss matched delimiters.
This is also commonly known as “Lost In Stupid Parenthesis” (*reference to LISP programing language*).
- 4) The entire world interprets leading zeroes as zero except “C” – leading zeroes signify OCTAL numbers.
`debug.ino:117:46: error: invalid digit '9' in octal constant`
- 5) Binary notation will only accept 8 digits --- count your digits.
Temporary placing a space in the middle sometimes helps: B0000 0000.
- 6) Improperly nested quotes (*double and single*) ---
- 7) Buffer overflow: “C” does not limit string writes like higher level languages such as Basic.
Insure that you are not writing beyond the end of allocated space.
- 8) Be sure that you use commas inside of the parameters clause of a “while” statement.
- 9) Be sure that you use semicolons inside of the parameters clause of a “for” statement.
- 10) Be sure “if equal” uses two equal signs “==” not one “=”.
- 11) The dreaded “off by one” syndrome ... you will be amazed at what a difference there can in having a one where you should have had a zero.
- 12) Lower case “i” looks a lot like lower case “l” that looks a lot like “1”. “0” also looks a lot like “O”.

Other Hints:

- 1) Years ago one of my college instructors told me that when Einstein was asked how to solve a particularly difficult physics problem that he responded: **“Simplify, Simplify, Simplify.”**
I would imagine that this applies to electronics and software as well as advanced physics.
- 2) Include lots of comments --- your logic may be sound and obvious today but totally incomprehensible tomorrow.
- 3) When you get something that works make a backup copy. You can clean it up later.
- 4) Reduce your function sizes so that you can view the entire function on a single screen.
- 5) Group functions that call each other together.
- 6) Define functions before they are called.
- 7) Write and debug small pieces code separately.
- 8) Use the statement: `Serial.println ("Got here");`
Start at the beginning and move the line through the code until it does not print.
- 9) There is no “stop” or “end” statement. Use the statement: `while (true);`
This should stop a runaway program.

- 10) Do not take the error messages at face value --- look at the statements ahead of the referenced line.
- 11) Temporarily comment out code sections (*use the edit menu functions*).
- 12) Do you Serial.print() a lot of strings in your code? Use the F() to reduce RAM usage (*it works*).
- 13) Close the IDE and reload the program --- sometime the IDE sometimes gets confused.
- 14) Copy the program to a new folder and rename it.
Strip out the pieces until you have the most basic template.
(I have a sketch folder called "debug" for this purpose)
Use a text editor to open the original program files.
Copy pieces one at a time from the Editor to the debug application in the IDE.
- 15) Have someone else look at you code --- you can get help on the Arduino user forum.
- 16) Shutdown the IDE and CLEAN your personal temp and build directories.

The Arduino IDE does NOT clean up after itself.

In the Temp directory look for folders named:

- console*.tmp,
- build*.tmp
- scoped_dir*
- untitled*.tmp

- 17) Do not use Microsoft Word as a code editor --- in that mode it is a royal pain in the southern most regions. On the other hand "Notepad ++" is an excellent code editor.
- 18) Remove or disable the 'CAPS LOCK' key from your keyboard (*perhaps not practical but very desirable*)
- 19) If all else fails --- get some rest. Fresh eyes see mistakes much better.

Received

```
void setup()
{ Serial.begin(9600);
}

void loop()
{ byte i;
  // this program is intended to demonstrate stoping a run-away program
  // by using the while() statement ....
  Serial.println ("Got here");
  // uncomment the next line ----
  // while (true);
  Serial.println ("Got here as well");
  // the error here is that i is declared as a byte.
  // its max value is limited to 255, after that it rolls back to 0
  // thus the program will rapidly repeat the print statement
  // until the end of time and you will most likely never see
  // the previous print statements flash by ....
  for (i=0; i<300; i++)
  { Serial.println (i, DEC);
    // uncomment the next line ----
    // if (i==255) while (true);
  }
}
```

RS232 Serial Monitor

Now that we have a Arduino program to report the temperature back to the PC we need something on the PC side to read that data (*other than using the serial terminal in the Arduino IDE*).

FreeBasic Compiler

FreeBasic is a modern multi-platform implementation of the basic language. It is an open source compiler that can generate standalone executable programs (*no runtime distribution package required*), libraries or object files for Windows or Linux operating systems. It can produce either CUI (*console/ character user interface*) or GUI (*Graphical User Interface*) programs. Like the Arduino IDE it uses GCC and/or GAS in the background to build the executable. It comes with its own small debugger and but you can also use the GDB/INSIGHT debugger. All of these are open source tools.

FreeBasic was originally written as a replacement for GWBasic that came with the early generation Microsoft operating systems. Microsoft has a tendency to 'upgrade' all of their software on a regular basis in a manner such that the new version is incompatible with the older versions. They do this in order to 'encourage' their customers to purchase the new version. Without the built in obsolescence Microsoft's profit margins would quickly start to plummet. Unfortunately it also appears that Microsoft is intentionally trying to eliminate all console based applications (see)

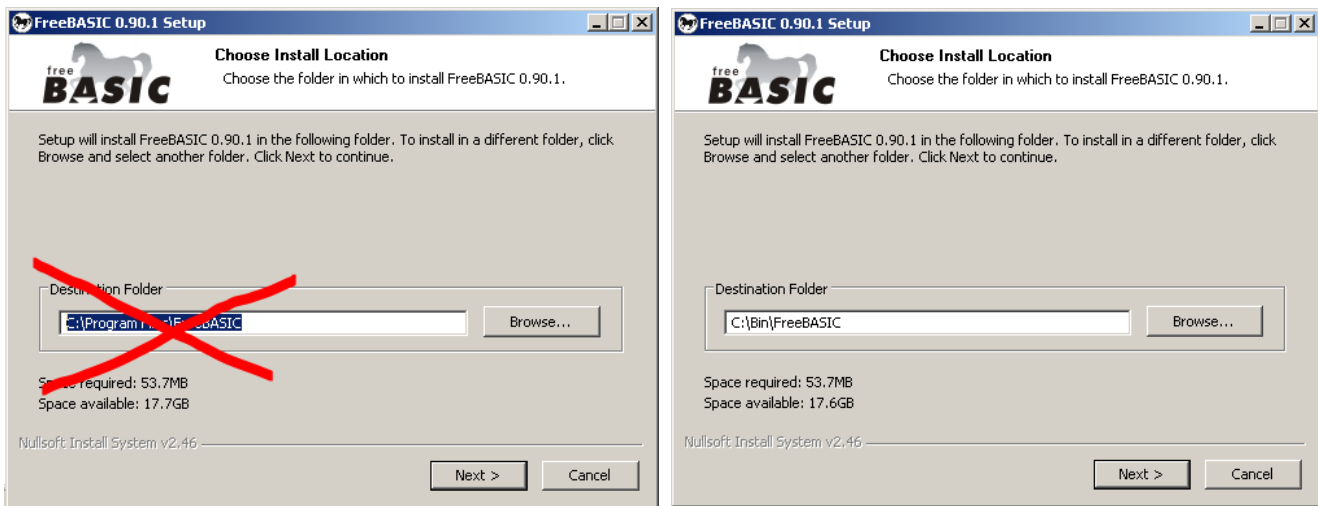
These marketing strategies made Bill Gates a billionaire at the considerable expense of the rest of the world.

Although FreeBasic can be used to produce GUI apps there is not a good GUI IDE for developing these kinds of applications. The ones that I tried were some combination of unintuitive, poorly documented or just downright flaky. On the other hand FreeBasic is EXCELLENT for console applications. There are several IDEs available for FreeBasic that work well for console based applications. The one I chose to use was FBIDE.

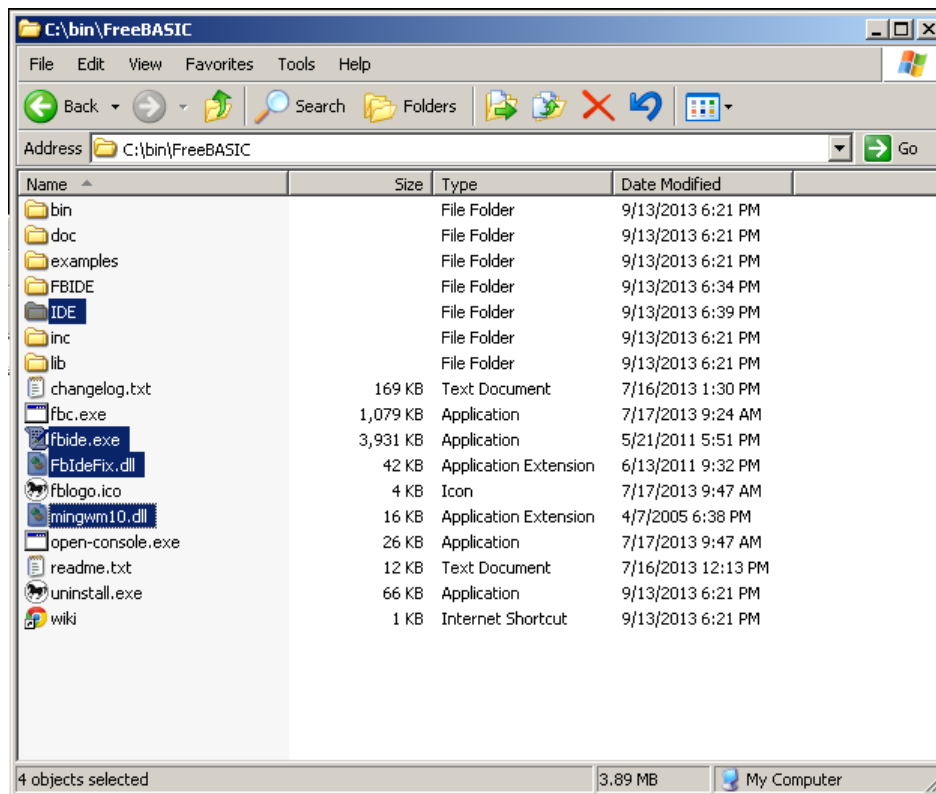
FreeBasic Web Site: <http://www.freebasic.net/>
FreeBasic Download: <http://sourceforge.net/projects/fbc/>
Free Basic Forum: <http://www.freebasic.net/forum/>

FBide Web Site: <http://fbide.freebasic.net/>
FBIDE: <http://fbide.freebasic.net/download>

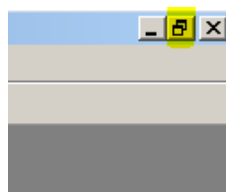
You can download and install both packages on your computer but it highly recommended that you **do NOT use** the default Installation directory: **C:\Program Files\FreeBASIC**. FreeBasic and FBIDE both default to storing projects and working directories under the Free Basic installation directory. The directory C:\Program Files\ and all of its subdirectories are "protected" directories. This will cause you numerous problems. I have created a **"c:\bin"** on my computer where I install such software (*in fairness to FreeBasic this was a common practice in the days of MS-DOS, Windows 3.1, Windows 95 and Windows NT even for programs produced by Microsoft*). This also provides someplace to install various linux/unix utilities that I sometimes find useful.



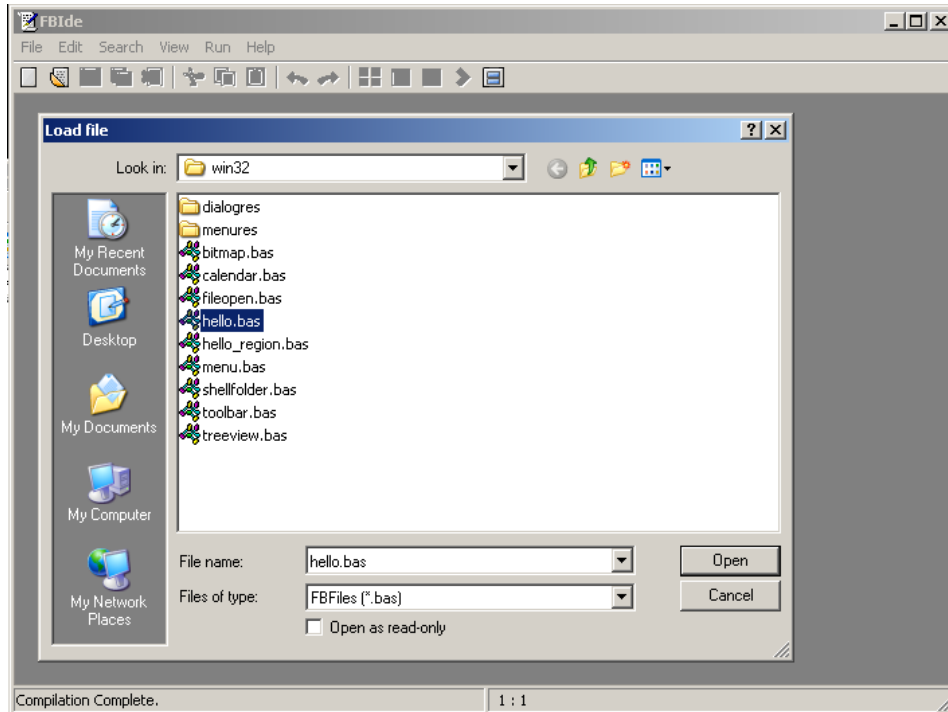
The FBIDE Program comes as a Zip file (*no installation*). Most current Operating Systems can open a Zip file without any additional software. Open the Zip file and drag the contents of the folder “FBide0.4.6r4” to a location under FreeBASIC. Your installation should look something like this.



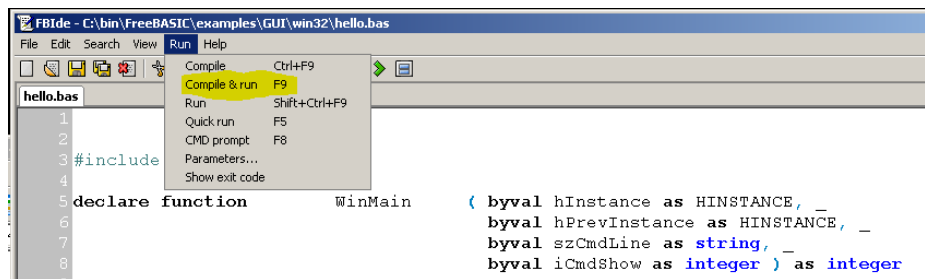
The selections shown with the **dark blue background** are the ones added by FBIDE. Double click on the fbide.exe file. It will open and immediately assume the entire screen. Click on the double Box icon in the upper right corner. It will then be much better behaved.



Select “File”, “Open” from the top menu. Navigate to the directory “C:\bin\FreeBASIC\examples\GUI\win32”. Open the file “Hello.bas”.



I am not about to try and explain the complexities of the code for this program. Thankfully that is not required because all we are trying to do is determine that the software works. From the top menu select “Run”, “Compile & run”.

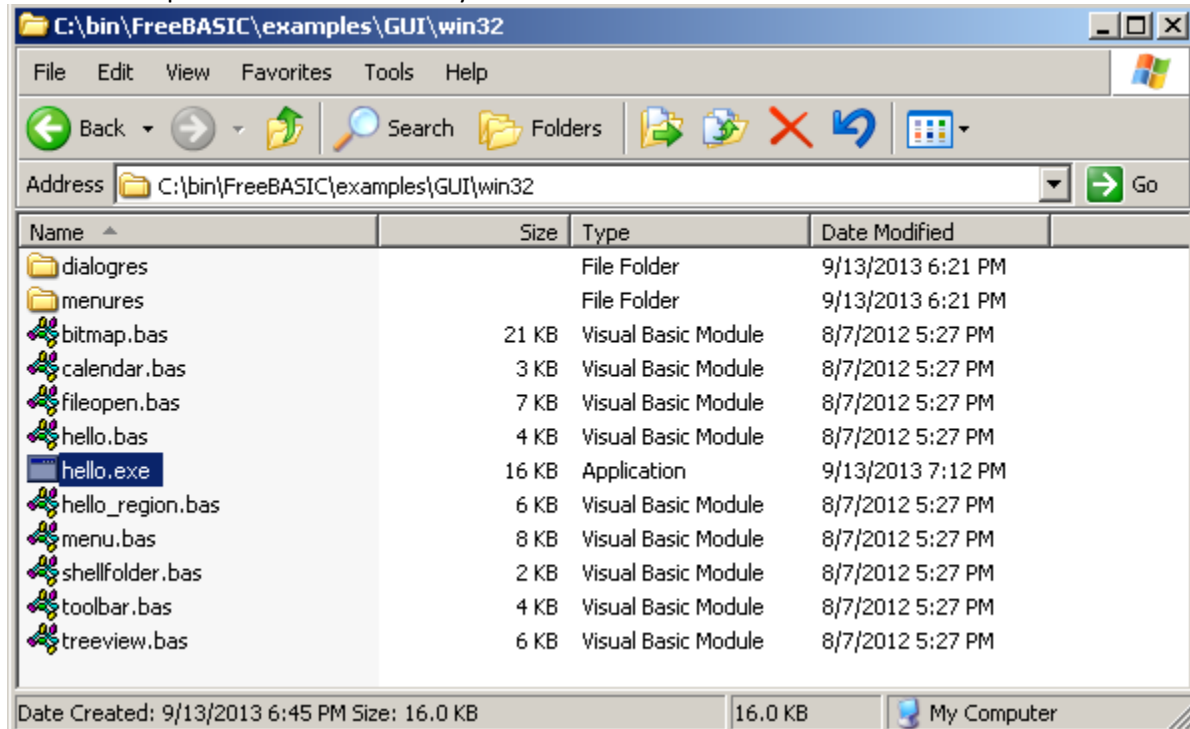


That should open a rather large GUI window with the words “Hello, World!” in the middle.



I would say that the Window for a bit oversized for the task but the good news is that you now have another compiler and IDE installed on your computer. With this one you can produce programs that run under any

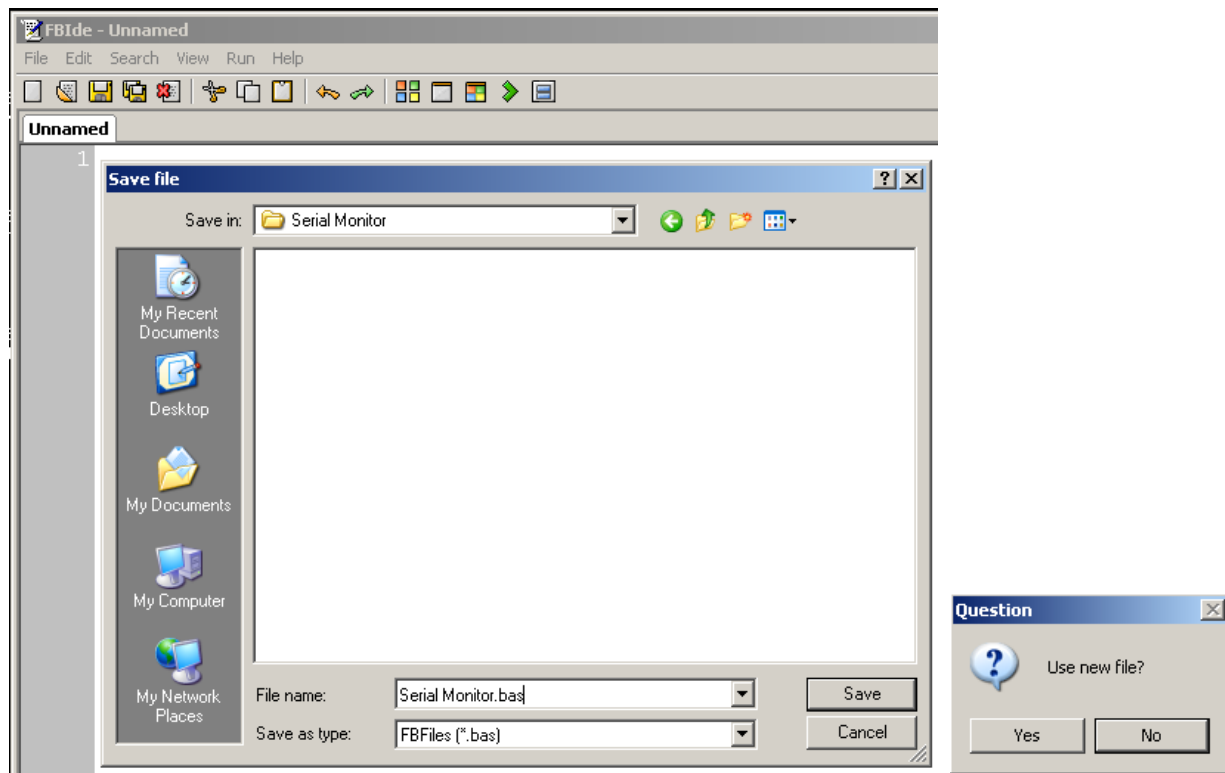
version of Microsoft Windows (*and in theory Linux*). Close this window as well as the program file in the FDIDE editor. Let's take a quick look at the directory where the source file was located.



That ".EXE" file that you see there is the executable program. You do not need any runtime packages, "DLL" files, libraries or installation programs. You can copy that file by itself to any computer running Microsoft Windows and the program can be run.

Serial Port Monitor Program

Navigate back up the directory tree and create a directory called Projects under the "FreeBasic" directory. Create another directory under it called "Serial Monitor". In FBIDE select "File", "New". This will open a new editor Window. Now select "File", "Save As" and navigate to the new project folder. Save the file with the name "Serial Monitor.bas". Answer "YES" to the dialog box that pops up (*it does not seem to make any difference if you answer yes or no*).



Now copy the sample code below and paste it into the editor Window. Then select “File”, “Save”.

```

/' This is my first FreeBasic program. I thought it would be easy because
I was sure that I would snatch some sample code and proceed on. Wrong!
Extensive searches revealed a bit here and a bit there but always just
a bit missing: and that bit always seemed to be enough to foil my efforts.

Sitting on the other end of my serial cable is an Auduino Nano
microprocessor board. It is sitting there printing "Hello World!" once
every second. All this program does is read the input from the com port
and print it to the screen. Pressing any key will cause the program to
close the com port and exit. It is amazing how much time I wasted on this.

It is implimented as a straight fall through process and one loop.
Not overly efficient as its wastes lots of CPU time sitting in the loop
polling for a character to come through. However it does have the advantage
that it does work fairly reliably.
'/'

#include "string.bi"           ' needed for format function
Dim LineCount As LongInt      ' just so we can see how many lines are read
Dim chrcount as Long
Dim C As Byte                 ' this is our incoming byte of data
Dim InBuffer As String        ' this is our buffer to collect the bytes
Dim PortStr as String
InBuffer=""
LineCount=0
chrcount=0

/' Any of these strings except the last will work
but the first is more reliable
Port = Com12
Parity = none
Data Bits = 8
Stop bits = 1
Carrier Detect Duration = 0

```

```

Clear to Send duration = 0
Data Set Ready duration = 0
Open Timeout = 0
Bin = Binary communications

PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP,BIN"
PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP,ASC,FE,TB0,RB0"
PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP"
PortStr = "COM12:9600,N,8,1,CD,CS,DS"
PortStr = "COM12:9600,N,8,1,CD,CS"      '/' does not work for Arduino '/'
// Open Com ("COM12:9600,N,8,1,CD,CS,DS,OP,BIN") as #2 //
'/

PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP,BIN"
Open Com(PortStr) AS #2
' loop until there is a keypress ... any keypress
While InKey = ""
' This first line is one of the bits that was missing.
' It checks to see if there is anything waiting in the Serial buffer
' Without it you are subject to reading a bunch of garbage
If Not(EOF(2)) then
' get a single byte from the serial port
Get #2,0,C,1
' characters below ASCII 32 are 'non-printing characters
' characters above ASCII 126 are not define (by ASCII)
' append any printable character to the string
If (C > 31) and (C < 127) Then
InBuffer = InBuffer + Chr(C)
' chrcount is not really needed. I added it while trying
' to figure out why I was getting garbage ... before the
' EOF() check was added.
chrcount=chrcount+1
else
' ignore this character
End If
' Linux/Unix terminate strings with a line feed (ASCII 10)
' MACs terminate lines with a carriage return (ASCII 13)
' Microsoft and Arduino use carriage return/linefeed (ASCII 13,10)
' If we get any of the above then increment the line count and
' print the string but only if we have something to print.
If ((C=13) Or (C=10)) And (chrcount > 0) Then
LineCount = LineCount + 1
Print Format(LineCount,"000000") + " (" + Format(chrcount,"000") + "): " + Inbuffer
' clear the buffer so that we can do it again
InBuffer=""
chrcount=0
End If
End If
' Call Sleep with 25ms or less to release time-slice when waiting
' for user input or looping inside a thread.This will prevent the program
' from unnecessarily hogging the CPU.
Sleep 25
Wend ' inkey = ""

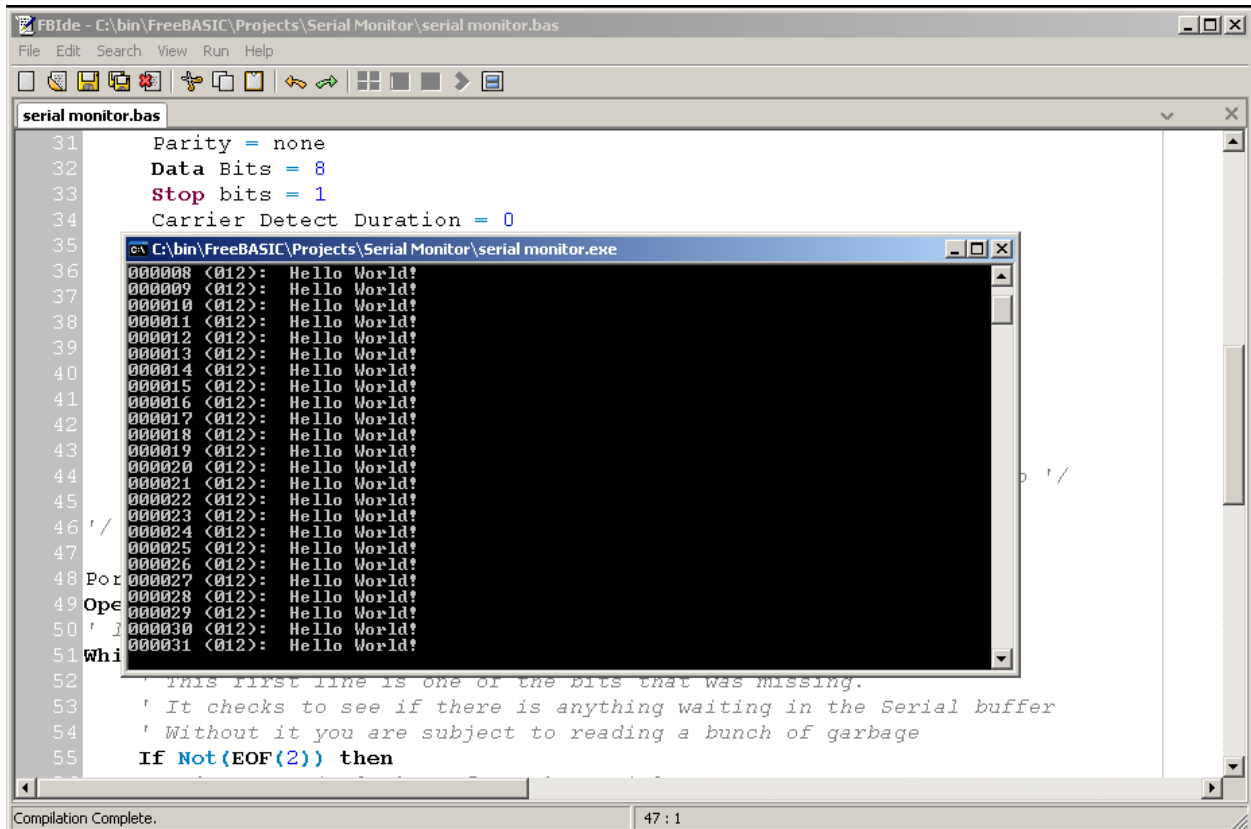
' we opend it, we close it
Close #2

' you'all come back, ya hear?
End

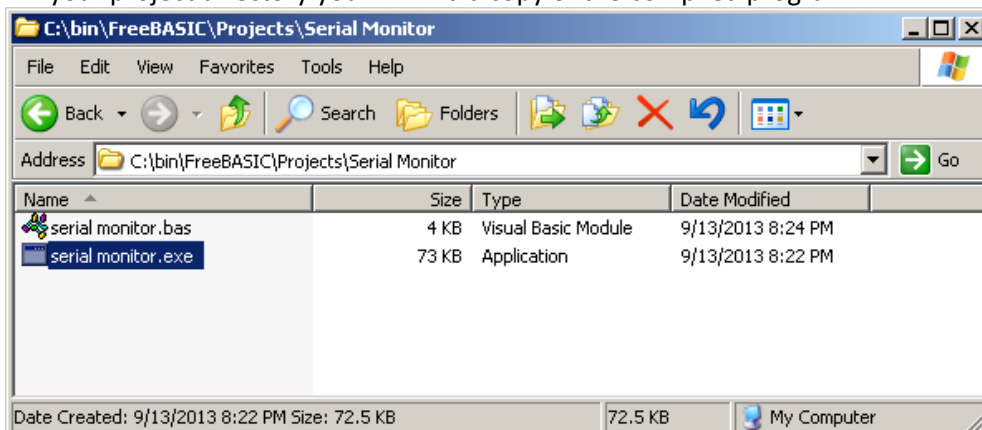
```

Like “C” FreeBasic has two conventions for comments. Single line comments are delimited by the single quote character. Multiple line comments are delimited by forward-slash plus single quote and a matching single quote plus forward-slash. As you look at this you might notice that there is very little actual program code. The

comments give a fairly complete explanation of the program. What you will want to pay attention to is the com port string: `"COM12:9600,N,8,1,CD,CS,DS,0P,BIN"`. You will most likely need to change this according to the Arduino and Computer that you are using. If you compile and run this then you should have a program that prints out any ASCII strings it receives on the serial port to the display.



Now if you look in your project directory you will find a copy of the compiled program.



You can run the AVR Temperature program on you Arduino and receive the data on your computer via the “serial monitor.exe” program. Because this program is printing to Standard Out you can redirect its output to a file via the command line:

`“serial monitor.exe” >> mylogfile.txt`

Still it does have a number of limitations most particularly the need to recompile it when you change the computers or the Arduino. See the “Appendix: Arduino Receiver” for a much more capable program. It uses an

INI file so that that it does not need to be recompiled when you change the COM port. You can also modify the program code to suite you own needs and/or preferences. It is compiled with the FreeBasic compiler.

PC Alternatives: Microsoft

There are several alternatives that you might wish to consider for producing PC based programs for you Arduino. Without question that list would have to contain Microsoft Visual Studio that provides Visual Basic, C#, J# and C++ variants. The current version is Visual Studio 2012 and is a very expensive option. However there is an "Express" version of the Visual Studio 2010 that is free (*requires registration*). When this document was being written it was available at the URL:

<http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express>

The 200 MB download does not include any help or documentation. That must be added on after the fact and from my experience is of limited value (*unlike the help system that came with Visual Basic 5 or 6*). I should also mention that the language elements (*at least Visual Basic*) bear no resemblance to the classical programming languages. Numerous keywords are completely missing with no equivalent. Where there is an equivalent it is likely buried somewhere deeply within a ".net" class library and in some cases requires a completely different syntax. One of these reasons for these changes was in order to use a single compiler for all of the languages. Another was that someone has decided that any program that is written should be written with managed object orienteed class libraries and conform to modern "C++" standards. That is all well and good for professionals that spend their life writing code for large, complex projects but it completely ignores the existence of the rest of us. Having used the BASIC programming language since the days of CPM, Atari, Apple II and Commodore Vic 20 computers I am personally offended by the effort Microsoft has put forth to destroy the Basic programming language.

To be fair I must mention that Microsoft has some lesser products available such as "Power Shell" and "Small Basic". Power shell is a Microsoft proprietary scripting utility. It is dependent on run time environment and has no interactive debugger. However it is quite powerful, extendable and used in a number of organizations for network management. "Small Basic" is very limited, short on documentation (*i.e. there is no keyword reference*), uses "C" type arrays and ".net" syntax. It has no resemblance to Basic. (*also not this is not the SmallBasic originally written for the Palm OS*).

If you happen to have a legal copy of Visual Basic 5 or Visual Basic 6 hang on to it. These are by far and without question the very best implementations of the Basic language. They are still viable for all versions of Microsoft Windows.

PC Alternatives: Non-Microsoft

The first scripting language that must be mentioned is BASH as used on the Linux and Unix systems. It is far older than and arguably as powerful as PowerShell as well as being multi-platform. It does not have the Microsoft Network management extensions that PowerShell has although I am almost certain that this is a limitation that could be addressed. I have Bash shell scripts that I use on both Windows and Linux platforms without any modifications.

Another popular Windows scripting implementation is AutoIt. I will use their description:

AutoIt v3 is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting. It uses a combination of simulated keystrokes, mouse movement and window/control manipulation in order to automate tasks in a way not possible or reliable with other languages (e.g. VBScript and SendKeys). AutoIt is also very small, self-contained and will run on all versions of Windows out-of-the-box with no annoying "runtimes" required!

- Easy to learn BASIC-like syntax

- Simulate keystrokes and mouse movements
- Manipulate windows and processes
- Interact with all standard windows controls
- Scripts can be compiled into standalone executables
- Create Graphical User Interfaces (GUIs)
- COM support
- Regular expressions
- Directly call external DLL and Windows API functions
- Scriptable RunAs functions
- Detailed helpfile and large community-based support forums
- Compatible with Windows 2000 / XP / 2003 / Vista / 2008 / 7
- Unicode and x64 support
- Digitally signed for peace of mind
- Works with Windows Vista's User Account Control (UAC)

AutoIt has been designed to be as small as possible and stand-alone with no external .dll files or registry entries required making it safe to use on Servers. Scripts can be compiled into stand-alone executables.

The problem with AutoIt is it has no native RS-232 support. There is an add but the source is not available and long term support depends on a single individual updating the software. AutoIt can be downloaded at the URL:

<http://www.autoitscript.com/site/autoit/>

Sharp Develop is an Open Source Development Environment for .NET. That being said it is dependent on Microsoft. It has the same advantages and disadvantages as Microsoft Visual Studio. In fact the help system must be downloaded from Microsoft. The two major advantages it has are open source and price: it is free. The Sharp Develop web site URL is:

<http://www.icsharpcode.net/OpenSource/SD/Default.aspx>

PowerBasic is commercial compiler that uses classical Basic language. It was written in assembler bas in the days of MS-DOS and Windows 3.1. It was an excellent product. They eventually produced a Windows based IDE and Windows console compiler. The IDE never could never compete with the “point and shoot” type IDE that Microsoft had in the Visual Basic line. In addition and perhaps more importantly the Visual Basic IDE had a fully integrated line by line debugger that allowed one in many cases to change the code without recompiling. However Microsoft has since abandoned classical Basic language. Thus PowerBasic may deserve a fresh look. One advantage they do have over FreeBasic is an effective integrated debugger.

Why FreeBasic

First I should mention that I have access to all of the products mentioned above including Microsoft Visual Studio 2012 Professional. I have used both Visual Studio 2012 and Microsoft Visual Basic 6 in a professional setting during previous employment. I chose not to use either of these products for this project because of cost. For this project I wanted something that was free to anyone with an internet connection.

I also wanted something that was available on multiple platforms (i.e. Windows, Linux and Mac). That eliminated anything based on “.net”. There is a multi-platform implementation of “.net” called “mono” but it has a bit of a reputation for being an immature product.

FreeBasic turned out to be the only free true compiler with a simple IDE that I could find that would operate on both the Linux and Windows environments (*unfortunately this leaves the MAC OS out*). It also has the advantage that it uses a single source file to produce a single executable file without any need for any run time libraries or “DLL” files (*until you get into the advance graphic functions, external databases, etc.*). It uses a classic implementation of the BASIC language and has extremely good help system that is integrated into the FBIDE environment. The debugger on the other hand leaves something to be desired.

ArduinoThermometer.exe

ArduinoThermometer is actually a set of programs written in FreeBasic to run in a console window under Windows or Linux and capture the data from the Arduino Thermometer application. Full source for both programs is included in the Appendix: Thermometer.exe. The main program is a customized version of the Arduino Receiver program mentioned in the previous section. The program supports all the standard options for the Receiver program plus it has been modified so that a number of single character keystroke can be used to control the Arduino:

```
Keys '1' to '0' set report Times
Key  'A' toggles AVR mode
Key  'B' Restore from Backup
Key  'C' toggles Celsius mode
Key  'D' toggles Debug mode
Key  'E' toggles EEPROM mode
Key  'F' toggles Fahrenheit mode
Key  'L' lists AVR commands
Key  'M' turns on Minimal mode (Fahrenheit only)
Key  'Q' prints AVR storage
Key  'R' toggles Rounding mode
Key  'S' prints AVR Status
Key  'V' toggles Raw Reading mode
Key  '>' increase Degree Offset by 0.25 Fahrenheit
Key  '<' decrease Degree Offset by 0.25 Fahrenheit
```

Noticeably missing are any commands to change the calibration or write new parameters to the EEPROM. That was intentional because this program is intended to be used to capture the data not to calibrate the device. However there is a way around that. The program allows the user to define five “user” strings via the “.ini” file. These commands are sent by pressing the “U” key followed immediately by a numeric key “1”, “2”, “3”, “4” or “5”. There is a one second timeout for the second key. As provide the first three of these are defined as follows:

```
Key  'U1' INI defined string: "Z1", write test data set 1 to EEPROM
Key  'U2' INI defined string: "Z2", write test data set 2 to EEPROM
Key  'U3' INI defined string: "ZD", Hex/ASCII dump of EEPROM
```

Because the Thermometer application allows the use of a space as a delimiter a series of commands may be included in a single user defined string. That is the way that the ‘M’ keystroke works. It sends the command string "RF CF IF FT ST".

```
Command Prompt - Arduino_Thermometer.exe

>Arduino_Thermometer.exe
-----
Report:      True
Debug:       False
Raw:         True
Fahrenheit:  True
Celsius:     True
Avr:         False
Round:       True
Minutes:     1
Voltage:     1.0699
Sensor ID:   SainSmartNano328
Offset:      0.0000
-----

===Arduino_Thermometer.exe Ver1.0.5 (8 October 2013) ===
Using INI file:
Arduino_Thermometer.ini
Using Port Str: COM12:9600,N,8,1,CD,CS,DS,OP,BIN
Appending date/time to data.
Logging data to: Thermometer_0001.LOG
Delimiter: Tab
EOL: carriage return plus line feed
Start: 10-07-2013 05:38:59
Press '?' or '/' to redisplay this message
Press 'Escape' key to exit.

Keys '1' to '0' set report Times
Key 'A' toggles AVR mode
Key 'B' Restore from Backup
Key 'C' toggles Celsius mode
Key 'D' toggles Debug mode
Key 'E' toggles EEPROM mode
Key 'F' toggles Fahrenheit mode
Key 'L' lists AVR commands
Key 'M' turns on Minimal mode (Fahrenheit only)
Key 'Q' prints AVR storage
Key 'R' toggles Rounding mode
Key 'S' prints AVR Status
Key 'U' toggles Raw Reading mode
Key '>' increase Degree Offset by 0.25 Fahrenheit
Key '<' decrease Degree Offset by 0.25 Fahrenheit
Key 'U1' INI defined string: "Z1"
Key 'U2' INI defined string: "Z2"
Key 'U3' INI defined string: "ZD"

2013-10-07 05:40:01 0703 23.00 73.50
2013-10-07 05:41:01 0703 23.00 73.50
2013-10-07 05:42:01 0703 23.00 73.50
2013-10-07 05:43:01 0703 23.00 73.50
```

Strip Semicolon Lines Utility

The main program writes everything it receives from the device to a tab demitted text file specified in the “.ini” file. Every line that Arduino application sends that is not an actual report line is prefixed with a semicolon. **StripSemicolonLines.exe** is a utility used to separate or extract the report lines. It was written with a number of options including the ability to mark a point in the input file where it last processed the lines. Running the program with a “?” as the parameter will print out the options.

Syntax: StripSemicolonLines.exe **file1 file2 file3** [options]

file1 = input filename

file2 = output filename

file3 = output filename with stripped lines (optional)

Options:

/O = Overwrite any existing output file

/A = Append to any existing output file (overrides /O)

/D = Delete input file

/R = Retain blank lines

/S = Split lines at semicolon and delete trailing portion

/X = Deletes all lines with semicolon regardless of location

/M = Mark end of file with “;;--PROCESSED--;;”

/E = Execute program with output file

/V = Verbose prints statistics before exiting

/? = display help and exit

The “**file1**” is the input file that was produced by the main program. “**file2**” is the output file where you want the report lines written. Both of these file names are required and may include a full or relative path specification. “**file3**” is optional. If included the lines that are stripped from the input file will be written to this file. The “**overwrite**” or “**append**” option tells the program what to do if you specify a filename for a file that already exists. If you do not specify an option and the file exists then the program aborts. If the “**append**” option is specified then the “**overwrite**” option is ignored. The “**delete**” option can be used to delete the original file after it is processed. Normally the program skips all blank lines however you can use the “**Retain**” option to keep them (*why you want to I have no idea*). The “**Split**” option divides any lines that have a semicolon somewhere other than the first character. It writes the first part to “**file2**” and the full line to “**file3**”. The “**X**” option has the opposite effect. It deletes any lines with semicolons anywhere in the line. The “**Mark**” option writes the line “;;--PROCESSED--;;” to the end of the input file after it has processed it. When the program is run with the “**Mark**” it reads the entire input file looking for the last occurrence of this line. It then begins processing at the next line. If the line is not found then it begins processing at the beginning of the file. This is useful for extracting data from an active log file. The “**Execute**” will pass the output file to a program such as a spreadsheet or charting program. The program name must include the full path and should be enclosed in quotes (*due to spaces in the path or file name*). The “**Verbose**” option prints the number of input lines, output lines, blank lines and semicolon lines before the program exits.

Although all the options are shown with forward slashes “/” the dash “-” can be used as well. The options may be in in order or case. These are some examples of valid command lines.

```
StripSemicolonLines.exe Thermometer.LOG work.txt
StripSemicolonLines.exe Thermometer.LOG work.txt dump.txt
StripSemicolonLines.exe Thermometer.LOG work.txt dump.txt /X /O /D /V
StripSemicolonLines.exe Thermometer.LOG work.txt dump.txt -o -D -X -V
StripSemicolonLines.exe Thermometer.LOG work.txt -O /x -V /M /d
StripSemicolonLines.exe Thermometer.LOG work.txt /E:"c:\program Files\suite\sheet.exe"
```

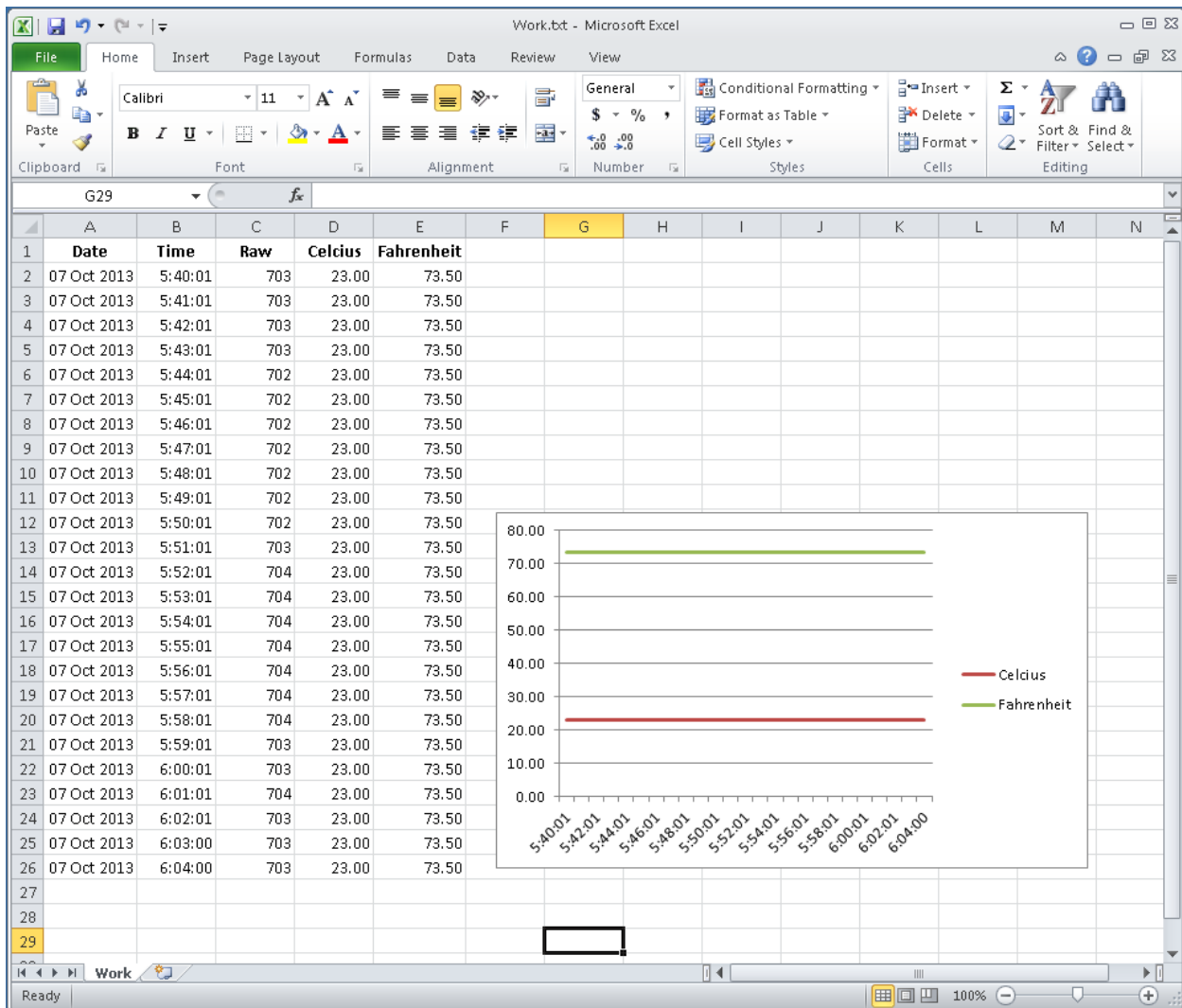
```
C:\Windows\system32\cmd.exe

>cd C:\bin\FreeBASIC\projects\Ardunio_Thermometer

>StripSemicolonLines.exe Thermometer_0001.LOG Work.txt Dump.txt /A /X /U /M /E:"
C:\Program Files (x86)\Microsoft Office\Office14\excel.exe"

    Total Lines: 38
Semicolon Lines: 13
    Output Lines: 25

>pause
Press any key to continue . . .
```



Receiver Modifications:

The changes to this program are NOT sophisticated. Rather the structure of the program is designed to allow someone to add new functionality fairly simply. The file "Arduinio_Thermometer_Globals.Bas" had several global variables added to support the new features. Those variables also have their default values set in this file.

```
Dim Shared DebugMode as Byte      ' False=0, True<>0;
Dim Shared CelsiusMode as Byte    ' False=0, True<>0;
Dim Shared FahrenheitMode as Byte ' False=0, True<>0;
Dim Shared AvrMode as Byte        ' False=0, True<>0;
Dim Shared EEMode as Byte         ' False=0, True<>0;
Dim Shared RawMode as Byte        ' False=0, True<>0;
Dim Shared RoundMode as Byte      ' False=0, True<>0;
Dim Shared UserStr1 as String      ' user defined string in INI file
Dim Shared UserStr2 as String      ' user defined string in INI file
Dim Shared UserStr3 as String      ' user defined string in INI file
Dim Shared UserStr4 as String      ' user defined string in INI file
Dim Shared UserStr5 as String      ' user defined string in INI file
Dim Shared OffsetVal as Single     ' used to store current degree offset

' .....
' These are the default modes
DebugMode=False
CelsiusMode=True
FahrenheitMode=True
AvrMode=False
EEMode=False
RawMode=True
RoundMode=True
UserStr1 ="Z1"
UserStr2 ="Z2"
UserStr3 ="ZD"
UserStr4 =""
UserStr5 =""
OffsetVal=0
```

The additions to the "Arduinio_Thermometer_Functions.Bas" are a bit more involved but not difficult to follow. First there were changes to read the new user strings from the ".ini" file in the function ReadIniFile ():

```
Case "USERSTR1"
    If Lcase(ValStr)<>"" then UserStr1=Trim(ValStr)
Case "USERSTR2"
    If Lcase(ValStr)<>"" then UserStr2=Trim(ValStr)
Case "USERSTR3"
    If Lcase(ValStr)<>"" then UserStr3=Trim(ValStr)
Case "USERSTR4"
    If Lcase(ValStr)<>"" then UserStr4=Trim(ValStr)
Case "USERSTR5"
    If Lcase(ValStr)<>"" then UserStr5=Trim(ValStr)
```

Next the new functions were added to the status string in the function BuildStatusStr():

```
' -----
' Add Application help here
Status = Status & EOL
Status = Status & " Keys '1' to '0' set report Times" & EOL
Status = Status & " Key 'A' toggles AVR mode" & EOL
Status = Status & " Key 'B' Restore from Backup" & EOL
Status = Status & " Key 'C' toggles Celsius mode" & EOL
Status = Status & " Key 'D' toggles Debug mode" & EOL
Status = Status & " Key 'E' toggles EEPROM mode" & EOL
Status = Status & " Key 'F' toggles Fahrenheit mode" & EOL
Status = Status & " Key 'L' lists AVR commands" & EOL
Status = Status & " Key 'M' turns on Minimal mode (Fahrenheit only)" & EOL
```

```

Status = Status & " Key 'Q' prints AVR storage" & EOL
Status = Status & " Key 'R' toggles Rounding mode" & EOL
Status = Status & " Key 'S' prints AVR Status" & EOL
Status = Status & " Key 'V' toggles Raw Reading mode" & EOL
Status = Status & " Key '>' increase Degree Offset by 0.25 Fahrenheit" & EOL
Status = Status & " Key '<' decrease Degree Offset by 0.25 Fahrenheit" & EOL
if (UserStr1<>"" ) then Status = Status & " Key 'U1' INI defined string: " & _
chr(34) & UserStr1 & chr(34) & EOL
if (UserStr2<>"" ) then Status = Status & " Key 'U2' INI defined string: " & _
chr(34) & UserStr2 & chr(34) & EOL
if (UserStr3<>"" ) then Status = Status & " Key 'U3' INI defined string: " & _
chr(34) & UserStr3 & chr(34) & EOL
if (UserStr4<>"" ) then Status = Status & " Key 'U4' INI defined string: " & _
chr(34) & UserStr4 & chr(34) & EOL
if (UserStr5<>"" ) then Status = Status & " Key 'U5' INI defined string: " & _
chr(34) & UserStr5 & chr(34) & EOL

```

The most extensive changes were to the Communications() function where the key handling had to be added.

```

' application key inserted here-----
Case 49 ' numeric key "1"
Print #2, "T1" ' set timing to 1 minute
Print "Report Time set to 1 minute"
Case 50 ' numeric key "2"
Print #2, "T2" ' set timing to 2 minutes
Print "Report Time set to 2 minutes"
Case 51 ' numeric key "3"
Print #2, "T3" ' set timing to 3 minutes
Print "Report Time set to 3 minutes"
Case 52 ' numeric key "4"
Print #2, "T4" ' set timing to 4 minutes
Print "Report Time set to 4 minutes"
Case 53 ' numeric key "5"
Print #2, "T5" ' set timing to 5 minutes
Print "Report Time set to 5 minutes"
Case 54 ' numeric key "6"
Print #2, "T6" ' set timing to 10 minutes
Print "Report Time set to 10 minutes"
Case 55 ' numeric key "7"
Print #2, "T7" ' set timing to 15 minutes
Print "Report Time set to 15 minutes"
Case 55 ' numeric key "8"
Print #2, "T8" ' set timing to 20 minutes
Print "Report Time set to 20 minutes"
Case 55 ' numeric key "9"
Print #2, "T9" ' set timing to 30 minutes
Print "Report Time set to 30 minutes"
Case 55 ' numeric key "0"
Print #2, "T0" ' set timing to 60 minutes
Print "Report Time set to 60 minutes"
Case 68,100 ' Alpha Key "D" or "d"
Print #2, "DB" ' set debug mode
if (DebugMode=False) then
DebugMode=True
Print "Turn Debug mode on"
else
DebugMode=False
Print "Turn Debug mode off"
End if
Case 70,102 ' Alpha Key "F" or "f"
if (FahrenheitMode=False) then
FahrenheitMode=True
Print "Turn Fahrenheit Mode on"
Print #2, "FT"
else

```



```

        FahrenheitMode=False
        Print "Turn Fahrenheit Mode off"
        Print #2, "FF"
    End if
Case 67,99          ' Alpha Key "C" or "c"
    if (CelsiusMode=False) then
        CelsiusMode=True
        Print "Turn Celsius Mode on"
        Print #2, "CT"
    else
        CelsiusMode=False
        Print "Turn Celsius Mode off"
        Print #2, "CF"
    End if
Case 65,97          ' Alpha Key "A" or "a"
    if (AVRMode=False) then
        AVRMode=True
        Print "Turn Avr Internal Mode on"
        Print #2, "IT"
    else
        AVRMode=False
        Print "Turn AVR Internal Mode off"
        Print #2, "IF"
    End if
Case 69,101         ' Alpha Key "E" or "e"
    if (EEMode=False) then
        EEMode=True
        Print "Turn EEMode Mode on, ***NEXT AVR RESTART***"
        Print #2, "E+"
    else
        EEMode=False
        Print "Turn EEMode Mode off, ***NEXT AVR RESTART***"
        Print #2, "E-"
    End if
Case 82,114         ' Alpha Key "R" or "r"
    Print #2, "00"
    if (RoundMode=False) then
        RoundMode=True
        Print "Turn Rounding Mode on"
    else
        RoundMode=False
        Print "Turn Rounding Mode off"
    End if
Case 86,118         ' Alpha Key "V" or "v"
    if (RawMode=False) then
        RawMode=True
        Print "Turn Raw Reading Mode on"
        Print #2, "RT"
    else
        RawMode=False
        Print "Turn Raw Reading Mode off"
        Print #2, "RF"
    End if

Case 77,109         ' Alpha Key "M" or "m"
    Print "Setting minimal mode (Fahrenheit only)"
    if (DebugMode=True) then Print #2, "DB"
    DebugMode=False
    if (RoundMode=False) then Print #2, "00"
    DebugMode=True
    Print #2, "RF CF IF FT ST" ' Set minimal mode
    RawMode=False
    CelsiusMode=False
    AvrMode=False
    FahrenheitMode=True

```

```

Case 66,98      ' Alpha Key "B" or "b"
  Print #2, "W-" ' restore from backup
Case 83,115     ' Alpha Key "S" or "s"
  Print #2, "ST" ' print Status
Case 81,113     ' Alpha Key "Q" or "q"
  Print #2, "ED" ' Dump EEPROM Storage
Case 76,108     ' Alpha Key "L" or "l"
  Print #2, "??" ' print AVR help
Case 85,117,26,122 ' Alpha Key "U" or "u"
  ' Send user defined String, String is defined in INI file
  ' This violates the 'Keep it shut' rule
  ' but this application only sends data once a minute
  Sleep (1000)   ' allow up to one second second keypress
  K=ASC(InKey)
  if (K=49) and UserStr1<>"" then print #2, UserStr1
  if (K=50) and UserStr2<>"" then print #2, UserStr2
  if (K=51) and UserStr3<>"" then print #2, UserStr3
  if (K=52) and UserStr4<>"" then print #2, UserStr4
  if (K=53) and UserStr5<>"" then print #2, UserStr5
case 60,44      ' keys '<' and ','
  OffsetVal=OffsetVal-0.25
  if OffsetVal=0 then OffsetVal=0.0001
  Print #2, "D0 " & OffsetVal
case 62,46      ' keys '>' and '.'
  OffsetVal=OffsetVal+0.25
  if OffsetVal=0 then OffsetVal=0.0001
  Print #2, "D0 " & OffsetVal

```

The most challenging changes were to the ProcessData() function. In this case we needed to do two things. First we need to capture the “Degree Offset” value when it is sent from the Arduino as part of the startup or when the user requests the Status to be printed. Two variables were added to the function to support this effort.

```

' added for Thermometer Application
Dim C as String      ' used for first character of string
Dim P as Byte = 0    ' index to location in string

' . . . .
'--- added for thermometer application -----
' check to see if this is a non-report line
' we do not want to add date and time to non-report lines
' secondly we want to capture the degree offset if we can
C=Left(Buffer,1)
if C=";" then      ' we have a non-report line
  if instr(Buffer,"Offset:")>0 then ' check for Degree Offset
    P=Instr(Buffer, chr(9))         ' find TAB character
    OffsetVal=Val(trim(Mid(Buffer,P+1))) ' capture and convert value
  end if
end if

```

Secondly we need to AVOID printing the data and time for non-report lines. The previously defined variable “C” was added as a condition to the control statements.

```

'----- display write -----
' If (AddDateTime) then
If ((AddDateTime) and (C <>"")) then

' . . .

'----- file write -----
If SendToFile<>0 then
  ' setup for file flush
  LogFileHandle=cast(FILE Ptr,Fileattr(3,fbFileAttrHandle))
  ' If (AddDateTime) then
  If ((AddDateTime) and (C <>"")) then

```



Note:

During testing two bugs were found in the original Receiver program. These were corrected and the changes were rolled back into the original code.

Conclusion

It has taken a lot more time than I thought --- months instead of days. In the end I believe that I have succeeded in creating the device that I was desired. It will require some of amount of time and observation to verify that. It is however something that can be built upon and enhanced.

Possible Enhancements

Temperature Accuracy

Increasing the accuracy is obvious. That comes down to three variables. The first is the sensor. The good news is that if one has implemented the sensor extension via the 3.5mm audio jack then replacing the sensor is very simple. If one shops carefully one might be able to obtain the more accurate versions at a discount. In the end I found a supplier with the **LM34CAH** deeply discounted.

The second variable is the voltage reference. The internal reference that we are using is convenient but not very accurate. Atmel's specifications give it a variance of 10%. Replacing the Reference voltage source with a precision external IC such as the [LT1004-1.2](#) would yield a much more predictable output as well as raising the top of the temperature range to 120 degrees Fahrenheit. This would require some additional capacitors and a bit more wiring.

The last variable is the accuracy/quality of our temperature calibration source. The more accurate our reference thermometer is the more accurate our calibration can be. An alternative is shaved ice bath calibration but that only gives us one point of reference. NIST certified fractionally calibrated laboratory thermometers may sometimes be acquired used from sources such as EBAY.com. These retail new in the range of US\$300 to US\$600.

EEPROM Storage Mode

An alternate power source is needed for this mode. A convenient solution is a compatible iPhone charge or battery pack. The Nano has 13 digital pins. The use of a resistor and a two position switch on one of these pins might be a more convenient method of controlling the EEPROM storage mode. A more sophisticated storage algorithm could be developed to increase the amount of data stored in this mode (*especially in the case of the ATmega168*). One might want to incorporate a SD Card for massive data storage.

Number of sensors

The Nano has 8 analog pins. It can support up to eight sensors. This would require a number of changes in the software but the underlying structure is provided in both the code and protocol. A larger breadboard or [shield](#) would be needed or perhaps even a dedicated purpose built circuit board.

Remote Data Collection

There are numerous wireless and Ethernet based modules available for the Arduino line. One of the more interesting ones is called the [Electric IMP](#). One of these might be used to allow collecting data from the Nano in remote locations.

LCD Display

There are several LCD displays available and an Arduino library as well. The application could be converted to a standalone application with its own power supply and display. In this case you might want to consider using a UNO instead of a Nano.

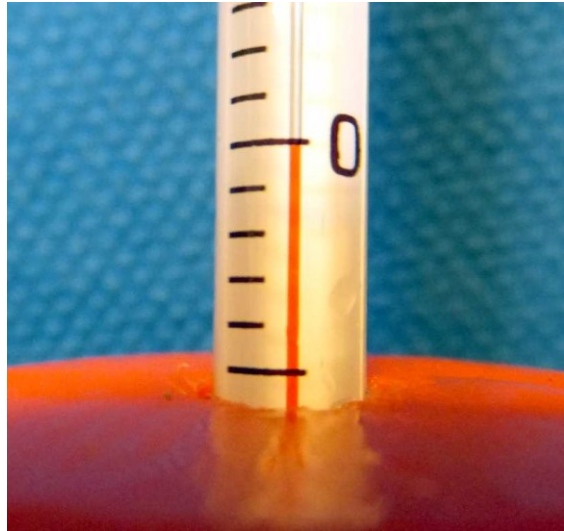
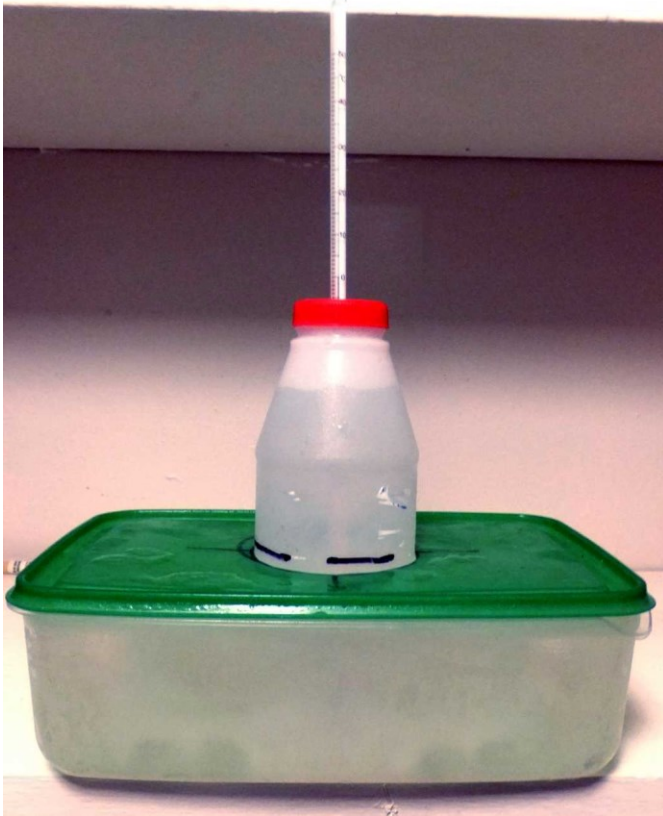
GUI Interface

We have defined a protocol for communicating with the Arduino Temperature device. That protocol could be used to develop a GUI (*Graphic User Interface*) program to control the device. Unfortunately at this time the

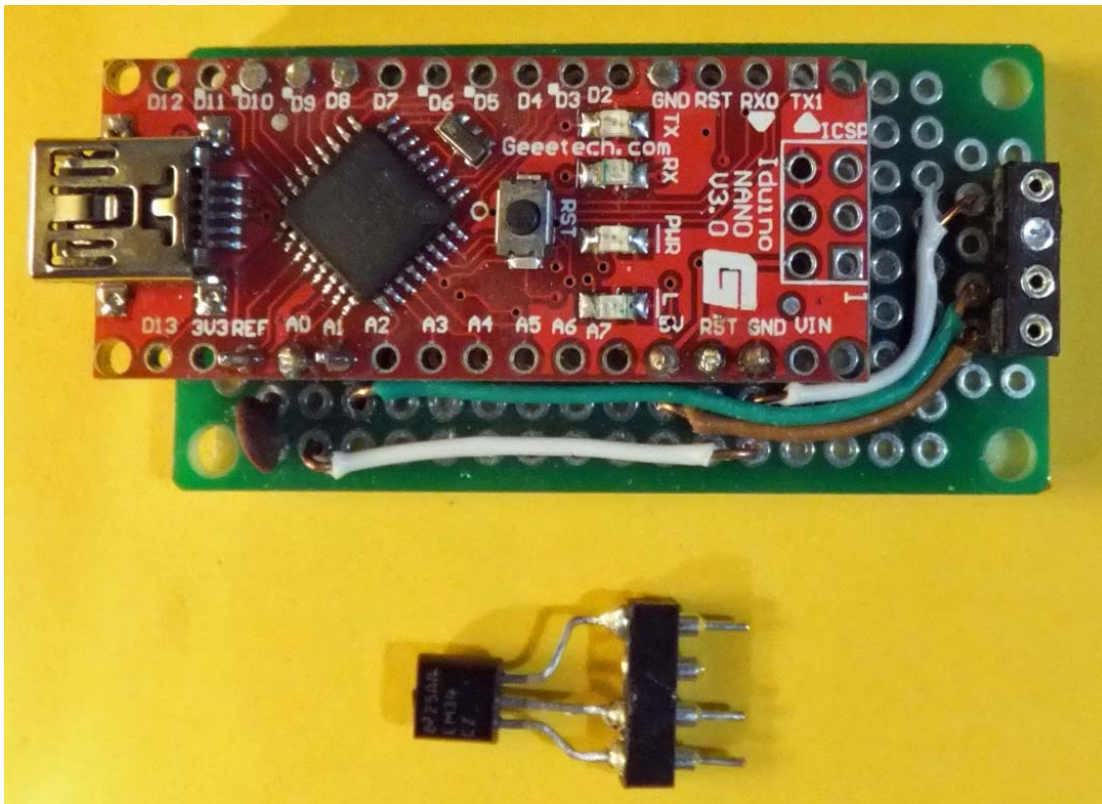
availability of a stable cross platform GUI API may limit this to a single OS. That is the principle reason that the program developed in this document targeted used a CUI (*Character User Interface*) environment.

Photo Gallery

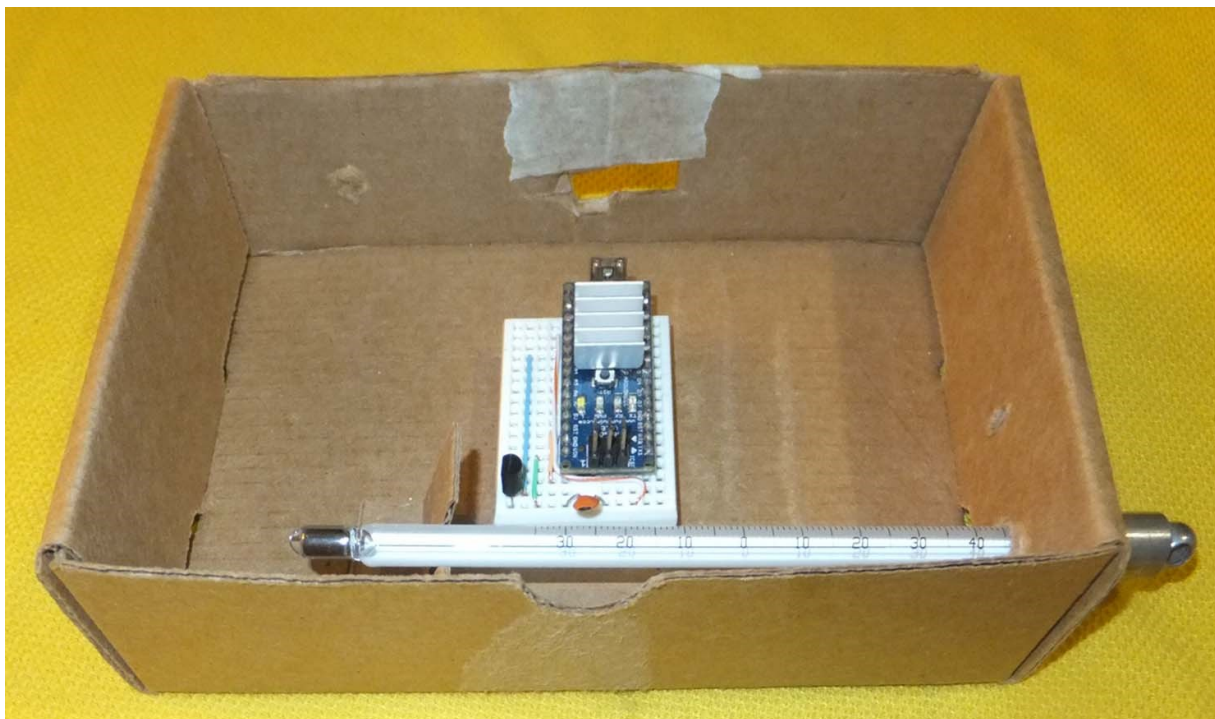
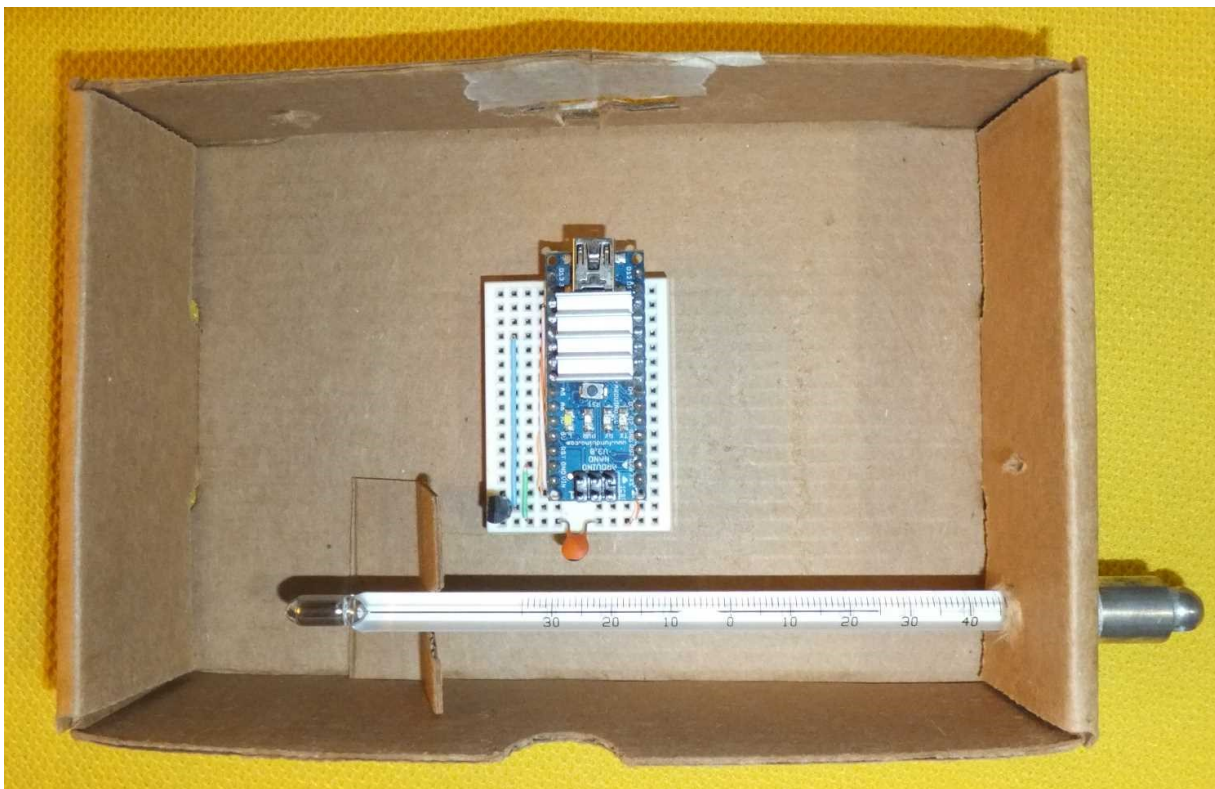
These are some random pictures of the hardware used during the development.



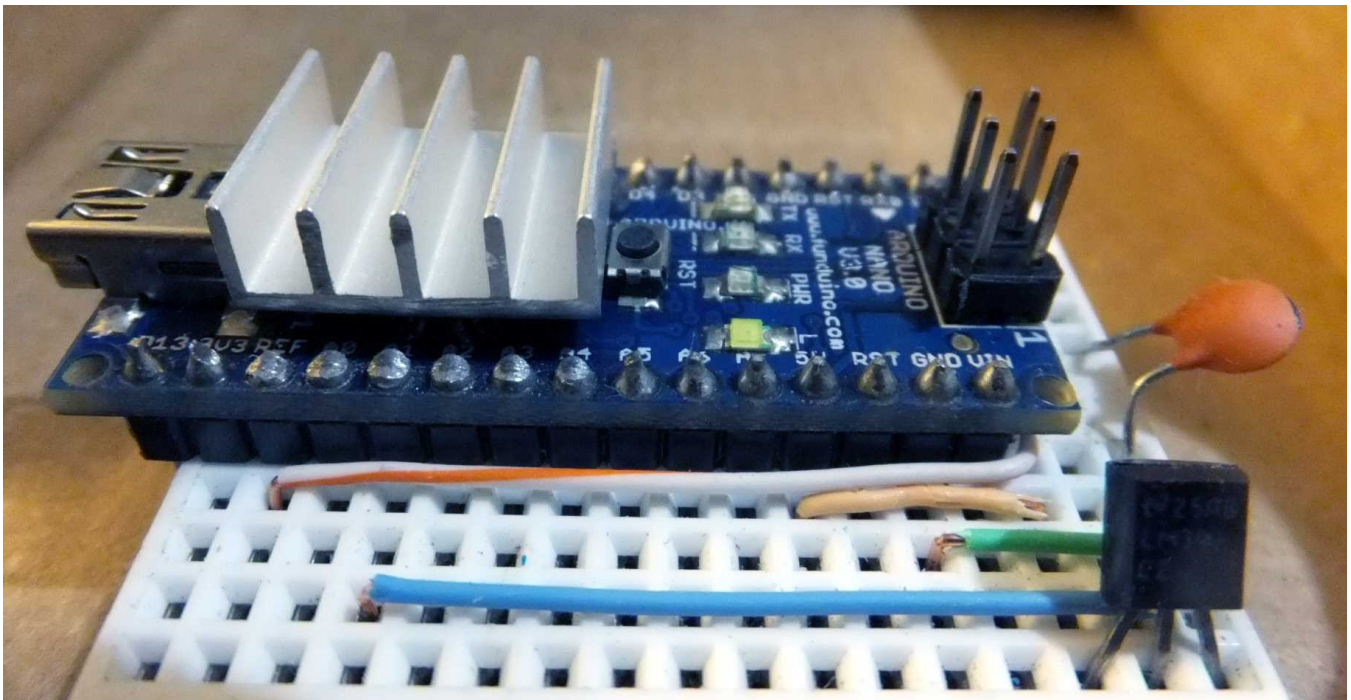
This is an 305mm Omega -50 to 50 Degree Celsius full immersion spirit thermometer (*Part number: GT-736620*) frozen in ice to check the calibration at zero degrees. It looks close enough (*there is a different background because I had to move it to get a close up*). It was purchased from Omega but it is actually manufactured by [Sper Scientific](#) in Taiwan. This is a new acquisition. I have not had the opportunity yet to compare it the Arduino but I did compare it the BCR and the two seem to coincide with each other (*for a change*).



This was my first attempt to make a plug-in sensor that could have an extension. I tried to attach use CAT 6 LAN cable with pieces of an 8 pin DIP socket. The cable was way too stiff even when striped down to two pair. Trying to solder the wires to the tiny pins also proved to be a challenge. That is when I came up with the idea of using the stereo cables. The missing pin was so that the sensor could not be plugged in backwards. Note that the matching hole is filled with solder.



This was the primary development/testing platform. The cardboard box served two purposes. One was to minimize the effects of any stray air currents. The other and more important was to hold the mercury bulb thermometer in close proximity where I could read it.



This is a close up of the Nano on the breadboard in the cardboard box. The heat sink was added to the top of the MPU to try and bring it closer to the air temperature (did not work). That is the LM34 temperature sensor in the lower right corner.



BCR mercury bulb thermometer acquired via EBAY. As far as I can tell it is fairly accurate. It is a bit difficult to read (*due to the small size and my less than perfect 6 decade old eyes*).

Appendix: Atmel MPU Table

These two tables list some of the common Atmel AVR MPU's that you may encounter.

Atmel MPU	Sig Byte 0X000	Sig Byte 0X001	Sig Byte 0X002	FLASH (bytes)	EEPROM (bytes)	SRAM (bytes)	Boot Loader	I/O Pins	PWM	ADC	ADC Chns	USAR T	Voltage (range)	Speed (max)	USB	Serial #
ATmega1280	0x1E	0x97	0x03	131,072	4,096	8,192	Yes	86	12	10 bits	16	4	2.7-5.5	16Mhz	No	No
ATmega1280V	0x1E	0x97	0x03	131,072	4,096	8,192	Yes	86	12	10 bits	16	4	1.8-5.5	8Mhz	No	No
ATmega1281	0x1E	0x97	0x04	131,072	4,096	8,192	Yes	54	6	10 bits	8	2	2.7-5.5	16Mhz	No	No
ATmega1281V	0x1E	0x97	0x04	131,072	4,096	8,192	Yes	54	6	10 bits	8	2	1.8-5.5	8Mhz	No	No
ATmega168 ***	0x1E	0x94	0x06	16,384	512	1,024	Yes	23	6	10 bits	6 or 8	1	2.7-5.5	20MHz	No	No
ATmega168A	0x1E	0x94	0x06	16,384	512	1,024	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	1.8-5.5	20MHz	No	No
ATmega168P	0x1E	0x92	0x0B	16,384	512	1,024	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	2.7-5.5	20MHz	No	No
ATmega168PA	0x1E	0x94	0x0B	16,384	512	1,024	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	1.8-5.5	20MHz	No	No
ATmega168PV	0x1E	0x92	0x0B	16,384	512	1,024	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	1.8-5.5	10MHz	No	No
ATmega16U2	0x1E	0x94	0x89	16,384	512	512	Yes	22	2	No	No	1	2.7-5.5	16Mhz	Yes	Yes
ATmega16U4	0x1E	0x94	0x88	16,384	512	1280	Yes	26	8	10 bits	12 <input checked="" type="checkbox"/>	1	2.7-5.5	16Mhz	Yes	Yes
ATmega2560	0x1E	0x98	0x01	262,144	4,096	8,192	Yes	86	12	10 bits	16	4	4.5-5.5	16Mhz	No	No
ATmega2560V	0x1E	0x98	0x01	262,144	4,096	8,192	Yes	86	12	10 bits	16	4	1.8-5.5	8Mhz	No	No
ATmega2561	0x1E	0x98	0x02	262,144	4,096	8,192	Yes	54	6	10 bits	8	2	4.5-5.5	16Mhz	No	No
ATmega2561V	0x1E	0x98	0x02	262,144	4,096	8,192	Yes	54	6	10 bits	8	2	1.8-5.5	8Mhz	No	No
ATmega328	0x1E	0x95	0x14	32,768	1,024	2,048	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	1.8-5.5	20MHz	No	No
ATmega328P ***	0x1E	0x95	0x0F	32,768	1,024	2,048	Yes	23	6	10 bits	6 or 8 <input checked="" type="checkbox"/>	1	1.8-5.5	20MHz	No	No
ATmega32U2	0x1E	0x95	0x8A	32,768	1,024	1,024	Yes	22	2	No	No	1	2.7-5.5	16Mhz	Yes	Yes
ATmega32U4 ***	0x1E	0x95	0x87	32,768	1,024	2560	Yes	26	8	10 bits	12 <input checked="" type="checkbox"/>	1	2.7-5.5	16Mhz	Yes	Yes
ATmega48P	0x1E	0x92	0x0A	4,096	256	512	No	23	6	10 bits	6 or 8	1	2.7-5.5	20MHz	No	No
ATmega48PA	0x1E	0x92	0x0A	4,096	256	512	No	23	6	10 bits	6 or 8	1	1.8-5.5	20MHz	No	No
ATmega48PV	0x1E	0x92	0x0A	4,096	256	512	No	23	6	10 bits	6 or 8	1	1.8-5.5	10MHz	No	No
ATmega640	0x1E	0x96	0x08	64,096	4,096	8,192	Yes	86	12	10 bits	16	4	2.7-5.5	16Mhz	No	No
ATmega640V	0x1E	0x96	0x08	64,096	4,096	8,192	Yes	86	12	10 bits	16	4	1.8-5.5	8Mhz	No	No

Atmel MPU	Sig Byte 0X000	Sig Byte 0X001	Sig Byte 0X002	FLASH (bytes)	EEPROM (bytes)	SRAM (bytes)	Boot Loader	I/O Pins	PWM	ADC	ADC Chns	USAR T	Voltage (max)	Speed (max)	USB	Serial #
ATmega88A	0x1E	0x93	0x0A	8,192	512	1,024	Yes	23	6	10 bits	6 or 8	1	1.8-5.5	20MHz	No	No
ATmega88P	0x1E	0x92	0x0F	8,192	512	1,024	Yes	23	6	10 bits	6 or 8	1	2.7-5.5	20MHz	No	No
ATmega88PA	0x1E	0x93	0x0F	8,192	512	1,024	Yes	23	6	10 bits	6 or 8	1	1.8-5.5	20MHz	No	No
ATmega88PV	0x1E	0x92	0x0F	8,192	512	1,024	Yes	23	6	10 bits	6 or 8	1	1.8-5.5	10MHz	No	No
ATmega8U2	0x1E	0x93	0x89	8,192	512	512	Yes	22	2	No	No	1	2.7-5.5	16Mhz	Yes	Yes
ATtiny10	0x1E	0x90	0x03	1,024	No	32	No	4	2	8 bits	4	No	1.8-5.5	12MHz	No	No
ATtiny25	0x1E	0x91	0x08	2,048	128	128	Yes	6	2	10 bits	4	No	2.7-5.5	20MHz	No	No
ATtiny25V	0x1E	0x91	0x08	2,048	128	128	Yes	6	2	10 bits	4	No	1.8-5.5	10MHz	No	No
ATtiny4	0x1E	0x8F	0x0A	512	No	32	No	4	2	No	No	No	1.8-5.5	12MHz	No	No
ATtiny45	0x1E	0x92	0x06	4,096	256	256	Yes	6	2	10 bits	4	No	2.7-5.5	20MHz	No	No
ATtiny45V	0x1E	0x92	0x06	4,096	256	256	Yes	6	2	10 bits	4	No	1.8-5.5	10MHz	No	No
ATtiny5	0x1E	0x8F	0x09	512	No	32	No	4	2	8 bits	4	No	1.8-5.5	12MHz	No	No
ATtiny85	0x1E	0x93	0x08	8,192	512	512	Yes	6	2	10 bits	4	No	2.7-5.5	20MHz	No	No
ATtiny85V	0x1E	0x93	0x08	8,192	512	512	Yes	6	2	10 bits	4	No	1.8-5.5	16Mhz	No	No
ATtiny9	0x1E	0x90	0x08	1,024	No	32	No	4	2	No	No	No	1.8-5.5	12MHz	No	No

*** These chips are known to have been used in boards sold as Arduino Nano or labeled Arduino Nano compatible

☒ in the “ADC chns” column means that an internal temperature sensor is also included (*I have not verified the rest*).

These values were taken directly from the Atmel datasheets. Any inaccuracies are the results (*particularly in the Speed and voltage columns*) of deciphering the sometimes confusing layout of those datasheets. Actual operating speed and voltage are directly linked.

All Atmel microcontrollers have a three-byte signature code which identifies the device. Those are identified in this table as: Sig Byte. These are NOT unique identifiers as several MPUs share the same signature bytes.

Depending on the MPU package the number of available ADC channels may vary.

USART is the acronym for “[Universal asynchronous receiver/transmitter](#)”. In this table it is used to indicate the number of hardware based TTL serial communication channels supported.

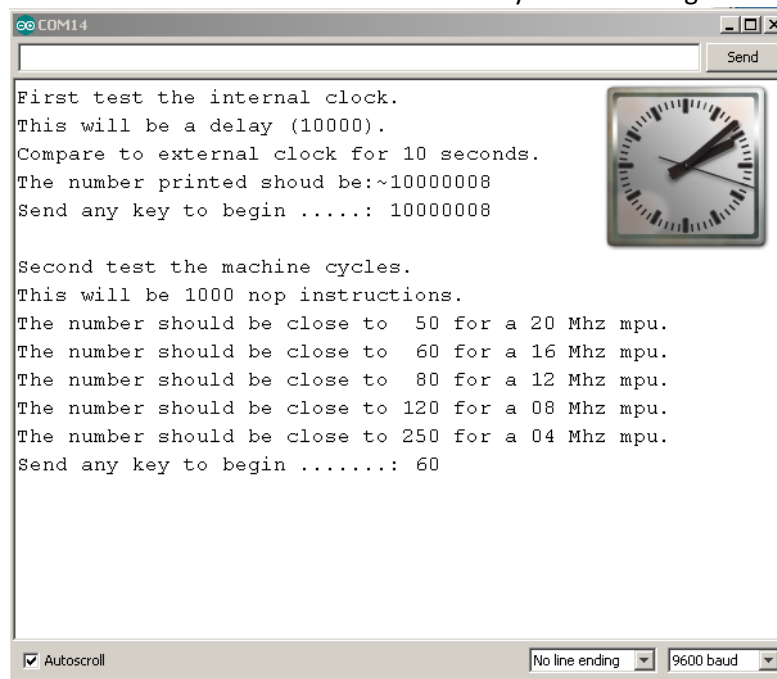
Appendix: Arduino Check Speed

What if you have gotten an Arduino from a source off the internet that claimed it was a 16 Mhz device but you think it might be somewhat slower. You look at the crystal first but it turns out that the speed is not listed or illegible. Is there a way to verify the speed?

Yes there is. The Arduino IDE allows you to imbed assembly instructions. The one of interest here is called "nop" and means no operation. It takes does nothing but requires exactly one machine cycle. Well how long does a machine cycle take ?

```
At 20 MHS one cycle = 1/20,000,000 = 0.0000000500 seconds
At 16 MHS one cycle = 1/16,000,000 = 0.0000000625 seconds
At 12 MHS one cycle = 1/12,000,000 = 0.0000000833 seconds
At 8 MHS one cycle = 1/8,000,000 = 0.0000001250 seconds
At 6 MHS one cycle = 1/6,000,000 = 0.0000001667 seconds
At 4 MHS one cycle = 1/4,000,000 = 0.0000002500 seconds
```

The concept of this program is first check the internal clock of the Arduino by comparing a delay (10000) instruction to an external clock. Then we execute 1,000 "nop" instructions and compare the time to what we expect the value to be. Both numbers should be close. If not then you need to dig a bit further.



(The font size had to be reduced to get this in to MS word).

```
/* CheckSpeed */
/* what speed is the Arduino running at and how can it be verified ?

http://playground.arduino.cc/Main/AVR
"for shorter delays use assembly language call 'nop' (no operation).
Each 'nop' statement executes in one machine cycle
(at 16 MHz) yielding a 62.5 ns (nanosecond) delay. "

one nano second = 1 second /1,000,000,000
At 20 MHS one cycle = 1/20,000,000 = 0.0000000500
At 16 MHS one cycle = 1/16,000,000 = 0.0000000625
At 12 MHS one cycle = 1/12,000,000 = 0.0000000833
At 8 MHS one cycle = 1/8,000,000 = 0.0000001250
At 6 MHS one cycle = 1/6,000,000 = 0.0000001667
At 4 MHS one cycle = 1/4,000,000 = 0.0000002500
*/
```

```

byte junk = 0;          /* incoming serial byte */
unsigned long Time;
unsigned long Start;

void setup()
{
  // First verify the timer is correct by check the number of seconds against an external clock
  Serial.begin (9600);
  Serial.println ("First test the internal clock.");
  Serial.println ("This will be a delay (10000).");
  Serial.println ("Compare to external clock for 10 seconds.");
  Serial.println ("The number printed should be:~10000000");
  Serial.print ("Send any key to begin ..... ");
  while (Serial.available() == 0);          // wait for key from user
  while (Serial.available() > 0) {junk = Serial.read();} // collect and discard user input
  Start=micros();
  delay (10000);
  Time=micros()-Start;
  // Serial.println("Check Time 10 seconds: ");
  Serial.println (Time);

  Serial.println ();
  Serial.println ("Second test the machine cycles.");
  Serial.println ("This will be 1000 nop instructions.");
  Serial.println ("The number should be close to 50 for a 20 Mhz mpu.");
  Serial.println ("The number should be close to 60 for a 16 Mhz mpu.");
  Serial.println ("The number should be close to 80 for a 12 Mhz mpu.");
  Serial.println ("The number should be close to 120 for a 08 Mhz mpu.");
  Serial.println ("The number should be close to 250 for a 04 Mhz mpu.");
  Serial.print ("Send any key to begin ..... ");
  while (Serial.available() == 0);          // wait for key from user
  while (Serial.available() > 0) {junk = Serial.read();} // collect and discard user input

  Start=micros();
  // each line is 10 nop instructions
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");

  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  //----1
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");

  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  //----2
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");

  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  //----3
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");

  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");
  __asm__ ("nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t" "nop\n\t");

```


Appendix: AVR ADC Sensor Registers

ADC Links:

<http://www.marulaberry.co.za/index.php/tutorials/code/arduino-adc/> (Marulaberry Projects)

[Arduino Playground, Internal Temperature Sensor](#) (arduino.cc)

[AVR122: Calibration of the AVR's internal temperature reference](#) (Atmel)

[Arduino / AVR internal temperature sensor interface](#) (avdweb)

[ANALOG INPUTS \(ANALOG TO DIGITAL CONVERTER\)](#) (QEEWiki)

[Analogue to Digital Conversion on an ATmega168](#) (protostack)

ADMUX, ADCH, ADCL are AVR eight bit registers (*a “register” is special dedicated memory location in the heart of the processor*). The **ADMUX** register holds the operational settings for the Analog to Digital converter. **ADSCRA** is the Control and Status register for the Analog to Digital converter. **ADCH** (*high byte*) and **ADCL** (*low byte*) are used to store the result from the conversion.

Register	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

The top two bits of ADMUX (**REFS1** and **REFS0**) control the source of the reference voltage.

REFS1	REFS0	Source	Reference Voltage Source
0	0	External	AREF Pin, Internal voltage reference turned off, Nano Pin number 18
0	1	Default	AVcc with external capacitor on AREF pin
1	0	n/a	Reserved
1	1	Internal***	Internal, 1.1 volts for ATmega168/328 or 2.56 for the ATmega8

*** The external AREF pin is directly connected to the ADC, and the reference voltage can be made more immune to noise by connecting a capacitor between the AREF pin and ground. VREF can also be measured at the AREF pin with a high impedance voltmeter.

The bottom four bits of ADMUX (**MUX3**, **MUX2**, **MUX1** and **MUX0**) control the source of the voltage to be read. The ADC is optimized for analog signals with an output impedance of approximately 10 k ohms or less.

MUX 3,2,1,0 (binary)	Input Voltage Source
0000	ADC0, analog pin 0, Nano pin number 19, (A0)
0001	ADC1, analog pin 1, Nano pin number 20, (A1)
0010	ADC2, analog pin 2, Nano pin number 21, (A2)
0011	ADC3, analog pin 3, Nano pin number 22, (A3)
0100	ADC4, analog pin 4, Nano pin number 23, (A4)
0101	ADC5, analog pin 5, Nano pin number 24, (A5)
0110	ADC6, analog pin 5, Nano pin number 24, (A6)
0111	ADC7, analog pin 6, Nano pin number 25, (A7)
1000	ADC8, analog pin 7, Nano pin number 26, (Internal Temperature Sensor)
1001 – 1101	(reserved)
1110	1.1 volt (internal reference voltage)
1111	0 volts (ground)

ADALAR controls how the Analog to Digital converter stores the result in the two registers ADCH and ADCL. One must be careful when reading these two storage registers. Reading ADCH causes the ADC to update. So always read ADHL first. The second mode is useful if you only want an 8 bit AD conversion and read ADHC. You might use this if you are trying to build a really fast routine where speed is more important than range (*perhaps a software Oscilloscope*).

ADALAR	Register	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ADCH	-	-	-	-	-	-	ADC9	ADC8
0	ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
1	ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
1	ADCL	ADC1	ADC0	-	-	-	-	-	-

Now about the ADSCSRB register ... I found this one a bit confusing.

Register	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN (Analog to Digital Enable) enables the AD converter subsystem. This bit needs to be set before any conversion takes place.

ADSC (Analog to Digital Start) is set to 1 when you want to start an AD conversion process. When the conversion is finished, the value reverts back to 0.

ADATE (Analog to Digital Auto Trigger Enable) is used to set the mode of operation. The default is 0 for single read. The alternative is a “triggered” operation (see ADCSRB below).

ADIF (Analog to Digital Interrupt Finished)

ADIE (Analog to Digital Interrupt Enable)

ADPS, **ADPS1** and **ADPS0** are used to specify a system clock division factor for ADC speed.

ADPS 2,1,0 (binary)	Division Factor
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

The ADC has a recommended maximum ADC clock speed of 200 kHz. “However, frequencies up to 1 MHz (50,000 samples per second) do not reduce the ADC resolution significantly”. The default division factor is 128:

At 16 Mhz: $16,000,000/128 = 125,000$ (or 125 kHz or approximately 8,621 samples per second)

The ATmega168 and ATmega328 have an additional ADC control register **ADCSRB** to use with “Free Running” mode. We will not be using it but for the sake of completeness the description is included.

Register	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
ADCSRB	-	ACME	-	-	-	ADTS2	ADTS1	ADTS0

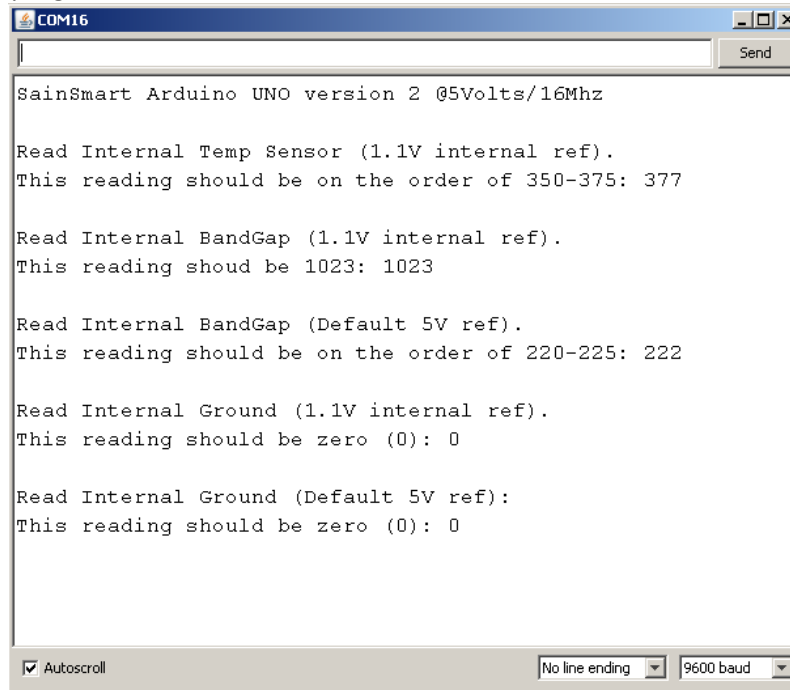
ACME (Analog Comparator Multiplexer Enable) must be set to 1 to use the ADC multiplexer.

The bottom three bits of ADCSRB (**ADTS2**, **ADTS1** and **ADTS0**) set the trigger source to begin an ADC conversion.

ADTS 2,1,0 (binary)	ADC Trigger Source
000	Free Running. When a ADC conversion is done another begins This is the same as as ATMega. (note: a conversion takes ~ 13.5 clock cycles)
001	Analog Comparator (compares
010	External Interrupt request 0
011	Timer/Counter0 Compare Match A
100	Timer/Counter0 Overflow
101	Timer/Counter1 Compare Match B
110	Timer/Counter1 Overflow
111	Timer/Counter1 Capture Event

Appendix: ADC Function test

I have some "Iduino Nano version 3.0" boards with Atmega168 mpu chips. I was perplexed that I was getting strange readings from the internal temperature sensor. So I wrote this program to test the operation of the ADC using the internal references. Then I found out that the ATmega168 does NOT have an internal temperature sensor. Still maybe the program will be useful.



```
COM16
SainSmart Arduino UNO version 2 @5Volts/16Mhz

Read Internal Temp Sensor (1.1V internal ref).
This reading should be on the order of 350-375: 377

Read Internal BandGap (1.1V internal ref).
This reading should be 1023: 1023

Read Internal BandGap (Default 5V ref).
This reading should be on the order of 220-225: 222

Read Internal Ground (1.1V internal ref).
This reading should be zero (0): 0

Read Internal Ground (Default 5V ref):
This reading should be zero (0): 0

Autoscroll No line ending 9600 baud
```

Main Program Code:

```
/* Program to test Internal function ADC using internal references */
unsigned long Time;
float save;

void setup()
{ word raw;
  Serial.begin(9600);
  // put arduino board description here
  Serial.println("Board Description -----");
  Serial.println();

  raw = avrRawTemp(1024);
  Serial.println("Read Internal Temp Sensor (1.1V internal ref).");
  Serial.print("This reading should be on the order of 325-400: ");
  Serial.println(raw);
  Serial.println();

  raw = avrBandGap1(1024);
  Serial.println("Read Internal BandGap (1.1V internal ref).");
  Serial.print("This reading should be 1023: ");
  Serial.println(raw);
  Serial.println();

  raw = avrBandGap2(1024);
  Serial.println("Read Internal BandGap (Default 5V ref).");
  Serial.print("This reading should be on the order of 220-225: ");
  Serial.println(raw);
  Serial.println();
```

```

raw = avrGround1(1024);
Serial.println("Read Internal Ground (1.1V internal ref).");
Serial.print("This reading should be zero (0): ");
Serial.println(raw);
Serial.println();

raw = avrGround2(1024);
Serial.println("Read Internal Ground (Default 5V ref): ");
Serial.print("This reading should be zero (0): ");
Serial.println(raw);
Serial.println();
}

void loop() {

}

```

Function Code:

```

// cbi and sbi are standard (AVR) methods for setting,
// or clearing, bits in PORT (and other) variables.
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif
//-----

word avrRawTemp(word samples)
{ // gets raw reading in the range 0 to 1023 from internal temperature sensor
  // using internal volt 1.1 voltage ref
  // Should return ~325-400 for normal enviroment temperatures
  unsigned long RawSum=0; // used to sum samples for averaging
  word RawTemp=0; // used to accumulate 10 bit ADC readings
  word test=0; // used to count samples
  byte exp=0; // samples = 2 to the exp power, used as
shift operand
  unsigned long Start=micros(); // this was used for benchmark timing

  // set system clock divisor to 128
  // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
  sbi(ADCSRA, ADPS2); // bit 2 of ADCSRA, system clock divisor
  sbi(ADCSRA, ADPS1); // bit 1 of ADCSRA, system clock divisor
  sbi(ADCSRA, ADPS0); // bit 0 of ADCSRA, system clock divisor
  cbi(ADCSRA, ADSC); // bit 5 of ADCSRA, disable auto trigger mode
  sbi(ADCSRA, ADEN); // bit 7 of ADCSRA, enable ADC

  // turn on internal reference, right-shift ADC buffer, ADC channel = internal temp sensor
  ADMUX = B11001000; // High Nibble: 1100 = internal 1.1 Vref
  // Low Nibble: 1000 = ADC chn 8 Temp sensor
  delay(10); // wait a bit for the analog reference to stabilize

  // as "C" lacks an expodential or power function (or operator)
  // we must use a resort to loops to calculate the binary exponent
  while (samples>1) { samples /=2; exp++;} // calculate power of 2
  samples=1; // make sure samples = 1 (not 0)
  while (test++ < exp) { samples *=2;} // set samples value to power of two

  test=0; // reset test because we have abused it
  while (test++ < samples) // oversampling loop (for averaging)
  { ADCSRA |= _BV(ADSC); // start the conversion by setting ADSC=1
    while (bit_is_set(ADCSRA, ADSC)); // ADSC is cleared when the conversion
finishes

```

```

        RawTemp = (ADCL | (ADCH << 8));           // get the ADC reading (low byte first)
        RawSum += RawTemp;                         // accumulate 10 bit ADC value
    }
    Time=micros()-Start;                           // record benchmark time
    return ((RawSum)>>exp);                         // averag by shifting bit position,
                                                    // LSBs lost
}

//-----
word avrBandGap1(word samples)
{ // gets raw reading in the range 0 to 1023 from internal Band Gap reference
  // using internal volt 1.1 voltage ref
  // Should return 1023
  unsigned long RawSum=0;
  word RawTemp=0;
  word test=0;
  byte exp=0;
  unsigned long Start=micros();

  sbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
  cbi(ADCSRA, ADSC);
  sbi(ADCSRA, ADEN);

  ADMUX = B11001110;                             // High Nibble: 1100 = internal 1.1 Vref
  delay(10);                                       // Low Nibble: 1110 = 1.1V BandGap (Vbg)

  while (samples>1) { samples /=2; exp++;}
  samples=1;
  while (test++ < exp) { samples *=2;}

  test=0;
  while (test++ < samples)
  { ADCSRA |= _BV(ADSC);
    while (bit_is_set(ADCSRA, ADSC));
    RawTemp = (ADCL | (ADCH << 8));
    RawSum += RawTemp;
    RawTemp=0;
  }
  Time=micros()-Start;
  return ((RawSum)>>exp);
}

//-----
word avrBandGap2(word samples)
{ // gets raw reading in the range 0 to 1023 from internal Band Gap reference
  // using internal volt ref AVCC= 5 voltage ref
  // should return ~225
  unsigned long RawSum=0;
  word RawTemp=0;
  word test=0;
  byte exp=0;
  unsigned long Start=micros();

  sbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
  cbi(ADCSRA, ADSC);
  sbi(ADCSRA, ADEN);

  ADMUX = B01001110;                             // High Nibble: 0100 = internal 5.0 Vref
  delay(10);                                       // Low Nibble: 1110 = 1.1V BandGap (Vbg)

  while (samples>1) { samples /=2; exp++;}

```

```

    samples=1;
    while (test++ < exp) { samples *=2;}

    test=0;
    while (test++ < samples)
    { ADCSRA |= _BV(ADSC);
      while (bit_is_set(ADCSRA, ADSC));
      RawTemp = (ADCL | (ADCH << 8));
      RawSum += RawTemp;
    }
    Time=micros()-Start;
    return ((RawSum)>>exp);
}

//-----
word avrGround1(word samples)
{ // gets raw reading in the range 0 to 1023 from from internal ground reference
  // using internal volt 1.1 voltage ref
  // This should return zero
  unsigned long RawSum=0;
  word RawTemp=0;
  word test=0;
  byte exp=0;
  unsigned long Start=micros();

  sbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
  cbi(ADCSRA, ADIFSC);
  sbi(ADCSRA, ADIFR);

  ADMUX = B11001111; // High Nibble: 1100 = internal 1.1 Vref
  delay(10);         // Low Nibble: 1111 = Internal Ground Ref

  while (samples>1) { samples /=2; exp++;}
  samples=1;
  while (test++ < exp) { samples *=2;}

  test=0;
  while (test++ < samples)
  { ADCSRA |= _BV(ADSC);
    while (bit_is_set(ADCSRA, ADSC));
    RawTemp = (ADCL | (ADCH << 8));
    RawSum += RawTemp;
  }
  Time=micros()-Start;
  return ((RawSum)>>exp);
}

//-----
word avrGround2(word samples)
{ // gets raw reading in the range 0 to 1023 from from internal ground reference
  // using internal volt ref AVCC= 5 voltage ref
  // This should return zero
  unsigned long RawSum=0;
  word RawTemp=0;
  word test=0;
  byte exp=0;
  unsigned long Start=micros();

  sbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
  cbi(ADCSRA, ADIFSC);
  sbi(ADCSRA, ADIFR);

```

```

ADMUX = B01001111;           // High Nibble: 0100 = internal 5.0 Vref
delay(10);                    // Low Nibble: 1111 = Internal Ground Ref

while (samples>1) { samples /=2; exp++;}
samples=1;
while (test++ < exp) { samples *=2;}

test=0;
while (test++ < samples)
{ ADCSRA |= _BV(ADSC);
  while (bit_is_set(ADCSRA, ADSC));
  RawTemp = (ADCL | (ADCH << 8));
  RawSum += RawTemp;
}
Time=micros()-Start;
return ((RawSum)>>exp);
}

```

Appendix: Disabling Auto Reset

Someone in their infinite wisdom decided that the Arduino should automatically reset every time the serial port is opened by the PC.

Reference: <http://arduino.cc/en/Main/ArduinoBoardNano>

Automatic (Software) Reset

Rather than requiring a physical press of the reset button before an upload, the Arduino Nano is designed in a way that allows it to be reset by software running on a connected computer. One of the hardware flow control lines (DTR) of the FT232RL is connected to the reset line of the ATmega168 or ATmega328 via a 100 nanofarad capacitor. When this line is asserted (taken low), the reset line drops long enough to reset the chip. The Arduino software uses this capability to allow you to upload code by simply pressing the upload button in the Arduino environment. This means that the bootloader can have a shorter timeout, as the lowering of DTR can be well-coordinated with the start of the upload.

This setup has other implications. When the Nano is connected to either a computer running Mac OS X or Linux, it resets each time a connection is made to it from software (via USB). For the following half-second or so, the bootloader is running on the Nano. While it is programmed to ignore malformed data (i.e. anything besides an upload of new code), it will intercept the first few bytes of data sent to the board after a connection is opened. If a sketch running on the board receives one-time configuration or other data when it first starts, make sure that the software with which it communicates waits a second after opening the connection and before sending this data.

That makes it easier to use with the IDE because the IDE can reset it to upload a new program. This sounds like a fine idea if the only time that the device is attached to the PC is when it is being programmed. There are however some situations where the auto reset feature is a problem: However two implications that are not mentioned in the Arduino guide are:

The Arduino Nano (*and most other Arduino boards*) cannot be used with a ISCP hardware debugger.

The Arduino Nano (*and most other Arduino boards*) cannot be used as an ISCP programmer.

A third situation is a the target application where the Nano is ALWAYS attached to the PC and one would rather not have it resetting every time the open serial port is opened so one could use multiple applications to communicate with the device. Some versions of the Arduino have a small trace that can be cut to disable this irritating behavior but appears that GRAVITECH did not consider this a useful feature. There are a couple of alternatives:

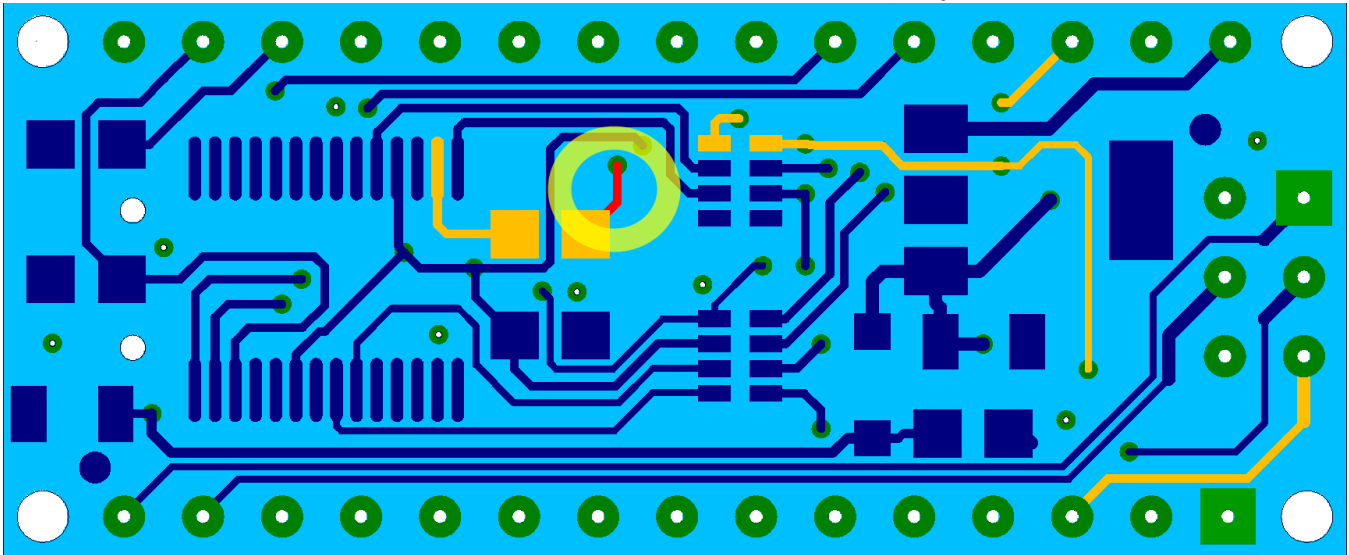
Reference: <http://playground.arduino.cc/Main/DisablingAutoResetOnSerialConnection>

"Stick a 120 ohm resistor in the headers between 5v and reset (you can find these on the isp connector too). 120 is hard to find so just combine resistors. Don't go below 110 ohms or above 124 ohms, and don't do this with an isp programmer attached. You can just pull out the resistor when you want auto-reset back."

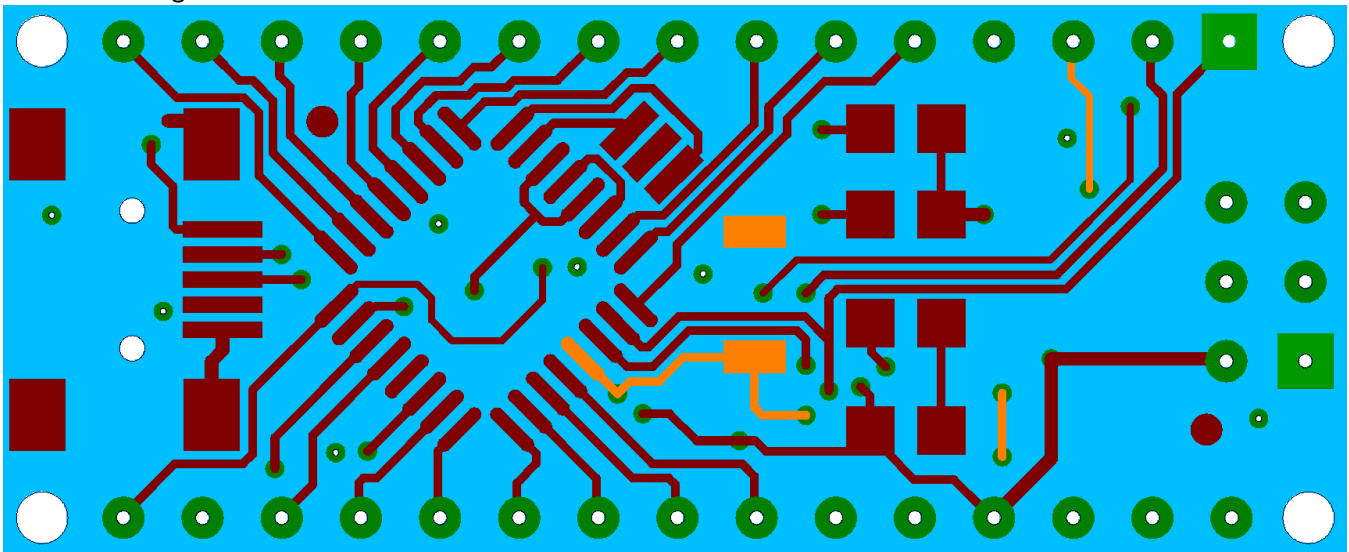
"Another way to avoid autoreset is connecting a capacitor between reset pin and ground. 10 uf should be enough. The ATmega168 is reset by pulsing its reset pin to GND. The Arduino IDE itself cannot create such pulses, but by setting the DTR line to LOW and adding a capacitor (R3 on the pcb, marked red), the reset pin gets sucked to LOW until the capacitor is charged through the internal pull up resistor and R1 - which resets the chip. This works in the same spirit as adding "auto reset" to a chip for proper startup after connecting power."

Neither method is useful in the case of using a hardware debugger.

This is a drawing of the traces on the underside of the Nano board (*looking from the bottom point of view*). The rest circuit traces are shown in orange. The offending trace is shown in red inside the yellow highlighted circle. Cut this trace and Auto Reset will be disabled. The rest of the reset circuit (*and functions*) will be left intact.



For reference this is the top of the Nano board (*looking from the top point of view*) with the reset circuit traces shown in orange.



Appendix: Arduino ElfDump

This is a small utility written in FreeBasic to read the Arduino preferences file and find the build.path. Then it searches for the matching sketch directory and writes a CMD file to use avr-readelf to create a header file (foo.hdr.txt) and avr-objdump to create an assembler file (foo.asm.txt). The CMD files can be edited to change the options. The program is open source and public domain. The command lines for these two utilities are somewhat long and complex. This simplifies the task of creating those command lines. This is a sample of the output “.CMD” file. The command lines are “wrapped” as three lines in the example. Note that the basic options for each of the utilities are included as well.

```
"C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avr-objdump.exe" -S
"c:\users\your.name\documents\arduino\build\ThermometerOne.cpp.elf" > "c:\users\ your.name
\documents\arduino\sketches\ThermometerOne\ThermometerOne.asm.txt"

"C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avr-readelf.exe" -e "c:\users\ your.name
\documents\arduino\build\ThermometerOne.cpp.elf" > "c:\users\ your.name
\documents\arduino\sketches\ThermometerOne\ThermometerOne.hrd.txt"

exit

avr-objdump
Options are:
-a, --archive-headers      Display archive header information
-f, --file-headers        Display the contents of the overall file header
-p, --private-headers     Display object format specific file header contents
-h, --[section-]headers   Display the contents of the section headers
-x, --all-headers         Display the contents of all headers
-d, --disassemble         Display assembler contents of executable sections
-D, --disassemble-all    Display assembler contents of all sections
-S, --source              Intermix source code with disassembly
-s, --full-contents       Display the full contents of all sections requested
-g, --debugging           Display debug information in object file
-e, --debugging-tags      Display debug information using ctags style
-G, --stabs               Display (in raw form) any STABS info in the file
-W, --dwarf               Display DWARF info in the file
-t, --syms                Display the contents of the symbol table(s)
-T, --dynamic-syms        Display the contents of the dynamic symbol table
-r, --reloc               Display the relocation entries in the file
-R, --dynamic-reloc       Display the dynamic relocation entries in the file

readelf
Options are:
-a --all                  Equivalent to: -h -l -S -s -r -d -V -A -I
-h --file-header          Display the ELF file header
-l --program-headers      Display the program headers
-S --section-headers      Display the sections' header
-g --section-groups       Display the section groups
-t --section-details      Display the section details
-e --headers              Equivalent to: -h -l -S
-s --symbols              Display the symbol table
-n --notes                Display the core notes (if present)
-r --relocs               Display the relocations (if present)
-u --unwind               Display the unwind info (if present)
-d --dynamic              Display the dynamic section (if present)
-V --version-info         Display the version sections (if present)
-A --arch-specific        Display architecture specific information (if any).
-c --archive-index        Display the symbol/file index in an archive
-D --use-dynamic          Use the dynamic section info when displaying symbols
```

This is the FreeBasic Source code for the utility.

```
/' small program to automate dumping disassembled elf file
<ARDUINOPATH>/hardware/tools/avr/bin/avr-objdump -d
<BUILDPATH>/<PROJECTNAME>.cpp.elf > <PROJECTNAME>.asm

Source code is placed in public domain: August 2013, Lewis Balentine, lewis@keywild.com
Target compiler: FreeBasic, http://www.freebasic.net/
Note: This is written to run as a console application under Windows.
      I suspect numerous changes would be needed for linux.
```



```

' /
' -----
# Include "dir.bi"
dim HomePath as String      '' users HOMEDRIVE and HOMEPATH
dim AppDataPath as String   '' users HOMEDRIVE and APPDATA
dim PrefFile as String      '' users's Arduino preferencesfile
dim SketchPath as String    '' users's Arduino Sketchbook
dim BuildPath as String     '' Arduino build directory
dim ElfFile as String       '' Elf file name (with path)
dim ProjName as String      '' Arduino project name
dim AvrObjDump as String    '' Avr-ObjDump.exe with full path
dim AvrReadElf as String    '' Avr-readelf.exe with full path
dim CmdFile as String       '' CMD file with full path
dim Cr as String            '' Carriage return/line feed (ASCII 13,10)
dim HelpStr as String       '' command line Help
dim DmpOpts as String       '' Options for Avr-ObjDump.exe
dim ElfOpts as String       '' Options for Avr-readelf.exe

Dim Buffer as String        '' working buffer for reading files
Dim TempStr as String      '' working string

Dim P as Integer           '' position from instr
' -----
Cr=chr(13) & chr(10)
' -----
HelpStr= "Arduino-elf-dump.exe is a small program written in FreeBasic to automate the" & Cr & _
        "process of creating dume files from the elf file produced by the Arduinio IDE." & Cr & _
        "The program attempts to write a CMD file to the project directory and execute" & Cr & _
        "it. The CMD file allows the Avr-ObjDump and/or Avr-readelf options to be" & Cr & _
        "edited as desired. The dump files are also placed in the project directory." & Cr & _
        Cr & _
        "This program requires that 'build.path' be specified in the user's Arduinio" & Cr & _
        "preferences file. Please set up a directory for builds and add it to your " & Cr & _
        "preferences file. Please also set preproc.save_build_files to true." & Cr & _
        Cr & _
        "The program searches the following directories for Avr-ObjDump & Avr-readelf:" & Cr & _
        "    C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\" & Cr & _
        "    C:\Program Files\Arduino\hardware\tools\avr\bin\" & Cr & _
        "    C:\Arduino\hardware\tools\avr\bin\" & Cr & _
        "    C:\bin\Arduino\hardware\tools\avr\bin\" & Cr & _
        Cr & _
        "The program searches the following directories for project directory:" & Cr & _
        "    Sketchbook (as defined in Arduino preferences file)" & Cr & _
        "    <users home path>\Documents\Arduino" & Cr & _
        "    <users home path>\My Documents\Arduino" & Cr & _
        "    <users home path>\Documents\Projects" & Cr & _
        "    <users home path>\My Documents\Projects" & Cr & _
        "If project directory can be found then the CMD file is written to the build" & Cr & _
        "is written to the build directory." & Cr & _
        Cr & _
        "There are no command line options." & Cr & _
        "This program is Public Domain open source." & Cr
' -----
DmpOpts= Cr & Cr & "exit" & Cr & Cr & _
        "avr-objdump" & Cr & _
        "Options are:" & Cr & _
        "    -a, --archive-headers    Display archive header information" & Cr & _
        "    -f, --file-headers          Display the contents of the overall file header" & Cr & _
        "    -p, --private-headers          Display object format specific file header contents" & Cr & _
        "    -h, --[section-]headers        Display the contents of the section headers" & Cr & _
        "    -x, --all-headers              Display the contents of all headers" & Cr & _
        "    -d, --disassemble              Display assembler contents of executable sections" & Cr & _
        "    -D, --disassemble-all          Display assembler contents of all sections" & Cr & _
        "    -S, --source                   Intermix source code with disassembly" & Cr & _
        "    -s, --full-contents             Display the full contents of all sections requested" & Cr & _
        "    -g, --debugging                Display debug information in object file" & Cr & _
        "    -e, --debugging-tags            Display debug information using ctags style" & Cr & _
        "    -G, --stabs                     Display (in raw form) any STABS info in the file" & Cr & _
        "    -W, --dwarf                     Display DWARF info in the file" & Cr & _
        "    -t, --syms                     Display the contents of the symbol table(s)" & Cr & _
        "    -T, --dynamic-syms              Display the contents of the dynamic symbol table" & Cr & _
        "    -r, --reloc                     Display the relocation entries in the file" & Cr & _
        "    -R, --dynamic-reloc             Display the dynamic relocation entries in the file" & Cr & Cr
' -----
ElfOpts= Cr & _
        "readelf" & Cr & _
        "Options are:" & Cr & _
        "    -a --all                      Equivalent to: -h -l -S -s -r -d -V -A -I" & Cr & _

```

```

" -h --file-header      Display the ELF file header" & Cr & _
" -l --program-headers  Display the program headers" & Cr & _
" -S --section-headers  Display the sections' header" & Cr & _
" -g --section-groups   Display the section groups" & Cr & _
" -t --section-details  Display the section details" & Cr & _
" -e --headers          Equivalent to: -h -l -S" & Cr & _
" -s --symbols          Display the symbol table" & Cr & _
" -n --notes            Display the core notes (if present)" & Cr & _
" -r --relocs           Display the relocations (if present)" & Cr & _
" -u --unwind           Display the unwind info (if present)" & Cr & _
" -d --dynamic          Display the dynamic section (if present)" & Cr & _
" -V --version-info     Display the version sections (if present)" & Cr & _
" -A --arch-specific    Display architecture specific information (if any)." & Cr & _
" -c --archive-index    Display the symbol/file index in an archive" & Cr & _
" -D --use-dynamic      Use the dynamic section info when displaying symbols" & Cr
' -----

' check for command line arguments
TempStr = Command (-1)
If TempStr <> "" then
    Print HelpStr
    End 0
End if

' Get users HOMEDRIVE and HOMEPATH from enviroment
HomePath = Environ("HomeDrive") & Environ("HomePath") & "\"
' Print HomePath

' Get users HOMEDRIVE and APPDATA from enviroment
AppDataPath = Environ("AppData") & "\"
' Print AppDataPath

' Find Arduino preferences file
' <AppData>\Arduino\preferences.txt
PrefFile=AppDataPath & "\\Arduino\\preferences.txt"
If Dir(PrefFile) = "" then
    Print "***ERROR** Can not find Ardunio preferences file:"
    Print PrefFile
    Sleep ' sleep waits for a keypress before continueing
    End -1
End If

'Open file and look for build.path & sketchbook.path
' build.path=<users path>\Documents\Arduino\Build
' sketchbook.path=<users path>\Documents\Arduino
' preproc.save_build_files=true
Open PrefFile for input as #1
If Err>0 Then
    Print "***Error** opening Ardunio preferences file:"
    Print PrefFile
    sleep
    End -1
End If

BuildPath=""
SketchPath=""
Do Until EOF(1)
    Line Input #1, buffer          '' read a line of text
    buffer=Lcase(Trim(buffer))     '' we should not need this but ...
    If InStr (buffer, "build.path")=1 then BuildPath=buffer
    If InStr (buffer, "sketchbook.path")=1 then SketchPath=buffer
Loop
Close #1

' for this to work we reuire that a build path be set in the Ardunio preferences file
If Len(BuildPath) < 20 then
    Print "***ERROR** This program requires that the 'build.path' is specified"
    Print "          in the user's Ardunio preferences file. Please set up a"
    Print "          directory for builds and add it to your preferences file."
    Print "  Example:"
    Print "          build.path=" & HomePath & "Documents\Arduino\Build"
    Print ""
    Print "          Please also set preproc.save_build_files to true."
    Print ""
    Print "Your Ardunio preferences file is:"
    Print PrefFile
    Sleep

```

```

    End -1
End If

' Now go find the the build
' First we need to strip off "build.path="
P=InStr (BuildPath, "=")
BuildPath = Trim(Mid (BuildPath, P+1)) & "\"
ElfFile = ""
ElfFile = Dir (BuildPath & "*.elf")
If ElfFile = "" then
    Print "***ERROR** Can not find '*.elf' file in build directory:"
    Print BuildPath
    Sleep
    End -1
End If

' Extract project name ... example: HelloWorld_001.cpp.elf
' this should be everything in front of .cpp.elf
P=InStr (ElfFile, ".cpp.elf")
ProjName=Trim(Left (ElfFile,P-1))
' Now see if we can find the project directory
if SketchPath<>"" then
    P=InStr (SketchPath, "=")
    ' SketchPath = Trim(Mid (SketchPath, P+1)) & "\sketches\" & ProjName
    TempStr = Trim(Mid (SketchPath, P+1)) & "\sketches\" & ProjName & "\"
end if
if dir(TempStr & ProjName & ".ino")="" then
    ' try an alternative
    TempStr = Trim(Mid (SketchPath, P+1)) & "\" & ProjName & "\"
end if
if dir(TempStr & ProjName & ".ino")="" then
    ' try an alternative
    TempStr = HomePath & "Documents\Projects\" & ProjName & "\"
end if
if dir(TempStr & ProjName & ".ino")="" then
    ' try an alternative
    TempStr = HomePath & "My Documents\Projects\" & ProjName & "\"
end if
if dir(TempStr & ProjName & ".ino")="" then
    ' try an alternative
    TempStr = HomePath & "Documents\Arduino\" & ProjName & "\"
end if
if dir(TempStr & ProjName & ".ino")="" then
    ' try an alternative
    TempStr = HomePath & "My Documents\Arduino\" & ProjName & "\"
end if

If dir(TempStr & ProjName & ".ino")="" then
    ' instead of an error we are going to dump the assembly output to
    ' the build directory
    SketchPath=BuildPath
else
    SketchPath=TempStr
end If

' now find the avr-objdump
' <ARDUINOPATH>/hardware/tools/avr/bin/avr-objdump
' C:\Program Files (x86)\Arduino\hardware\tools\avr\bin
AvrObjDump=""
TempStr = "C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avr-objdump.exe"
If Dir(TempStr)<>"" then AvrObjDump=TempStr
TempStr = "C:\Program Files\Arduino\hardware\tools\avr\bin\avr-objdump.exe"
If Dir(TempStr)<>"" then AvrObjDump=TempStr
TempStr = "C:\Arduino\hardware\tools\avr\bin\avr-objdump.exe"
If Dir(TempStr)<>"" then AvrObjDump=TempStr
TempStr = "C:\bin\Arduino\hardware\tools\avr\bin\avr-objdump.exe"
If Dir(TempStr)<>"" then AvrObjDump=TempStr
If AvrObjDump = "" then
    Print "***ERROR** Can not find 'avr-objdump.exe'."
    Print "Searched:"
    Print "    C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\"
    Print "    C:\Program Files\Arduino\hardware\tools\avr\bin\"
    Print "    C:\Arduino\hardware\tools\avr\bin\"
    Print "    C:\bin\Arduino\hardware\tools\avr\bin\"
    Sleep
    End -1
End If

```

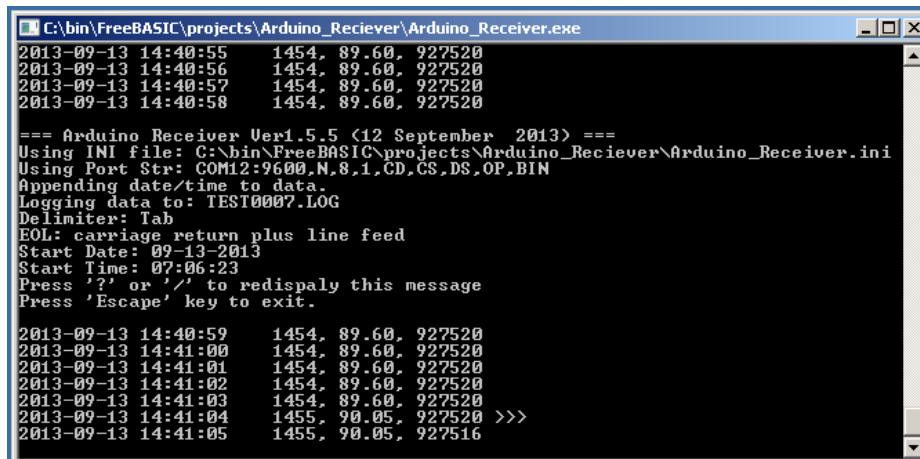
```

P=Instr(AvrObjDump,"avr-objdump.exe")
AvrReadElf=Left(AvrObjDump,P-1) & "avr-readelf.exe"

' Now build our command string ...
CmdFile = SketchPath & ProjName & "_dump.cmd"
open CmdFile for output as #1
TempStr = chr(34) & AvrObjDump & chr(34) & " -S "
TempStr = TempStr & chr(34) & BuildPath & ElfFile & chr(34)
TempStr = TempStr & " > " & chr(34) & SketchPath & ProjName & ".asm.txt" & chr(34)
Print #1, TempStr
TempStr = chr(34) & AvrReadElf & chr(34) & " -e "
TempStr = TempStr & chr(34) & BuildPath & ElfFile & chr(34)
TempStr = TempStr & " > " & chr(34) & SketchPath & ProjName & ".hrd.txt" & chr(34)
Print #1, TempStr
Print #1, DmpOpts
Print #1, ElfOpts
Close #1
Shell CmdFile
end 0

```

Appendix: Arduino Receiver



```
C:\bin\FreeBASIC\projects\Arduino_Receiver\Arduino_Receiver.exe
2013-09-13 14:40:55      1454. 89.60. 927520
2013-09-13 14:40:56      1454. 89.60. 927520
2013-09-13 14:40:57      1454. 89.60. 927520
2013-09-13 14:40:58      1454. 89.60. 927520

=== Arduino Receiver Ver1.5.5 <12 September 2013> ===
Using INI file: C:\bin\FreeBASIC\projects\Arduino_Receiver\Arduino_Receiver.ini
Using Port Str: COM12:9600,N,8,1,CD,CS,DS,OP,BIN
Appending date/time to data.
Logging data to: TEST0007.LOG
Delimiter: Tab
EOL: carriage return plus line feed
Start Date: 09-13-2013
Start Time: 07:06:23
Press '?' or '/' to redisplay this message
Press 'Escape' key to exit.

2013-09-13 14:40:59      1454. 89.60. 927520
2013-09-13 14:41:00      1454. 89.60. 927520
2013-09-13 14:41:01      1454. 89.60. 927520
2013-09-13 14:41:02      1454. 89.60. 927520
2013-09-13 14:41:03      1454. 89.60. 927520
2013-09-13 14:41:04      1455. 90.05. 927520 >>>
2013-09-13 14:41:05      1455. 90.05. 927516
```

Arduino Receiver is an enhanced version of the Serial Port Monitor program presented in the main text. The program was developed and tested under the Windows 7 operating system. It is also known to be fully functional on Windows XP. It was written in a manner such that it should also function under X86 Linux operating systems with appropriate COM port strings in the INI file but has not been tested in this environment. This program (along with the source code) is available for download from:

<http://www.keywild.com/arduino/index.htm> (current version)

A copy was posted in The Arduino “Other Software Development: Arduino Receiver, PC RS-232 data logger” Forum Thread at:

<http://forum.arduino.cc/index.php?topic=187396.0> (Version 1.5.5)

```
Program Name: Arduino_Receiver.exe
Program Vers: 1.5.5
Program Date: September 13, 2013
Author: Lewis Balentine, http://www.keywild.com
This program placed in the Public Domain by the author.
No warranties of any kind either expressed or implied.
Please read 'Arduino_Receiver.ini' & 'Arduino_Receiver_Notes.txt' for help.

arduino_reciever(1.5.5).zip
ZIP file contains EXE, INI, TXT and BAS(source) files.

Description:
This program monitors a serial port and displays the received ASCII lines in a console window.
All Parameters are controlled by a INI file of the same name but there is a command line option
to use an alternate INI file. Options in the INI file include:

...PortStr      = All parameters for COM port (Range=ANY: see notes for detailed options)
...SwitchHrs    = Sequential log file names rotated on the hour (range 0-24 hours, default 0)
...ApndTime     = Appends Date/time to front of each line of received data (default false)
...StdOut       = Suppress status messages. Error message are always output. (default false)
...FileStr      = If defined will send data to a log file as well as Standard Out (default False)
...Delimiter    = Specify delimiter to go between Date/Time and received data (default TAB)
...EOL          = Specify end of line character for output file (CR/LF, CR, or LF)
...ExitKey      = Define specific key for exit (default Escape Key)

PortStr is the ONLY required parameter line.
All INI parameter lines, port Options and error codes are fully Documented in included files.
```

If PortStr is the only INI Parameter line used then operation is exactly like the Serial Port Monitor program with the exception that Error Checking has been implemented for all COM port and File Input/Output operations. Sixteen (16) error codes/messages are defined to assist in trouble shooting. Exit codes are produced on every exit except abnormal termination (*i.e. program closed by Operation System*).

```
'      exit code 50:  normal exit, user pressed exit key
'      exit code 51:  normal exit, End Of Transmission received
'      exit code 52:  normal exit, user requested help
'      exit code 53:  ini file not found
'      exit code 54:  error opening ini file
'      exit code 55:  error reading ini file
'      exit code 56:  error closing ini file
'      exit code 57:  PortStr not found in ini file
'      exit code 58:  error opening COM port
'      exit code 59:  error opening output file
'      exit code 60:  error reading COM port
'      exit code 61:  error writing output file
'      exit code 62:  error closing COM port
'      exit code 63:  error closing output file
'      exit code 64:  error closing output file during file switch
'      exit code 65:  error opening output file during file switch
```

File flushing takes place on every log file write to guard against data-loss in the event of abnormal exit, power interruption or similar problems. Existing log files are NOT overwritten and may be opened by other applications during logging operations. All output goes to “standard out” which defaults to the console window display. The source code is HEAVILY commented and written to be used as a template for other applications. The source code is broken into three files:

Arduino_Receiver.Bas

Main source code

Arduino_Receiver_Globals.Bas

Global Variables (*shared among all modules/functions*)

Arduino_Receiver_Functions.Bas

Functions used by Main Program

The main program code is 20 lines long:

```
' ----- libraries -----
#include Once "string.bi"          ' needed for format function
#include once "crt.bi"             ' for file flush (see notes)
#include once "file.bi"            ' for file flush (see notes)#include once '
include global varriables
#include once "Arduinio_Receiver_Globals.Bas" ' Global Variables
#include once "Arduino_Receiver_Functions.bas" ' function defined for this prog
'----- main program code -----
VoidByte = InititalizeGlobals()
' find, read and parse INI file
If ReadIniFile()<>0 then End(ExitCode)
' evaluate global variables read from INI
If EvalGlobals()<>0 then End(ExitCode)
' opens Com Port and optional Log File
If OpenCommunications()<>0 then End(ExitCode)
' process serial data loop
VoidByte = Communications()
' close com port and optional log file
End (CloseCommunications())
'----- End Of File-----
```

The Functions module is somewhat longer: approximately 650 lines. There are probably more comments than actual source code. Functions included are:

```
GetIniFileName()      ' searches for INI file among several options provided
CleanIniStr()          ' used to strip comments and stray characters
ReadIniFile()          ' used to read and parse INI file
BuildFileStr()         ' used for log file
BuildStatusStr()       ' used to display start time, log file, INI options
EvalGlobals()          ' used to initialize operations, validate INI parameters, etc
OpenCommunications()   ' used to open com port and optional log file
CloseCommunications()  ' used to close com port and optional log file
CheckTime()            ' used to switch optional log file
ProcessData()          ' used to write serial data to display and optional log file
Communications()        ' main program loop used to read serial port and keyboard
```

The function "ReadIniFile()" uses a case structure so that adding additional options to the INI file is simple. The function "Communications()" includes provisions to allow commands to be sent to Arduino device (*or other serial device for that matter*).

A provision has been included in "GetIniFileName()" to test the command line for "user help request". This request may be any of several help request conventions (*i.e. HELP /? -? --? /h -h --h*). As the program is written this request is answer with the program name, version and suggestion to read the INI file. It could be easily expanded (*two alternatives are launching a PDF reader or Web Browser with a specific reference*).

Of course there is always the option to use it just as it is.

Appendix: Thermometer.exe

All code that was specifically added for the Thermometer application is shown in **BOLD** characters.

Main Program Code

Code File for Thermometer.exe = Arduino_Thermometer.bas

This is the file that **must** be active when you select Compile in the FBIDE editor.

```
' ---- libraries -----
#include Once "string.bi"          ' needed for format function
#include once "crt.bi"             ' for file flush (see notes)
#include once "file.bi"           ' for file flush (see notes)
' include global varriables
#include once "Arduino_Thermometer_Globals.Bas" ' Global Variables
#include once "Arduino_Thermometer_Functions.Bas" ' function defined for this prog
'----- main program code -----
VoidByte = InitalizeGlobals()
' find, read and parse INI file
If ReadIniFile()<>0 then End(ExitCode)
' evaluate global variables read from INI
If EvalGlobals()<>0 then End(ExitCode)
' opens Com Port and optional Log File
If OpenCommunications()<>0 then End(ExitCode)
' process serial data loop
VoidByte = Communications()
' close com port and optional log file
End (CloseCommunications())
'----- End Of File-----
```

Global Variables

Code File for Thermometer.exe = Arduino_Thermometer_Globals.Bas

```
' Global Variables for the Arduino Reciever Program
' ---- libraries -----
#include Once "string.bi"          ' needed for format function
#include once "crt.bi"             ' for file flush (see notes)
#include once "file.bi"           ' for file flush (see notes)
'-----define False-----
#ifndef FALSE
#define FALSE 0
#endif
#ifndef False
#define False 0
#endif
#ifndef false
#define false 0
#endif
'-----define True-----
#ifndef TRUE
#define TRUE -1
#endif
#ifndef true
#define true -1
#endif
#ifndef True
#define True -1
#endif
'-----

Dim SHARED VoidByte as Byte = 0          ' BECASUE FREEBASIC DOES NOT SUPPORT VOID
'-----
```



```

' revision October 2013 -- moved from checktime function in order to eliminate
' first pass requirement
Dim SHARED PrevHour as Byte          ' holds last hour counted
Dim SHARED NowHour as Byte           ' holds last current hour
Dim SHARED HourCount as Byte = 0     ' holds hour count
'-----
Dim SHARED ProgramName as String     ' used to display program Name ...
Dim SHARED ProgramVer as String      ' used to display program Version ...
Dim SHARED IdString as String        ' imbed program name, version, date
Dim SHARED Status as String          ' used to display program status ...
Dim SHARED IniName as String         ' name of ini file
Dim SHARED PortStr as String         ' hold parameters for opening com port
Dim SHARED AddDateTime as Byte       ' if not zero then append time to string
Dim SHARED FileStr as String         ' optional file name for output
Dim SHARED SendToFile as Byte        ' if not zero then append to file
Dim SHARED SwitchHrs as Byte         ' used to increment file name
Dim SHARED Delimiter as String       ' string used between date/time and data
Dim SHARED EOL as String             ' Carriage Return / Line Feed
Dim SHARED StdOutFlg as Byte         ' Flag to send to Standard out only
Dim SHARED LogFileHandle as FILE ptr ' used to flush file buffer to disk
Dim SHARED ExitKey as Byte           ' used to define specific key for exit
Dim SHARED TStart as String          ' used to calculate run time
Dim SHARED DStart as String          ' hold start day, used to calculate run time
Dim SHARED ExitCode as Integer       ' used to pass exit code
'-----
' variables added for Thermometer application
Dim Shared DebugMode as Byte         ' False=0, True<>0;
Dim Shared CelsiusMode as Byte       ' False=0, True<>0;
Dim Shared FahrenheitMode as Byte    ' False=0, True<>0;
Dim Shared AvrMode as Byte           ' False=0, True<>0;
Dim Shared EEMode as Byte            ' False=0, True<>0;
Dim Shared RawMode as Byte           ' False=0, True<>0;
Dim Shared RoundMode as Byte         ' False=0, True<>0;
Dim Shared UserStr1 as String         ' user defined strin in INI file
Dim Shared UserStr2 as String         ' user defined strin in INI file
Dim Shared UserStr3 as String         ' user defined strin in INI file
Dim Shared UserStr4 as String         ' user defined strin in INI file
Dim Shared UserStr5 as String         ' user defined strin in INI file
Dim Shared OffsetVal as Single        ' used to store current degree offset
'-----
' Initialize all global variables
Function InitializeGlobals() as byte
    EOL = Chr(13) & Chr(10)          ' This is the default EOL until INI is read
    ProgramName = "Arduino_Thermometer.exe"
    ProgramVer = "1.0.5 (8 October 2013)"
    ' ----- IdString is defined Globally to imbed within Object code -----
    IdString = "Program Name: " & ProgramName & EOL & _
        "Program Vers: " & ProgramVer & EOL & _
        "Author: Lewis Balentine, http://www.keywild.com" & EOL & _
        "This program placed in the Public Domain by the author." & EOL & _
        "No warranties of any kind either expressed or implied." & EOL & _
        "Please read 'Arduino_Thermometer.ini' for help." & EOL
    PortStr = ""
    FileStr = ""
    ExitCode = 0
    SendToFile = 0
    AddDateTime = 0
    StdOutFlg = 0
    SwitchHrs = 0
    ExitKey = 27
    Delimiter = Chr(9)
    DStart = Date                    ' get the start date
    TStart = Time                    ' get the start time (24 hour format)
'-----

```

```

' revision October 2013 -- initialize checktime parameters
NowHour=val(Left(time,2))
PrevHour=NowHour
HourCount=0

' -----
' These are the default modes
DebugMode=False
CelsiusMode=True
FahrenheitMode=True
AvrMode=False
EEMode=False
RawMode=True
RoundMode=True
UserStr1 ="Z1"
UserStr2 ="Z2"
UserStr3 ="Z0"
UserStr4 =""
UserStr5 =""
OffsetVal=0
Return 0
End Function
'---End of File -----

```

Thermometer Functions

Code File for Thermometer.exe = Arduinio_Thermometer_Functions.bas

```

' Functions defined for the program Arduino Reciever
' Functions defined before USE do not require declarations
' Place this module at the TOP of the main source module
'
' ---- libraries -----
#include Once "string.bi"          ' needed for format function
#include once "crt.bi"             ' for file flush (see notes)
#include once "file.bi"            ' for file flush (see notes)

' include global varriables
#include once "Arduinio_Thermometer_Globals.Bas"    ' Global Variables
'
'=====
Function GetIniFileName (HelpStr as String, Default as String) as Byte
' This function is used to get the ini file name as well as
' check for user requested help.
' If help request is found this function terminates the program
' If INI file is not found this function terminates the program
' Input:
'   HelpStr = string to be printed if help is requested on command line
' OutPut:
'   Returns INI file name. Aborts if file not found.
' -----
Dim Buffer As String              ' used to read command line arguments
Dim P as Byte                    ' working numeric variable
Dim TryString as String          ' Various INI names tried
'-----
TryString="These filneames were tried:" & EOL
' check for command line help request
Buffer=Trim(Ucase(Command(-1))) ' returns ENTIRE command line
If Buffer<>"" then
' there are number of possible conventions
If (Buffer = "?") or _
   (Buffer = "HELP") or _
   (Instr(Buffer, "?")>0) or _
   (Instr(Buffer, "/H")>0) or _

```

```

(Instr(Buffer, "-?")>0) or _
(Instr(Buffer, "-H")>0) or _
(Instr(Buffer, "--?")>0) or _
(Instr(Buffer, "--H")>0) or _
(Instr(Buffer, "/HELP")>0) then
Print HelpStr
Print "Normal exit, ExitCode: 52"
ExitCode=52
Return(ExitCode)
End If
End If
' Check for command line specifying INI file -----
Buffer=Trim(Command(1)) ' returns first parameter
If Buffer <> "" then
' we must have a alternate ini file name, but is it valid ?
TryString= TryString & Buffer & EOL
If DIR(Buffer)<>"" then
IniName = Buffer
return 0
End If
End If
' Check for INI file matching EXE name -----
' get the actual command string and set the default ini filename to match
' This allows the executable to be renamed with not change to source
Buffer=Trim(Command(0)) ' includes path if used
if lcase(right(Buffer,4))=".exe" then
P=Len(Buffer)-3
Buffer=Left(Buffer,P) & ".ini"
Else
Buffer=Buffer & ".ini"
End If
' do we have a valid file name
TryString= TryString & Buffer & EOL
If DIR(Buffer)<>"" then
IniName = Buffer
return 0
End If
' Check for default INI file name -----
TryString= TryString & Default & EOL
If DIR(Default)<>"" then
IniName = Default
Return 0
End If
' If we fall through to this point then report error and abort
Print HelpStr
Print "*** FATAL ERROR ***: No INI file found."
Print TryString
Print "****ERROR*** exit, ExitCode: 53"
ExitCode=62
Return(ExitCode)
End Function
'
'=====
Function CleanIniStr (Buffer as String, _
ByRef KeyStr as String, _
ByRef ValStr as String ) As Byte
' This function is used to strip comments, non-printing characters and
' quotes from a INI file String.
' Input:
' Buffer = String to be cleaned
' KeyStr = Holds Parmeter name string on return
' ValStr = Holds Parmeter value string on return
' OutPut:
' Returns 0 if Buffer is NOT a valid parameter line
' Returns -1 if Buffer is a valid parameter line

```

```

'-----
Dim P as Byte = 0          ' used as index into buffer
Dim Work as String = ""    ' working storage
Dim C as Byte              ' holds value for single character
'-----

' remove any spaces
Buffer=Trim(Buffer)
' strip off comments
P=Instr(Buffer, ";")
If P=1 then Return 0
If P>1 then Buffer=Trim(Left(Buffer, P-1))
' check for empty string
If Buffer="" then Return 0
' clean up strange characters ...
' mostly to eliminate any tab characters or quotes
For P = 1 to Len(Buffer)
    C=Asc(Mid(Buffer,P,1))
    ' only accept characters that match our criteria
    ' chr(34) = double quote, chr(39) = single quote
    If (C>31) and (C<127) and (C<>34) and (C<>39) then
        Work=Work & Chr(C)
    End If
Next P
' again check for spaces after possible tabs/quotes removed
Work=Trim(Work)
If Work="" then Return 0
P=Instr(Work, "=")
If (P=0) or (P=1) or (P=len(work)) then Return 0
KeyStr=Ucase(Trim(Left(Buffer, P-1)))
ValStr=Trim(Mid(Buffer, P+1))
If (KeyStr="") or (ValStr="") then Return 0
Return -1
End Function
'
'=====
Function ReadIniFile () As Byte
' This function is used to read the ini file. Should be first function used.
' Uses previous defined function: "GetIniFileName"
' Uses previous defined function: "CleanIniStr"
' Input:
'   none
' OutPut:
'   Returns ExitCode on Error, Else 0
'-----

Dim ErrCode as Integer    ' used to hold Err code
Dim P as Byte             ' used as index into string
Dim Buffer as String       ' used to hold input line
Dim KeyStr as String       ' used in parsing ini file
Dim ValStr as String       ' used in parsing ini file
Dim EnvStr as String       ' used to expand enviroment string
'-----

If GetIniFileName (IdString, "Arduino_Receiver.ini")<>0 then
    Return ExitCode
End If
open IniName for input as #1
ErrCode=Err               ' reading error code destroys it
If ErrCode<>0 then
    Print "**** FATAL ERROR ****: opening: " & IniName
    Print "Error code returned was: " & ErrCode
    Print "****ERROR*** exit, ExitCode: 54"
    ExitCode=54
    Return ExitCode
End If
'-----
' read each line of the ini file checking for parameters

```

```

While not (eof(1))
  Line Input #1, Buffer
  ErrCode=Err          ' reading error code destroys it
  If ErrCode<>0 then
    Print "*** FATAL ERROR ***" reading: " & IniName
    Print "Error code returned was: " & ErrCode
    Print "***ERROR*** exit, ExitCode: 55"
    ExitCode=55
    Return ExitCode
  End If
  ' CleanIniStr return 0 for non-parameter line
  If CleanIniStr(Buffer, KeyStr, ValStr)<>0 then
    ' print KeyStr, ValStr: sleep ' for debugging
    Select Case KeyStr
      Case "PORTSTR"
        PortStr=ValStr
      Case "APNDTIME"
        ValStr=Ucase(ValStr)
        if (ValStr="TRUE") or (ValStr="YES") then AddDateTime=1
      Case "STDOUT"
        ValStr=Ucase(ValStr)
        if (ValStr="TRUE") or (ValStr="YES") then StdOutFlg=1
      Case "FILESTR"
        FileStr=ValStr
        SendToFile=1
        ' expand enviromental variable
        ' I am a bit perplexed by the offsets used
        ' but they seem to work (trial & error until success)
        If Left(FileStr,1) = "%" then
          'Print FileStr          'debugging
          P=Instr(2,Buffer, "%")
          EnvStr=Mid(Left(FileStr,P-2),2)
          'Print EnvStr          'debugging
          EnvStr=Environ(EnvStr)
          'Print EnvStr          'debugging
          FileStr=EnvStr & Mid(FileStr, P)
        End If
      Case "SWITCHHRS"
        ' print KeyStr, ValStr: sleep ' for debugging
        SwitchHrs=Abs(Int(Val(ValStr)))
        If SwitchHrs>24 then SwitchHrs=24
        ' print SwitchHrs, ValStr: sleep ' for debugging
      Case "DELIMITER"
        if ValStr="TAB" then: Delimiter=chr(09)
        elseif ValStr="COMMA" then Delimiter=", "
        elseif ValStr="COLLON" then Delimiter=": "
        elseif Abs(Val(ValStr)) >0 then _
          Delimiter=Space(Int(Val(ValStr)))
        elseif Left(ValStr,1)="0" then Delimiter=""
        elseif Lcase(ValStr)="zero" then Delimiter=""
        else DELIMITER=chr(9)
        end if
      Case "EOL"
        ValStr=Left(ValStr,1)
        if ValStr="0" then: EOL=chr(13) & chr(10)
        elseif ValStr="1" then EOL=chr(10)
        elseif ValStr="2" then EOL=chr(13)
        else EOL=chr(13) & chr(10)
        end if
      Case "EXITKEY"
        If Lcase(ValStr)="escape" then
          ExitKey=27
        ElseIf Lcase(ValStr)="ctlx" then
          ExitKey=24
        Else

```

```

ExitKey=Abs(Int(Val(ValStr)))
If ExitKey<>0 then
    If ExitKey>126 then ExitKey=27
    If ExitKey<32 then ExitKey=27
End If
end if
'-----
' add additional case statements for new options in INI file
' Example:
'Case "BULLWINKLE"
'    BULLWINKLE code goes here
'Case "ROCKYRACoon"
'    ROCKYRACoon code goes here
'case keyword MUST be UPPERCASE (in INI file it can be upper or lower)
'for reliability include spaces in keyword
'-----

Case "USERSTR1"
    If Lcase(ValStr)<>"" then UserStr1=Trim(ValStr)
Case "USERSTR2"
    If Lcase(ValStr)<>"" then UserStr2=Trim(ValStr)
Case "USERSTR3"
    If Lcase(ValStr)<>"" then UserStr3=Trim(ValStr)
Case "USERSTR4"
    If Lcase(ValStr)<>"" then UserStr4=Trim(ValStr)
Case "USERSTR5"
    If Lcase(ValStr)<>"" then UserStr5=Trim(ValStr)
Case Else
    'ignore it
End Select ' case KeyStr
End If ' CleanIniStr
Wend 'not (eof(1))
Close #1
ErrCode=Err ' reading error code destroys it
If ErrCode<>0 then
    Print "**** FATAL ERROR **** closing: " & IniName
    Print "Error code returned was: " & ErrCode
    Print "****ERROR*** exit, ExitCode: 56"
    ExitCode=56
    Return ExitCode
End If
Return ExitCode
End Function
'
'=====
Function BuildFileStr () as String
' This function is used build sequential file names when SwitchHrs>0
' Input:
'    none
' OutPut:
'    Returns valid FileStr UNLESS SendToFile=0
'-----
Static BaseFileStr0 as String ' holds FileStr as defined in INI file
Static BaseFileStr1 as String ' holds first part of FileStr
Static BaseFileStr2 as String ' holds last part of FileStr
Static SwitchHrsNdx as Integer=-1 ' holds current iteration of filename
'-----
Dim P as Integer ' index into string
Dim Exist as String="xxx" ' used to check for existence of file
'-----
' by definition if SendToFile=0 then FileStr=""
If SendToFile=0 then Return ""
' If SwitchHrs=0 then default FileStr is to be used
If SwitchHrs=0 then Return FileStr
' This is only executed on the first call
If SwitchHrsNdx = -1 then

```

```

' This is the first time into the function
P=Instr(FileStr, "0000")
If P=0 then
    ' user has failed to specify appropriate file string
    ' disable SwitchHrs and return
    SwitchHrs=0
    Return FileStr
Else
    BaseFileStr0=FileStr          ' save it just in case
    BaseFileStr1=Left(FileStr,P-1)
    BaseFileStr2=Mid(FileStr,P)
    ' regardless of what was in the INI file
    ' we are only going to use four zeros
    While Left(BaseFileStr2,1)="0"
        BaseFileStr2=Mid(BaseFileStr2,2)
    Wend
End If
End If
' This is executed on the every call
While Exist <> ""
    SwitchHrsNdx=SwitchHrsNdx +1
    FileStr=BaseFileStr1 & _
        Right("0000" & Trim(Str(SwitchHrsNdx)),4) & _
        BaseFileStr2
    Exist=Dir(FileStr)
Wend
Return (FileStr)
End Function
'
'=====
Function BuildStatusStr () as Byte
    ' This function is used to build the status line message that is displayed
    ' when the user presses "?" or "/".
    ' Input:
    '   none
    ' OutPut:
    '   Returns StdOutFlg
    '-----

    Status = EOL & _
        "=== Arduino Receiver Ver" & ProgramVer & " ===" & EOL & _
        "Using INI file: " & EOL & IniName & EOL & _
        "Using Port Str: " & PortStr & EOL
    If AddDateTime<>0 then Status = Status & "Appending date/time to data." & EOL
    If FileStr<>"" then
        Status = Status & "Logging data to: " & FileStr & EOL
        ' display EOL and Delimiter are a bit more complicated
        ' Fortunately there are a limited numebr of possibilities.
        If Delimiter=", " then: Status = Status & "Delimiter: Comma" & EOL
        ElseIf Delimiter=":" then Status = Status & "Delimiter: Collon" & EOL
        ElseIf Delimiter=chr(9) then Status = Status & "Delimiter: Tab" & EOL
        ElseIf Delimiter="" then Status = Status & "Delimiter: none" & EOL
        Else Status = Status & "Delimiter: " & chr(34) & Delimiter & Chr(34) & EOL
        End if
        If EOL=chr(13) then: Status = Status & "EOL: carriage return" & EOL
        ElseIf EOL=chr(10) then Status = Status & "EOL: line feed" & EOL
        Else Status = Status & "EOL: carriage return plus line feed" & EOL
        End if
    End If
    Status = Status & "Start: " & Dstart & " " & Tstart & EOL
    Status = Status & "Press '?' or '/' to redispaly this message" & EOL
    If ExitKey = 24 then
        Status = Status & "Press 'Ctrl' and 'X' to exit." & EOL
    ElseIf ExitKey = 27 then
        Status = Status & "Press 'Escape' key to exit." & EOL
    End If
End Function

```

```

Else
    Status = Status & "Press '" & Chr(ExitKey) & "' key to exit." & EOL
End If
'-----
' Add Application help here
Status = Status & EOL
Status = Status & " Keys '1' to '0' set report Times" & EOL
Status = Status & " Key 'A' toggles AVR mode" & EOL
Status = Status & " Key 'B' Restore from Backup" & EOL
Status = Status & " Key 'C' toggles Celsius mode" & EOL
Status = Status & " Key 'D' toggles Debug mode" & EOL
Status = Status & " Key 'E' toggles EEPROM mode" & EOL
Status = Status & " Key 'F' toggles Fahrenheit mode" & EOL
Status = Status & " Key 'L' lists AVR commands" & EOL
Status = Status & " Key 'M' turns on Minimal mode (Fahrenheit only)" & EOL
Status = Status & " Key 'Q' prints AVR storage" & EOL
Status = Status & " Key 'R' toggles Rounding mode" & EOL
Status = Status & " Key 'S' prints AVR Status" & EOL
Status = Status & " Key 'V' toggles Raw Reading mode" & EOL
Status = Status & " Key '>' increase Degree Offset by 0.25 Fahrenheit" & EOL
Status = Status & " Key '<' decrease Degree Offset by 0.25 Fahrenheit" & EOL
if (UserStr1<>"" ) then Status = Status & " Key 'U1' INI defined string: " & _
    chr(34) & UserStr1 & chr(34) & EOL
if (UserStr2<>"" ) then Status = Status & " Key 'U2' INI defined string: " & _
    chr(34) & UserStr2 & chr(34) & EOL
if (UserStr3<>"" ) then Status = Status & " Key 'U3' INI defined string: " & _
    chr(34) & UserStr3 & chr(34) & EOL
if (UserStr4<>"" ) then Status = Status & " Key 'U4' INI defined string: " & _
    chr(34) & UserStr4 & chr(34) & EOL
if (UserStr5<>"" ) then Status = Status & " Key 'U5' INI defined string: " & _
    chr(34) & UserStr5 & chr(34) & EOL

Return StdOutFlg
End Function
'
'=====
Function EvalGlobals () as Byte
' This function is used to read the ini file. Should be first function used.
' Uses previous defined function: "BuildFileStr"
' Uses previous defined function: "BuildStatusStr"
' Input:
'     none
' Output:
'     Returns ExitCode on failure, Else 0
'-----
' Port String is required
If PortStr="" then
    Print "*** FATAL ERROR ***: PortStr not found in " & IniName
    Print "***ERROR*** exit, ExitCode: 57"
    ExitCode=57
    Return ExitCode
End If
' If SwitchHrs<>0 then we need to build our FileStr
If SwitchHrs<> 0 then FileStr=BuildFileStr
If BuildStatusStr()<> 0 then print Status
Return ExitCode
End Function
'
'=====
Function OpenCommunications() as Byte
' This function is used to open the Serial port and the optional log file
' Input:
'     none
' Output:

```



```

' Returns ExitCode
'-----
Dim ErrCode as Integer          ' used to hold Err code
'-----
' One alternative is to use this function to open a Database
' Data could then be sent to the Database in the ProcessData function
' Add code to CloseCommunications to close DataBase.
'-----

Open Com(PortStr) AS #2
ErrCode=Err                    ' reading error code destroys it
If ErrCode<>0 then
    Print "***Fatal Error*** opening com port using string: "
    Print PortStr
    Print "Error code was: " & ErrCode
    Print "***ERROR*** exit, ExitCode: 58"
    ExitCode=58
End If
'-----

If SendToFile<>0 then
    Open FileStr for Append as #3
    ErrCode=Err
    If ErrCode<>0 then
        Print "***Error*** opening file: " & FileStr
        Print "Error reported was: " & ErrCode
        Close #2
        print "***ERROR*** exit, ExitCode: 59"
        ExitCode=59
    End If
End If
Return ExitCode
End Function
'
'=====
Function CloseCommunications() as Byte
' This function is used to close the Serial port and the optional log file
' Input:
'   none
' Output:
'   always Returns 0
'-----

Dim ErrCode as Integer          ' used to hold Err code
Dim SaveCode as Integer         ' used to hold Err code
'-----

SaveCode = ExitCode             ' save the reason we are exiting
Close #2
ErrCode=Err
If ErrCode<>0 then
    Print "***Error*** closing file COM port"
    Print "Error reported was: " & ErrCode
    ExitCode=62
End If
'-----

If SendToFile<>0 then
    ' if open failed then we never got here ...
    ' log the reason the program is exiting
    Print #3, "Exit Code: " & SaveCode & " at " & Time & " on " & Date
    Close #3
    ErrCode=Err
    If ErrCode<>0 then
        Print "***Error*** closing file: " & FileStr
        Print "Error reported was: " & ErrCode
        ExitCode=63
    End If
End If
'-----

```

```

Select Case SaveCode
Case 50
    print "Normal exit, User pressed exit key"
Case 51
    print "Normal exit, EOT recieved in data stream"
Case Else
    print "***ERROR*** exit, ExitCode: " & SaveCode
End Select
Return ExitCode
End Function
'
'=====
Function CheckTime() as Byte
' This function is used tally hours for sequential files filanems.
' If the total hours is greater than ot equal to SwitchHrs then
' the current log file is close and a new one is opened.
' Uses previous defined function: "BuildFileStr"
' Uses previous defined function: "BuildStatusStr"
' Input:
'     none
' OutPut:
'     returns ExitCode on Failure, else 0
'-----
' revision October 2013 --
' NowHour, PrevHour, HourCount made into Global Variables
' Eliminated first pass code -- Variables initialized Globally
' Faster exit when hours have not changed
' Fixed updating PrevHour bug
'-----
Dim ErrCode as Integer      ' used to hold Err code
Dim SaveCode as Integer     ' used to hold Err code
Dim DateStr as String       ' used to reformat date/time into useable format
Dim K as Byte = 0           ' error trigger
'-----
' always check for unintended calls ....
If SwitchHrs=0 then return 0
' check hours
NowHour=val(Left(time,2))
if NowHour=PrevHour then return 0
' ExitCode=0 ' keep current exit code unless ther is an error

' if we get this far then we need to increment our parameters
PrevHour=NowHour
HourCount=HourCount+1
' now check for the file switch
If HourCount> SwitchHrs then
    ' time to change our socks ...
    ' print "changeing socks ..." : sleep
    Close #3
    ErrCode=Err
    If ErrCode<>0 then
        Print "***Error*** closing file during file switch: " & FileStr
        Print "Error reported was: " & ErrCode
        ExitCode = 64
        Return 64
    End If
    Open BuildFileStr() for Append as #3
    ErrCode=Err
    If ErrCode<>0 then
        Print "***Error*** opening file during file switch: " & FileStr
        Print "Error reported was: " & ErrCode
        print "***ERROR*** ExitCode: 65"
        ExitCode = 65
        Return 65
    End If
End If

```

```

        VoidByte=BuildStatusStr()
        HourCount=0
    End If
    Return ExitCode
End Function
'
'=====
Function ProcessData(ByRef Buffer as String ) as Byte
' This function is used to process a single line of filtered ASCII data
' that has been collected from the serial port.
' All data writes take place in this function.
' Input:
'   none
' OutPut:
'   Returns ExitCode=60 on error, else 0
'-----
' revision October 2013 --
' moved checktime function call to this function
'-----
Dim ErrCode as Integer      ' used to hold Err code
Dim SaveCode as Integer    ' used to hold Err code
Dim DateStr1 as String     ' used to reformat date/time into useable format
Dim DateStr2 as String     ' used to reformat date/time into useable format
Dim Counter as Integer     ' used to trigger CheckTime
Dim K as Byte = 0          ' error trigger
'-----
' added for Thermometer Application
Dim C as String            ' used for first character of string
Dim P as Byte = 0
'-----
' Place any addition processing code in this function.
' Keep it SHORT. Reading the COM is 'paused' for this function.
'----- added for thermometer application -----
' check to see if this is a non-report line
' we do not want to add date and time to non-report lines
' secondly we want to capture the degree offset if we can
C=Left(Buffer,1)
if C=";" then
    if Instr(Buffer,"Offset:")>0 then
        P=Instr(Buffer, chr(9))
        OffsetVal=Val(trim(Mid(Buffer,P+1)))
    end if
end if

'----- display write -----
' next line modified for Thermometer application
' If (AddDateTime) then
If ((AddDateTime) and (C <>"")) then
    DateStr1=mid(Date,7,4) & "-"
    DateStr1=DateStr1 & Left(Date,2) & "-"
    DateStr1=DateStr1 & Mid(Date,4,2)
    DateStr2=DateStr1 & Delimiter & Time
    DateStr1=DateStr1 & " " & Time
    If StdOutFlg=0 then
        Print DateStr1 & " " & buffer
    Else
        Print DateStr2 & Delimiter & buffer
    End If
Else
    Print buffer
End If

'----- file write -----
If SendToFile<>0 then

```

```

' revision October 2013 --
' The only time that we actually need to check the time
' is when we are writing to a file
CheckTime
' setup for file flush
LogFileHandle=cast(FILE Ptr,Fileattr(3,fbFileAttrHandle))
' next line modified for Thermometer application
' If (AddDateTime) then
If ((AddDateTime) and (C <>"")) then
    Print #3,DateStr2 & Delimiter & buffer & EOL;
    ErrCode=Err
    fflush(LogFileHandle)      ' no error to check for
Else
    Print #3, buffer & EOL;
    ErrCode=Err
    fflush(LogFileHandle)      ' no error to check for
End If
If ErrCode<>0 then
    Print "***Error*** writting file: " & FileStr
    Print "Error reported was: " & ErrCode
    ExitCode = 61
    Return 61
End If
End If ' SendToFile<>0
'----- clear input buffer -----
Buffer=""
Return ExitCode
End Function
'
'=====
Function Communications() as Byte
' This function is the main process loop for collecting data from both the
' serial port and the keyboard. All data reads take place in this function.
' The Serial data is filtered for ASCII characters only. When an EOL
' is recieved then the data string is sent to the ProcessData function.
' Com port & optional log file must be opened prior to calling this function.
' Uses previous defined function: "ProcessData"
' Uses previous defined function: "CheckTime"
' Input:
'   none
' OutPut:
'   Returns ExitCode
'-----
' revision October 2013 --
' moved checktime function call from this function to ProcessData function
'-----
Dim ErrCode as Integer      ' used to hold Err code
Dim C As Byte = 0           ' this is our incoming byte of data
Dim K as Integer =0         ' used to read keyboard & trigger exit
Dim Buffer As String = ""    ' this is our buffer to collect the bytes
' Dim Counter as Integer    ' used to trigger CheckTime
'-----
While InKey<> ""             'empty the keyboard buffer ... just in case
Wend

' loop untill there is an exit trigger
While ExitCode=0
' The first line checks to see if there is anything waiting in the COM
' buffer. Without it one is subject to reading a bunch of garbage.
If EOF(2) then
' Call Sleep with 25ms or less to release time-slice when waiting
' for user input or looping inside a thread.This will prevent the
' program from unnecessarily hogging the CPU.
Sleep 25
Else

```

```

' get a single byte from the serial port
Get #2,0,C,1
ErrCode=Err
If ErrCode<>0 then
    Print "***Fatal Error*** reading com port"
    Print PortStr
    Print "Error code was: " & ErrCode
    ExitCode=60
End If
' ----- filter recieved data -----
' characters below ASCII 32 are 'non-printing characters
' characters above ASCII 126 are not defined (by ASCII)
' append any printable character to the string
' include an exception for horizontal tab characters
If ((C > 31) and (C < 127)) or (C = 9) Then Buffer = Buffer + Chr(C)
' Linux/Unix terminate strings with a line feed (ASCII 10)
' MACs terminate lines with a carriage return (ASCII 13)
' Microsoft and Arduino use carriage return/linefeed (ASCII 13,10)
' End of Transmission is ASCII 04
' If we get any of the above then print the string
' but only if we have something to process.
If ((C=13) Or (C=10) or (C=04)) And (Len(Buffer) > 0) Then
    ExitCode = ProcessData(buffer)
End If ' ((C=13) Or (C=10) ...
' check for end of tranmission code in data stream
If C=04 then
    ExitCode=51
End If
End If ' Not(EOF(2))
'
' revision October 2013 --
'
' if we are writing to sequential file names then we need to
' check if a log file needs to be changed
If SwitchHrs>0 then
    Counter=Counter+1
    If Counter > 2048 then ' arbitrary number, change as needed
        ExitCode=CheckTime
        Counter=0
    End If
End If
'
' if we do not already have an exit flag then read the key board
If ExitCode=0 then
    K=ASC(InKey)
    Select Case K
    Case 63,47 ' ASCII 63 ="?" question mark
        Print Status ' ASCII 47 ="/" question mark
    Case ExitKey
        ExitCode=50 ' set exit code, normal exit
    ' thermometer application key inserted below-----
    Case 49 ' numeric key "1"
        Print #2, "T1" ' set timing to 1 minute
        Print "Report Time set to 1 minute"
    Case 50 ' numeric key "2"
        Print #2, "T2" ' set timing to 2 minutes
        Print "Report Time set to 2 minutes"
    Case 51 ' numeric key "3"
        Print #2, "T3" ' set timing to 3 minutes
        Print "Report Time set to 3 minutes"
    Case 52 ' numeric key "4"
        Print #2, "T4" ' set timing to 4 minutes
        Print "Report Time set to 4 minutes"
    Case 53 ' numeric key "5"
        Print #2, "T5" ' set timing to 5 minutes
        Print "Report Time set to 5 minutes"
    Case 54 ' numeric key "6"
        Print #2, "T6" ' set timing to 10 minutes

```

```

Print "Report Time set to 10 minutes"
Case 55 ' numeric key "7"
Print #2, "T7" ' set timing to 15 minutes
Print "Report Time set to 15 minutes"
Case 55 ' numeric key "8"
Print #2, "T8" ' set timing to 20 minutes
Print "Report Time set to 20 minutes"
Case 55 ' numeric key "9"
Print #2, "T9" ' set timing to 30 minutes
Print "Report Time set to 30 minutes"
Case 55 ' numeric key "0"
Print #2, "T0" ' set timing to 60 minutes
Print "Report Time set to 60 minutes"
Case 68,100 ' Alpha Key "D" or "d"
Print #2, "DB" ' set debug mode
if (DebugMode=False) then
    DebugMode=True
    Print "Turn Debug mode on"
else
    DebugMode=False
    Print "Turn Debug mode off"
End if
Case 70,102 ' Alpha Key "F" or "f"
if (FahrenheitMode=False) then
    FahrenheitMode=True
    Print "Turn Fahrenheit Mode on"
    Print #2, "FT"
else
    FahrenheitMode=False
    Print "Turn Fahrenheit Mode off"
    Print #2, "FF"
End if
Case 67,99 ' Alpha Key "C" or "c"
if (CelsiusMode=False) then
    CelsiusMode=True
    Print "Turn Celsius Mode on"
    Print #2, "CT"
else
    CelsiusMode=False
    Print "Turn Celsius Mode off"
    Print #2, "CF"
End if
Case 65,97 ' Alpha Key "A" or "a"
if (AVRMode=False) then
    AVRMode=True
    Print "Turn Avr Internal Mode on"
    Print #2, "IT"
else
    AVRMode=False
    Print "Turn AVR Internal Mode off"
    Print #2, "IF"
End if
Case 69,101 ' Alpha Key "E" or "e"
if (EEMode=False) then
    EEMode=True
    Print "Turn EEMode Mode on, ***NEXT AVR RESTART***"
    Print #2, "E+"
else
    EEMode=False
    Print "Turn EEMode Mode off, ***NEXT AVR RESTART***"
    Print #2, "E-"
End if
Case 82,114 ' Alpha Key "R" or "r"
Print #2, "00"
if (RoundMode=False) then

```

```

        RoundMode=True
        Print "Turn Rounding Mode on"
    else
        RoundMode=False
        Print "Turn Rounding Mode off"
    End if
Case 86,118          ' Alpha Key "V" or "v"
    if (RawMode=False) then
        RawMode=True
        Print "Turn Raw Reading Mode on"
        Print #2, "RT"
    else
        RawMode=False
        Print "Turn Raw Reading Mode off"
        Print #2, "RF"
    End if

Case 77,109          ' Alpha Key "M" or "m"
    Print "Setting minimal mode (Fahrenheit only)"
    if (DebugMode=True) then Print #2, "DB"
    DebugMode=False
    if (RoundMode=False) then Print #2, "00"
    DebugMode=True
    Print #2, "RF CF IF FT ST" ' Set minimal mode
    RawMode=False
    CelsiusMode=False
    AvrMode=False
    FahrenheitMode=True

Case 66,98           ' Alpha Key "B" or "b"
    Print #2, "W-"      ' restore from backup
Case 83,115          ' Alpha Key "S" or "s"
    Print #2, "ST"      ' print Status
Case 81,113          ' Alpha Key "Q" or "q"
    Print #2, "ED"      ' Dump EEPROM Storage
Case 76,108          ' Alpha Key "L" or "l"
    Print #2, "??"      ' print AVR help
Case 85,117,26,122   ' Alpha Key "U" or "u"
'      Send user defined String, String is defined in INI file
'      This violates the 'Keep it shut' rule
'      but this application only sends data once a minute
Sleep (1000)         ' allow up to one second second keypress
K=ASC(InKey)
if (K=49) and UserStr1<>"" then print #2, UserStr1
if (K=50) and UserStr2<>"" then print #2, UserStr2
if (K=51) and UserStr3<>"" then print #2, UserStr3
if (K=52) and UserStr4<>"" then print #2, UserStr4
if (K=53) and UserStr5<>"" then print #2, UserStr5
case 60,44           ' keys '<' and '>'
    OffsetVal=OffsetVal-0.25
    if OffsetVal=0 then OffsetVal=0.0001
    Print #2, "D0 " & OffsetVal
case 62,46           ' keys '>' and '<'
    OffsetVal=OffsetVal+0.25
    if OffsetVal=0 then OffsetVal=0.0001
    Print #2, "D0 " & OffsetVal
' thermometer application key inserted above -----
Case Else
    ' Add additional case statements for other Keyboard codes.
    ' Use TestKeyCode.exe to identify ASC() keycodes.
    ' Hint:You can use mapped keys for longer sequences:
    ' Case 65,97 then print #2, "Long command line Here".
    ' "A"=65, "a"=97
    '
    ' Or place Keyboard handling code here.
    ' Example:

```

```

' if you need to send special commands
' out the serial port to control the Arduino
' A=Ucase(chr(K))
' If instr("0123456789ABCDEF",A) then Print #2, A;
'
' Keep it SHORT ... the COM port will not be read again
' until this sequence is completed.
' It is best handle one key stroke at a time at a time.
' Otherwise:
' 1) Increase the size of the recieve buffer using the PortStr
' extended option RB (i.e. RB64 or RB128 or RB256 or RB1024)
' 2) Launch a separte thread for keyboard handling/processing
' and replace this section with access to a shared variable
'
End Select
End If
Wend
Return ExitCode
End Function
'
'====End of File=====

```

Ini File for Main Program

Arduinio_Thermometer.ini

```

PortStr = COM12:9600,N,8,1,CD,CS,DS,OP,BIN
;-----
; Arduino Thermometer monitors a specific COM port and displays any ASCII strings it receives.
; No warranties of any kind either expressed or implied.
; Note that the program only updates the display when it recieves an "end of line" character.
; any end of line characer: carriage return and/or linefeed
; empty lines (end of line character only) are discarded
; only ASCII characters between 32 and 126 are used
; with the exception horizontal tabs (ASCII 09)
;
; To exit the program press the Escape Key.
; Press "?" or "/" to print status information.
; (see parameter line "ExitKey" to define a different keypress for exit)
; The ASCII EOT character in the data stream will also terminate the application.
; EOT (End Of Transmission) = decimal 04.
;
; This file "Arduinio_Thermometer.ini" is the default INI file.
; The program looks for an INI file with the same name as the executable (including path).
; A different INI file may be specified on the command line to allow for multiple
; instances or alternate configurations.
;
; In the INI file:
; semicolons indicate remarks, the program ignores anything on a line following a semicolon
; parameters are NOT case sensitive
; either YES or TRUE evaluate to True
; a parameter line may appear anywhere in the ini file
; the last instance of any parameter line is used
;
; The only REQUIRED parameter line is PortStr.
; There are number of optional parameter lines.
; See details below for each parameter line.
;-----
; PortStr is used directly (without any changes) by the program to open a com port.
; Any of these strings except the last will work in the X86 Windows.
; The first seems to be reliable.
;
; PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP,BIN"
; PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP,ASC,FE,TB0,RB0"

```



```

; PortStr = "COM12:9600,N,8,1,CD,CS,DS,OP"
; PortStr = "COM12:9600,N,8,1,CD,CS,DS"
; PortStr = "COM12:9600,N,8,1,CD,CS"      '/' does not work for Arduino '/'
; "Com##: [ #### ][ , [ parity ][ , [ data_bits ][ , [ stop_bits ][ , [ extended_options ]]]]"
; (see Arduino_Receiver_Notes.bas for specific details)
; This is a ***REQUIRED*** parameter line.
;=====
; PortStr = COM12:9600,N,8,1,CD,CS,DS,OP,BIN
; (PortStr moved to top of file for 'obvious' easy access for new users)
;
;-----
; If ApndStr = True then the Date and Time are appended to the beginning of each string
; A delimiter (see below) is used to separate that value from the string
; The format of the date/time string is: YYYY-MM-DD HH:MM:SS
; Resolution is limited to one second.
; This is an optional parameter. Default is false.
;=====
ApndTime = TRUE
;
;-----
; StdOut is used for all display output.
; Setting this parameter to true suppresses status and some error messages.
; (**Fatal Error** messages are NOT suppressed)
; It also forces the delimiter parameter to be used for display output.
; This is an optional parameter. Default is false.
;=====
StdOut = False
;
;-----
; FileStr, if not empty, is used by the program to open a file.
; Whatever is received on the com port is appended to the file as well as the display.
; If you want the file in specific directory then include the directory path as well.
; Quotes are NOT required around FileStr (and deleted if found before any other processing).
; A ***SINGLE*** envirometnal variable of the form %VARIABLENAME% may be used at the
; beginning of the file/path name (i.e. %HOMEPATH%\Documents\My Logs\XXX.log).
; Relative paths may be used as well.
; This is an optional parameter. Default is no output file.
;=====
FileStr = Thermometer_0000.LOG
;
;-----
; SwitchHrs will cause filename to be incremented periodically based on hours.
; The Valid range is from 1 (one) to 24 (twenty-four).This is done by looking for "0000"
; in the FileStr and incremetning it. Existing matching file names are skipped.
; This is an optional parameter. Default is 0.
;=====
SwitchHrs = 12
;
;-----
; Delimiter options are:
; TAB ... Places one character code 09 between Date/Time and Data in file
; COMMA ... Places ", " between Date/Time and Data in file
; Collon ... Places ": " between Date/Time and Data in file
; Number ... Places number spaces between Date/Time and Data in file
; 0 (zero) is a valid option and effectively eliminates the Delimiter
; This is an optional parameter. Default is TAB character.
;=====
Delimiter = TAB
;
;-----
; EOL sets the end of line termiantion used in the output file.
; EOL = 0 for carriage return plus line feed (Windows/DOS)
; EOL = 1 for line feed (Linux ?)
; EOL = 2 for carriage return (MAC ?)
; EOL when defined is also used for terminations of Display strings.

```

```

; (Except for the initial command line help request which is called before the INI file is read.)
; This is an optional parameter. Default is carriage return plus line feed.
;=====
EOL = 0
;
;-----
; ExitKey allows a specif key to be used to exit the program. This is usefule to avoid
; accidental exits. The ASCII value must be in the range 32 to 126 a keyword is used:
; "Escape" = ASCII 27 (the escape Key).
; "CtlX" = ASCII 24 (Control X)
; Note: Control "C" will cause an 'abnormal termination'.
; The utility TestKeyCode.exe can be used to indetify specific keycodes.
; This is an optional parameter. Default is Escape key.
;=====
ExitKey = Escape
;
;-----
; USERSTR#: USERSTR1, USERSTR2, USERSTR3, USERSTR4, USERSTR5
; This was added for the Arduino Thermometer application.
; The user can define up to five string that will be sent when the user presses
; the "U" key followed by "1", "2", "3", "4" or "5" within one second.
; These are an optional parameters.
;=====
USERSTR1=Z1 ;default is Z1 = load default data set 1
USERSTR2=Z2 ; default is Z2 = load default data set 2
USERSTR3=ZD ; default is ZD = Dump EEPROM
; End of File

```

Utility Program

Code File for stripsemicolonlines.exe = stripsemicolonlines.bas

This is the file that **must** be active when you select Compile in the FBIDE editor.

```

'-----define False-----
#ifndef FALSE
#define FALSE 0
#endif
#ifndef False
#define False 0
#endif
#ifndef false
#define false 0
#endif
'-----define True-----
#ifndef TRUE
#define TRUE -1
#endif
#ifndef true
#define true -1
#endif
#ifndef True
#define True -1
#endif

'--declare variables -----
Dim SHARED FileIn as String
Dim SHARED FileOut as String
Dim SHARED SemiOut as String
Dim SHARED Work as String
Dim SHARED Buffer as String
Dim SHARED HlpStr as String
Dim SHARED PrgStr as String
Dim SHARED EOL as String
Dim SHARED C as String

```

```

Dim SHARED P as Integer
Dim SHARED I as Integer
Dim SHARED OverWrite as Byte
Dim SHARED AppendMode as Byte
Dim SHARED DeleteFile as Byte
Dim SHARED SplitLine as Byte
Dim SHARED AllLines as Byte
Dim SHARED Retain as Byte
Dim SHARED MarkMode as Byte
Dim SHARED Verbose as Byte
Dim SHARED DeBugMe as Byte
Dim SHARED RtnCode as Integer
Dim SHARED SemiCount as Long
Dim SHARED LineCount as Long
Dim SHARED BlankCount as Long
Dim SHARED PartialCount as Long
Dim SHARED MarkLine as Long

' set defaults -----
PrgStr=""
EOL=chr(13) & Chr(10)
OverWrite=False
AppendMode=False
DeleteFile=False
SplitLine=False
AllLines=False
Retain=False
MarkMode=False
DeBugMe=False
Verbose=False
SemiCount=0
LineCount=0
BlankCount=0
PartialCount=0
MarkLine=0
I=0
P=0

' define Help string-----
HlpStr="Syntax: StripSemicolonLines.exe file1 file2 file3 [options]" & EOL & _
"  file1 = input filename" & EOL & _
"  file2 = output filename" & EOL & _
"  file3 = output filename with stripped lines (optional)" & EOL & _
"Options:" & EOL & _
"  /O = Overwrite any existing output file" & EOL & _
"  /A = Append to any existing output file (overrides /O)" & EOL & _
"  /D = Delete input file" & EOL & _
"  /R = Retain blank lines" & EOL & _
"  /S = Split lines at semicolon and delete trailing portion" & EOL & _
"  /X = Deletes all lines with semicolon regardless of location" & EOL & _
"  /M = Mark end of file with " & chr(34) & "--PROCESSED--;" & chr(34) & EOL & _
"        On next run seeks to last marker before processing" & EOL & _
"  /E = Execute program with output file" & EOL & _
"        Program name is delimited by a collon" & EOL & _
"        Example /E:" & chr(34) & "full path\MyProgram.exe" & chr(34) & EOL & _
"  /V = Verbose prints statistics before exiting" & EOL & EOL & _
"  /? = display help and exit" & EOL & EOL & _
"All lines that begin with semicolons will be removed from oputput file" & EOL & _
"Unless /R option is used all blank lines will also be removed from oputput file" & EOL & _
"Trailing blanks are deleted in any case" & EOL & _
"-----This program placed in the PUBLIC DOMAIN October 2013-----"

' get the command line -----
Buffer=Command(1)
If Buffer="/DB" then DeBugMe=True          ' undocumented debug mode

```

```

p=1
While (Buffer<>"" )
  Buffer=Trim(Ucase(Command(P)))
  If (DebugMe=True) then print Buffer: Sleep
  If Buffer=""? then Print HlpStr: End: end If

  If len(Buffer)=2 then
    If Buffer="/"? then Print HlpStr: End: end If
    If Buffer="-?" then Print HlpStr: End: end If
    If Buffer="??" then Print HlpStr: End: end If

    If Buffer="/O" then OverWrite=True
    If Buffer="/A" then AppendMode=True
    If Buffer="/D" then DeleteFile=True
    If Buffer="/S" then SplitLine=True
    If Buffer="/X" then AllLines=True
    If Buffer="/R" then Retain=True
    If Buffer="/M" then MarkMode=True
    If Buffer="/V" then Verbose=True

    If Buffer="-O" then OverWrite=True
    If Buffer="-A" then AppendMode=True
    If Buffer="-D" then DeleteFile=True
    If Buffer="-S" then SplitLine=True
    If Buffer="-X" then AllLines=True
    If Buffer="-R" then Retain=True
    If Buffer="-M" then MarkMode=True
    If Buffer="-V" then Verbose=True

  else
    If Left(Buffer,3)= "/E:" then
      ' allow for upper/lower case
      PrgStr=Trim(Mid(Trim(Command(P)),4))
    else
      If Buffer="/DB" then
        ' do nothing
      Else ' must be a filename
        I=I+1
        If I=1 then FileIn=Trim(Command(P))
        If I=2 then FileOut=Trim(Command(P))
        If I=3 then SemiOut=Trim(Command(P))
      End If
    end If
  end If
  ' next command parameter
  P=P+1
Wend

' Appendmode overrides Overwrite mode -----
if AppendMode=true then OverWrite=False

' debug -----
if (DebugMe=True) then
  Print "   FileIn: " & FileIn
  Print "   FileOut: " & FileOut
  Print "   SemiOut: " & SemiOut
  Print "   PrgStr: " & PrgStr
  Print " OverWrite: " & OverWrite
  Print "AppendMode: " & AppendMode
  Print "DeleteFile: " & DeleteFile
  Print " SplitLine: " & SplitLine
  Print " AllLines: " & AllLines
  Print "   Retain: " & Retain
  Print " MarkMode: " & MarkMode

```

```

    Print "    Verbose: " & Verbose
    sleep
end if

' check file names -----
if (FileIn="") then Print EOL & HlpStr: End: end If
if dir(FileIn)=" " then
    Print EOL & "****ERROR*** Input file not found" & EOL
    Print FileIn
    end
End If

if (FileOut="") then
    Print EOL & "****ERROR*** No output file specified" & EOL
    End
end If

If dir(FileOut)<> "" then
    if (OverWrite=true) then
        kill (FileOut)
        RtnCode=Err
        if(RtnCode<>0) then
            Print EOL & "****ERROR*** Cannot delete existing output file"
            Print "    " & FileOut
            Print "    Error= " & RtnCode
            End
        end if
    else
        If AppendMode=false then
            Print EOL & "****ERROR*** Output file exists."
            Print " Overwrite option not specified."
            Print " Append option not specified."
            Print "    " & FileOut
            Print EOL & HlpStr
        End if
    end if
End If

If SemiOut<>"" then
    If dir(SemiOut)<> "" then
        if (OverWrite=true) then
            kill (SemiOut)
            RtnCode=Err
            if(RtnCode<>0) then
                Print EOL & "****ERROR*** Cannot delete existing output file"
                Print "    " & SemiOut
                Print "    Error= " & RtnCode
                End
            end if
        else
            If AppendMode=false then
                Print EOL & "****ERROR*** Output file exists."
                Print " Overwrite option not specified."
                Print " Append option not specified."
                Print "    " & SemiOut
                Print EOL & HlpStr
            End if
        end if
    End If
End If

' Find last mark ?? -----
if (MarkMode=True) then
    Open FileIn for Input as #1
    While not(eof(1))

```

```

        Line Input #1, Buffer
        LineCount=LineCount+1
        if Buffer="";--PROCESSED--;" then MarkLine=LineCount
    Wend
    Close #1
    LineCount=0
    If (DebugMe=True) then print "Mark Line: " & MarkLine : Sleep
end If

' Open Files -----
Open FileIn for Input as #1
RtnCode=Err
If (RtnCode) <>0 then
    Print EOL & "****ERROR*** Can not open Input file."
    Print "    " & FileIn
    Print "    Error= " & RtnCode
End
End If

Open Fileout for Append as #2
RtnCode=Err
If(RtnCode) <>0 then
    Print EOL & "****ERROR*** Can not open Output file."
    Print "    " & FileOut
    Print "    Error= " & RtnCode
End
End If

If SemiOut<>"" then
    Open SemiOut for Append as #3
    RtnCode=Err
    If(RtnCode) <>0 then
        Print EOL & "****ERROR*** Can not open Output file."
        Print "    " & SemiOut
        Print "    Error= " & RtnCode
    End
End If
End If
If (DebugMe=True) then print "files opened": Sleep
'

' ---- process file -----
' see to last file mark
While LineCount<MarkLine
    Line Input #1, Buffer
    LineCount=LineCount+1
Wend
LineCount=0

While not (EOF(1))
    Line Input #1, Buffer
    if Trim(buffer)="" then
        BlankCount=BlankCount+1
        If (Retain=True) then Print #2,""
    Else
        C=left(trim(buffer),1)
        If C=";" then
            SemiCount=SemiCount+1
            If SemiOut<>"" then Print #3,rtrim(buffer)
        else
            P=Instr(Buffer,";")
            if (AllLines=True) and (P>0) then
                SemiCount=SemiCount+1
                If SemiOut<>"" then Print #3,rtrim(buffer)
            Else

```

```

        if (SplitLine=True) and (P>0) then
            SemiCount=SemiCount+1
            PartialCount=PartialCount+1
            If SemiOut<>"" then Print #3,rtrim(buffer)
            Buffer = Left(Buffer, P-1)
            Print #2, rtrim(Buffer)
        Else
            LineCount=LineCount+1
            Print #2, rtrim(Buffer)
        end if
    end if
end If
end If
Wend
If (DebugMe=True) then print "files processed": Sleep

' close files -----
close #1
close #2
If SemiOut<>"" then Close #3
if DeleteFile=True then
    kill (FileIn)
    RtnCode=Err
    if(RtnCode<>0) then
        Print EOL & "****ERROR*** Cannot delete input file"
        Print " " & FileIn
        Print " Error= " & RtnCode
        End
    end if
End If
If (DebugMe=True) then print "files closed": Sleep

' Mark file as processed -----
if (MarkMode=True) then
    Open FileIn for Append as #1
    print #1, ";;--PROCESSED--;;"
    Close #1
    If (DebugMe=True) then print "File Marked": Sleep
end If

' report -----
If Verbose=True then
    Print EOL & " Total Lines: " & BlankCount+SemiCount+LineCount
    Print "Semicolon Lines: " & SemiCount
    if (SplitLine=True) and (PartialCount>0) then Print " Partial Lines: " & PartialCount
    if (BlankCount>0) then Print " Blank Lines: " & BlankCount
    if (SplitLine=True) then LineCount=LineCount+PartialCount
    If (Retain=True) then LineCount=LineCount+BlankCount
    Print " Output Lines: " & LineCount
End If

' Execute ?? -----
If PrgStr<>"" then
    If (DebugMe=True) then
        print "Executing Program: "
        print PrgStr & " " & chr(34) & Fileout & chr(34)
        Sleep
    end if
    RtnCode=Run (PrgStr, chr(34) & Fileout & chr(34))
    If RtnCode<>0 then
        print "****ERROR*** Execution failed:"
        print PrgStr & " " & chr(34) & Fileout & chr(34)
    end if
end If
If (DebugMe=True) then print "Program Complete": Sleep

```

```
' sleep waits for a key press  
' go away -----  
end
```


Appendix: Thermometer One Program Code (Plan “A”)

Thermometer One Main Program File

```
/* ThermometerOne */

#include <avr/sleep.h> // needed for shutdown function
#include <EEPROM.h> // needed for EEPROM read and write
#include <HexDecAsc.h> // used for EEPROM dump All

// EEPROM Address Constants
const word EEmask = 0; // 1 byte location of EEPROM storage mode mask
const word EEflag = 1; // 1 byte location of EEPROM storage mode flag
const word EEOffsetR= 2; // 2 byte location of CovrtOffset
const word EEcelsius= 4; // 2 byte location of Covrt2Celsius
const word EEminutes= 6; // 2 byte location of Report Target Minutes
const word EEunused0= 8; // 2 byte location -- unused --
const word EEunused1= 10; // 2 byte location -- unused --
const word EEunused2= 12; // 2 byte location -- unused --
const word EEunused3= 14; // 2 byte location -- unused --
const word EEidtring= 16; // ID string w/o termination size (16)
const word EEidsize = 16; // 24 byte location of IdString
const word EEwdsiz = EEidtring+EEidsize; // Working data storage size (32)
//-----
const word StorageWorking=EEmask; // EEPROM start for working calibration data
// EEPROM start for backup copy of constants
// note we have to add 1 to the value
// Becasue addresses begin with zero not one
const word StorageBackup=((E2END-(EEwdsiz))+1);

// EEPROM addresses variables
word StorageBegin =StorageWorking+EEwdsiz; // begin storage for report data
word StorageMark =StorageBegin; // marks start of current segment
word StorageEnd =StorageBackup; // marks end of current segment
word StorageIndex =StorageBegin; // index for next EEPROM write

// Conversion Factors/Calibration Data
const float CovrtFactor=65532;
char IdString[EEidsize+1]; // ID/Location string for this device
word CovrtOffsetR; // Raw Reading offset
const word CovrtOffsetC=20; // Celsius temperature offset (default 20)
const word CovrtOffsetF=68; // Fahrenheit temperature offset (default 68)
float Covrt2Celsius; // Linear scale factor for Celsius (default 0.25)
float Covrt2Fahrenheit; // Linear scale factor for Fahrenheit (default 0.45)
float Celsius; // Last conversion to Celsius Temperature
float Fahrenheit; // Last conversion to Fahrenheit Temperature
byte newflg=0; // used to indicate new conversion factors in memory

// Global operational mode Variables // set default operation modes
boolean ReportMode = true; // True = reporting, False = Command Mode
boolean RtnRawRead = true; // True = include Raw
boolean RtnFahrenh = true; // True = include Fahrenheit
boolean RtnCelsius = true; // True = include Celsius
boolean DeBug = false; // True = extended reporting for debugging
boolean SleepMode = false; // False - sleepmode not implimented
boolean EepromMode = false; // False - write data to EEPROM

// Global work Variables
char cmd[] = {0,0,0}; // used to store two character command
char prevcmd[] = {0,0,0}; // used to store previous two character command
word MinuteTarget = 1; // Number of minutes between report lines
// word SecondsMinute = 10000; // --- to speed things up a bit for debugging
word SecondsMinute = 60000; // added so calibration timing can be reduced
unsigned long SecondsTarget = 0; // Number of seconds between report lines
```

```

unsigned long Accumalator    = 0;           // Accumalate temperature reads
unsigned long CycleCount     = 0;           // Cycles per Report line
unsigned long RptStartTime   = 0;           // Time between report lines
unsigned long RptTrigger     = 0;           // Target Time for report
unsigned long CycleStart     = 0;           // Target Time for report
unsigned long CycleTime      = 0;           // Target Time for report
word          LastRead       = 0;           // Stores previous RawRead Average
byte          Consecutive    = 0;           // used to count consective equal readings
byte          gap            = 0;           // used to increase gap between reads
                                           // There are 1000 milliseconds in a second

//-----
void setup()
{
  char c;
  Serial.begin (9600);
  pinMode(13, OUTPUT);                      // so we can blink it later during writes
  EnableADC();                             // enables the ADC and set ADC clock factor
  delay (1000);                             // let serial library complete setup
  while (Serial.available()>0)              // drain any data from the serial buffer
    c=Serial.read();
  Read_Calibration_Data();                  // read and set conversion factors from EEPROM
  Check_EEPROM();                          // see if we are writing to EEPROM vs Serial
  // calculate seconds between report lines
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=long(MinuteTarget)*long(SecondsMinute);
  //delay (5000);                             // allow PC 5 seconds to get setup
  if (EepromMode==false) ReportStatus();    // report default parameters
  Accumalator = 0;                           // set startup parameters
  CycleCount  = 0;
  RptTrigger  = millis() + SecondsTarget;
  RptStartTime= millis();
  //-----debugging stuff-----
  // Serial.println ("Got here");
  // while (true);
}

//-----
void loop()
{
  char c1, c2;
  word wtemp;

  // ---- This is where we check for command input
  if ((Serial.available()>2) && (EepromMode==false))
    { if(ReadTwoCharacters()) CmdProcessor();}

  // ---- This is where we collect our temperature data
  // gap is used to increase the amount of time between reading sampling the ADC.
  // This Insures that we will not miss any data transmitted on the serial port.
  if (gap++ == 9)
    { gap=0;
      // cycle times are only used if debugging is turned on
      if (DeBug == true) CycleStart= millis();
      avrRawTemp();
      if (DeBug == true) CycleTime = CycleTime+(millis()-CycleStart);
    }

  // ---- This is where we output the teperature data
  // The time required to read 64 samples is about 119-120 milliseconds. If we get
  // within 125 milliseconds of the Report Trigger Time then we wait for it.
  // With the these timing numbers there are 500 reads of
  // 64 virtual 12 bit samples per minute.
  // Added condition for millis exceeding report trigger (possible with long commands)
  if (((RptTrigger-millis())< 125) || (millis()>RptTrigger))
    { // Serial.println ("Got here: RptTrigger ");
      while (millis() < RptTrigger);
    }
}

```

```

        // We want the new trigger time set as close as possible to when the previous trigger
        // went off --- so we put ti first.
        RptTrigger= (millis() + (SecondsTarget));
        // Serial.print (F("Got Here: RptTrigger, milliseconds to wait= "));
        // Serial.println (SecondsTarget);
        Report();
    }
}

//=====
void CmdProcessor()
{
    // this function is the main command handler
    // not many comments because I think the code is obvious
    if (Debug == true)
    {
        Serial.print (F("; Command Processor "));
        DebugPrintCharacters (cmd[0],cmd[1]);
    }

    if ((cmd[0]=='I') && (cmd[1]=='D')) {Serial.print(F("; ")); Print_IdString();}
    else if ((cmd[0]=='S') && (cmd[1]=='T')) ReportStatus();
    else if ((cmd[0]=='O') && (cmd[1]==':')) NewOffsetR();
    else if ((cmd[0]=='C') && (cmd[1]==':')) NewCelsius();
    else if ((cmd[0]=='C') && (cmd[1]=='=')) CelsiusEquals();
    else if ((cmd[0]=='F') && (cmd[1]==':')) NewFahrenheit();
    else if ((cmd[0]=='F') && (cmd[1]=='=')) FahrenheitEquals();
    else if ((cmd[0]=='L') && (cmd[1]==':')) NewIdString();
    else if ((cmd[0]=='D') && (cmd[1]=='B')) ToggleDebugMode();
    else if ((cmd[0]=='W') && (cmd[1]=='W')) Write_Calibration_Data();
    else if ((cmd[0]=='W') && (cmd[1]=='+')) OverwriteBackup();
    else if ((cmd[0]=='W') && (cmd[1]=='-')) RestoreFromBackup();
    else if ((cmd[0]=='L') && (cmd[1]=='L')) HelpMe();
    else if ((cmd[0]=='?') && (cmd[1]=='?')) HelpMe();
    else if ((cmd[0]=='S') && (cmd[1]=='S')) ShutDown();
    else if ((cmd[0]=='I') && (cmd[1]=='I')) software_Reset();
    else if ((cmd[0]=='E') && (cmd[1]=='+')) EEmodeFlagSet();
    else if ((cmd[0]=='E') && (cmd[1]=='-')) EEmodeFlagClear();
    else if ((cmd[0]=='E') && (cmd[1]=='C')) ClearStorage();
    else if ((cmd[0]=='E') && (cmd[1]=='D')) DumpStorage();
    // else if ((cmd[0]=='T') && (cmd[1]=='T')) TestTest();
    else if (cmd[0]=='R') SetRawReadMode();
    else if (cmd[0]=='F') SetFahrenheitMode();
    else if (cmd[0]=='C') SetCelsiusMode();
    else if (cmd[0]=='T') NewReportTime();
    else if (cmd[0]=='P') SetReportMode();

    // example of commands not implemented
    else if ((cmd[0]=='A') && (cmd[1]==':')) PrintNotImplemented();
    else if ((cmd[0]=='S') && (cmd[1]==':')) PrintNotImplemented();
    // example of application specific command implemented
    // these two commands write test data to the EEPROM working storage
    else if ((cmd[0]=='Z') && (cmd[1]=='1')) TestData1();
    else if ((cmd[0]=='Z') && (cmd[1]=='2')) TestData2();
    // this command used for calibration, changes reporting to 5 seconds
    else if ((cmd[0]=='Z') && (cmd[1]=='Z')) CalibrationMode();
    // this command used to dump entire EEPROM to Serial Port
    else if ((cmd[0]=='Z') && (cmd[1]=='D')) EepromDumpAll();
    else PrintNotRecognized(); // not recognized
}

//-----
//void TestTest()
// { for (byte i=0; i< 20; i++)
//     { QuickBlink();
//       delay (200);
//     }
// }

```

```

//-----
void HelpMe()
//Serial.println(F("This string will be stored in flash memory"));
{ PrintSeperatorLine();
  Serial.println(F("; Arduino AtMega328 Internal Temperature Sensor 1.0"));
  Serial.println(F("; ID Output ID string"));
  Serial.println(F("; ST Output Status"));
  Serial.println(F("; RT Raw=True"));
  Serial.println(F("; RF Raw=False"));
  Serial.println(F("; FT Fahrenheit=True"));
  Serial.println(F("; FF Fahrenheit=False"));
  Serial.println(F("; F= Enter Current Fahrenheit"));
  Serial.println(F("; CT Celsius=True"));
  Serial.println(F("; CF Celsius=False"));
  Serial.println(F("; C= Enter Current Celsius"));
  Serial.println(F("; T1 Report time = 01 minutes"));
  Serial.println(F("; T2 Report time = 02 minutes"));
  Serial.println(F("; T3 Report time = 03 minutes"));
  Serial.println(F("; T4 Report time = 04 minutes"));
  Serial.println(F("; T5 Report time = 05 minutes"));
  Serial.println(F("; T6 Report time = 10 minutes"));
  Serial.println(F("; T7 Report time = 15 minutes"));
  Serial.println(F("; T8 Report time = 20 minutes"));
  Serial.println(F("; T9 Report time = 30 minutes"));
  Serial.println(F("; T0 Report time = 60 minutes"));
  Serial.println(F("; TA Report time = 02 hours"));
  Serial.println(F("; TB Report time = 04 hours"));
  Serial.println(F("; TC Report time = 06 hours"));
  Serial.println(F("; TD Report time = 08 hours"));
  Serial.println(F("; TE Report time = 12 hours"));
  Serial.println(F("; TF Report time = 24 hours"));
  Serial.println(F("; PF Print mode = False"));
  Serial.println(F("; PT Print mode = True"));
  Serial.println(F("; DB Debug mode toggle"));
  Serial.println(F("; L: New Location"));
  Serial.println(F("; O: New Raw Offset"));
  Serial.println(F("; F: New Fahrenheit Factor"));
  Serial.println(F("; C: New Celsius Factor"));
  Serial.println(F("; WW Write Calibraton data to EEPROM"));
  Serial.println(F("; W+ Overwrite Backup Calibraton data"));
  Serial.println(F("; W- Restore from Backup Calibraton data"));
  Serial.println(F("; E+ Set Flag to send next run to EEPROM"));
  Serial.println(F("; E- Clear Flag to send next run to EEPROM"));
  Serial.println(F("; EC Clear EEPROM Storage"));
  Serial.println(F("; ED Dump data stored in EEPROM"));
  Serial.println(F("; LL List implimented commands"));
  Serial.println(F("; ?? List implimented commands"));
  Serial.println(F("; SS Shutdown (send twice)"));
  Serial.println(F("; !! Reset (send twice)"));
  Serial.println(F("; Responce 'XX' = not implemented"));
  Serial.println(F("; Responce '??' = not recognized"));
  PrintSeperatorLine();
  // example of application specific command implimneted
  // these two commands write test data to the EEPROM working storage
  Serial.println(F("; Z1 Write test data 1"));
  Serial.println(F("; Z2 Write test data 2"));
  // special calibration mode
  Serial.println(F("; ZZ 5 Second reporting for calibration"));
  // Dump all EEPROM memory to Serial in Hex and ASCII
  Serial.println(F("; ZD Dump ALLL EEPROM to serial"));
  PrintSeperatorLine();
}

//-----
void PrintSeperatorLine()

```

```

{ Serial.println(F("; -----"));
}

//=====
boolean ReadTwoCharacters()
{ char c1=0,c2=0,c3=-1;
  byte m=0;
  boolean EOC=true; // End of Command Terminator
  boolean OurReturn=false;

  // It is not to be believed how much effort went into creating this simple function to read
  // two characters. I noted a bit of problem reading characters from the serial port when
  // the loop was too fast therefore I have added a bit of a delay to insure the serial port
  // library can keep up. Worst case senario this function can take more than 250 milliseconds.
  // Normally when this function is called we expect the three bytes we need to be in the buffer
  // but if there is noise on the line or a parrot randomly pecking at the keyboard it could
  // take a bit longer.
  //
  // by definition we are looking for two characters followed by a terminator
  // we define a command terminator to be a carriage return, new line or null character
  // --- for good measure we are including the tab character and space as well
  // space was added because it is impossible to send a tab character from the Arduino IDE
  // we will accept any combination of those characters as a single terminator
  // we will accept the last two printable ASCII characters before a terminator for our command
  // we keep reading until we get a terminator, but we will only read for a short period

  // but before we do anything else we are going to save the previous command for posterity
  prevcmd[0]=cmd[0]; // actually we are saving it so that shutdown
  prevcmd[1]=cmd[1]; // and reset can check it before they execute

  while ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32) && (m<25))
  { // if we have a valid ASCII character for c3 then roll the characters down
    if (Serial.available()>0)
    { c3=Serial.read();
      if (c3>32) {c1=c2; c2=c3;}
    }
    // we need a bit of a delay to let the serial interface catch up
    // after 25 empty reads we give up
    else { delay (10); m++;}
  }
  // DebugPrintCharacters (c1,c2,c3,m);

  // we are very liberal about what we will accept for a command terminator
  // but we insist on having one.
  if ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32)) EOC= false;
  // we need to drain any remaining command terminator characters from the serial buffer
  else DrainCmdTerminators();

  // now check for valid ASCII characters and End of Line
  if ((c1>32) && (c2>32) && EOC)
  { // OK... we have something to work with
    // Convert lower case to UPPER case except "w"
    // DebugPrintCharacters (c1,c2,c3);
    if ((c1 != 'w') && (c1 >96) && (c1 <123)) c1 = (c1 -32);
    if ((c2 != 'w') && (c2 >96) && (c2 <123)) c2 = (c2 -32);
    // DebugPrintCharacters (c1,c2);
    cmd[0]=c1;
    cmd[1]=c2;
    OurReturn=true;
  }
  // whatever it was that was sent did not meet our criteria
  // inform the parrot that he or she must do better
  else Serial.println(F("; ?? ??"));
  return OurReturn;
}

```

```

//-----
void DrainCmdTerminators()
{ char c3=0;
  // removed leading command terminators from serial buffer
  delay (10); c3=Serial.peek();
  while ((c3==13) || (c3==10) || (c3==9) || (c3==0) || (c3 == 32))
  { c3=Serial.read();
    delay (10);
    c3=Serial.peek();
  }
  // c3 should at this point should be -1 unless there are more commands/charaters in the
  buffer
}

//=====
// overloaded debugging function for debugging the above input routine
void DebugPrintCharacters (char c1, char c2, char c3, byte m)
{
  if (DeBug == true)
  { Serial.print ("Received: ");
    Serial.print (c1);
    // Serial.print (" ");
    Serial.print (c2);
    Serial.print (" ");
    if (c3 != 0)
    { Serial.print (c3, DEC);
      Serial.print (" ");
    }
    if (m != 0) Serial.print (m, DEC);
    Serial.println ();
  }
}

void DebugPrintCharacters (char c1, char c2, char c3)
{ byte m=0;
  DebugPrintCharacters (c1,c2,c3,m);
}

void DebugPrintCharacters (char c1, char c2)
{ byte m=0;
  char c3=0;
  DebugPrintCharacters (c1,c2,c3,m);
}

//-----End of Main File-----

```

Thermometer One Functions Module

```

// cbi and sbi are standard (AVR) methods for setting,
// or clearing, bits in PORT (and other) variables.
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

//-----
void EnableADC()
{ // This is probably not needed but ....
  // set system clock divisor to 128
  // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
  sbi(ADCSRA, ADPS2); // bit 2 of ADCSRA, system clock divisor
  sbi(ADCSRA, ADPS1); // bit 1 of ADCSRA, system clock divisor
}

```

```

    sbi(ADCSRA, ADPS0);           // bit 0 of ADCSRA, system clock divisor
    cbi(ADCSRA, ADSC);           // bit 5 of ADCSRA, disable auto trigger mode
    sbi(ADCSRA, ADEN);           // bit 7 of ADCSRA, enable ADC
}

//-----
void Read_Calibration_Data()
{ // This is more or less taken straight from the
  // EEPROM_TempSensor_Calibration_Constants program
  byte i=0, j=0;
  float saveflt;
  word TempWord;

  // detect a virgin device --- well at least try
  if ((EEPROM.read(EEminutes)==0xFF) &&
      (EEPROM.read(EEminutes+1)==0xFF))
    TestData1();

  // Get the rawreading offset;
  CovrtOffsetR= EEPROM.read(EEOffsetR)<<8;
  CovrtOffsetR= (CovrtOffsetR + EEPROM.read(EEOffsetR +1));

  // Get the Covrt2Celsius factor;
  TempWord= EEPROM.read(EEcelsius)<<8;
  TempWord= TempWord + EEPROM.read(EEcelsius +1);
  // now we need to convert it
  Covrt2Celsius = float(TempWord)/CovrtFactor ;

  // calculate Covrt2Fahrenheit factor;
  Covrt2Fahrenheit = Covrt2Celsius * 1.8000;

  // ID String -----
  i=0; c=1;
  while (c!=0,i< EEidsize)
  {c=EEPROM.read(EEidtring + i);      // read the ID string
    IdString [i++]=c;
  }
  IdString [EEidsize]=0;              // just in case

  // Set MinuteTarget from default minutes
  MinuteTarget=(EEPROM.read(EEminutes)<<8) + EEPROM.read(EEminutes +1);
  if (MinuteTarget<1)MinuteTarget=1;

  // this is a bit flag to indicate when the current constants
  // in memory are different from those stored in working storage
  newflg=0;

  // Get the unused-----
  // TempWord= EEPROM.read(EEOffsetR)<<8;
  // TempWord= TempWord + EEPROM.read(EEOffsetR+1);
  // unused=TempWord
  // CovrtOffsetR=1389;
  // Covrt2Celsius = 0.25;
  // Covrt2Fahrenheit = 0.45;
  // MinuteTarget=1;
}

//-----
void Write_Calibration_Data()
{ word i=0;
  // float saveflt;
  word TempWord;
  char c=-1;

  // New location

```

```

    if ((newflg & B00000001) == B00000001)
        { // Serial.println ("Got Here: Write_Calibration_Data, Location");
          for (i=0; i<EEidsize; i++) {EEPROM.write((EEidtring +i), IdString[i]);}
        }

    // New Offset Constant
    if ((newflg & B00000010) == B00000010)
        { // Serial.println ("Got Here: Write_Calibration_Data, Offset");
          TempWord=word(CovrtOffsetR);
          EEPROM.write(EEoffsetR , highByte(TempWord));
          EEPROM.write(EEoffsetR +1, lowByte (TempWord));
        }
    // New Celsius Factor
    if ((newflg & B00000100) == B00000100)
        { //Serial.println ("Got Here: Write_Calibration_Data, Celsius Factor");
          TempWord = word(Covrt2Celsius * CovrtFactor);
          EEPROM.write(EEcelsius , highByte(TempWord));
          EEPROM.write(EEcelsius +1, lowByte (TempWord));
          // Note:
          // When the EEPROM data is read
          // Covrt2Fahrenheit is calculated from Covrt2Celsius
        }

    // New MinuteTarget
    if ((newflg & B00010000) == B00010000)
        { // Serial.println ("Got Here: Write_Calibration_Data, MinuteTarget");
          EEPROM.write(EEminutes , highByte(MinuteTarget));
          EEPROM.write(EEminutes +1, lowByte (MinuteTarget));
        }

    // write the unused word(s)
    i=EEunused0;
    while (i<EEidtring)
        {if (EEPROM.read(i)!= 0xFF) EEPROM.write(i, 0xFF);
          i++;
        }

    // clear the EEMODE flag
    EEmodeFlagClear();

    // Serial.println ("Got Here: Write_Calibration_Data, Read");
    // note newflg is reset by Read_Calibration_Data
    Read_Calibration_Data();
    // PrintOKStr(); is sent by Report_Reset
    Report_Reset();
}

//-----
void ClearStorage()
{ // this is used to clear/erase the EEPROM data storage (except for constant areas)
  word addr;
  byte b;
  for (addr=( StorageBegin); addr<StorageBackup; addr++)
      { if (EEPROM.read(addr) != 0xFF) EEPROM.write(addr,0xFF);
        } // note: each byte requires 6-8 machine cycles
  PrintOKStr ();
}

//-----
void EEmodeFlagSet()
{ // toggle the flag forthe next run to write to EEPROM
  // does not affect current run
  // Here is the thing. We have setup wearleveling for our EEPROM data storage
  // but modeflag gets hit twice for every EEPROM run. So it will wearout
  // long before the bulk of the storage. We could just increment the byte and

```



```

// look for odd or even values but that would continually toggle the low bit
// wearing it out before there rest. As it turns out it is writing a zero to
// a bit that wears then out. So we want to minimize the zero bit writes.
// We are going to move a zero bit right to left. This extends our life by a
// factor of eight. At that point you need to swap the backup and working
// data locations by changing the EEPROM address locations and reprograming
// the Arduino. That would double the life (2 * 8 = 16).
//
byte flag;
if(EEmodeFlagTF()==false)
{ // Serial.println (F("Got here: EEmodeFlagToggle, make not equal"));
  // make them not equal
  // shift left and add a one to the right
  flag=EEPROM.read(Eeflag);
  // Serial.println(flag);
  flag=(flag<<1)+1;
  // Serial.println(flag);
  // if we have all ones start over again at the right
  if (flag==B11111111) flag=B11111110;
  // Serial.println(flag);
  // now save it
  EEPROM.write(Eeflag,flag);
}
PrintOKStr();
}

//-----
void EEmodeFlagClear()
{ byte flag;
  if(EEmodeFlagTF())
  { // Serial.println (F("Got here: EEmodeFlagClear"));
    // make them equal
    flag=EEPROM.read(Eeflag);
    EEPROM.write(EEmask,flag);
  }
  if (EepromMode==false) PrintOKStr();
}

//-----
boolean EEmodeFlagTF()
{ // returns true if EEmodeFlag is set
  byte flag,mask;
  flag=EEPROM.read(Eeflag);
  mask=EEPROM.read(EEmask);
  if(flag==mask) return false;
  else return true;
}

//-----
void Check_EEPROM()
{ byte b[4],i;
  word w;
  word addr;

  EepromMode=false;
  if (EEmodeFlagTF())
  { // Serial.println ("got here: Check_EEPROM");
    EepromMode=true;
    // clear the flag
    EEmodeFlagClear();
    // disable serial reporting and debug mode
    ReportMode=false;
    DeBug=false;
    // we need to find the beginning of EEPROM that has not been used
    // we need at least four bytes to begin a new section.

```

```

// so we need to find the first place where there are four bytes with FFh
// zero our test pattern
for (i=0; i<4; i++) b[i]=0;
addr=StorageBegin;
while ((addr<StorageBackup) && ((b[0]!=0xFF) || (b[1]!=0xFF)
                                || (b[2]!=0xFF) || (b[3]!=0xFF)))
{
    b[0]=b[1];
    b[1]=b[2];
    b[2]=b[3];
    b[3]=EEPROM.read(addr++);
}
// did we read until the end ??
if (addr >= StorageEnd) StorageMark=StorageBegin;
// we found 4 bytes that have not been written to
else StorageMark=addr-4;
// in either case we clear the storage
ClearStorage();
// mark the beginning
EEPROM.write(StorageMark,0);
EEPROM.write(StorageMark+1,0);
// set then beginning and end of the current segment
StorageIndex=StorageMark+2;
StorageEnd=StorageBackup;
// -----
// while(true); // stop here so we can check a memory dump
}
}

//-----
void Print_IdString()
{ // Serial.print(F("; ")); // prefix
  Serial.println(IdString); // print it
}

//-----
void PrintTrueFalse(byte T)
{ // used to report True or False for boolean Globals
  if(T == 0) Serial.println(F("False"));
  else Serial.println(F("True"));
}

//-----
void ReportStatus()
{ // report settings
  PrintSeperatorLine();
  Serial.print (F("; Report Mode: "));
  PrintTrueFalse (ReportMode);
  Serial.print (F("; Debugging Active: "));
  PrintTrueFalse (DeBug);
  Serial.print (F("; Report Raw Reading: "));

  PrintTrueFalse (RtnRawRead);
  Serial.print (F("; Report Fahrenheit: "));
  PrintTrueFalse (RtnFahrenh);
  Serial.print (F("; Report Celsius: "));
  PrintTrueFalse (RtnCelsius);
  Serial.print (F("; Minutes Bewteen: "));
  Serial.println (MinuteTarget, DEC);

  // single sensor, no prefix needed
  Serial.print (F("; Sensor ID/Location: "));
  Print_IdString ();
  Serial.print (F("; Raw Offset: "));
  Serial.println (CovrtOffsetR, DEC);
  Serial.print (F("; Celsius Factor: "));

```

```

Serial.println (Covrt2Celsius, 4);
Serial.print (F("; Fahrenheit Factor: "));
Serial.println (Covrt2Fahrenheit, 4);

if (newflg != 0)
  Serial.println (F("; Current parameters have *NOT* been written to EEPROM."));
if(EEmodeFlagTF())
  Serial.println (F("; *** Next run will write Data to EEPROM ***"));

PrintSeperatorLine();

}

//-----
void avrRawTemp()
{ /* each sample has 16 ADC reads for a 12 bit virtual ADC */
  /* REF: Atmel document number AVR121.pdf */
  //-----
  /* on 16Mhz ATmega328 512 samples requires just under 1 second */
  /* 16 samples (16*16=256) gives fairly consistent results */
  /* on a steady-state system in under 40K microseconds */
  //-----
  unsigned long RawSum=0; // used to sum samples for averaging
  word RawTemp=0; // used to accumulate 10 bit ADC readings
  word test=0; // used to count samples
  byte k; // counter for 10 bit ADC reads

  // turn on internal reference, right-shift ADC buffer, ADC channel = internal temp sensor
  ADMUX = 0xC8;
  delay(5); // wait a bit for the analog ref to stabilize

  while (test++ < 64 ) // oversampling loop (for averaging)
  { for (k=0; k<16; k++) // virtual ADC loop,
    // 16 consecutive readings
    { ADCSRA |= _BV(ADSC); // start the conversion
      while (bit_is_set(ADCSRA, ADSC)); // ADSC is cleared when the
      // conversion finishes
      RawTemp += (ADCL | (ADCH << 8)); // accumulate the reading (low byte first)
    }
    RawSum += (RawTemp >>2); // accumulate virtual 12 bit ADC value
    RawTemp=0; // zero ADC accumulator for
    // the next sequence
  }
  // Serial.println ("Got here: avrRawTemp");
  Accumulator += ((RawSum)>>6); // average by shifting bit position,
  // LSBs are lost
  CycleCount++; // used by functions to average readings
}

//-----
void Convert(word RawReading)
{ //converts Raw Reading to Celsius and Fahrenheit

  // we need to use an integer for the delta to allow for negative values
  int Delta;
  byte fraction;

  // get the difference between the current reading and the raw reading offset
  Delta=RawReading-CovrtOffsetR;

  //Serial.println (F("; got here: Convert"));
  //Serial.println (RawReading);
  //Serial.println (CovrtOffsetR);
  //Serial.println (Delta);
  //Serial.println (Covrt2Celsius);

```

```

//Serial.println (CovrtOffsetC);

Celsius= ((Delta * Covrt2Celsius) + CovrtOffsetC)+40;
// now find closest 1/4 degree
fraction=100*(Celsius-word(Celsius));
if      (fraction > 87) fraction=4;
else if (fraction > 62) fraction=3;
else if (fraction > 37) fraction=2;
else if (fraction > 12) fraction=1;
else      fraction=0;
// the extra 40 degrees had to be inserted to fix a problem
// where conversion below zero rolled the value for
// word(Celsius) to positive number
Celsius=(word(Celsius)+(float(fraction)*0.25))-40;

Fahrenheit= (Delta * Covrt2Fahrenheit) + CovrtOffsetF;
// now find closest 1/2 degree
fraction=100*(Fahrenheit-word(Fahrenheit));
if      (fraction > 75) fraction=2;
else if (fraction > 25) fraction=1;
else      fraction=0;
Fahrenheit=word(Fahrenheit)+(float(fraction)*0.5);
}

//-----
void Report()
{ word AvgSumRead;
  AvgSumRead = Accumalator/CycleCount;
  if (EepromMode == true) Report2EEPROM(AvgSumRead);
  else if (ReportMode == true)
  { Convert (AvgSumRead);
    if (RtnRawRead == true)
    { Serial.print (AvgSumRead);
      Serial.print (char(9));
    }
    if (RtnCelsius == true)
    { Serial.print (Celsius,2);
      Serial.print (char(9));
    }
    if (RtnFahrenh == true)
    { Serial.print (Fahrenheit,2);
      Serial.print (char(9));
    }
    if (DeBug == true)
    { Serial.print (CycleTime/CycleCount);
      Serial.print (char(9));
      Serial.print (CycleCount);
      Serial.print (char(9));
      Serial.print (millis()-RptStartTime);
      RptStartTime=millis();
    }
    Serial.println();
  }
  Accumalator   = 0;
  CycleCount    = 0;
  CycleTime     = 0;
  LastRead=AvgSumRead;
}

//-----
void QuickBlink()
{ // on the UNO 1 mullisecond will surface
  // adjusted up to 3 for Nano
  digitalWrite(13, HIGH);    // turn on LED

```

```

    delay(3);
    digitalWrite(13, LOW);    // turn off LED
}

//-----
void Report2EEPROM(word AvgSumRead)
{ // We are implimenting both data compression and wearleveling.
  // Our data is only 12 bits. Becuase we should never get a reading
  // over 2047 in our high bit will always be zero.
  // We are going to use the top four bits to count consecutive equal
  // readings. In that manner we may be able to store 16 readings in
  // a singal word value.
  word makeword;
  QuickBlink();

  // Serial.println(F("Get here: Report2EEPROM"));
  // send this string for testing: EC EE ST !! !!
  // we need to skip the firs pass because we have nothing to work with
  if (LastRead !=0)
  { if (LastRead == AvgSumRead) Consecutive++;
    if ((Consecutive == 15) || (LastRead != AvgSumRead))
    { // Serial.println(F("Get here: Report2EEPROM, write record"));
      // we are going to try two blinks everytime that there is a write
      makeword = (Consecutive <<12)+LastRead;
      EEPROM.write (StorageIndex++, highByte(makeword));
      EEPROM.write (StorageIndex++, lowByte(makeword));
      Consecutive=0;
      // now we need to check our storage space
      if ((StorageEnd-StorageIndex)<2)
      { // folks there is Trouble in river city !
        if (StorageMark==StorageBegin)
          {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        if (StorageEnd ==StorageMark )
          {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        // OK, start at the beginning ....
        StorageIndex=StorageBegin;
        StorageEnd=StorageMark;
      }
      delay (50);    // force a bit of a delay so both blinks can be seen
      QuickBlink();
    }
  }
  LastRead == AvgSumRead;
}

//-----
void DumpStorage()
{ // print data stored in eeprom
  byte b1, b2, c;
  word reading;
  word countreading=0;
  word countwords=0;
  boolean savemode;
  // save the current reporting mode
  savemode=ReportMode;
  // find the beginning of the data defined to be two zero bytes
  b1=1;
  b2=1;
  while ((StorageIndex<StorageBackup) && ((b1!=0) || (b2!=0)))
  { b1=b2;
    b2=EEPROM.read(StorageIndex++);
  }
  StorageEnd=StorageBackup;
  StorageMark=StorageIndex-2;
  // 4 high bits are the count, low 12 bits are the reading

```

```

Serial.println ("; Begin EEPROM data dump -----");
Serial.print ("Raw Reading");
Serial.print (char(9));
Serial.print ("Celsius");
Serial.print (char(9));
Serial.print ("Fahrenheit");
Serial.println ();
while (((StorageEnd-StorageIndex)>=2) && ((b1 != 0xFF)||(b2 != 0xFF)))
{ b1=EEPROM.read(StorageIndex++);
  b2=EEPROM.read(StorageIndex++);
  //----- debuggin code
  // Serial.print ("; Location: ");
  // Serial.print (StorageIndex);
  // Serial.print (" , ");
  // Serial.print (b1,HEX);
  // Serial.print (" , ");
  // Serial.print (b2,HEX);
  countwords++;
  // two bytes of FFh will mark the end
  if ((b1 != 0xFF) || (b2 != 0xFF))
  { Consecutive=b1>>4;
    reading= ((b1 & B00001111)<<8)+b2;
    Convert(reading);
    //----- debuggin code
    // Serial.print (" , ");
    // Serial.print (Consecutive);
    // Serial.print (" , ");
    // Serial.print (reading);
    // Serial.println();
    // while (Serial.available() ==0);
    // c=Serial.read();
    // the logic here is we need to print every reading at least once ...
    // that is when it is zero. When we subtract one from zero we get 255
    while (Consecutive<255)
    { countreading++;
      Serial.print (reading);
      Serial.print (char(9));
      Serial.print (Celsius,2);
      Serial.print (char(9));
      Serial.print (Fahrenheit,2);
      Serial.println ();
      Consecutive--;
      //----- debuggin code
      // Serial.print (Consecutive);
      // Serial.print (" , ");
      // Serial.println();
      // while (Serial.available() ==0);
      // c=Serial.read();
    }
    // now check the addresses
    if ((StorageEnd-StorageIndex)<2)
    { if (StorageMark != StorageBegin)
      { StorageEnd = StorageMark;
        StorageMark = StorageBegin;
        StorageIndex = StorageMark +2;
      }
    }
  }
}
Serial.println (F("; End EEPROM data dump -----"));
Serial.print (F("; readings: "));
Serial.println (countreading, DEC);
Serial.print (F("; storage words: "));
Serial.println (countwords, DEC);
PrintSeperatorLine();

```

```

// restore the current reporting mode
ReportMode=savemode;
}

//-----
void PrintOKStr ()
{ // command was accepted and processed
  // this just serves to reduce command response memory usage a bit
  if (EepromMode == false)
  { // we do not want to get hung up
    // trying to write to something that is not connected
    Serial.print (F("; "));
    Serial.print (cmd);
    Serial.println (F(" OK"));
  }
}

//-----
void PrintNotRecognized()
{ // command was Not Recognized
  // this just serves to reduce command response memory usage a bit
  if (EepromMode == false)
  { // we do not want to get hung up
    // trying to write to something that is not connected
    Serial.print (F("; "));
    Serial.print (cmd);
    Serial.println (F(" ??"));
  }
}

//-----
void PrintNotImplemented()
{ // command was Not Recognized
  // this just serves to reduce command response memory usage a bit
  if (EepromMode == false)
  { Serial.print (F("; "));
    Serial.print (cmd);
    Serial.println (F(" XX"));
  }
}

//-----
void ShutDown()
{ // Note that no provision is made to wake up.
  // This is as close to shutdown as we can get.
  // Because of the inefficient voltage regulator this
  // mode still draws a lot of power (about 10mA).
  // A standard 9 volt battery may last about 16 hours.

  // Serial.println(prevcmd);
  if ((prevcmd[0]=='S') && (prevcmd[1]=='S'))
  { Serial.println (F("; SHUTDOWN"));
    // give device time to send string
    for (byte i=0; i< 25; i++)
    { QuickBlink();
      delay (100);
    }
    cbi(ADCSRA, ADEN); // bit 7 of ADCSRA, disable ADC
    SleepMode = true;
    noInterrupts();
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_enable();
    sleep_mode(); // all execution should stop here
    while(0==0); // endless loop (belts and suspenders)
  }
}

```

```

    else PrintOKStr();      // first time through only
}

//-----
void software_Reset()
{ // Restarts program from beginning but
  // does not reset the peripherals and registers
  // as we are not doing anything with the the
  // timers or peripherals or registers this
  // should be adequate (will not support updating)

  // Serial.println(prevcmd);
  if ((prevcmd[0]=='!') && (prevcmd[1]=='!'))
  { Serial.println(F("; RESETTING"));
    // give device time to send string
    delay(1000);
    asm volatile (" jmp 0");
  }
  else PrintOKStr();      // first time through only
}

//-----
void SetRawReadMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnRawRead = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnRawRead = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetFahrenheitMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnFahrenh = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnFahrenh = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetCelsiusMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnCelsius = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnCelsius = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetReportMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {ReportMode = true; PrintOKStr();}
  else if (cmd[1]=='F') {ReportMode = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void ToggleDebugMode()
{ // toggle Debug mode
  if (Debug == true) Debug = false;
  else if (Debug == false) Debug = true;
  PrintOKStr();
}

//-----
void NewReportTime()
{ // set report Minutes
  if (cmd[1]=='1') { MinuteTarget = 1; Report_Reset();}
}

```



```

else if (cmd[1]=='2') { MinuteTarget = 2; Report_Reset();}
else if (cmd[1]=='3') { MinuteTarget = 3; Report_Reset();}
else if (cmd[1]=='4') { MinuteTarget = 4; Report_Reset();}
else if (cmd[1]=='5') { MinuteTarget = 5; Report_Reset();}
//---- the timings below have not been tested -----
else if (cmd[1]=='6') { MinuteTarget = 10; Report_Reset();}
else if (cmd[1]=='7') { MinuteTarget = 15; Report_Reset();}
else if (cmd[1]=='8') { MinuteTarget = 20; Report_Reset();}
else if (cmd[1]=='9') { MinuteTarget = 30; Report_Reset();}
else if (cmd[1]=='0') { MinuteTarget = 60; Report_Reset();}
else if (cmd[1]=='A') { MinuteTarget = 120; Report_Reset();}
else if (cmd[1]=='B') { MinuteTarget = 240; Report_Reset();}
else if (cmd[1]=='C') { MinuteTarget = 360; Report_Reset();}
else if (cmd[1]=='D') { MinuteTarget = 480; Report_Reset();}
else if (cmd[1]=='E') { MinuteTarget = 720; Report_Reset();}
else if (cmd[1]=='F') { MinuteTarget = 1440; Report_Reset();}
// max=86,400,000 milliseconds and that is why we use four byte variables

else if (cmd[1]=='T') Serial.println(F("; TT XX")); // not implimented
else PrintNotRecognized(); // not recognized
}

//-----
void Report_Reset()
{ // this force the current data to be reported
  // and reset our clock using the new time
  unsigned long SaveMe=SecondsTarget;
  PrintOKStr();
  Serial.println (F("; Report Timing reset"));
  // calculate seconds between report lines
  // SecondsTarget=MinuteTarget*SecondsMinute;
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=long(MinuteTarget)*long(SecondsMinute);
  if (SecondsTarget != SaveMe) newflg = newflg | B00010000;
  // Serial.print (F("Got Here: report reset, milliseconds to wait= "));
  // Serial.println (SecondsTarget);

  Accumalator = 0; // reset report parameters
  CycleCount = 0;
  RptTrigger = millis() + SecondsTarget;
  RptStartTime= millis();
  // Serial.print (F("Got Here: report reset, trigger= "));
  // Serial.println (RptTrigger-millis());
}

//-----
void NewIdString()
{ // New Location ID String
  // Serial.println("got here: NewIdString");
  // set time out to 5 seconds
  unsigned long timelimit = millis() + (5000);
  boolean timeout=false;
  char c= -1;
  byte n= 0;
  while ((c != 0) && (c != 10) && (c != 9) && (c != 13) && (n<EEidsize) && (timeout==false))
  { delay(10);
    c = Serial.read();
    if (c > 31) IdString[n++]=c;
    // check for timeout
    if (millis()>timelimit) timeout=true;
  }
  if (timeout) Serial.println (F("; L: aborted due to timeout"));
  else
  { IdString[EEidsize]=0; // make certain last charater is null
    newflg = newflg | B00000001;
  }
}

```

```

        // Serial.println (IdString);
        PrintOKStr();
    }
    DrainCmdTermiantors();
}

//-----
void NewOffsetR()
{ // New Conversion Offset
    int tempfloat=0;
    delay (2000);
    tempfloat=Serial.parseFloat();
    if (tempfloat!=0)
    { CovrtOffsetR=tempfloat;
      newflg = newflg | B00000010;
      PrintOKStr();
    }
    else Serial.println (F("; 0: zero value not accepted"));
    DrainCmdTermiantors();
}

//-----
void CelsiusEquals()
{ // sets offset according to current reading and input Celsius
    float tempfloat=0;
    word deltaR=0;
    // Serial.println (F("Got Here: CelsiusEquals"));
    delay (2000);
    tempfloat=Serial.parseFloat();
    if (tempfloat != 0)
    { // get the current raw reading
      Serial.println (F("; Calculating new offset ... ."));
      deltaR = (tempfloat - CovrtOffsetC) / Covrt2Celsius;
      while (CycleCount<250) avrRawTemp();
      CovrtOffsetR = (Accumalator/CycleCount)-deltaR;
      newflg = newflg | B00000010;
      PrintOKStr();
    }
    else Serial.println (F("; C= zero value not accepted"));
    DrainCmdTermiantors();
}

//-----
void FahrenheitEquals()
{ // sets offset according to current reading and input Fahrenheit
    float tempfloat=0;
    float deltaR;
    // Serial.println (F("Got Here: FahrenheitEquals"));
    delay (2000);
    tempfloat=Serial.parseFloat();
    if (tempfloat != 0)
    { // get the current raw reading
      Serial.println (F("; Calculating new offset ... ."));
      deltaR = (tempfloat - CovrtOffsetF) / Covrt2Fahrenheit;
      while (CycleCount<250) avrRawTemp();
      CovrtOffsetR = (Accumalator/CycleCount)-deltaR;
      newflg = newflg | B00000010;
      PrintOKStr();
    }
    else Serial.println (F("; F: zero value not accepted"));
    DrainCmdTermiantors();
}

//-----
void NewCelsius()

```

```

{ // New Celsius Factor
  delay (2000);
  float tempfloat=0;
  // Serial.println ("Got Here: NewCelsius");
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0)
    { Covrt2Celsius=tempfloat;
      newflg = newflg | B00000100;
      // Calculate Fahrenheit factor
      Covrt2Fahrenheit=Covrt2Celsius * 1.8000;
      PrintOKStr();
    }
  else Serial.println (F("; C: zero value not accepted"));
  DrainCmdTermiantors();
}

//-----
void NewFahrenheit()
{ // New Fahrenheit constant
  delay (2000);
  float tempfloat=0;
  // Serial.println ("Got Here: NewFahrenheit");
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0)
    { Covrt2Fahrenheit=tempfloat;
      newflg = newflg | B00000100;
      // Calculate Celsius factor
      Covrt2Celsius = Covrt2Fahrenheit / 1.8000;
      PrintOKStr();
    }
  else Serial.println (F("; F= zero value not accepted"));
  DrainCmdTermiantors();
}

//-----
void RestoreFromBackup()
{ char TempString[EEwdsz];
  byte i;
  // read the backup copy
  for (i=0; i<EEwdsz; i++) TempString[i]=EEPROM.read(StorageBackup + i);
  // write working copy
  for (i=0; i<EEwdsz; i++) EEPROM.write(StorageWorking +i, TempString[i]);
  newflg=0;
  Read_Calibration_Data();
  Report_Reset();
}

//-----
void OverwriteBackup()
{ char TempString[EEwdsz];
  byte i;
  // read the working copy
  for (i=0; i<EEwdsz; i++)
    TempString[i]=EEPROM.read(StorageWorking + i);
  // write backup copy
  for (i=0; i<EEwdsz; i++) EEPROM.write(StorageBackup +i, TempString[i]);
  PrintOKStr();
}

//=====
void TestData1()
{ // This is NOT valid calibration data
  // These sets were picked for testing
  // so that one set look like the another set.
  char temp[]="(1)tst data, UNO ";

```

```

//.....1234567890123456
byte i;
Serial.println(F("; Test Data one being written to EEPROM"));
// clear the EEPROM report storage area
ClearStorage();
for (i=0; i< EEidsize; i++) IdString[i]=temp[i];
IdString[EEidsize]=0;
CovrtOffsetR=1500;
Covrt2Celsius =0.25000;
Covrt2Fahrenheit=Covrt2Celsius * 1.80000;
MinuteTarget=1;
newflg=0XFF;
Write_Calibration_Data();
}

//-----
void TestData2()
{ char temp[256];
//.....1234567890123456
byte i;
Serial.println(F("; Test Data two being written to EEPROM"));
// clear the EEPROM report storage area
ClearStorage();
for (i=0; i< EEidsize; i++) IdString[i]=temp[i];
IdString[EEidsize]=0;
CovrtOffsetR=1389;
Covrt2Celsius =0.2217;
Covrt2Fahrenheit=Covrt2Celsius * 1.80000;
MinuteTarget=1;
newflg=0XFF;
Write_Calibration_Data();
}

//-----
void CalibrationMode()
{ // used for calibration, reduces time between report lines to 5 seconds
// there should be about 40 samples per report which will still give a good average
Serial.println(F("; Entering 5 second calibration mode ..."));
SecondsMinute=5000;
MinuteTarget=1;
DeBug=false;
EepromMode=false;
ReportMode=true;
RtnRawRead = true;
RtnFahrenh = false;
RtnCelsius = false;
Report_Reset();
newflg=0;
}

//-----
void EepromDumpAll()
{ char buffer[60]; // allocate buffer
word addr=0; // set start address
PrintSeperatorLine();
Serial.println(F("; Dump all EEPROM in Hex and ASCII")); // inform the user
while (addr < E2END) // run until we reach the end
{ for (byte i=0; i<16; i++) // process 16 bytes at a time
{ buffer[i]=EEPROM.read(addr++); // read EEPROM
}
Serial.print ("; ");
Serial.println(formatRamDump(addr-16, buffer)); // print formatted string
}
PrintSeperatorLine();
PrintOKStr();
}

```

```
}  
// ----- end of thermometer functions code-----
```

Appendix: Thermometer One Program Code (Plan "B")

Thermometer One Main Program File

```
/* ThermometerOne, Plan "B" ATMEGA328 Version */
/* Release 1.0.0, October 2013, Public Domain */

#include <avr/sleep.h>           // needed for shutdown function
#include <EEPROM.h>              // needed for EEPROM read and write
#include <HexDecAsc.h>           // used for EEPROM dump All

// EEPROM Address Constants
const word EEmask = 0;          // 1 byte location of EEPROM storage mode mask
const word EEflag = 1;         // 1 byte location of EEPROM storage mode flag
const word EErefvolt= 2;       // 2 byte location of RefVoltage
const word EEOffset = 4;      // 2 byte location of DegreeOffset
const word EEminutes= 6;      // 2 byte location of Report Target Minutes
const word EEunused0= 8;      // 2 byte location -- reserved -- unused
const word EEunused1= 10;     // 2 byte location -- reserved -- unused
const word EEunused2= 12;     // 2 byte location -- reserved -- unused
const word EEunused3= 14;     // 2 byte location -- reserved -- unused
const word EEidtring= 16;     // ID string w/o termination size (16)
const word EEidsize = 16;     // 24 byte location of IdString
const word EEwdsz = EEidtring+EEidsz; // Working data storage size (32)
//-----
const word EEtable = 32;      // 40 word location of conversion table
const word EEtbsz = 80;      // Working data storage size (80)
//-----
const word StorageWorking=EEmask; // EEPROM start for working calibration data
// EEPROM start for backup copy of constants
// note we have to add 1 to the value
// Because addresses begin with zero not one
const word StorageBackup=((E2END-(EEwdsz))+1);

// EEPROM addresses variables // begin storage for report data
word StorageBegin =StorageWorking+EEwdsz+EEtbsz;
word StorageMark =StorageBegin; // marks start of current segment
word StorageEnd =StorageBackup; // marks end of current segment
word StorageIndex =StorageBegin; // index for next EEPROM write

// Conversion Factors/Calibration Data
float RefVoltage; // Analog Sensor Reference Voltage
const float CovrtFactorV= 8192;
float DegreeOffset; // Fudge factor to adjust output
const float CovrtFactor0= 1024;
char IdString[EEidsz+1]; // ID/Location string for this device
word MinuteTarget = 1; // Number of minutes between report lines
byte newflg=0; // used to indicate new factors in memory

float Celsius; // Last conversion to Celsius Temperature
float Fahrenheit; // Last conversion to Fahrenheit Temperature
float AvrCelsius; // Last AVR conversion to Celsius Temperature
float AVRFahrenheit; // Last AVR conversion to Fahrenheit Temperature

// Global operational mode Variables // set default operation modes
boolean ReportMode = true; // True = reporting, False = Command Mode
boolean RtnRawRead = true; // True = include Raw
boolean RtnCelsius = true; // True = include Celsius
boolean RtnFahrenh = true; // True = include Fahrenheit
boolean DeBug = false; // True = extended reporting for debugging
boolean RtnAvrRead = false; // True = include Avr internal Temperature line
boolean EepromMode = false; // True = write data to EEPROM on next run
boolean RoundMode = true; // True = round output to nearest 1/4 or 1/2
```

```

// Global work Variables
// word      SecondsMinute = 10000;      // There are 1000 milliseconds in a second
// word      SecondsMinute = 60000;      // --- to speed things up a bit for debugging
// added so calibration timing can be reduced
unsigned long SecondsTarget = 0;        // Number of seconds between report lines
unsigned long RptStartTime = 0;         // Time between report lines
byte gap = 0;                          // used to increase gap between reads
unsigned long RptTrigger = 0;           // Target Time for report
unsigned long CycleStart = 0;           // Target Time for report
unsigned long CycleTime = 0;            // Target Time for report

unsigned long Accumalator = 0;          // Accumalate temperature reads
unsigned long CycleCount = 0;           // Cycles per Report line
char cmd[] = {0,0,0};                  // used to store two character command
char prevcmd[] = {0,0,0};              // used to store previous two character command

word LastRead = 0;                     // Stores previous RawRead Average
byte Consecutive = 0;                  // used to count consective equal readings

//=====
void setup()
{
  char c;
  Serial.begin (9600);
  pinMode(13, OUTPUT);                  // so we can blink it later during writes
  EnableADC();                          // enables the ADC and set ADC clock factor
  delay (1000);                         // let serial library complete setup
  while (Serial.available()>0)          // drain any data from the serial buffer
    c=Serial.read();
  Read_Calibration_Data();              // read and set conversion factors from EEPROM
  Check_EEPROM();                      // see if we are writing to EEPROM vs Serial
  // calculate seconds between report lines
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=long(MinuteTarget)*long(SecondsMinute);
  //delay (5000);                       // allow PC 5 seconds to get setup
  if (EepromMode==false) ReportStatus(); // report default parameters
  Accumalator = 0;                      // set startup parameters
  CycleCount = 0;
  RptTrigger = millis() + SecondsTarget;
  RptStartTime= millis();
  //-----debugging stuff-----
  // Serial.println ("Got here");
  // while (true);
}

//-----
void loop()
{
  char c1, c2;
  word wtemp;

  // ---- This is where we check for command input
  if ((Serial.available()>2) && (EepromMode==false))
    { if(ReadTwoCharacters()) CmdProcessor();}

  // ---- This is where we collect our temperature data
  // gap is used to increase the amount of time between reading sampling the ADC.
  // This Insures that we will not miss any data transmitted on the serial port.
  if (gap++ == 9)
    { gap=0;
      // cycle times are only used if debugging is turned on
      if (DeBug == true) CycleStart= millis();
      ReadRawTempA1();
      if (DeBug == true) CycleTime = CycleTime+(millis()-CycleStart);
    }

  // ---- This is where we output the teperature data
  // The time required to read 64 samples is about 119-120 milliseconds. If we get

```

```

// within 150 milliseconds of the Report Trigger Time then we wait for it.
// With the these timing numbers there are 500 reads of
// 64 virtual 12 bit samples per minute.
// Added condition for millis exceeding report trigger (possible with long commands)
if (((RptTrigger-millis())< 150) || (millis())>RptTrigger))
{ // Serial.println ("Got here: RptTrigger ");
  while (millis() < RptTrigger);
  // We want the new trigger time set as close as possible to when the previous trigger
  // went off --- so we put ti first.
  RptTrigger= (millis() + (SecondsTarget));
  // Serial.print (F("Got Here: RptTrigger, milliseconds to wait= "));
  // Serial.println (SecondsTarget);
  Report();
}
}

//=====
void CmdProcessor()
{ // this function is the main command handler
  // not many comments because I think the code is obvious
  if (Debug == true)
  { Serial.print (F("; Command Processor "));
    DebugPrintCharacters (cmd[0],cmd[1]);
  }

  if ((cmd[0]=='C') && (cmd[1]=='=')) CelsiusEquals();
  else if ((cmd[0]=='D') && (cmd[1]=='B')) ToggleDebugMode();
  else if ((cmd[0]=='D') && (cmd[1]=='O')) NewDegreeOffset();
  else if ((cmd[0]=='D') && (cmd[1]=='0')) NewDegreeOffset(); // for typos
  else if ((cmd[0]=='E') && (cmd[1]=='+')) EModeFlagSet();
  else if ((cmd[0]=='E') && (cmd[1]=='C')) ClearStorage();
  else if ((cmd[0]=='E') && (cmd[1]=='D')) DumpStorage();
  else if ((cmd[0]=='E') && (cmd[1]=='-')) EModeFlagClear();
  else if ((cmd[0]=='F') && (cmd[1]=='=')) FahrenheitEquals();
  else if ((cmd[0]=='I') && (cmd[1]=='D')) {Serial.print(F("; ")); Print_IdString();}
  else if ((cmd[0]=='L') && (cmd[1]=='L')) HelpMe();
  else if ((cmd[0]=='L') && (cmd[1]==':')) NewIdString();
  else if ((cmd[0]=='R') && (cmd[1]=='V')) NewRefVolt();
  else if ((cmd[0]=='S') && (cmd[1]=='S')) ShutDown();
  else if ((cmd[0]=='S') && (cmd[1]=='T')) ReportStatus();
  else if ((cmd[0]=='W') && (cmd[1]=='W')) Write_Calibration_Data();
  else if ((cmd[0]=='W') && (cmd[1]=='+')) OverwriteBackup();
  else if ((cmd[0]=='W') && (cmd[1]=='-')) RestoreFromBackup();
  else if ((cmd[0]=='?') && (cmd[1]=='?')) HelpMe();
  else if ((cmd[0]=='!') && (cmd[1]=='!')) software_Reset();
  else if ((cmd[0]=='0') && (cmd[1]=='0')) ToggleRoundMode();

  else if (cmd[0]=='C') SetCelsiusMode();
  else if (cmd[0]=='F') SetFahrenheitMode();
  else if (cmd[0]=='I') SetAvrInternalMode();
  else if (cmd[0]=='P') SetReportMode();
  else if (cmd[0]=='R') SetRawReadMode();
  else if (cmd[0]=='T') NewReportTime();

  // example of commands not implemented
  else if ((cmd[0]=='A') && (cmd[1]==':')) PrintNotImplemented();
  else if ((cmd[0]=='S') && (cmd[1]==':')) PrintNotImplemented();

  // example of application specific command implemented
  // these two commands write test data to the EEPROM working storage
  else if ((cmd[0]=='Z') && (cmd[1]=='1')) TestData1();
  else if ((cmd[0]=='Z') && (cmd[1]=='2')) TestData2();
  // this command used for calibration, changes reporting to 5 seconds
  else if ((cmd[0]=='Z') && (cmd[1]=='Z')) CalibrationMode();
  // this command used to dump entire EEPROM to Serial Port
  else if ((cmd[0]=='Z') && (cmd[1]=='D')) EepromDumpAll();

```



```

else PrintNotRecognized();          // not recognized
// -----
// previously define commands not valid in this implimentation
// else if ((cmd[0]=='O') && (cmd[1]==':')) NewOffsetR();
// else if ((cmd[0]=='F') && (cmd[1]==':')) NewFahrenheit();
// else if ((cmd[0]=='C') && (cmd[1]==':')) NewCelsius();
// else if ((cmd[0]=='C') && (cmd[1]=='=')) CelsiusEquals();
// else if ((cmd[0]=='T') && (cmd[1]=='T')) TestTest();

}

//-----
//void TestTest()
// { for (byte i=0; i< 20; i++)
//   { QuickBlink();
//     delay (200);
//   }
// }

//-----
void HelpMe()
//Serial.println(F("This string will be stored in flash memory"));
{ PrintSeperatorLine();
  Serial.println(F(";\\t\\tArduino AtMega328 Temperature Sensor 1.0.0"));
  Serial.println(F(";\\tID\\tOutput ID string"));
  Serial.println(F(";\\tST\\tOutput Status"));
  Serial.println(F(";\\tRT\\tRaw=True"));
  Serial.println(F(";\\tRF\\tRaw=False"));
  Serial.println(F(";\\tFT\\tFahrenheit=True"));
  Serial.println(F(";\\tFF\\tFahrenheit=False"));
  Serial.println(F(";\\tF=\\tEnter Current Fahrenheit"));
  Serial.println(F(";\\tCT\\tCelsius=True"));
  Serial.println(F(";\\tCF\\tCelsius=False"));
  Serial.println(F(";\\tIT\\tAVR Internal Temperature=True"));
  Serial.println(F(";\\tIF\\tAVR Internal Temperature=False"));
  Serial.println(F(";\\tC=\\tEnter Current Celsius")); // to be done
  Serial.println(F(";\\tDO\\tNew Degree Offset (Fahrenheit)"));
  Serial.println(F(";\\tDF\\tSame as DO"));
  Serial.println(F(";\\tRV\\tNew Reference Voltage"));
  Serial.println(F(";\\tT1\\tReport time = 01 minutes"));
  Serial.println(F(";\\tT2\\tReport time = 02 minutes"));
  Serial.println(F(";\\tT3\\tReport time = 03 minutes"));
  Serial.println(F(";\\tT4\\tReport time = 04 minutes"));
  Serial.println(F(";\\tT5\\tReport time = 05 minutes"));
  Serial.println(F(";\\tT6\\tReport time = 10 minutes"));
  Serial.println(F(";\\tT7\\tReport time = 15 minutes"));
  Serial.println(F(";\\tT8\\tReport time = 20 minutes"));
  Serial.println(F(";\\tT9\\tReport time = 30 minutes"));
  Serial.println(F(";\\tT0\\tReport time = 60 minutes"));
  Serial.println(F(";\\tTA\\tReport time = 02 hours"));
  Serial.println(F(";\\tTB\\tReport time = 04 hours"));
  Serial.println(F(";\\tTC\\tReport time = 06 hours"));
  Serial.println(F(";\\tTD\\tReport time = 08 hours"));
  Serial.println(F(";\\tTE\\tReport time = 12 hours"));
  Serial.println(F(";\\tTF\\tReport time = 24 hours"));
  Serial.println(F(";\\tPF\\tPrint mode = False"));
  Serial.println(F(";\\tPT\\tPrint mode = True"));
  Serial.println(F(";\\tDB\\tDebug mode toggle"));
  Serial.println(F(";\\t00\\tRounding mode toggle"));
  Serial.println(F(";\\tL:\\tNew Location"));
  Serial.println(F(";\\tWW\\tWrite Calibration data to EEPROM"));
  Serial.println(F(";\\tW+\\tOverwrite Backup Calibration data"));
  Serial.println(F(";\\tW-\\tRestore from Backup Calibration data"));
  Serial.println(F(";\\tE+\\tSet Flag to send next run to EEPROM"));
  Serial.println(F(";\\tE-\\tClear Flag to send next run to EEPROM"));

```

```

Serial.println(F("\tEC\tClear EEPROM Storage"));
Serial.println(F("\tED\tDump data stored in EEPROM"));
Serial.println(F("\tLL\tList implemented commands"));
Serial.println(F("\t??\tList implemented commands"));
Serial.println(F("\tSS\tShutdown (send twice)"));
Serial.println(F("\t!!\tReset (send twice)"));
PrintSeperatorLine();
// example of application specific command implimneted
// these two commands write test data to the EEPROM working storage
Serial.println(F("\tZ1\tWrite test data 1"));
Serial.println(F("\tZ2\tWrite test data 2"));
// special calibration mode
Serial.println(F("\tZZ\tToggle 5 Second reporting"));
// Dump all EEPROM memory to Serial in Hex and ASCII
Serial.println(F("\tZD\tDump ALLL EEPROM to serial"));
PrintSeperatorLine();
Serial.println(F("\t\tResponse 'XX' = not implemented"));
Serial.println(F("\t\tResponse '??' = not recognized"));
PrintSeperatorLine();
}

//-----
void PrintSeperatorLine()
{ Serial.print(" ");
  for (byte i=0; i<36; i++) Serial.print('-');
  Serial.println();
}

//=====
boolean ReadTwoCharacters()
{ char c1=0,c2=0,c3=-1;
  byte m=0;
  boolean EOC=true; // End of Command Terminator
  boolean OurReturn=false;

  // It is not to be believed how much effort went into creating this simple function to read
  // two characters. I noted a bit of problem reading characters from the serail port when
  // the loop was too fast therefore I have added a bit of a delay to insure the serial port
  // library can keep up. Worst case senario this function can take more than 250 milliseconds.
  // Normally when this functionis called we expect the htree bytes we need to be in the buffer
  // but if there is noise on the line or a parrot randomly pecking at the keyboard it could
  // take a bit longer.
  //
  // by defintion we are looking for two characters followed by a terminator
  // we define a command terminatore to be a carriage return, new line or null character
  // --- for good measure we are including the tab character and space as well
  // space was added because it is impossible to send a tab character from the Ardunion IDE
  // we will accept any combination of those characters as a single terminator
  // we will accept the last two printable ASCII characters before a terminator for our command
  // we keep reading until we get a terminator, but we will only read for a short period

  // but before we do anything else we are going to save the rprevious command for posterity
  prevcmd[0]=cmd[0]; // actually we are saving it so that shutdown
  prevcmd[1]=cmd[1]; // and reset can check it before they execute

  while ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32) && (m<25))
  { // if we have a valid ASCII character for c3 then roll the charaters down
    if (Serial.available(>0))
    { c3=Serial.read();
      if (c3>32) {c1=c2; c2=c3;}
    }
    // we need a bit of a delay to let the serial interface catch up
    // after 25 empty reads we give up
    else { delay (10); m++;}
  }
}

```

```

// DebugPrintCharacters (c1,c2,c3,m);

// we are very liberal about what we will accept for a command terminator
// but we insist on having one.
if ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32)) EOC= false;
// we need to drain any remaining command terminator characters from the serial buffer
else DrainCmdTerminators();

// now check for valid ASCII characters and End of Line
if ((c1>32) && (c2>32) && EOC)
{ // OK... we have something to work with
  // Convert lower case to UPPER case except "w"
  // DebugPrintCharacters (c1,c2,c3);
  if ((c1 != 'w') && (c1 >96) && (c1 <123)) c1 = (c1 -32);
  if ((c2 != 'w') && (c2 >96) && (c2 <123)) c2 = (c2 -32);
  // DebugPrintCharacters (c1,c2);
  cmd[0]=c1;
  cmd[1]=c2;
  OurReturn=true;
}
// whatever it was that was sent did not meet our criteria
// inform the parrot that he or she must do better
else Serial.println(F("; ?? ??"));
return OurReturn;
}

//-----
void DrainCmdTerminators()
{ char c3=0;
  // removed leading command terminators from serial buffer
  delay (10); c3=Serial.peek();
  while ((c3==13) || (c3==10) || (c3==9) || (c3==0) || (c3 == 32))
  { c3=Serial.read();
    delay (10);
    c3=Serial.peek();
  }
  // c3 should at this point should be -1 unless there are more commands/characters in the
  buffer
}

//=====
// overloaded debugging function for debugging the above input routine
void DebugPrintCharacters (char c1, char c2, char c3, byte m)
{
  if (DeBug == true)
  { Serial.print ("Received: ");
    Serial.print (c1);
    // Serial.print (" ");
    Serial.print (c2);
    Serial.print (" ");
    if (c3 != 0)
    { Serial.print (c3, DEC);
      Serial.print (" ");
    }
    if (m != 0) Serial.print (m, DEC);
    Serial.println ();
  }
}

void DebugPrintCharacters (char c1, char c2, char c3)
{ byte m=0;
  DebugPrintCharacters (c1,c2,c3,m);
}

void DebugPrintCharacters (char c1, char c2)
{ byte m=0;
  char c3=0;

```

```

    DebugPrintCharacters (c1,c2,c3,m);
}

//-----End of Main File-----

```

Thermometer One Functions Module

```

// cbi and sbi are standard (AVR) methods for setting,
// or clearing, bits in PORT (and other) variables.
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

//-----
void EnableADC()
{ // This is probably not needed but ....
  // set system clock divisor to 128
  // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
  sbi(ADCSRA, ADPS2);          // bit 2 of ADCSRA, system clock divisor
  sbi(ADCSRA, ADPS1);          // bit 1 of ADCSRA, system clock divisor
  sbi(ADCSRA, ADPS0);          // bit 0 of ADCSRA, system clock divisor
  cbi(ADCSRA, ADSCF);          // bit 5 of ADCSRA, disable auto trigger mode
  sbi(ADCSRA, ADEN);           // bit 7 of ADCSRA, enable ADC
}

//-----
void Read_Calibration_Data()
{ // This is more or less taken straight from the
  // EEPROM_TempSensor_Calibration_Constants program
  byte i=0, j=0;
  word TempWord;
  char c;
  char sign;

  // detect a virgin device --- well at least try
  if ((EEPROM.read(EEminutes)==0xFF) &&
      (EEPROM.read(EEminutes+1)==0xFF))
    TestData1();

  // Get Degree Offset
  TempWord= EEPROM.read(EEoffset)<<8;
  TempWord= (TempWord + EEPROM.read(EEoffset +1));
  // Serial.print (F("Got Here: Degree Offset EEPROM word: "));
  // Serial.println (TempWord, HEX);
  // get the sign bit
  sign= 1;
  // Serial.print (F("High Nibble: "));
  // Serial.println (TempWord>>12,BIN);
  if ((TempWord>>12) == B1000) sign= -1;
  // Serial.print (F("Sign: "));
  // Serial.println (sign,DEC);
  // strip the sign bit
  TempWord=(TempWord<<1)>>1;
  // Serial.print (F("Degree Offset EEPROM word stripped: "));
  // Serial.println (TempWord, HEX);
  // now we need to convert it to a fraction
  DegreeOffset=(float(TempWord)/float(CovrtFactor0)) * sign;

  // Get the RefVoltage factor;
  TempWord= EEPROM.read(EErefvolt)<<8;
  TempWord= TempWord + EEPROM.read(EErefvolt +1);
  // now we need to convert it to a fraction

```

```

RefVoltage = float(TempWord)/CovrtFactorV ;

// ID String -----
i=0; c=1;
while (c!=0,i< EEidsz)
{c=EEPROM.read(EEidtring + i);      // read the ID string
 IdString [i++]=c;
}
IdString [EEidsz]=0;                // just in case

// Set MinuteTarget from default minutes
MinuteTarget=(EEPROM.read(EEminutes)<<8) + EEPROM.read(EEminutes +1);
if (MinuteTarget<1)MinuteTarget=1;

// this is a bit flag to indicate when the current constants
// in memory are different from those stored in working storage
newflg=0;

}

//-----
void Write_Calibration_Data()
{ word i=0;
  // float saveflt;
  word TempWord;
  char c=-1;
  byte sign;

  // New location
  if ((newflg & B00000001) == B00000001)
  { // Serial.println ("Got Here: Write_Calibration_Data, Location");
    for (i=0;i<EEidsz; i++) {EEPROM.write((EEidtring +i), IdString[i]);}
  }

  // New Degree Offset
  if ((newflg & B00000010) == B00000010)
  { // Serial.println (F("Got Here: Write_Calibration_Data, Offset"));
    // Serial.print  (F("Degree Offset: "));
    // Serial.println  (DegreeOffset);
    sign = 0;
    if (DegreeOffset <0) sign = B10000000;
    TempWord=word(abs(DegreeOffset) * CovrtFactor0);
    // Serial.print  (F("Degree Offset as word: "));
    // Serial.println  (TempWord);
    // Serial.print  (F("Sign: "));
    // Serial.println  (sign);
    EEPROM.write(EEoffset , highByte(TempWord)|sign);
    EEPROM.write(EEoffset +1, lowByte (TempWord));
  }

  // New Reference Voltage
  if ((newflg & B00000100) == B00000100)
  { //Serial.println ("Got Here: Write_Calibration_Data, Reference Voltage");
    TempWord = word(RefVoltage * CovrtFactorV);
    EEPROM.write(EErefvolt , highByte(TempWord));
    EEPROM.write(EErefvolt +1, lowByte (TempWord));
    // Note:
    // When the EEPROM data is read
    // Covrt2Fahrenheit is calculated from RefVoltage
  }

  // New MinuteTarget
  if ((newflg & B00010000) == B00010000)
  { // Serial.println ("Got Here: Write_Calibration_Data, MinuteTarget");
    EEPROM.write(EEminutes , highByte(MinuteTarget));
    EEPROM.write(EEminutes +1, lowByte (MinuteTarget));
  }
}

```

```

    }

    // write the unused word(s)
    i=EEunused0;
    while (i<EEidtring)
        {if (EEPROM.read(i)!= 0xFF) EEPROM.write(i, 0xFF);
        i++;
        }

    // clear the EEMODE flag
    EEmodeFlagClear();

    // Serial.println ("Got Here: Write_Calibration_Data, Read");
    // note newflg is reset by Read_Calibration_Data
    Read_Calibration_Data();
    // PrintOKStr(); is sent by Report_Reset
    Report_Reset();
}

//-----
void ClearStorage()
{ // this is used to clear/erase the EEPROM data storage (except for constant areas)
  word addr;
  byte b;
  for (addr=(StorageBegin); addr<StorageBackup; addr++)
    { if (EEPROM.read(addr) != 0xFF) EEPROM.write(addr,0xFF);
    } // note: each byte requires 6-8 machine cycles
  PrintOKStr ();
}

//-----
void EEmodeFlagSet()
{ // toggle the flag for the next run to write to EEPROM
  // does not affect current run
  // Here is the thing. We have setup wearleveling for our EEPROM data storage
  // but modeflag gets hit twice for every EEPROM run. So it will wearout
  // long before the bulk of the storage. We could just increment the byte and
  // look for odd or even values but that would continually toggle the low bit
  // wearing it out before there rest. As it turns out it is writing a zero to
  // a bit that wears then out. So we want to minimize the zero bit writes.
  // We are going to move a zero bit right to left. This extends our life by a
  // factor of eight. At that point you need to swap the backup and working
  // data locations by changing the EEPROM address locations and reprogramming
  // the Arduino. That would double the life (2 * 8 = 16).
  //
  byte flag;
  if(EEmodeFlagTF()==false)
    { // Serial.println (F("Got here: EEmodeFlagToggle, make not equal"));
      // make them not equal
      // shift left and add a one to the right
      flag=EEPROM.read(Eeflag);
      // Serial.println(flag);
      flag=(flag<<1)+1;
      // Serial.println(flag);
      // if we have all ones start over again at the right
      if (flag==B11111111) flag=B11111110;
      // Serial.println(flag);
      // now save it
      EEPROM.write(Eeflag,flag);
    }
  PrintOKStr();
}

//-----
void EEmodeFlagClear()

```

```

{ byte flag;
  if(EEmodeFlagTF())
  { // Serial.println (F("Got here: EEmodeFlagClear"));
    // make them equal
    flag=EEPROM.read(Eeflag);
    EEPROM.write(EEmask,flag);
  }
  if (EepromMode==false) PrintOKStr();
}

//-----
boolean EEmodeFlagTF()
{ // returns true if EEmodeFlag is set
  byte flag,mask;
  flag=EEPROM.read(Eeflag);
  mask=EEPROM.read(EEmask);
  if(flag==mask) return false;
  else return true;
}

//-----
void Check_EEPROM()
{ byte b[4],i;
  word w;
  word addr;

  EepromMode=false;
  if (EEmodeFlagTF())
  { // Serial.println ("got here: Check_EEPROM");
    EepromMode=true;
    // clear the flag
    EEmodeFlagClear();
    // disable serial reporting and debug mode
    ReportMode=false;
    DeBug=false;
    // we need to find the beginning of EEPROM that has not been used
    // we need at least four bytes to begin a new section.
    // so we need to find the first place where there are four bytes with FFh
    // zero our test pattern
    for (i=0; i<4; i++) b[i]=0;
    addr=StorageBegin;
    while ((addr<StorageBackup) && ((b[0]!=0xFF) || (b[1]!=0xFF)
                                     || (b[2]!=0xFF) || (b[3]!=0xFF)))
    { b[0]=b[1];
      b[1]=b[2];
      b[2]=b[3];
      b[3]=EEPROM.read(addr++);
    }
    // did we read until the end ??
    if (addr >= StorageEnd) StorageMark=StorageBegin;
    // we found 4 bytes that have not been written to
    else StorageMark=addr-4;
    // in either case we clear the storage
    ClearStorage();
    // mark the beginning
    EEPROM.write(StorageMark,0);
    EEPROM.write(StorageMark+1,0);
    // set then beginning and end of the current segment
    StorageIndex=StorageMark+2;
    StorageEnd=StorageBackup;
    // definitive notice of mode
    for (i=0; i < 30; i++) {QuickBlink(); delay (250);}
    // -----
    // while(true); // stop here so we can check a memory dump
  }
}

```

```

}

//-----
void Print_IdString()
{ // Serial.print(F("; "));          // pefix
  Serial.println(IdString);          // print it
}

//-----
void PrintTrueFalse(byte T)
{ // used to report True or False for boolean Globals
  if(T == 0) Serial.println(F("False"));
  else Serial.println(F("True"));
}

//-----
void ReportStatus()
{ // report settings
  PrintSeperatorLine();
  Serial.print (F("; Report:\t"));
  PrintTrueFalse (ReportMode);
  Serial.print (F("; Debug:\t"));
  PrintTrueFalse (DeBug);
  Serial.print (F("; Raw:\t"));
  PrintTrueFalse (RtnRawRead);
  Serial.print (F("; Fahrenheit:\t"));
  PrintTrueFalse (RtnFahrenh);
  Serial.print (F("; Celsius:\t"));
  PrintTrueFalse (RtnCelsius);
  Serial.print (F("; Avr:\t"));
  PrintTrueFalse (RtnAvrRead);
  Serial.print (F("; Round:\t"));
  PrintTrueFalse (RoundMode);

  Serial.print (F("; Minutes:\t"));
  Serial.println (MinuteTarget, DEC);
  Serial.print (F("; Voltage:\t"));
  Serial.println (RefVoltage, 4);

  // single sensor, no prefix needed
  Serial.print (F("; Sensor ID:\t"));
  Print_IdString ();
  Serial.print (F("; Offset:\t"));
  Serial.println (DegreeOffset, 4);

  if (newflg != 0)
    Serial.println (F(";Parameters not saved"));
  if(EEmodeFlagTF())
    Serial.println (F(";EEPROM Mode Flag Set"));

  PrintSeperatorLine();
}

//-----
// ---- See Below for the function the read the External LM34DZ Temperature Sensor ----
//-----
void AvrTemperature()
{ // Read AVR internal Temperature Sensor and Convert to Celsius and Degrees
  //
  unsigned long RawSum=0;          // used to sum samples for averaging
  word RawTemp=0;                  // used to accumulate 10 bit ADC readings
  word test=0;                     // used to count samples
  word RawReading;                 // averaged reading
  float RawVoltage;
  float DegreesKelvin;

```



```

// turn on internal reference, right-shift ADC buffer, ADC channel = Avr temperature
// B11000000: AMUX Voltage reference = Internal 1.1 volt      (bits 7,6)
// B00000000: AMUX Right Shift ADC Buffer                    (bit 5)
// B00001000: AMUX Input Source= internal temperature sensor (bits 3,2,1,0)
ADMUX = B11001000;
delay(10);                                // wait for the things to stabilize

while (test++ < 1024 )                    // oversampling loop (for averaging)
{ ADCSRA |= _BV(ADSC);                    // start the conversion
  while (bit_is_set(ADCSRA, ADSC));        // ADSC cleared when complete
  RawTemp = (ADCL | (ADCH << 8));          // collect the reading
  RawSum += RawTemp;                      // add it to out total
}
RawReading = (RawSum)>>10;                // averag the 1024 readings
//---- Convert It-----
// Convert to Raw Voltage
RawVoltage = (float(RawReading)/1024)* RefVoltage;
// Convert to Temperature in degrees Kelvin
DegreesKelvin= RawVoltage * 1000;
// Convert to Celsius
AvrCelsius= DegreesKelvin - 273;
// Convert to Fahrenheit
AVRFahrenheit=(AvrCelsius*1.8)+32;
//---- Print It-----
Serial.print ("");
Serial.print (RawReading);
Serial.print (char(9));
// Serial.print (RawVoltage);
// Serial.print (char(9));
// Serial.print (DegreesKelvin);
// Serial.print (char(9));
Serial.print (AvrCelsius);
Serial.print (char(9));
Serial.print (AVRFahrenheit);
Serial.print (char(9));
Serial.println (F("ARV"));

}

//-----
// ---- Below is the function the read the External LM34DZ Temperature Sensor ----
//-----

void ReadRawTempA1()
{ // Read ADC for Pin A1 (connected to LM34DZ Temperature Sensor)
  // Cycle time is aproximately 124 miliseconds
  unsigned long RawSum=0;                // used to sum samples for averaging
  word RawTemp=0;                        // used to accumalate 10 bit ADC readings
  word test=0;                          // used to count samples

  // Just in case ...
  // The INPUT mode explicitly disables the internal pullup resistors.
  pinMode(A1,INPUT);

  // turn on internal reference, right-shift ADC buffer, ADC channel = ADC1 (pin A1)
  // B11000000: AMUX Voltage reference = Internal 1.1 volt      (bits 7,6)
  // B00000000: AMUX Right Shift ADC Buffer                    (bit 5)
  // B00000001: AMUX Input Source= pin A1                    (bits 3,2,1,0)
  ADMUX = B11000001;
  delay(10);                                // wait for the things to stabilize

  while (test++ < 1024 )                    // oversampling loop (for averaging)
  { ADCSRA |= _BV(ADSC);                    // start the conversion
    while (bit_is_set(ADCSRA, ADSC));        // ADSC cleared when complete
    RawTemp = (ADCL | (ADCH << 8));          // collect the reading

```

```

        RawSum += RawTemp;                // add it to out total
    }
    Accumalator += ((RawSum)>>10);        // averag and add to Accumalator
                                           // LSBs are lost
    CycleCount++;                        // used by functions to average readings
}

//-----
void Convert(word RawReading)
{ // converts Raw Reading to Celsius and Fahrenheit
  // New plan: offset will only be used for minor correction
  // SCALE is actual voltage that is supposed to be 1.1 but reads 1.067
  // This temperature sensor reports in Fahrenheit 1 millivolt per degree
  // We need the correct voltage !!!
  // Covrt2Fahrenheit=1.067;
  Fahrenheit = (float(RawReading)/1024)* RefVoltage *100;
  Fahrenheit = Fahrenheit + DegreeOffset;
  if (RoundMode) Fahrenheit= nearesthalf(Fahrenheit);
  // Consistency is next to godliness.
  // We are working in Fahrenheit.
  // In our case Celsius is a function of Fahrenheit.
  // Thus we always complete our Fahrenheit calcs first.
  // That includes rounding.
  Celsius = (Fahrenheit-32)/1.8;
  if (RoundMode) Celsius=nearestquater(Celsius);
}

//-----
float nearestquater (float ValueIn)
{ // Return value rounded to nearest quater (0.25)
  byte sign=1;
  float fraction;
  if (ValueIn , 0)
  {
    sign = -1;
    ValueIn=abs(ValueIn);
  }
  fraction =ValueIn-long(ValueIn);
  if (fraction >= 0.875) fraction=1.00;
  else if (fraction >= 0.625) fraction=0.75;
  else if (fraction >= 0.375) fraction=0.50;
  else if (fraction >= 0.125) fraction=0.25;
  else fraction=0;
  // Serial.println (ValueIn);
  // Serial.println (ValueIn-long(ValueIn));
  // Serial.println (fraction);
  // Serial.println (sign);
  return (long(ValueIn)+fraction) * sign;
}

//-----
float nearesthalf (float ValueIn)
{ // Return value rounded to nearest half (0.50)
  byte sign=1;
  float fraction;
  if (ValueIn , 0)
  {
    sign = -1;
    ValueIn=abs(ValueIn);
  }
  fraction =ValueIn-long(ValueIn);
  if (fraction >= 0.750) fraction=1.00;
  else if (fraction >= 0.250) fraction=0.50;
  else fraction=0;
  return (long(ValueIn)+fraction) * sign;
}

```

```

}

//-----
void Report()
{ word AvgSumRead;
  AvgSumRead = Accumalator/CycleCount;
  if (EepromMode) Report2EEPROM(AvgSumRead);
  else if (ReportMode)
  { Convert (AvgSumRead);
    Serial.print ('0');
    if (RtnRawRead)
    { Serial.print (AvgSumRead);
      Serial.print (char(9));
    }
    if (RtnCelsius)
    { Serial.print (Celsius,2);
      Serial.print (char(9));
    }
    if (RtnFahrenh)
    { Serial.print (Fahrenheit,2);
      Serial.print (char(9));
    }
    if (DeBug)
    { Serial.print (CycleTime/CycleCount);
      Serial.print (char(9));
      Serial.print (CycleCount);
      Serial.print (char(9));
      Serial.print (millis()-RptStartTime);
      RptStartTime=millis();
    }
    Serial.println();
    // addin for AVR internal temperature line
    if (RtnAvrRead) AvrTemperature();
  }
  Accumalator = 0;
  CycleCount = 0;
  CycleTime = 0;
  LastRead=AvgSumRead;
}

//-----
void QuickBlink()
{ // on the UNO 1 mullisecond will surface
  // adjusted up to 3 for Nano
  digitalWrite(13, HIGH);    // turn on LED
  delay(3);
  digitalWrite(13, LOW);     // turn off LED
}

//-----
void Report2EEPROM(word AvgSumRead)
{ // We are implimenting both data compression and wearleveling.
  // Our data is only 12 bits. Becuase we should never get a reading
  // over 2047 in our high bit will always be zero.
  // We are going to use the top four bits to count consecutive equal
  // readings. In that manner we may be able to store 16 readings in
  // a singal word value.
  word makeword;
  QuickBlink();

  // Serial.println(F("Get here: Report2EEPROM"));
  // send this string for testing: EC EE ST !! !!
  // we need to skip the firs pass because we have nothing to work with
  if (LastRead !=0)
  { if (LastRead == AvgSumRead) Consecutive++;

```

```

    if ((Consecutive == 15) || (LastRead != AvgSumRead))
    { // Serial.println(F("Get here: Report2EEPROM, write record"));
      // we are going to try two blinks everytime that there is a write
      makeword = (Consecutive <<12)+LastRead;
      EEPROM.write (StorageIndex++, highByte(makeword));
      EEPROM.write (StorageIndex++, lowByte(makeword));
      Consecutive=0;
      // now we need to check our storage space
      if ((StorageEnd-StorageIndex)<2)
      { // folks there is Trouble in river city !
        if (StorageMark==StorageBegin)
        {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        if (StorageEnd ==StorageMark )
        {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        // OK, start at the beginning ....
        StorageIndex=StorageBegin;
        StorageEnd=StorageMark;
      }
      delay (50);      // force a bit of a delay so both blinks can be seen
      QuickBlink();
    }
  }
  LastRead == AvgSumRead;
}

```

//-----

void DumpStorage()

```

{ // print data stored in eeprom
  byte b1, b2, c;
  word reading;
  word countreading=0;
  word countwords=0;
  boolean savemode;
  // save the current reporting mode
  savemode=ReportMode;
  // find the beginning of the data defined to be two zero bytes
  b1=1;
  b2=1;
  while ((StorageIndex<StorageBackup) && ((b1!=0) || (b2!=0)))
  { b1=b2;
    b2=EEPROM.read(StorageIndex++);
  }
  StorageEnd=StorageBackup;
  StorageMark=StorageIndex-2;
  // 4 high bits are the count, low 12 bits are the reading
  PrintSeperatorLine();
  Serial.println (F("; Begin EEPROM data dump"));
  Serial.println (F("; Raw Reading\tCelsius\tFahrenheit"));
  while (((StorageEnd-StorageIndex)>=2) && ((b1 != 0xFF)||(b2 != 0xFF)))
  { b1=EEPROM.read(StorageIndex++);
    b2=EEPROM.read(StorageIndex++);
    //----- debuggin code
    // Serial.print  ("; Location: ");
    // Serial.print  (StorageIndex);
    // Serial.print  (" , ");
    // Serial.print  (b1,HEX);
    // Serial.print  (" , ");
    // Serial.print  (b2,HEX);
    countwords++;
    // two bytes of FFh will mark the end
    if ((b1 != 0xFF) || (b1 != 0xFF))
    { Consecutive=b1>>4;
      reading= ((b1 & B00001111)<<8)+b2;
      Convert(reading);
    }
    //----- debuggin code
  }
}

```

```

        // Serial.print  (" ");
        // Serial.print  (Consecutive);
        // Serial.print  (" ");
        // Serial.print  (reading);
        // Serial.println();
        // while (Serial.available() ==0);
        // c=Serial.read();
        // the logic here is we need to print every reading at least once ...
        // that is when it is zero. When we subtract one from zero we get 255
        while (Consecutive<255)
        { countreading++;
          Serial.print  (reading);
          Serial.print  (char(9));
          Serial.print  (Celsius,2);
          Serial.print  (char(9));
          Serial.print  (Fahrenheit,2);
          Serial.println ();
          Consecutive--;
          //----- debuggin code
          // Serial.print  (Consecutive);
          // Serial.print  (" ");
          // Serial.println();
          // while (Serial.available() ==0);
          // c=Serial.read();
        }
        // now check the addresses
        if ((StorageEnd-StorageIndex)<2)
        { if (StorageMark != StorageBegin)
          { StorageEnd = StorageMark;
            StorageMark = StorageBegin;
            StorageIndex = StorageMark +2;
          }
        }
      }
    }
    PrintSeperatorLine();
    Serial.println (F("; End EEPROM data dump"));
    Serial.print  (F("; Readings:\t"));
    Serial.println  (countreading, DEC);
    Serial.print  (F(";Storage Words:\t"));
    Serial.println  (countwords, DEC);
    PrintSeperatorLine();
    // restore the current reporting mode
    ReportMode=savemode;
  }

//-----
void Response (char str[])
{if (EepromMode == false)
  { // we do not want to get hung up
    // this just serves to reduce command response memory usage a bit
    // trying to wrtie to seomthing that is not connected
    Serial.print (F("; "));
    Serial.print (cmd);
    Serial.print (F(" "));
    Serial.println (str);
  }
}

//-----
void PrintOKStr ()
{ // command was accepted and processed
  // this just serves to reduce command response memory usage a bit
  Response ("OK");
}

```

```

//-----
void PrintNotRecognized()
{ // command was Not Recognized
  // this just serves to reduce command response memory usage a bit
  Response ("??");
}

//-----
void PrintNotImplemented()
{ // command was Not Recognized
  Response ("XX");
}

//-----
void ShutDown()
{ // Note that no provision is made to wake up.
  // This is as close to shutdown as we can get.
  // Because of the inefficient voltage regulator this
  // mode still draws a lot of power (about 10mA).
  // A standard 9 volt battery may last about 16 hours.

  // Serial.println(prevcmd);
  if ((prevcmd[0]=='S') && (prevcmd[1]=='S'))
  { Serial.println(F("; SHUTDOWN"));
    // give device time to send string
    for (byte i=0; i< 25; i++)
    { QuickBlink();
      delay (100);
    }
    cbi(ADCSRA, ADEN); // bit 7 of ADCSRA, disable ADC
    noInterrupts();
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_enable();
    sleep_mode(); // all execution should stop here
    while(0==0); // endless loop (belts and suspenders)
  }
  else PrintOKStr(); // first time through only
}

//-----
void software_Reset()
{ // Restarts program from beginning but
  // does not reset the peripherals and registers
  // as we are not doing anything with the the
  // timers or peripherals or registers this
  // should be adequate (will not support updating)

  // Serial.println(prevcmd);
  if ((prevcmd[0]=='!') && (prevcmd[1]=='!'))
  { Serial.println(F("; RESETTING"));
    // give device time to send string
    delay (1000);
    asm volatile (" jmp 0");
  }
  else PrintOKStr(); // first time through only
}

//-----
void SetRawReadMode()
{ // check for "T" or "F", true or false
  if (cmd[1]=='T') {RtnRawRead = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnRawRead = false; PrintOKStr();}
  else PrintNotRecognized();
}

```

```

//-----
void SetCelsiusMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnCelsius = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnCelsius = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetFahrenheitdMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnFahrenh = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnFahrenh = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetReportMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {ReportMode = true; PrintOKStr();}
  else if (cmd[1]=='F') {ReportMode = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void ToggleDebugMode()
{ // toggle Debug mode
  if (Debug == true) Debug = false;
  else if (Debug == false) Debug = true;
  PrintOKStr();
}

//-----
void SetAvrInternalMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnAvrRead = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnAvrRead = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void ToggleRoundMode()
{ // check for "T" or "F", true of false
  if (RoundMode) {RoundMode = false; PrintOKStr();}
  else {RoundMode = true; PrintOKStr();}
}

//-----
void NewReportTime()
{ // set report Minutes
  if (cmd[1]=='1') { MinuteTarget = 1; Report_Reset();}
  else if (cmd[1]=='2') { MinuteTarget = 2; Report_Reset();}
  else if (cmd[1]=='3') { MinuteTarget = 3; Report_Reset();}
  else if (cmd[1]=='4') { MinuteTarget = 4; Report_Reset();}
  else if (cmd[1]=='5') { MinuteTarget = 5; Report_Reset();}
  //---- the timings below have not been tested -----
  else if (cmd[1]=='6') { MinuteTarget = 10; Report_Reset();}
  else if (cmd[1]=='7') { MinuteTarget = 15; Report_Reset();}
  else if (cmd[1]=='8') { MinuteTarget = 20; Report_Reset();}
  else if (cmd[1]=='9') { MinuteTarget = 30; Report_Reset();}
  else if (cmd[1]=='0') { MinuteTarget = 60; Report_Reset();}
  else if (cmd[1]=='A') { MinuteTarget = 120; Report_Reset();}
  else if (cmd[1]=='B') { MinuteTarget = 240; Report_Reset();}
  else if (cmd[1]=='C') { MinuteTarget = 360; Report_Reset();}
}

```

```

else if (cmd[1]=='D') { MinuteTarget = 480; Report_Reset();}
else if (cmd[1]=='E') { MinuteTarget = 720; Report_Reset();}
else if (cmd[1]=='F') { MinuteTarget = 1440; Report_Reset();}
// max=86,400,000 milliseconds and that is why we use four byte variables

else if (cmd[1]=='T') Serial.println(F("; TT XX")); // not implimented
else PrintNotRecognized(); // not recognized
}

//-----
void Report_Reset()
{ // this force the current data to be reported
  // and reset our clock using the new time
  unsigned long SaveMe=SecondsTarget;
  PrintOKStr();
  Serial.println (F("; Report Timing reset"));
  // calculate seconds between report lines
  // SecondsTarget=MinuteTarget*SecondsMinute;
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=(long(MinuteTarget)*long(SecondsMinute));

  if (SecondsTarget != SaveMe) newflg = newflg | B00010000;
  // Serial.print (F("Got Here: report reset, milliseconds to wait= "));
  // Serial.println (SecondsTarget);

  Accumalator = 0; // reset report parameters
  CycleCount = 0;
  RptTrigger = millis() + SecondsTarget;
  RptStartTime= millis();
  // Serial.print (F("Got Here: report reset, trigger= "));
  // Serial.println (RptTrigger-millis());
}

//-----
void NewIdString()
{ // New Location ID String
  // Serial.println("got here: NewIdString");
  // set time out to 5 seconds
  unsigned long timelimit = millis() + (5000);
  boolean timeout=false;
  char c= -1;
  byte n= 0;
  while ((c != 0) && (c != 10) && (c != 9) && (c != 13) && (n<EEidsize) && (timeout==false))
  { delay(10);
    c = Serial.read();
    if (c > 31) IdString[n++]=c;
    // check for timeout
    if (millis()>timelimit) timeout=true;
  }
  while (n<EEidsize) IdString[n++]=0;
  if (timeout) Serial.println (F("; L: aborted due to timeout"));
  else
  { IdString[EEidsize]=0; // make certain last charater is null
    newflg = newflg | B00000001;
    // Serial.println (IdString);
    PrintOKStr();
  }
  DrainCmdTermiantors();
}

//-----
void PrintDegreeOffsetEffect(float NewOffset)
{ // new offset must be in Degrees Fahrenheit
  boolean SaveRoundMode;
  SaveRoundMode=RoundMode;

```



```

    while (CycleCount<50) ReadRawTempA1();
    DegreeOffset=0;
    RoundMode=false;
    Convert(Accumalator/CycleCount);
    DegreeOffset=NewOffset;
    Serial.print (F("; Offset(F):\t"));
    Serial.println (DegreeOffset);
    Serial.print (F("; Fahrenheit:\t"));
    Serial.println (Fahrenheit);
    Serial.print (F("; Adjusted:\t"));
    Fahrenheit=Fahrenheit+NewOffset;
    if (SaveRoundMode) Fahrenheit=nearesthalf(Fahrenheit);
    Serial.println (Fahrenheit);
    //-----
    Serial.print (F("; Celsius:\t"));
    Serial.println (Celsius);
    Serial.print (F("; Adjusted:\t"));
    Celsius=Celsius+(DegreeOffset/1.8000);
    if (SaveRoundMode) Celsius=nearestquater(Celsius);
    Serial.println (Celsius);
    newflg = newflg | B00000010;
    RoundMode=SaveRoundMode;
    PrintOKStr();
}

//-----
void ValueNotAccepted()
{ Serial.print("; ");
  Serial.print(cmd);
  Serial.print(" invalid/no input");
}

//-----
void NewDegreeOffset()
{ // New Degree Offset
  float tempfloat=0;
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat!=0)
  { if (abs(tempfloat)<0.010) tempfloat = 0;
    PrintDegreeOffsetEffect(tempfloat);
    newflg = newflg | B00000010;
  }
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void CalculateDegreeOffset(float tempfloat)
{ // calculate a new degree offset, TempF is Temperature in degrees Fahrenheit
  // get the current raw reading
  boolean SaveRoundMode;

  Serial.println (F("; Calculating new Degree offset ... ..."));
  while (CycleCount<50) ReadRawTempA1();
  // set the current offset to zero so that it
  // does not affect the Conversion
  DegreeOffset=0;
  SaveRoundMode=RoundMode;
  RoundMode=false;
  Convert(Accumalator/CycleCount);
  RoundMode=SaveRoundMode;
  PrintDegreeOffsetEffect(tempfloat-Fahrenheit);
}

```

```

//-----
void FahrenheitEquals()
{ // sets offset according to current reading and input Fahrenheit
  float tempfloat=0;
  float deltaR;
  word RawReading;
  // Serial.println (F("Got Here: FahrenheitEquals"));
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0) CalculateDegreeOffset(tempfloat);
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void CelsiusEquals()
{ // sets offset according to current reading and input Fahrenheit
  float tempfloat=0;
  float deltaR;
  word RawReading;
  // Serial.println (F("Got Here: CelsiusEquals"));
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0) CalculateDegreeOffset((tempfloat*1.8)+32);
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void NewRefVolt()
{ // New Degree Offset
  float tempfloat=0;
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat!=0)
  { // Serial.println(tempfloat,4);
    // Serial.println(RefVoltage,4);
    RefVoltage=tempfloat;
    // Serial.println(RefVoltage,4);
    newflg = newflg | B00000100;
    PrintOKStr();
  }
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void RestoreFromBackup()
{ char TempString[EEwdsz];
  byte i;
  // read the backup copy
  for (i=0; i<EEwdsz; i++) TempString[i]=EEPROM.read(StorageBackup + i);
  // write working copy
  for (i=0; i<EEwdsz; i++) EEPROM.write(StorageWorking +i, TempString[i]);
  newflg=0;
  Read_Calibration_Data();
  Report_Reset();
}

//-----
void OverwriteBackup()
{ char TempString[EEwdsz];
  byte i;
  // read the working copy

```

```

    for (i=0; i<EEwdsz; i++)
        TempString[i]=EEPROM.read(StorageWorking + i);
    // write backup copy
    for (i=0; i<EEwdsz; i++) EEPROM.write(StorageBackup +i, TempString[i]);
    PrintOKStr();
}

//=====
void TestData1()
{ // These sets were picked for testing
  // so that one set look like the another set.
  char temp[]="(1)tst data,Nano  ";
  //.....1234567890123456
  byte i;
  Serial.println(F("; Test Data one being written to EEPROM"));
  // clear the EEPROM report storage area
  ClearStorage();
  for (i=0; i< EEidsz; i++) IdString[i]=temp[i];
  // Insert null terminator at end
  IdString[EEidsz]=0;
  DegreeOffset=0;
  RefVoltage =1.075;
  MinuteTarget=1;
  newflg=0XFF;
  Write_Calibration_Data();
}

//-----
void TestData2()
{ char temp[]="(2)tst data, UNO  ";
  //.....1234567890123456
  byte i;
  Serial.println(F("; Test Data two being written to EEPROM"));
  // clear the EEPROM report storage area
  ClearStorage();
  for (i=0; i< EEidsz; i++) IdString[i]=temp[i];
  // Insert null terminator at end
  IdString[EEidsz]=0;
  DegreeOffset=0;
  RefVoltage =1.1000;
  MinuteTarget=1;
  newflg=0XFF;
  Write_Calibration_Data();
}

//-----
void CalibrationMode()
{ // used for calibration, reduces time between report lines to 5 seconds
  // there should be about 40 samples per report which will still give a good average

  if (SecondsMinute==5000)
  {
    Serial.println(F("; Exiting 5 second calibration mode <<<"));
    SecondsMinute=60000;
  }
  else
  {
    Serial.println(F("; Entering 5 second calibration mode >>>"));
    SecondsMinute=5000;
  }
  MinuteTarget=1;
  DeBug=false;
  EepromMode=false;
  ReportMode=true;
  RtnRawRead = true;
}

```

```

    RtnCelsius = true;
    RtnFahrenheit = true;
    Report_Reset();
}

//-----
void EepromDumpAll()
{ char buffer[60];           // allocate buffer
  word addr=0;               // set start address
  PrintSeparatorLine();
  Serial.println(F("; Dump all EEPROM in Hex and ASCII")); // inform the user
  while (addr < E2END)        // run until we reach the end
  { for (byte i=0; i<16; i++) // process 16 bytes at a time
    { buffer[i]=EEPROM.read(addr++); // read EEPROM
    }
    Serial.print ("; ");
    Serial.println(formatRamDump(addr-16, buffer)); // print formatted string
  }
  PrintSeparatorLine();
  PrintOKStr();
}
// ===== end of thermometer functions code ===== //

```

Appendix: Thermometer ATmega168

Main Program File (ATmega168)

```
/* ThermometerOne, ATMEGA168 Version */
/* Release 1.0.0, October 2013, Public Domain */

#include <avr/sleep.h>                // needed for shutdown function
#include <EEPROM.h>                   // needed for EEPROM read and write

// EEPROM Address Constants
const word EEmask = 0;                // 1 byte location of EEPROM storage mode mask
const word EEflag = 1;                // 1 byte location of EEPROM storage mode flag
const word ERefvoltage = 2;           // 2 byte location of RefVoltage
const word EEOffset = 4;              // 2 byte location of DegreeOffset
const word EEminutes = 6;             // 2 byte location of Report Target Minutes
const word EEunused0 = 8;             // 2 byte location -- reserved -- unused
const word EEunused1 = 10;            // 2 byte location -- reserved -- unused
const word EEunused2 = 12;            // 2 byte location -- reserved -- unused
const word EEunused3 = 14;            // 2 byte location -- reserved -- unused
const word EEidtring = 16;            // ID string w/o termination size (16)
const word EEidsize = 16;             // 24 byte location of IdString
const word EEwdsize = EEidtring+EEidsize; // Working data storage size (32)
//----- ----- // -- This area reserved for Table based system --
const word EETable = 32;              // 40 word location of conversion table
const word EETbsize = 80;             // Working data storage size (80)
//----- ----- // -- This area reserved for Table based system --
const word StorageWorking=EEmask;     // EEPROM start for working calibration data
// EEPROM start for backup copy of constants
// note we have to add 1 to the value
// Because addresses begin with zero not one
const word StorageBackup=((E2END-(EEwdsize))+1);

// EEPROM addresses variables          // begin storage for report data
word StorageBegin =StorageWorking+EEwdsize+EETbsize;
word StorageMark =StorageBegin;       // marks start of current segment
word StorageEnd =StorageBackup;       // marks end of current segment
word StorageIndex =StorageBegin;      // index for next EEPROM write

// Conversion Factors/Calibration Data
float RefVoltage;                     // Analog Sensor Reference Voltage
const float CovrtFactorV= 8192;
float DegreeOffset;                   // Fudge factor to adjust output
const float CovrtFactor0= 1024;
char IdString[EEidsize+1];           // ID/Location string for this device
word MinuteTarget = 1;               // Number of minutes between report lines
byte newflg=0;                       // used to indicate new factors in memory

float Celsius;                        // Last conversion to Celsius Temperature
float Fahrenheit;                    // Last conversion to Fahrenheit Temperature

// Global operational mode Variables  // set default operation modes
boolean ReportMode = true;           // True = reporting, False = Command Mode
boolean RtnRawRead = true;           // True = include Raw
boolean RtnCelsius = true;           // True = include Celsius
boolean RtnFahrenheit = true;        // True = include Fahrenheit
boolean DeBug = false;               // True = extended reporting for debugging
boolean EepromMode = false;          // True = write data to EEPROM on next run
boolean RoundMode = true;            // True = round output to nearest 1/4 or 1/2

// Global work Variables              // There are 1000 milliseconds in a second
// word SecondsMinute = 10000;        // --- to speed things up a bit for debugging
word SecondsMinute = 60000;          // added so calibration timing can be reduced
unsigned long SecondsTarget = 0;     // Number of seconds between report lines
```

```

unsigned long RptStartTime = 0;           // Time between report lines
byte gap = 0;                           // used to increase gap between reads
unsigned long RptTrigger = 0;            // Target Time for report
unsigned long CycleStart = 0;            // Target Time for report
unsigned long CycleTime = 0;             // Target Time for report

unsigned long Accumalator = 0;           // Accumalate temperature reads
unsigned long CycleCount = 0;            // Cycles per Report line
char cmd[] = {0,0,0};                   // used to store two character command
char prevcmd[] = {0,0,0};               // used to store previous two character command

word LastRead = 0;                       // Stores previous RawRead Average
byte Consecutive = 0;                    // used to count consective equal readings

//=====
void setup()
{
  char c;
  Serial.begin (9600);
  pinMode(13, OUTPUT);                    // so we can blink it later during writes

  EnableADC();                           // enables the ADC and set ADC clock factor
  delay (1000);                           // let serial library complete setup
  while (Serial.available()>0)            // drain any data from the serial buffer
    c=Serial.read();
  Read_Calibration_Data();                 // read and set conversion factors from EEPROM
  Check_EEPROM();                         // see if we are writing to EEPROM vs Serial
  // calculate seconds between report lines
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=long(MinuteTarget)*long(SecondsMinute);
  //delay (5000);                          // allow PC 5 seconds to get setup
  if (EepromMode==false) ReportStatus();   // report default parameters
  Accumalator = 0;                         // set startup parameters
  CycleCount = 0;
  RptTrigger = millis() + SecondsTarget;
  RptStartTime= millis();
}

//-----
void loop()
{
  char c1, c2;
  word wtemp;

  // ---- This is where we check for command input
  if ((Serial.available()>2) && (EepromMode==false))
    { if(ReadTwoCharacters()) CmdProcessor();}

  // ---- This is where we collect our temperature data
  // gap is used to increase the amount of time between reading sampling the ADC.
  // This Insures that we will not miss any data transmitted on the serial port.
  if (gap++ == 9)
    { gap=0;
      // cycle times are only used if debugging is turned on
      if (Debug == true) CycleStart= millis();
      ReadRawTempA1();
      if (Debug == true) CycleTime = CycleTime+(millis()-CycleStart);
    }

  // ---- This is where we output the teperature data
  // The time required to read 64 samples is about 119-120 milliseconds. If we get
  // within 150 milliseconds of the Report Trigger Time then we wait for it.
  // With the these timing numbers there are 500 reads of
  // 64 virtual 12 bit samples per minute.
  // Added condition for millis exceeding report trigger (possible with long commands)
  if (((RptTrigger-millis())< 150) || (millis()>RptTrigger))
    { // Serial.println ("Got here: RptTrigger ");

```

```

        while (millis() < RptTrigger);
        // We want the new trigger time set as close as possible to when the previous trigger
        // went off --- so we put ti first.
        RptTrigger= (millis() + (SecondsTarget));
        Report();
    }
}

//=====
void CmdProcessor()
{
    // this function is the main command handler
    // not many comments because I think the code is obvious
    if (Debug == true)
    {
        Serial.print(F("; Command Processor "));
        DebugPrintCharacters (cmd[0],cmd[1]);
    }
    if ((cmd[0]=='C') && (cmd[1]=='=')) CelsiusEquals();
    else if ((cmd[0]=='D') && (cmd[1]=='B')) ToggleDebugMode();
    else if ((cmd[0]=='D') && (cmd[1]=='O')) NewDegreeOffset();
    else if ((cmd[0]=='E') && (cmd[1]=='+')) EEmodeFlagSet();
    else if ((cmd[0]=='E') && (cmd[1]=='C')) ClearStorage();
    else if ((cmd[0]=='E') && (cmd[1]=='D')) DumpStorage();
    else if ((cmd[0]=='E') && (cmd[1]=='-')) EEmodeFlagClear();
    else if ((cmd[0]=='F') && (cmd[1]=='=')) FahrenheitEquals();
    else if ((cmd[0]=='I') && (cmd[1]=='D')) {Serial.print(F("; ")); Print_IdString();}
    else if ((cmd[0]=='L') && (cmd[1]=='L')) HelpMe();
    else if ((cmd[0]=='L') && (cmd[1]==':')) NewIdString();
    else if ((cmd[0]=='R') && (cmd[1]=='V')) NewRefVolt();
    else if ((cmd[0]=='S') && (cmd[1]=='S')) ShutDown();
    else if ((cmd[0]=='S') && (cmd[1]=='T')) ReportStatus();
    else if ((cmd[0]=='W') && (cmd[1]=='W')) Write_Calibration_Data();
    else if ((cmd[0]=='W') && (cmd[1]=='+')) OverwriteBackup();
    else if ((cmd[0]=='W') && (cmd[1]=='-')) RestoreFromBackup();
    else if ((cmd[0]=='?') && (cmd[1]=='?')) HelpMe();
    else if ((cmd[0]=='!') && (cmd[1]=='!')) software_Reset();
    else if ((cmd[0]=='0') && (cmd[1]=='0')) ToggleRoundMode();

    else if (cmd[0]=='C') SetCelsiusMode();
    else if (cmd[0]=='F') SetFahrenheitMode();
    else if (cmd[0]=='P') SetReportMode();
    else if (cmd[0]=='R') SetRawReadMode();
    else if (cmd[0]=='T') NewReportTime();

    // ATMEGA186 does not have internal temperature sensor
    else if ((cmd[0]=='I') && (cmd[1]=='T')) PrintNotImplemented();
    else if ((cmd[0]=='I') && (cmd[1]=='F')) PrintNotImplemented();

    // example of application specific command implemented
    // this command writes default data to the EEPROM working storage
    else if ((cmd[0]=='Z') && (cmd[1]=='1')) TestData1();
}

//-----
void HelpMe()
{
    // We need to seriously reduce the size of this function
    // Reduced by: 1,922 bytes
    Serial.println(F(
        "; AtMega168 Temperature Sensor 1.0\n"
        "; ID ST RT RF FT FF F= CT CF DO RV\n"
        "; T# PF PT DB 00 L: WW W+ W- E+ E-\n"
        "; EC ED LL SS !! Z1"));
    PrintSeparatorLine();
}

```

```

//-----
void PrintSeperatorLine()
{ Serial.print("; ");
  for (byte i=0; i<36; i++) Serial.print('-');
  Serial.println();
}

//=====
boolean ReadTwoCharacters()
{ char c1=0,c2=0,c3=-1;
  byte m=0;
  boolean EOC=true; // End of Command Terminator
  boolean OurReturn=false;

  // It is not to be believed how much effort went into creating this simple function to read
  // two characters. I noted a bit of problem reading characters from the serial port when
  // the loop was too fast therefore I have added a bit of a delay to insure the serial port
  // library can keep up. Worst case senario this function can take more than 250 milliseconds.
  // Normally when this function is called we expect the three bytes we need to be in the buffer
  // but if there is noise on the line or a parrot randomly pecking at the keyboard it could
  // take a bit longer.
  //
  // by definition we are looking for two characters followed by a terminator
  // we define a command terminator to be a carriage return, new line or null character
  // --- for good measure we are including the tab character and space as well
  // space was added because it is impossible to send a tab character from the Arduion IDE
  // we will accept any combination of those characters as a single terminator
  // we will accept the last two printable ASCII characters before a terminator for our command
  // we keep reading until we get a terminator, but we will only read for a short period

  // but before we do anything else we are going to save the previous command for posterity
  prevcmd[0]=cmd[0]; // actually we are saving it so that shutdown
  prevcmd[1]=cmd[1]; // and reset can check it before they execute

  while ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32) && (m<25))
  { // if we have a valid ASCII character for c3 then roll the characters down
    if (Serial.available()>0)
    { c3=Serial.read();
      if (c3>32) {c1=c2; c2=c3;}
    }
    // we need a bit of a delay to let the serial interface catch up
    // after 25 empty reads we give up
    else { delay (10); m++;}
  }
  // DebugPrintCharacters (c1,c2,c3,m);

  // we are very liberal about what we will accept for a command terminator
  // but we insist on having one.
  if ((c3 != 13) && (c3 != 10) && (c3 != 9) && (c3 != 0) && (c3 != 32)) EOC= false;
  // we need to drain any remaining command terminator characters from the serial buffer
  else DrainCmdTerminators();

  // now check for valid ASCII characters and End of Line
  if ((c1>32) && (c2>32) && EOC)
  { // OK... we have something to work with
    // Convert lower case to UPPER case except "w"
    // DebugPrintCharacters (c1,c2,c3);
    if ((c1 != 'w') && (c1 >96) && (c1 <123)) c1 = (c1 -32);
    if ((c2 != 'w') && (c2 >96) && (c2 <123)) c2 = (c2 -32);
    // DebugPrintCharacters (c1,c2);
    cmd[0]=c1;
    cmd[1]=c2;
    OurReturn=true;
  }
  // whatever it was that was sent did not meet our criteria

```



```

    // inform the parrot that he or she must do better
    else Serial.println(F("; ?? ??"));
    return OurReturn;
}

//-----
void DrainCmdTerminators()
{ char c3=0;
  // removed leading command terminators from serial buffer
  delay (10); c3=Serial.peek();
  while ((c3==13) || (c3==10) || (c3==9) || (c3==0) || (c3 == 32))
  { c3=Serial.read();
    delay (10);
    c3=Serial.peek();
  }
  // c3 should at this point should be -1 unless there are more commands/charaters in the
  buffer
}

//=====
// overloaded debugging function for debugging the above input routine
void DebugPrintCharacters (char c1, char c2, char c3, byte m)
{
  if (DeBug == true)
  { Serial.print ("Received: ");
    Serial.print (c1);
    // Serial.print (" ");
    Serial.print (c2);
    Serial.print (" ");
    if (c3 != 0)
    { Serial.print (c3, DEC);
      Serial.print (" ");
    }
    if (m != 0) Serial.print (m, DEC);
    Serial.println ();
  }
}

void DebugPrintCharacters (char c1, char c2, char c3)
{ byte m=0;
  DebugPrintCharacters (c1,c2,c3,m);
}

void DebugPrintCharacters (char c1, char c2)
{ byte m=0;
  char c3=0;
  DebugPrintCharacters (c1,c2,c3,m);
}

//-----End of Main File-----

```

Thermometer Function ATmega168 File

```

// cbi and sbi are standard (AVR) methods for setting,
// or clearing, bits in PORT (and other) variables.
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

//-----
void EnableADC()
{ // This is probably not needed but ....
  // set system clock divisor to 128
  // 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.

```

```

    sbi(ADCSRA, ADPS2);           // bit 2 of ADCSRA, system clock divisor
    sbi(ADCSRA, ADPS1);           // bit 1 of ADCSRA, system clock divisor
    sbi(ADCSRA, ADPS0);           // bit 0 of ADCSRA, system clock divisor
    cbi(ADCSRA, ADSCF);           // bit 5 of ADCSRA, disable auto trigger mode
    sbi(ADCSRA, ADSCF);           // bit 7 of ADCSRA, enable ADC
}

//-----
void Read_Calibration_Data()
{
    byte i=0, j=0;
    word TempWord;
    char c;
    char sign;

    // detect a virgin device --- well at least try
    if ((EEPROM.read(EEminutes)==0xFF) &&
        (EEPROM.read(EEminutes+1)==0xFF))
        TestData1();

    // Get Degree Offset
    TempWord= EEPROM.read(EEoffset)<<8;
    TempWord= (TempWord + EEPROM.read(EEoffset +1));
    sign= 1;
    if ((TempWord>>12) == B1000) sign= -1;
    // strip the sign bit
    TempWord=(TempWord<<1)>>1;
    // now we need to convert it to a fraction
    DegreeOffset=(float(TempWord)/float(CovrtFactor0)) * sign;

    // Get the RefVoltage factor;
    TempWord= EEPROM.read(EErefvolt)<<8;
    TempWord= TempWord + EEPROM.read(EErefvolt +1);
    // now we need to convert it to a fraction
    RefVoltage = float(TempWord)/CovrtFactorV ;

    // ID String -----
    i=0; c=1;
    while (c!=0,i< EEidsize)
    {c=EEPROM.read(EEidtring + i);      // read the ID string
     IdString [i++]=c;
    }
    IdString [EEidsize]=0;              // just in case

    // Set MinuteTarget from default minutes
    MinuteTarget=(EEPROM.read(EEminutes)<<8) + EEPROM.read(EEminutes +1);
    if (MinuteTarget<1)MinuteTarget=1;

    // this is a bit flag to indicate when the current constants
    // in memory are different from those stored in working storage
    newflg=0;

}

//-----
void Write_Calibration_Data()
{
    word i=0;
    // float saveflt;
    word TempWord;
    char c=-1;
    byte sign;

    // New location
    if ((newflg & B00000001) == B00000001)
    {
        for (i=0;i<EEidsize; i++) {EEPROM.write((EEidtring +i), IdString[i]);}
    }
}

```

```

    }

    // New Degree Offset
    if ((newflg & B00000010) == B00000010)
    { sign = 0;
      if (DegreeOffset < 0) sign = B10000000;
      TempWord = word(abs(DegreeOffset) * CovrtFactor0);
      EEPROM.write(EEoffset, highByte(TempWord)|sign);
      EEPROM.write(EEoffset + 1, lowByte (TempWord));
    }
    // New Reference Voltage
    if ((newflg & B00000100) == B00000100)
    { TempWord = word(RefVoltage * CovrtFactorV);
      EEPROM.write(EErefvolt, highByte(TempWord));
      EEPROM.write(EErefvolt + 1, lowByte (TempWord));
    }

    // New MinuteTarget
    if ((newflg & B00010000) == B00010000)
    {
      EEPROM.write(EEminutes, highByte(MinuteTarget));
      EEPROM.write(EEminutes + 1, lowByte (MinuteTarget));
    }

    // write the unused word(s)
    i = EEunused0;
    while (i < EEidtring)
    { if (EEPROM.read(i) != 0xFF) EEPROM.write(i, 0xFF);
      i++;
    }

    // clear the EEMODE flag
    EEmodeFlagClear();

    // note newflg is reset by Read_Calibration_Data
    Read_Calibration_Data();
    // PrintOKStr(); is sent by Report_Reset
    Report_Reset();
}

//-----
void ClearStorage()
{ // this is used to clear/erase the EEPROM data storage (except for constant areas)
  word addr;
  byte b;
  for (addr = (StorageBegin); addr < StorageBackup; addr++)
  { if (EEPROM.read(addr) != 0xFF) EEPROM.write(addr, 0xFF);
  } // note: each byte requires 6-8 machine cycles
  PrintOKStr ();
}

//-----
void EEmodeFlagSet()
{ // toggle the flag for the next run to write to EEPROM
  // does not affect current run
  // Here is the thing. We have setup wearleveling for our EEPROM data storage
  // but modeflag gets hit twice for every EEPROM run. So it will wearout
  // long before the bulk of the storage. We could just increment the byte and
  // look for odd or even values but that would continually toggle the low bit
  // wearing it out before there rest. As it turns out it is writing a zero to
  // a bit that wears then out. So we want to minimize the zero bit writes.
  // We are going to move a zero bit right to left. This extends our life by a
  // factor of eight. At that point you need to swap the backup and working
  // data locations by changing the EEPROM address locations and reprogramming
  // the Arduino. That would double the life (2 * 8 = 16).

```

```

//
byte flag;
if(EEmodeFlagTF()==false)
{
    // make them not equal
    // shift left and add a one to the right
    flag=EEPROM.read(Eeflag);
    // Serial.println(flag);
    flag=(flag<<1)+1;
    // Serial.println(flag);
    // if we have all ones start over again at the right
    if (flag==B11111111) flag=B11111110;
    // Serial.println(flag);
    // now save it
    EEPROM.write(Eeflag,flag);
}
PrintOKStr();
}

//-----
void EEmodeFlagClear()
{ byte flag;
  if(EEmodeFlagTF())
  { // make them equal
    flag=EEPROM.read(Eeflag);
    EEPROM.write(EEmask,flag);
  }
  if (EepromMode==false) PrintOKStr();
}

//-----
boolean EEmodeFlagTF()
{ // returns true if EEmodeFlag is set
  byte flag,mask;
  flag=EEPROM.read(Eeflag);
  mask=EEPROM.read(EEmask);
  if(flag==mask) return false;
  else return true;
}

//-----
void Check_EEPROM()
{ byte b[4],i;
  word w;
  word addr;

  EepromMode=false;
  if (EEmodeFlagTF())
  {
    EepromMode=true;
    // clear the flag
    EEmodeFlagClear();
    // disable serial reporting and debug mode
    ReportMode=false;
    DeBug=false;
    // we need to find the beginning of EEPROM that has not been used
    // we need at least four bytes to begin a new section.
    // so we need to find the first place where there are four bytes with FFh
    // zero our test pattern
    for (i=0; i<4; i++) b[i]=0;
    addr=StorageBegin;
    while ((addr<StorageBackup) && ((b[0]!=0xFF) || (b[1]!=0xFF)
                                     || (b[2]!=0xFF) || (b[3]!=0xFF)))
    { b[0]=b[1];
      b[1]=b[2];

```

```

        b[2]=b[3];
        b[3]=EEPROM.read(addr++);
    }
    // did we read until the end ??
    if (addr >= StorageEnd) StorageMark=StorageBegin;
    // we found 4 bytes that have not been written to
    else StorageMark=addr-4;
    // in either case we clear the storage
    ClearStorage();
    // mark the beginning
    EEPROM.write(StorageMark,0);
    EEPROM.write(StorageMark+1,0);
    // set then beginning and end of the current segment
    StorageIndex=StorageMark+2;
    StorageEnd=StorageBackup;
    // definitive notice of mode
    for (i=0; i < 30; i++) {QuickBlink(); delay (250);}
}

//-----
void Print_IdString()
{ // Serial.print(F("; "));           // prefix
  Serial.println(IdString);           // print it
}

//-----
void PrintTrueFalse(byte T)
{ // used to report True or False for boolean Globals
  if(T == 0) Serial.println(F("False"));
  else Serial.println(F("True"));
}

//-----
void ReportStatus()
{ // report settings
  PrintSeperatorLine();
  Serial.print (F("; Report:\t"));
  PrintTrueFalse (ReportMode);
  Serial.print (F("; Debug:\t"));
  PrintTrueFalse (DeBug);
  Serial.print (F("; Raw:\t"));
  PrintTrueFalse (RtnRawRead);
  Serial.print (F("; Fahrenheit:\t"));
  PrintTrueFalse (RtnFahrenh);
  Serial.print (F("; Celsius:\t"));
  PrintTrueFalse (RtnCelsius);
  Serial.print (F("; Round:\t"));
  PrintTrueFalse (RoundMode);

  Serial.print (F("; Minutes:\t"));
  Serial.println (MinuteTarget, DEC);
  Serial.print (F("; Voltage:\t"));
  Serial.println (RefVoltage, 4);

  // single sensor, no prefix needed
  Serial.print (F("; Sensor ID:\t"));
  Print_IdString ();
  Serial.print (F("; Offset:\t"));
  Serial.println (DegreeOffset, 4);

  if (newflg != 0)
    Serial.println (F(";Parameters not saved"));
  if(EEmodeFlagTF())
    Serial.println (F(";EEPROM Mode Flag Set"));
}

```

```

    PrintSeperatorLine();

}

//-----
//    ---- Below is the function the read the External LM34DZ Temperature Sensor ----
//-----

void ReadRawTempA1()
{ // Read ADC for Pin A1 (connected to LM34DZ Temperature Sensor)
  // Cycle time is aproximately 124 miliseconds
  unsigned long RawSum=0;           // used to sum samples for averaging
  word RawTemp=0;                   // used to accumulate 10 bit ADC readings
  word test=0;                      // used to count samples

  // Just in case ...
  // The INPUT mode explicitly disables the internal pullup resistors.
  pinMode(A1,INPUT);

  // turn on internal reference, right-shift ADC buffer,ADC channel = ADC1 (pin A1)
  // B11000000: AMUX Voltage reference = Internal 1.1 volt      (bits 7,6)
  // B00000000: AMUX Right Shift ADC Buffer                    (bit 5)
  // B00000001: AMUX Input Source= pin A1                     (bits 3,2,1,0)
  ADMUX = B11000001;
  delay(10);           // wait for the things to stabilize

  while (test++ < 1024 )           // oversampling loop (for averaging)
  { ADCSRA |= _BV(ADSC);           // start the conversion
    while (bit_is_set(ADCSRA, ADSC)); // ADSC cleared when complete
    RawTemp = (ADCL | (ADCH << 8)); // collect the reading
    RawSum += RawTemp;              // add it to out total
  }
  Accumalator += ((RawSum)>>10);    // averag and add to Accumalator
                                   // LSBs are lost
  CycleCount++;                    // used by functions to average readings
}

//-----

void Convert(word RawReading)
{ // converts Raw Reading to Celsius and Fahrenheit
  // New plan: offset will only be used for minor correction
  // SCALE is actual voltage that is supposed to be 1.1 but reads 1.067
  // This temperature sensor reports in Fahrenheit 1 millivolt per degree
  // We need the correct voltage !!!
  // Covrt2Fahrenheit=1.067;
  Fahrenheit = (float(RawReading)/1024)* RefVoltage *100;
  Fahrenheit = Fahrenheit + DegreeOffset;
  if (RoundMode) Fahrenheit= nearesthalf(Fahrenheit);
  // Consistancy is next to godliness.
  // We are working in Fahrenheit.
  // In our case Celsius is a function of Fahrenheit.
  // Thus we always complete our Fahrenheit calcs first.
  // That includes rounding.
  Celsius = (Fahrenheit-32)/1.8;
  if (RoundMode) Celsius=nearestquater(Celsius);
}

//-----

float nearestquater (float ValueIn)
{ // Return value rounded to nearest quater (0.25)
  byte sign=1;
  float fraction;
  if (ValueIn , 0)
  {
    sign = -1;

```

```

    ValueIn=abs(ValueIn);
    }
    fraction =ValueIn-long(ValueIn);
    if      (fraction >= 0.875) fraction=1.00;
    else if (fraction >= 0.625) fraction=0.75;
    else if (fraction >= 0.375) fraction=0.50;
    else if (fraction >= 0.125) fraction=0.25;
    else      fraction=0;
    return (long(ValueIn)+fraction) * sign;
}

//-----
float nearestHalf (float ValueIn)
{ // Return value rounded to nearest half (0.50)
  byte sign=1;
  float fraction;
  if (ValueIn , 0)
  {
    sign = -1;
    ValueIn=abs(ValueIn);
  }
  fraction =ValueIn-long(ValueIn);
  if      (fraction >= 0.750) fraction=1.00;
  else if (fraction >= 0.250) fraction=0.50;
  else      fraction=0;
  return (long(ValueIn)+fraction) * sign;
}

//-----
void Report()
{ word AvgSumRead;
  AvgSumRead = Accumalator/CycleCount;
  if (EepromMode) Report2EEPROM(AvgSumRead);
  else if (ReportMode)
  { Convert (AvgSumRead);
    Serial.print ('0');
    if (RtnRawRead)
    { Serial.print (AvgSumRead);
      Serial.print (char(9));
    }
    if (RtnCelsius)
    { Serial.print (Celsius,2);
      Serial.print (char(9));
    }
    if (RtnFahrenh)
    { Serial.print (Fahrenheit,2);
      Serial.print (char(9));
    }
    if (DeBug)
    { Serial.print (CycleTime/CycleCount);
      Serial.print (char(9));
      Serial.print (CycleCount);
      Serial.print (char(9));
      Serial.print (millis()-RptStartTime);
      RptStartTime=millis();
    }
    Serial.println();
  }
  Accumalator   = 0;
  CycleCount    = 0;
  CycleTime     = 0;
  LastRead=AvgSumRead;
}

//-----

```

```

void QuickBlink()
{ // on the UNO 1 millisecond will suffice
  // adjusted up to 3 for Nano
  digitalWrite(13, HIGH);    // turn on LED
  delay(3);
  digitalWrite(13, LOW);    // turn off LED
}

//-----
void Report2EEPROM(word AvgSumRead)
{ // We are implementing both data compression and wearleveling.
  // Our data is only 12 bits. Because we should never get a reading
  // over 2047 in our high bit will always be zero.
  // We are going to use the top four bits to count consecutive equal
  // readings. In that manner we may be able to store 16 readings in
  // a single word value.
  word makeword;
  QuickBlink();

  // Serial.println(F("Get here: Report2EEPROM"));
  // send this string for testing: EC EE ST !! !!
  // we need to skip the first pass because we have nothing to work with
  if (LastRead != 0)
  { if (LastRead == AvgSumRead) Consecutive++;
    if ((Consecutive == 15) || (LastRead != AvgSumRead))
    { // Serial.println(F("Get here: Report2EEPROM, write record"));
      // we are going to try two blinks everytime that there is a write
      makeword = (Consecutive << 12) + LastRead;
      EEPROM.write (StorageIndex++, highByte(makeword));
      EEPROM.write (StorageIndex++, lowByte(makeword));
      Consecutive=0;
      // now we need to check our storage space
      if ((StorageEnd-StorageIndex)<2)
      { // folks there is trouble in river city !
        if (StorageMark==StorageBegin)
          {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        if (StorageEnd ==StorageMark )
          {prevcmd[1]=prevcmd[0]=cmd[1]=cmd[0]='S'; ShutDown();}
        // OK, start at the beginning ....
        StorageIndex=StorageBegin;
        StorageEnd=StorageMark;
      }
      delay (50);    // force a bit of a delay so both blinks can be seen
      QuickBlink();
    }
  }
  LastRead == AvgSumRead;
}

//-----
void DumpStorage()
{ // print data stored in eeprom
  byte b1, b2, c;
  word reading;
  word countreading=0;
  word countwords=0;
  boolean savemode;
  // save the current reporting mode
  savemode=ReportMode;
  // find the beginning of the data defined to be two zero bytes
  b1=1;
  b2=1;
  while ((StorageIndex<StorageBackup) && ((b1!=0) || (b2!=0)))
  { b1=b2;
    b2=EEPROM.read(StorageIndex++);
  }
}

```



```

    }
    StorageEnd=StorageBackup;
    StorageMark=StorageIndex-2;
    // 4 high bits are the count, low 12 bits are the reading
    PrintSeperatorLine();
    Serial.println (F("; Begin EEPROM data dump"));
    Serial.println (F("; Raw Reading\tCelsius\tFahrenheit"));
    while (((StorageEnd-StorageIndex)>=2) && ((b1 != 0xFF)||(b2 != 0xFF)))
    { b1=EEPROM.read(StorageIndex++);
      b2=EEPROM.read(StorageIndex++);
      //----- debuggin code
      // Serial.print  ("; Location: ");
      // Serial.print  (StorageIndex);
      // Serial.print  (" ", " ");
      // Serial.print  (b1,HEX);
      // Serial.print  (" ", " ");
      // Serial.print  (b2,HEX);
      countwords++;
      // two bytes of FFh will mark the end
      if ((b1 != 0xFF) || (b2 != 0xFF))
      { Consecutive=b1>>4;
        reading= ((b1 & B00001111)<<8)+b2;
        Convert(reading);
        //----- debuggin code
        // Serial.print  (" ", " ");
        // Serial.print  (Consecutive);
        // Serial.print  (" ", " ");
        // Serial.print  (reading);
        // Serial.println();
        // while (Serial.available() ==0);
        // c=Serial.read();
        // the logic here is we need to print every reading at least once ...
        // that is when it is zero. When we subtract one from zero we get 255
        while (Consecutive<255)
        { countreading++;
          Serial.print  (reading);
          Serial.print  (char(9));
          Serial.print  (Celsius,2);
          Serial.print  (char(9));
          Serial.print  (Fahrenheit,2);
          Serial.println ();
          Consecutive--;
        }
        // now check the addresses
        if ((StorageEnd-StorageIndex)<2)
        { if (StorageMark != StorageBegin)
          { StorageEnd = StorageMark;
            StorageMark = StorageBegin;
            StorageIndex = StorageMark +2;
          }
        }
      }
    }
    PrintSeperatorLine();
    Serial.println (F("; End EEPROM data dump"));
    Serial.print  (F("; Readings:\t"));
    Serial.println  (countreading, DEC);
    Serial.print  (F(";Storage Words:\t"));
    Serial.println  (countwords, DEC);
    PrintSeperatorLine();
    // restore the current reporting mode
    ReportMode=savemode;
  }
  //-----

```

```

void Response (char str[])
{
    if (EepromMode == false)
    {
        // we do not want to get hung up
        // this just serves to reduce command response memory usage a bit
        // trying to write to something that is not connected
        Serial.print (F("; "));
        Serial.print (cmd);
        Serial.print (F(" "));
        Serial.println (str);
    }
}

//-----
void PrintOKStr ()
{
    // command was accepted and processed
    Response ("OK");
}

//-----
void PrintNotRecognized()
{
    // command was Not Recognized
    Response ("??");
}

//-----
void PrintNotImplemented()
{
    // command was Not Recognized
    Response ("XX");
}

//-----
void ShutDown()
{
    // Note that no provision is made to wake up.
    // This is as close to shutdown as we can get.
    // Because of the inefficient voltage regulator this
    // mode still draws a lot of power (about 10mA).
    // A standard 9 volt battery may last about 16 hours.

    // Serial.println(prevcmd);
    if ((prevcmd[0]=='S') && (prevcmd[1]=='S'))
    {
        Serial.println (F("; SHUTDOWN"));
        // give device time to send string
        for (byte i=0; i< 25; i++)
        {
            QuickBlink();
            delay (100);
        }
        cbi(ADCSRA, ADEN); // bit 7 of ADCSRA, disable ADC
        noInterrupts();
        set_sleep_mode(SLEEP_MODE_PWR_DOWN);
        sleep_enable();
        sleep_mode(); // all execution should stop here
        while(0==0); // endless loop (belts and suspenders)
    }
    else PrintOKStr(); // first time through only
}

//-----
void software_Reset()
{
    // Restarts program from beginning but
    // does not reset the peripherals and registers
    // as we are not doing anything with the the
    // timers or peripherals or registers this
    // should be adequate (will not support updating)

    // Serial.println(prevcmd);

```

```

    if ((prevcmd[0]!='!') && (prevcmd[1]!='!'))
    { Serial.println(F("; RESETTING"));
      // give device time to send string
      delay(1000);
      asm volatile (" jmp 0");
    }
    else PrintOKStr();      // first time through only
}

//-----
void SetRawReadMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnRawRead = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnRawRead = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetCelsiusMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnCelsius = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnCelsius = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetFahrenheitMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {RtnFahrenh = true; PrintOKStr();}
  else if (cmd[1]=='F') {RtnFahrenh = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void SetReportMode()
{ // check for "T" or "F", true of false
  if (cmd[1]=='T') {ReportMode = true; PrintOKStr();}
  else if (cmd[1]=='F') {ReportMode = false; PrintOKStr();}
  else PrintNotRecognized();
}

//-----
void ToggleDebugMode()
{ // toggle Debug mode
  if (Debug == true) Debug = false;
  else if (Debug == false) Debug = true;
  PrintOKStr();
}

//-----
void ToggleRoundMode()
{ // check for "T" or "F", true of false
  if (RoundMode) {RoundMode = false; PrintOKStr();}
  else {RoundMode = true; PrintOKStr();}
}

//-----
void NewReportTime()
{ // set report Minutes
  if (cmd[1]=='1') { MinuteTarget = 1; Report_Reset();}
  else if (cmd[1]=='2') { MinuteTarget = 2; Report_Reset();}
  else if (cmd[1]=='3') { MinuteTarget = 3; Report_Reset();}
  else if (cmd[1]=='4') { MinuteTarget = 4; Report_Reset();}
  else if (cmd[1]=='5') { MinuteTarget = 5; Report_Reset();}
  //---- the timings below have not been tested -----

```

```

else if (cmd[1]=='6') { MinuteTarget = 10; Report_Reset();}
else if (cmd[1]=='7') { MinuteTarget = 15; Report_Reset();}
else if (cmd[1]=='8') { MinuteTarget = 20; Report_Reset();}
else if (cmd[1]=='9') { MinuteTarget = 30; Report_Reset();}
else if (cmd[1]=='0') { MinuteTarget = 60; Report_Reset();}

else if (cmd[1]=='A') PrintNotImplemented(); // not implimented
else if (cmd[1]=='B') PrintNotImplemented(); // not implimented
else if (cmd[1]=='C') PrintNotImplemented(); // not implimented
else if (cmd[1]=='D') PrintNotImplemented(); // not implimented
else if (cmd[1]=='E') PrintNotImplemented(); // not implimented
else if (cmd[1]=='F') PrintNotImplemented(); // not implimented
else if (cmd[1]=='T') PrintNotImplemented(); // not implimented
else PrintNotRecognized(); // not recognized
}

//-----
void Report_Reset()
{ // this force the current data to be reported
  // and reset our clock using the new time
  unsigned long SaveMe=SecondsTarget;
  PrintOKStr();
  Serial.println (F("; Report Timing reset"));
  // calculate seconds between report lines
  // SecondsTarget=MinuteTarget*SecondsMinute;
  // we have to "cast" the two word values or we will get a word value for the result
  SecondsTarget=(long(MinuteTarget)*long(SecondsMinute));

  if (SecondsTarget != SaveMe) newflg = newflg | B00010000;

  Accumalator = 0; // reset report parameters
  CycleCount = 0;
  RptTrigger = millis() + SecondsTarget;
  RptStartTime= millis();
}

//-----
void NewIdString()
{ // New Location ID String
  // Serial.println("got here: NewIdString");
  // set time out to 5 seconds
  unsigned long timelimit = millis() + (5000);
  boolean timeout=false;
  char c= -1;
  byte n= 0;
  while ((c != 0) && (c != 10) && (c != 9) && (c != 13) && (n<EEidsize) && (timeout==false))
  { delay(10);
    c = Serial.read();
    if (c > 31) IdString[n++]=c;
    // check for timeout
    if (millis()>timelimit) timeout=true;
  }
  while (n<EEidsize) IdString[n++]=0;
  if (timeout) Serial.println (F("; aborted due to timeout"));
  else
  { IdString[EEidsize]=0; // make certain last charater is null
    newflg = newflg | B00000001;
    // Serial.println (IdString);
    PrintOKStr();
  }
  DrainCmdTermiantors();
}

//-----

```

```

void PrintDegreeOffsetEffect(float NewOffset)
{ // new offset must be in Degrees Fahrenheit
  boolean SaveRoundMode;
  SaveRoundMode=RoundMode;

  while (CycleCount<50) ReadRawTempA1();
  DegreeOffset=0;
  RoundMode=false;
  Convert(Accumulator/CycleCount);
  DegreeOffset=NewOffset;
  Serial.print (F("; Offset(F):\t"));
  Serial.println (DegreeOffset);
  Serial.print (F("; Fahrenheit:\t"));
  Serial.println (Fahrenheit);
  Serial.print (F("; Adjusted:\t"));
  Fahrenheit=Fahrenheit+NewOffset;
  if (SaveRoundMode) Fahrenheit=nearesthalf(Fahrenheit);
  Serial.println (Fahrenheit);
  //-----
  Serial.print (F("; Celsius:\t"));
  Serial.println (Celsius);
  Serial.print (F("; Adjusted:\t"));
  Celsius=Celsius+(DegreeOffset/1.8000);
  if (SaveRoundMode) Celsius=nearestquater(Celsius);
  Serial.println (Celsius);
  newflg = newflg | B00000010;
  RoundMode=SaveRoundMode;
  PrintOKStr();
}

//-----
void ValueNotAccepted()
{ Serial.print("; ");
  Serial.print(cmd);
  Serial.print(" invalid/no input");
}

//-----
void NewDegreeOffset()
{ // New Degree Offset
  float tempfloat=0;
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat!=0)
  { if (abs(tempfloat)<0.010) tempfloat = 0;
    PrintDegreeOffsetEffect(tempfloat);
    newflg = newflg | B00000010;
  }
  else Serial.println (F("; value not accepted"));
  DrainCmdTermiantors();
}

//-----
void CalculateDegreeOffset(float tempfloat)
{ // calculate a new degree offset, TempF is Temperature in degrees Fahrenheit
  // get the current raw reading
  boolean SaveRoundMode;

  Serial.println (F("; Calculating new offset ..."));
  while (CycleCount<50) ReadRawTempA1();
  // set the current offset to zero so that it
  // does not affect the Conversion
  DegreeOffset=0;
  SaveRoundMode=RoundMode;
  RoundMode=false;

```

```

    Convert(Accumulator/CycleCount);
    RoundMode=SaveRoundMode;
    PrintDegreeOffsetEffect(tempfloat-Fahrenheit);
}

//-----
void FahrenheitEquals()
{ // sets offset according to current reading and input Fahrenheit
  float tempfloat=0;
  float deltaR;
  word RawReading;
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0) CalculateDegreeOffset(tempfloat);
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void CelsiusEquals()
{ // sets offset according to current reading and input Fahrenheit
  float tempfloat=0;
  float deltaR;
  word RawReading;
  // Serial.println (F("Got Here: CelsiusEquals"));
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat != 0) CalculateDegreeOffset((tempfloat*1.8)+32);
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void NewRefVolt()
{ // New Degree Offset
  float tempfloat=0;
  delay (2000);
  tempfloat=Serial.parseFloat();
  if (tempfloat!=0)
  { // Serial.println(tempfloat,4);
    // Serial.println(RefVoltage,4);
    RefVoltage=tempfloat;
    // Serial.println(RefVoltage,4);
    newflg = newflg | B00000100;
    PrintOKStr();
  }
  else ValueNotAccepted();
  DrainCmdTermiantors();
}

//-----
void RestoreFromBackup()
{ char TempString[EEwdsz];
  byte i;
  // read the backup copy
  for (i=0; i<EEwdsz; i++) TempString[i]=EEPROM.read(StorageBackup + i);
  // write working copy
  for (i=0; i<EEwdsz; i++) EEPROM.write(StorageWorking +i, TempString[i]);
  newflg=0;
  Read_Calibration_Data();
  Report_Reset();
}

//-----
void OverwriteBackup()

```

```

{ char TempString[EEwdsz];
  byte i;
  // read the working copy
  for (i=0; i<EEwdsz; i++)
    TempString[i]=EEPROM.read(StorageWorking + i);
  // write backup copy
  for (i=0; i<EEwdsz; i++) EEPROM.write(StorageBackup +i, TempString[i]);
  PrintOKStr();
}

//=====
void TestData1()
{ // These sets were picked for testing
  // so that one set look like the another set.
  char temp[]="(1)tst data,Nano ";
  //.....1234567890123456
  byte i;
  Serial.println(F("; Load Default Parameters"));
  // clear the EEPROM report storage area
  ClearStorage();
  for (i=0; i< EEidsz; i++) IdString[i]=temp[i];
  // Insert null terminator at end
  IdString[EEidsz]=0;
  DegreeOffset=0;
  RefVoltage =1.075;
  MinuteTarget=1;
  newflg=0XFF;
  Write_Calibration_Data();
}

```

Appendix: ASCII Table

ASCII characters are equivalent to the first 126 characters of UTF-8. (Some notable ASCII omissions are the British pound sterling character "£", the copyright symbol "©" and registered trademark symbol). ASCII does not define characters codes above 127. Those characters above 127 in the table below are from the true type font "MS Linedraw" that came with early MS Windows operating systems (*replicates the MS DOS extended character set*). As current versions of Microsoft Word have this font disabled these pages have been inserted as "pictures".

Binary	Hex	Decimal	esc	Char	Description
0000 0000	0	0	\0	NUL	Null character
0000 0001	1	1		SOH	Start of Header
0000 0010	2	2		STX	Start of Text
0000 0011	3	3		ETX	End of Text
0000 0100	4	4		EOT	End of Transmission
0000 0101	5	5		ENQ	Enquiry
0000 0110	6	6		ACK	Acknowledgment
0000 0111	7	7	\a	BEL	Bell
0000 1000	8	8	\b	BS	Backspace
0000 1001	9	9	\t	HT	Horizontal Tab
0000 1010	A	10	\n	LF	Line Feed
0000 1011	B	11	\v	VT	Vertical Tab
0000 1100	C	12	\r	FF	Form Feed
0000 1101	D	13		CR	Carriage Return
0000 1110	E	14		SO	Shift Out
0000 1111	F	15		SI	Shift In
0001 0000	10	16		DLE	Data Link Escape
0001 0001	11	17		C1 (XON)	Device Control 1
0001 0010	12	18		DC2	Device Control 2
0001 0011	13	19		DC3(XOFF)	Device Control 3
0001 0100	14	20		DC4	Device Control 4
0001 0101	15	21		NAK	Negative Acknowledgment
0001 0110	16	22		SYN	Synchronous Idle
0001 0111	17	23		ETB	End of Transmission Block
0001 1000	18	24		CAN	Cancel
0001 1001	19	25		EM	End of Medium
0001 1010	1A	26		SUB	Substitute
0001 1011	1B	27	\e	ESC	Escape
0001 1100	1C	28		FS	File Separator
0001 1101	1D	29		GS	Group Separator
0001 1110	1E	30		RS	Reqst to Send
0001 1111	1F	31		US	Unit Separator

Binary	Hex	Decimal	Char
0010 0000	20	32	(space)
0010 0001	21	33	!
0010 0010	22	34	"
0010 0011	23	35	#
0010 0100	24	36	\$
0010 0101	25	37	%
0010 0110	26	38	&
0010 0111	27	39	'
0010 1000	28	40	(
0010 1001	29	41)
0010 1010	2A	42	*
0010 1011	2B	43	+
0010 1100	2C	44	,
0010 1101	2D	45	-
0010 1110	2E	46	.
0010 1111	2F	47	/
0011 0000	30	48	0
0011 0001	31	49	1
0011 0010	32	50	2
0011 0011	33	51	3
0011 0100	34	52	4
0011 0101	35	53	5
0011 0110	36	54	6
0011 0111	37	55	7
0011 1000	38	56	8
0011 1001	39	57	9
0011 1010	3A	58	:
0011 1011	3B	59	;
0011 1100	3C	60	<
0011 1101	3D	61	=
0011 1110	3E	62	>
0011 1111	3F	63	?

Binary	Hex	Decimal	Char	Binary	Hex	Decimal	Char
0100 0000	40	64	@	0110 0000	60	96	~
0100 0001	41	65	A	0110 0001	61	97	a
0100 0010	42	66	B	0110 0010	62	98	b
0100 0011	43	67	C	0110 0011	63	99	c
0100 0100	44	68	D	0110 0100	64	100	d
0100 0101	45	69	E	0110 0101	65	101	e
0100 0110	46	70	F	0110 0110	66	102	f
0100 0111	47	71	G	0110 0111	67	103	g
0100 1000	48	72	H	0110 1000	68	104	h
0100 1001	49	73	I	0110 1001	69	105	i
0100 1010	4A	74	J	0110 1010	6A	106	j
0100 1011	4B	75	K	0110 1011	6B	107	k
0100 1100	4C	76	L	0110 1100	6C	108	l
0100 1101	4D	77	M	0110 1101	6D	109	m
0100 1110	4E	78	N	0110 1110	6E	110	n
0100 1111	4F	79	O	0110 1111	6F	111	o
0101 0000	50	80	P	0111 0000	70	112	p
0101 0001	51	81	Q	0111 0001	71	113	q
0101 0010	52	82	R	0111 0010	72	114	r
0101 0011	53	83	S	0111 0011	73	115	s
0101 0100	54	84	T	0111 0100	74	116	t
0101 0101	55	85	U	0111 0101	75	117	u
0101 0110	56	86	V	0111 0110	76	118	v
0101 0111	57	87	W	0111 0111	77	119	w
0101 1000	58	88	X	0111 1000	78	120	x
0101 1001	59	89	Y	0111 1001	79	121	y
0101 1010	5A	90	Z	0111 1010	7A	122	z
0101 1011	5B	91	[0111 1011	7B	123	{
0101 1100	5C	92	\	0111 1100	7C	124	
0101 1101	5D	93]	0111 1101	7D	125	}
0101 1110	5E	94	^	0111 1110	7E	126	~
0101 1111	5F	95	_	0111 1111	7F	127	

Binary	Hex	Decimal	Char	Binary	Hex	Decimal	Char
1000 0000	80	128		1010 0000	A0	160	á
1000 0001	81	129	ù	1010 0001	A1	161	í
1000 0010	82	130	é	1010 0010	A2	162	ó
1000 0011	83	131	â	1010 0011	A3	163	ú
1000 0100	84	132	ä	1010 0100	A4	164	ñ
1000 0101	85	133	à	1010 0101	A5	165	Ñ
1000 0110	86	134	å	1010 0110	A6	166	ª
1000 0111	87	135	ç	1010 0111	A7	167	º
1000 1000	88	136	ê	1010 1000	A8	168	¿
1000 1001	89	137	ë	1010 1001	A9	169	¸
1000 1010	8A	138	è	1010 1010	AA	170	¬
1000 1011	8B	139	ï	1010 1011	AB	171	½
1000 1100	8C	140	î	1010 1100	AC	172	¾
1000 1101	8D	141	ì	1010 1101	AD	173	¡
1000 1110	8E	142	Ä	1010 1110	AE	174	«
1000 1111	8F	143	Å	1010 1111	AF	175	»
1001 0000	90	144	É	1011 0000	B0	176	░
1001 0001	91	145	æ	1011 0001	B1	177	▒
1001 0010	92	146	Æ	1011 0010	B2	178	▓
1001 0011	93	147	ø	1011 0011	B3	179	
1001 0100	94	148	ö	1011 0100	B4	180	└
1001 0101	95	149	õ	1011 0101	B5	181	├
1001 0110	96	150	û	1011 0110	B6	182	└┘
1001 0111	97	151	ü	1011 0111	B7	183	┐
1001 1000	98	152	ÿ	1011 1000	B8	184	┑
1001 1001	99	153	Ö	1011 1001	B9	185	├┤
1001 1010	9A	154	Û	1011 1010	BA	186	▯
1001 1011	9B	155	◊	1011 1011	BB	187	┑┒
1001 1100	9C	156	£	1011 1100	BC	188	┓
1001 1101	9D	157	¥	1011 1101	BD	189	└┘
1001 1110	9E	158	₯	1011 1110	BE	190	┘
1001 1111	9F	159	f	1011 1111	BF	191	┐

Binary	Hex	Decimal	Char	Binary	Hex	Decimal	Char
1100 0000	C0	192	ℓ	1110 0000	E0	224	α
1100 0001	C1	193	⊥	1110 0001	E1	225	β
1100 0010	C2	194	⌞	1110 0010	E2	226	Γ
1100 0011	C3	195	⌟	1110 0011	E3	227	π
1100 0100	C4	196	—	1110 0100	E4	228	Σ
1100 0101	C5	197	⊕	1110 0101	E5	229	σ
1100 0110	C6	198	⌞	1110 0110	E6	230	μ
1100 0111	C7	199	⌟	1110 0111	E7	231	τ
1100 1000	C8	200	ℓ	1110 1000	E8	232	Φ
1100 1001	C9	201	℞	1110 1001	E9	233	Θ
1100 1010	CA	202	⊥	1110 1010	EA	234	Ω
1100 1011	CB	203	⌞	1110 1011	EB	235	δ
1100 1100	CC	204	⌟	1110 1100	EC	236	∞
1100 1101	CD	205	=	1110 1101	ED	237	∅
1100 1110	CE	206	⊕	1110 1110	EE	238	ε
1100 1111	CF	207	⊥	1110 1111	EF	239	∩
1101 0000	D0	208	⊥	1111 0000	F0	240	≡
1101 0001	D1	209	⌞	1111 0001	F1	241	±
1101 0010	D2	210	⌞	1111 0010	F2	242	≥
1101 0011	D3	211	ℓ	1111 0011	F3	243	≤
1101 0100	D4	212	⌞	1111 0100	F4	244	⌈
1101 0101	D5	213	℞	1111 0101	F5	245	⌋
1101 0110	D6	214	℞	1111 0110	F6	246	÷
1101 0111	D7	215	⊕	1111 0111	F7	247	≈
1101 1000	D8	216	⊕	1111 1000	F8	248	°
1101 1001	D9	217	⌞	1111 1001	F9	249	•
1101 1010	DA	218	⌞	1111 1010	FA	250	•
1101 1011	DB	219	■	1111 1011	FB	251	√
1101 1100	DC	220	■	1111 1100	FC	252	π
1101 1101	DD	221	■	1111 1101	FD	253	²
1101 1110	DE	222	■	1111 1110	FE	254	■
1101 1111	DF	223	■	1111 1111	FF	255	□

Appendix: Celsius vs. Fahrenheit Table

Celsius	Fahrenheit	Celsius	Fahrenheit
-40	-40.00	-39.5	-39.10
-39	-38.20	-38.5	-37.30
-38	-36.40	-37.5	-35.50
-37	-34.60	-36.5	-33.70
-36	-32.80	-35.5	-31.90
-35	-31.00	-34.5	-30.10
-34	-29.20	-33.5	-28.30
-33	-27.40	-32.5	-26.50
-32	-25.60	-31.5	-24.70
-31	-23.80	-30.5	-22.90
-30	-22.00	-29.5	-21.10
-29	-20.20	-28.5	-19.30
-28	-18.40	-27.5	-17.50
-27	-16.60	-26.5	-15.70
-26	-14.80	-25.5	-13.90
-25	-13.00	-24.5	-12.10
-24	-11.20	-23.5	-10.30
-23	-9.40	-22.5	-8.50
-22	-7.60	-21.5	-6.70
-21	-5.80	-20.5	-4.90
-20	-4.00	-19.5	-3.10
-19	-2.20	-18.5	-1.30
-18	-0.40	-17.5	0.50
-17	1.40	-16.5	2.30
-16	3.20	-15.5	4.10
-15	5.00	-14.5	5.90
-14	6.80	-13.5	7.70
-13	8.60	-12.5	9.50
-12	10.40	-11.5	11.30
-11	12.20	-10.5	13.10
-10	14.00	-9.5	14.90
-9	15.80	-8.5	16.70
-8	17.60	-7.5	18.50
-7	19.40	-6.5	20.30
-6	21.20	-5.5	22.10
-5	23.00	-4.5	23.90
-4	24.80	-3.5	25.70
-3	26.60	-2.5	27.50
-2	28.40	-1.5	29.30
-1	30.20	-0.5	31.10

Fahrenheit	Celsius	Fahrenheit	Celsius
-40	-40.00	-39	-39.44
-38	-38.89	-37	-38.33
-36	-37.78	-35	-37.22
-34	-36.67	-33	-36.11
-32	-35.56	-31	-35.00
-30	-34.44	-29	-33.89
-28	-33.33	-27	-32.78
-26	-32.22	-25	-31.67
-24	-31.11	-23	-30.56
-22	-30.00	-21	-29.44
-20	-28.89	-19	-28.33
-18	-27.78	-17	-27.22
-16	-26.67	-15	-26.11
-14	-25.56	-13	-25.00
-12	-24.44	-11	-23.89
-10	-23.33	-9	-22.78
-8	-22.22	-7	-21.67
-6	-21.11	-5	-20.56
-4	-20.00	-3	-19.44
-2	-18.89	-1	-18.33
0	-17.78	1	-17.22
2	-16.67	3	-16.11
4	-15.56	5	-15.00
6	-14.44	7	-13.89
8	-13.33	9	-12.78
10	-12.22	11	-11.67
12	-11.11	13	-10.56
14	-10.00	15	-9.44
16	-8.89	17	-8.33
18	-7.78	19	-7.22
20	-6.67	21	-6.11
22	-5.56	23	-5.00
24	-4.44	25	-3.89
26	-3.33	27	-2.78
28	-2.22	29	-1.67
30	-1.11	31	-0.56

Celsius	Fahrenheit	Celsius	Fahrenheit
0	32.00	0.5	32.90
1	33.80	1.5	34.70
2	35.60	2.5	36.50
3	37.40	3.5	38.30
4	39.20	4.5	40.10
5	41.00	5.5	41.90
6	42.80	6.5	43.70
7	44.60	7.5	45.50
8	46.40	8.5	47.30
9	48.20	9.5	49.10
10	50.00	10.5	50.90
11	51.80	11.5	52.70
12	53.60	12.5	54.50
13	55.40	13.5	56.30
14	57.20	14.5	58.10
15	59.00	15.5	59.90
16	60.80	16.5	61.70
17	62.60	17.5	63.50
18	64.40	18.5	65.30
19	66.20	19.5	67.10
20	68.00	20.5	68.90
21	69.80	21.5	70.70
22	71.60	22.5	72.50
23	73.40	23.5	74.30
24	75.20	24.5	76.10
25	77.00	25.5	77.90
26	78.80	26.5	79.70
27	80.60	27.5	81.50
28	82.40	28.5	83.30
29	84.20	29.5	85.10
30	86.00	30.5	86.90
31	87.80	31.5	88.70
32	89.60	32.5	90.50
33	91.40	33.5	92.30
34	93.20	34.5	94.10
35	95.00	35.5	95.90
36	96.80	36.5	97.70
37	98.60	37.5	99.50
38	100.40	38.5	101.30
39	102.20	39.5	103.10
40	104.00	40.5	104.90

Fahrenheit	Celsius	Fahrenheit	Celsius
32	0.00	33	0.56
34	1.11	35	1.67
36	2.22	37	2.78
38	3.33	39	3.89
40	4.44	41	5.00
42	5.56	43	6.11
44	6.67	45	7.22
46	7.78	47	8.33
48	8.89	49	9.44
50	10.00	51	10.56
52	11.11	53	11.67
54	12.22	55	12.78
56	13.33	57	13.89
58	14.44	59	15.00
60	15.56	61	16.11
62	16.67	63	17.22
64	17.78	65	18.33
66	18.89	67	19.44
68	20.00	69	20.56
70	21.11	71	21.67
72	22.22	73	22.78
74	23.33	75	23.89
76	24.44	77	25.00
78	25.56	79	26.11
80	26.67	81	27.22
82	27.78	83	28.33
84	28.89	85	29.44
86	30.00	87	30.56
88	31.11	89	31.67
90	32.22	91	32.78
92	33.33	93	33.89
94	34.44	95	35.00
96	35.56	97	36.11
98	36.67	99	37.22
100	37.78	101	38.33
102	38.89	103	39.44
104	40.00	105	40.56

Celsius	Fahrenheit		Celsius	Fahrenheit
1	105.80		41.5	106.70
42	107.60		42.5	108.50
43	109.40		43.5	110.30
44	111.20		44.5	112.10
45	113.00		45.5	113.90
46	114.80		46.5	115.70
47	116.60		47.5	117.50
48	118.40		48.5	119.30
49	120.20		49.5	121.10
50	122.00		50.5	122.90
51	123.80		51.5	124.70
52	125.60		52.5	126.50
53	127.40		53.5	128.30
54	129.20		54.5	130.10
55	131.00		55.5	131.90
56	132.80		56.5	133.70
57	134.60		57.5	135.50
58	136.40		58.5	137.30
59	138.20		59.5	139.10
60	140.00		60.5	140.90
61	141.80		61.5	142.70
62	143.60		62.5	144.50
63	145.40		63.5	146.30
64	147.20		64.5	148.10
65	149.00		65.5	149.90
66	150.80		66.5	151.70
67	152.60		67.5	153.50
68	154.40		68.5	155.30
69	156.20		69.5	157.10
70	158.00		70.5	158.90
71	159.80		71.5	160.70
72	161.60		72.5	162.50
73	163.40		73.5	164.30
74	165.20		74.5	166.10
75	167.00		75.5	167.90
76	168.80		76.5	169.70
77	170.60		77.5	171.50
78	172.40		78.5	173.30
79	174.20		79.5	175.10
80	176.00		80.5	176.90

Fahrenheit	Celsius		Fahrenheit	Celsius
106	41.11		107	41.67
108	42.22		109	42.78
110	43.33		111	43.89
112	44.44		113	45.00
114	45.56		115	46.11
116	46.67		117	47.22
118	47.78		119	48.33
120	48.89		121	49.44
122	50.00		123	50.56
124	51.11		125	51.67
126	52.22		127	52.78
128	53.33		129	53.89
130	54.44		131	55.00
132	55.56		133	56.11
134	56.67		135	57.22
136	57.78		137	58.33
138	58.89		139	59.44
140	60.00		141	60.56
142	61.11		143	61.67
144	62.22		145	62.78
146	63.33		147	63.89
148	64.44		149	65.00
150	65.56		151	66.11
152	66.67		153	67.22
154	67.78		155	68.33
156	68.89		157	69.44
158	70.00		159	70.56
160	71.11		161	71.67
162	72.22		163	72.78
164	73.33		165	73.89
166	74.44		167	75.00
168	75.56		169	76.11
170	76.67		171	77.22
172	77.78		173	78.33
174	78.89		175	79.44
176	80.00		177	80.56

Celsius	Fahrenheit		Celsius	Fahrenheit
81	177.80		81.5	178.70
82	179.60		82.5	180.50
83	181.40		83.5	182.30
84	183.20		84.5	184.10
85	185.00		85.5	185.90
86	186.80		86.5	187.70
87	188.60		87.5	189.50
88	190.40		88.5	191.30
89	192.20		89.5	193.10
90	194.00		90.5	194.90
91	195.80		91.5	196.70
92	197.60		92.5	198.50
93	199.40		93.5	200.30
94	201.20		94.5	202.10
95	203.00		95.5	203.90
96	204.80		96.5	205.70
97	206.60		97.5	207.50
98	208.40		98.5	209.30
99	210.20		99.5	211.10
100	212.00		100.5	212.90

Fahrenheit	Celsius		Fahrenheit	Celsius
178	81.11		179	81.67
180	82.22		181	82.78
182	83.33		183	83.89
184	84.44		185	85.00
186	85.56		187	86.11
188	86.67		189	87.22
190	87.78		191	88.33
192	88.89		193	89.44
194	90.00		195	90.56
196	91.11		197	91.67
198	92.22		199	92.78
200	93.33		201	93.89
202	94.44		203	95.00
204	95.56		205	96.11
206	96.67		207	97.22
208	97.78		209	98.33
210	98.89		211	99.44
212	100.00		213	100.56

Appendix: LM34 Data Sheet

The full datasheet (TI Literature Number: SNIS161B) is included herein by reference to last known valid TI URL:
<http://www.ti.com/lit/ds/symlink/lm34.pdf>

When reviewing that document the reader may find the following sections to be of interest:



November 2000

LM34 Precision Fahrenheit Temperature Sensors

General Description

The LM34 series are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Fahrenheit temperature. The LM34 thus has an advantage over linear temperature sensors calibrated in degrees Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Fahrenheit scaling. The LM34 does not require any external calibration or trimming to provide typical accuracies of $\pm 1/2^\circ\text{F}$ at room temperature and $\pm 1 1/2^\circ\text{F}$ over a full -50 to $+300^\circ\text{F}$ temperature range. Low cost is assured by trimming and calibration at the wafer level. The LM34's low output impedance, linear output, and precise inherent calibration make interfacing to readout or control circuitry especially easy. It can be used with single power supplies or with plus and minus supplies. As it draws only $75\text{ }\mu\text{A}$ from its supply, it has very low self-heating, less than 0.2°F in still air. The LM34 is rated to operate over a -50° to $+300^\circ\text{F}$ temperature range, while the LM34C is rated for a -40° to $+230^\circ\text{F}$ range (0°F with improved accuracy). The LM34 series is available packaged in

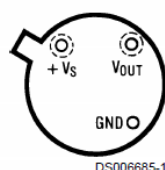
hermetic TO-46 transistor packages, while the LM34C, LM34CA and LM34D are also available in the plastic TO-92 transistor package. The LM34D is also available in an 8-lead surface mount small outline package. The LM34 is a complement to the LM35 (Centigrade) temperature sensor.

Features

- Calibrated directly in degrees Fahrenheit
- Linear $+10.0\text{ mV}/^\circ\text{F}$ scale factor
- 1.0°F accuracy guaranteed (at $+77^\circ\text{F}$)
- Rated for full -50° to $+300^\circ\text{F}$ range
- Suitable for remote applications
- Low cost due to wafer-level trimming
- Operates from 5 to 30 volts
- Less than $90\text{ }\mu\text{A}$ current drain
- Low self-heating, 0.18°F in still air
- Nonlinearity only $\pm 0.5^\circ\text{F}$ typical
- Low-impedance output, 0.4Ω for 1 mA load

Connection Diagrams

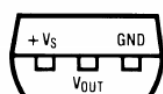
TO-46
Metal Can Package
(Note 1)



DS006685-1

Order Numbers LM34H,
LM34AH, LM34CH,
LM34CAH or LM34DH
See NS Package
Number H03H

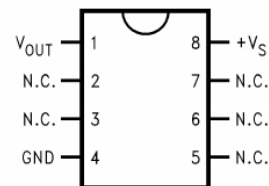
TO-92
Plastic Package



BOTTOM VIEW
DS006685-2

Order Number LM34CZ,
LM34CAZ or LM34DZ
See NS Package
Number Z03A

SO-8
Small Outline
Molded Package



DS006685-20

N.C. = No Connection

Top View
Order Number LM34DM
See NS Package Number M08A

Note 1: Case is connected to negative pin (GND).

Absolute Maximum Ratings (Note 11)

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/Distributors for availability and specifications.

Supply Voltage	+35V to -0.2V
Output Voltage	+6V to -1.0V
Output Current	10 mA
Storage Temperature,	
TO-46 Package	-76°F to +356°F
TO-92 Package	-76°F to +300°F
SO-8 Package	-65°C to +150°C
ESD Susceptibility (Note 12)	800V
Lead Temp.	

TO-46 Package	
(Soldering, 10 seconds)	+300°C
TO-92 Package	
(Soldering, 10 seconds)	+260°C
SO Package (Note 13)	
Vapor Phase (60 seconds)	215°C
Infrared (15 seconds)	220°C
Specified Operating Temp. Range (Note 3)	
	T_{MIN} to T_{MAX}
LM34, LM34A	-50°F to +300°F
LM34C, LM34CA	-40°F to +230°F
LM34D	+32°F to +212°F

DC Electrical Characteristics (Notes 2, 7)

Parameter	Conditions	LM34A			LM34CA			Units (Max)
		Typical	Tested Limit (Note 5)	Design Limit (Note 6)	Typical	Tested Limit (Note 5)	Design Limit (Note 6)	
Accuracy (Note 8)	T _A = +77°F	±0.4	±1.0		±0.4	±1.0		°F
	T _A = 0°F	±0.6			±0.6		±2.0	°F
	T _A = T _{MAX}	±0.8	±2.0		±0.8	±2.0		°F
	T _A = T _{MIN}	±0.8	±2.0		±0.8		±3.0	°F
Nonlinearity (Note 9)	T _{MIN} ≤ T _A ≤ T _{MAX}	±0.35		±0.7	±0.30		±0.6	°F
Sensor Gain (Average Slope)	T _{MIN} ≤ T _A ≤ T _{MAX}	+10.0	+9.9, +10.1		+10.0		+9.9, +10.1	mV/°F, min mV/°F, max
Load Regulation (Note 4)	T _A = +77°F	±0.4	±1.0		±0.4	±1.0		mV/mA
	T _{MIN} ≤ T _A ≤ T _{MAX} 0 ≤ I _L ≤ 1 mA	±0.5		±3.0	±0.5		±3.0	mV/mA
Line Regulation (Note 4)	T _A = +77°F	±0.01	±0.05		±0.01	±0.05		mV/V
	5V ≤ V _S ≤ 30V	±0.02		±0.1	±0.02		±0.1	mV/V
Quiescent Current (Note 10)	V _S = +5V, +77°F	75	90		75	90		μA
	V _S = +5V	131		160	116		139	μA
	V _S = +30V, +77°F	76	92		76	92		μA
	V _S = +30V	132		163	117		142	μA
Change of Quiescent Current (Note 4)	4V ≤ V _S ≤ 30V, +77°F	+0.5	2.0		0.5	2.0		μA
	5V ≤ V _S ≤ 30V	+1.0		3.0	1.0		3.0	μA
Temperature Coefficient of Quiescent Current		+0.30		+0.5	+0.30		+0.5	μA/°F
Minimum Temperature for Rated Accuracy	In circuit of Figure 1, I _L = 0	+3.0		+5.0	+3.0		+5.0	°F
Long-Term Stability	T _J = T _{MAX} for 1000 hours	±0.16			±0.16			°F

Parameter	Conditions	LM34			LM34C, LM34D			Units (Max)
		Typical	Tested Limit (Note 5)	Design Limit (Note 6)	Typical	Tested Limit (Note 5)	Design Limit (Note 6)	
Accuracy, LM34, LM34C (Note 8)	$T_A = +77^{\circ}\text{F}$	± 0.8	± 2.0		± 0.8	± 2.0		$^{\circ}\text{F}$
	$T_A = 0^{\circ}\text{F}$	± 1.0			± 1.0		± 3.0	$^{\circ}\text{F}$
	$T_A = T_{\text{MAX}}$	± 1.6	± 3.0		± 1.6		± 3.0	$^{\circ}\text{F}$
	$T_A = T_{\text{MIN}}$	± 1.6		± 3.0	± 1.6		± 4.0	$^{\circ}\text{F}$
Accuracy, LM34D (Note 8)	$T_A = +77^{\circ}\text{F}$				± 1.2	± 3.0		$^{\circ}\text{F}$
	$T_A = T_{\text{MAX}}$				± 1.8		± 4.0	$^{\circ}\text{F}$
	$T_A = T_{\text{MIN}}$				± 1.8		± 4.0	$^{\circ}\text{F}$
Nonlinearity (Note 9)	$T_{\text{MIN}} \leq T_A \leq T_{\text{MAX}}$	± 0.6		± 1.0	± 0.4		± 1.0	$^{\circ}\text{F}$
Sensor Gain (Average Slope)	$T_{\text{MIN}} \leq T_A \leq T_{\text{MAX}}$	$+10.0$	$+9.8,$ $+10.2$		$+10.0$		$+9.8,$ $+10.2$	mV/ $^{\circ}\text{F}$, min mV/ $^{\circ}\text{F}$, max
Load Regulation (Note 4)	$T_A = +77^{\circ}\text{F}$	± 0.4	± 2.5		± 0.4	± 2.5		mV/mA
	$T_{\text{MIN}} \leq T_A \leq +150^{\circ}\text{F}$	± 0.5		± 6.0	± 0.5		± 6.0	mV/mA
	$0 \leq I_L \leq 1 \text{ mA}$							
Line Regulation (Note 4)	$T_A = +77^{\circ}\text{F}$	± 0.01	± 0.1		± 0.01	± 0.1		mV/V
	$5\text{V} \leq V_S \leq 30\text{V}$	± 0.02		± 0.2	± 0.02		± 0.2	mV/V
Quiescent Current (Note 10)	$V_S = +5\text{V}, +77^{\circ}\text{F}$	75	100		75	100		μA
	$V_S = +5\text{V}$	131		176	116		154	μA
	$V_S = +30\text{V}, +77^{\circ}\text{F}$	76	103		76	103		μA
	$V_S = +30\text{V}$	132		181	117		159	μA
Change of Quiescent Current (Note 4)	$4\text{V} \leq V_S \leq 30\text{V}, +77^{\circ}\text{F}$	+0.5	3.0		0.5	3.0		μA
	$5\text{V} \leq V_S \leq 30\text{V}$	+1.0		5.0	1.0		5.0	μA
Temperature Coefficient of Quiescent Current		+0.30		+0.7	+0.30		+0.7	$\mu\text{A}/^{\circ}\text{F}$
Minimum Temperature for Rated Accuracy	In circuit of Figure 1, $I_L = 0$	+3.0		+5.0	+3.0		+5.0	$^{\circ}\text{F}$
Long-Term Stability	$T_j = T_{\text{MAX}}$ for 1000 hours	± 0.16			± 0.16			$^{\circ}\text{F}$

Note 2: Unless otherwise noted, these specifications apply: $-50^{\circ}\text{F} \leq T_j \leq +300^{\circ}\text{F}$ for the LM34 and LM34A; $-40^{\circ}\text{F} \leq T_j \leq +230^{\circ}\text{F}$ for the LM34C and LM34CA; and $+32^{\circ}\text{F} \leq T_j \leq +212^{\circ}\text{F}$ for the LM34D. $V_S = +5 \text{ Vdc}$ and $I_{\text{LOAD}} = 50 \mu\text{A}$ in the circuit of Figure 2; $+6 \text{ Vdc}$ for LM34 and LM34A for $230^{\circ}\text{F} \leq T_j \leq 300^{\circ}\text{F}$. These specifications also apply from $+5^{\circ}\text{F}$ to T_{MAX} in the circuit of Figure 1.

Note 3: Thermal resistance of the TO-46 package is 720°F/W junction to ambient and 43°F/W junction to case. Thermal resistance of the TO-92 package is 324°F/W junction to ambient. Thermal resistance of the small outline molded package is 400°F/W junction to ambient. For additional thermal resistance information see table in the Typical Applications section.

Note 4: Regulation is measured at constant junction temperature using pulse testing with a low duty cycle. Changes in output due to heating effects can be computed by multiplying the internal dissipation by the thermal resistance.

Note 5: Tested limits are guaranteed and 100% tested in production.

Note 6: Design limits are guaranteed (but not 100% production tested) over the indicated temperature and supply voltage ranges. These limits are not used to calculate outgoing quality levels.

Note 7: Specification in **BOLDFACE TYPE** apply over the full rated temperature range.

Note 8: Accuracy is defined as the error between the output voltage and $10 \text{ mV}/^{\circ}\text{F}$ times the device's case temperature at specified conditions of voltage, current, and temperature (expressed in $^{\circ}\text{F}$).

Note 9: Nonlinearity is defined as the deviation of the output-voltage-versus-temperature curve from the best-fit straight line over the device's rated temperature range.

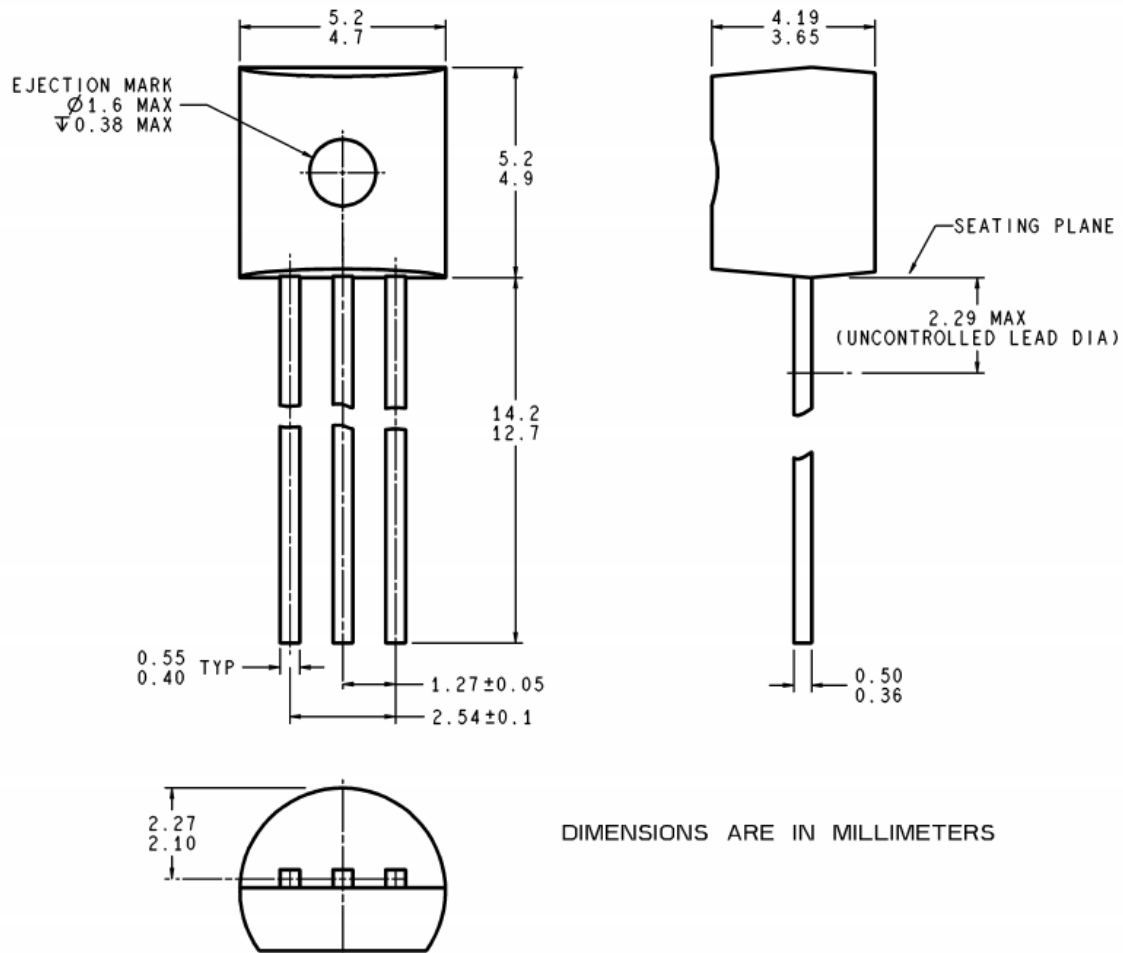
Note 10: Quiescent current is defined in the circuit of Figure 1.

Note 11: Absolute Maximum Ratings indicate limits beyond which damage to the device may occur. DC and AC electrical specifications do not apply when operating the device beyond its rated operating conditions (Note 2).

Note 12: Human body model, 100 pF discharged through a $1.5 \text{ k}\Omega$ resistor.

Note 13: See AN-450 "Surface Mounting Methods and Their Effect on Product Reliability" or the section titled "Surface Mount" found in a current National Semiconductor Linear Data Book for other methods of soldering surface mount devices.

Physical Dimensions inches (millimeters) unless otherwise noted (Continued)



Order Number LM34CZ, LM34CAZ or LM34DZ
 NS Package Z03A

Typical Applications

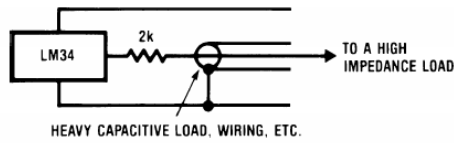


FIGURE 3. LM34 with Decoupling from Capacitive Load

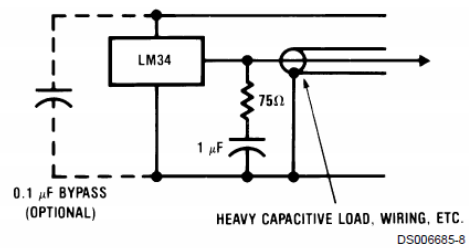


FIGURE 4. LM34 with R-C Damper

Temperature Rise of LM34 Due to Self-Heating (Thermal Resistance)

Conditions	TO-46, No Heat Sink	TO-46, Small Heat Fin (Note 14)	TO-92, No Heat Sink	TO-92, Small Heat Fin (Note 15)	SO-8 No Heat Sink	SO-8 Small Heat Fin (Note 15)
Still air	720°F/W	180°F/W	324°F/W	252°F/W	400°F/W	200°F/W
Moving air	180°F/W	72°F/W	162°F/W	126°F/W	190°F/W	160°F/W
Still oil	180°F/W	72°F/W	162°F/W	126°F/W		
Stirred oil	90°F/W	54°F/W	81°F/W	72°F/W		
(Clamped to metal, infinite heat sink)		(43°F/W)				(95°F/W)

Note 14: Wakefield type 201 or 1" disc of 0.020" sheet brass, soldered to case, or similar.

Note 15: TO-92 and SO-8 packages glued and leads soldered to 1" square of 1/16" printed circuit board with 2 oz copper foil, or similar.

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.



National Semiconductor Corporation
Americas
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com
www.national.com

National Semiconductor Europe
Fax: +49 (0) 180-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 69 9508 6208
English Tel: +44 (0) 870 24 0 2171
Français Tel: +33 (0) 1 41 91 8790

National Semiconductor Asia Pacific Customer Response Group
Tel: 65-2544466
Fax: 65-2504466
Email: ap.support@nsc.com

National Semiconductor Japan Ltd.
Tel: 81-3-5639-7560
Fax: 81-3-5639-7507

Appendix: MS takes Bow Shot at console applications

With Windows 7 Microsoft took a bow shot at all console based applications. They reduced the functionality of "**SetConsoleCtrlHandler()**", "**atexit()**" and "**_onexit()**" such that console based applications can no longer do a proper shutdown when the OS closes the application. In all previous versions of the Windows operating system programmers could use any of the above to determine that the Operating system was going to terminate the console application. There was also a special key code that was generated. With Windows Vista they reduced the application's time to respond to that event to approximately two seconds. In Windows 7 *(and later versions)* Microsoft has eliminated that capability entirely. The most disastrous result of this action is that console based applications may be closed with the cached data never having been written to disk *(the OS does NOT flush the write buffers it has implemented on behalf of the application)*.

Microsoft's response to this issue is for developer/programmer to rewrite the application as Windows based GUI *(even if they only use the GUI Window to launch a CUI window)*.

With one modification of the core operating system code Microsoft has seriously compromised the viability of all console applications ever written as well as making it extremely difficult or at least overly complex to create a viable console application for any future Microsoft OS. There seems to be some controversy if this was an internal marketing decision or just a monumental goof on Microsoft's part. Personally I think the answer to that question is obvious.