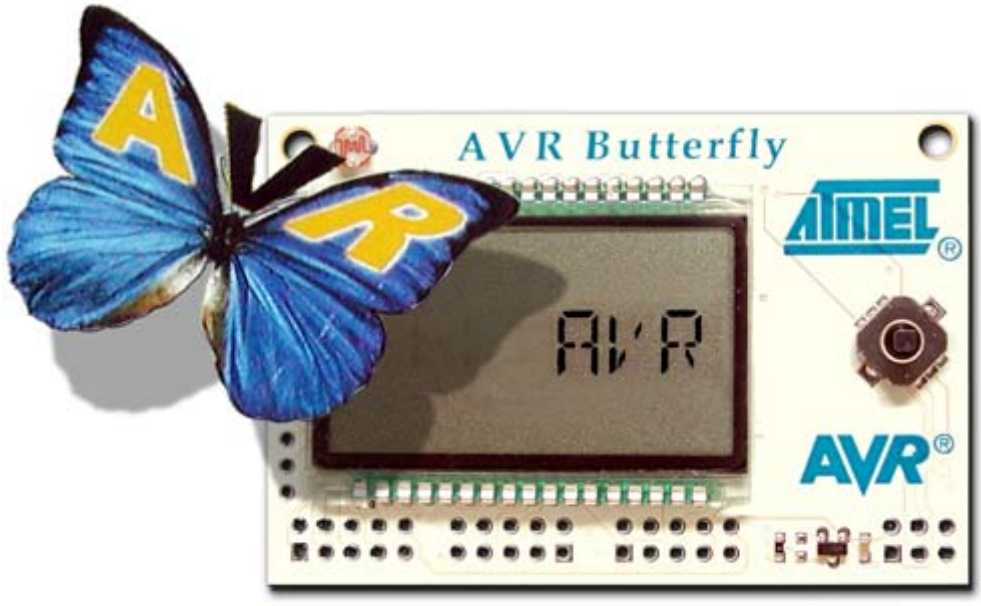
C Programming for Microcontrollers

Quick Start Guide: The Next Generation



Joe Pardue

SmileyMicros.com

Published by Smiley Micros - October 2008

Smiley Micros
5601 Timbercrest Trail
Knoxville, TN 37909
Email: book@SmileyMicros.com
Web: http://www.SmileyMicros.com

For Marcia

God only knows what I'd be without you

Chapter 0: Introduction To the Next Generation.

PLEASE READ THIS: The upgraded project source code located in the www.smileymicros.com downloads menu item ‘C Projects Source Code Next Generation’ will not exactly match the original source code in the book. The differences are mainly in header files and register name changes that are of no import to learning C.

The original Quick Start Guide was a barely modified reprint of the first three chapter of the book *C Programming for Microcontrollers* by Joe Pardue. It was intended to allow the reader to get a feel for the book before purchasing. It also provided, in Chapter 2, a guide to getting the Butterfly set up for use with the free WinAVR C compiler toolset and AVRStudio (and a few other things). But time moves on and the tools have changed enough to warrant a new Quick Start Guide. Chapters 1 and 3 are the same as in the book. Chapter 2 is totally rewritten.

Chapter 1: Introduction

C Programming and microcontrollers are two big topics, practically continental in size, and like continents, are easy to get lost in. Combining the two is a little like traipsing from Alaska to Tierra del Fuego. Chances are you'll get totally lost and if the natives don't eat you, your infected blisters will make you want to sit and pout. I've been down this road so much that I probably have my own personal rut etched in the metaphorical soil, and I can point to all the sharp rocks I've stepped on, all the branches that have whacked me in the face, and the bushes from which the predators leapt. If you get the image of a raggedy bum stumbling through the jungle, you've got me right. Consider this book a combination roadmap, guidebook, and emergency first aid kit for your journey into this fascinating, but sometimes dangerous world.

I highly recommend that you get the book, 'The C Programming Language – second edition' by Kernighan and Ritchie, here after referred to as K&R. Dennis Ritchie, Figure 1, wrote C, and his book is the definitive source on all things C.

Figure 1: C being invented

In Figure 1, Dennis Ritchie, inventor of the C programming language stands next to Ken Thompson, original inventor of Unix, designing the original Unix operating system at Bell Labs on a PDP-11

I have chosen to follow that book's organization in this book's structure. The main difference is that their book is machine independent and gives lots of examples based on manipulating text, while this book is machine dependent, specifically based on the AVR microcontroller, and the examples are as microcontroller oriented as I can make them.

Why C?

Back in the dark ages of microprocessors, software development was done exclusively in the specific assembly language of the specific device. These assembly languages were character based 'mnemonic' substitutions for the numerical machine language codes. Instead of writing something like: 0x12 0x07 0xA4 0x8F to get the device to load a value into a memory location, you could write something like: MOV 22 MYBUFFER+7. The assembler would translate that statement into the machine language for you. I've written code in machine language (as a learning experiment) and believe me when I tell you that assembly language is a major step up in productivity. But a device's assembly language is tied to the device and the way the device works. They are hard to master, and become obsolete for you the moment you change microcontroller families. They are specific purpose languages that work only on specific microprocessors. C is a general-purpose programming language that can work on any microprocessor that has a C compiler written for it. C abstracts the concepts of what a computer does and provides a text based logical and readable way to get computers to do what computers do. Once you learn C, you can move easily between microcontroller families, write software much faster, and create code that is much easier to understand and maintain.

Why AVR?

As microprocessors evolved, devices increased in complexity with new hardware and new instructions to accomplish new tasks. These microprocessors became known as CISC or Complex Instruction Set Computers. Complex is often an understatement; some of the CISCs that I've worked with have mind-numbingly complex instruction sets. Some of the devices have so many instructions that it becomes difficult to figure out the most efficient way to do anything that isn't built into the hardware.

Then somebody figured that if they designed a very simple core processor that only did a few things but did them very fast and efficiently, they could make a much cheaper and easier to program computer. Thus was born the RISC, Reduced Instruction Set Computers. The downside was that you had to write additional assembly language software to do all the things that the CISC computer had built in. For instance, instead of calling a divide instruction in a CISC device, you would have to do a series of subtractions to accomplish a division using a RISC device. This ‘disadvantage’ was offset by price and speed, and is completely irrelevant when you program with C since the complier generates the assembly code for you.

Although I’ll admit that ‘CISC versus RISC’ and ‘C versus assembly language’ arguments often seem more like religious warfare than logical discourse, I have come to believe that the AVR, a RISC device, programmed in C is the best way to microcontroller salvation (halleluiah brother).

The folks that designed the AVR as a RISC architecture and instruction set while keeping C programming language in mind. In fact they worked with C compiler designers from IAR to help them with the hardware design to help optimize it for C programming.

Since this is an introductory text I won’t go into all the detailed reasons I’ve chosen the AVR, I’ll just state that I have a lot of experience with other microcontrollers such as Intel’s 8051, Motorola’s 68xxxes, Zilog’s Z’s, and Microchip’s PIC’s and I’m done with them (unless adequately paid – hey, I’m no zealot). These devices are all good, but they require expensive development boards, expensive programming boards, and expensive software development tools (don’t believe them about the ‘free’ software, in most cases the ‘free’ is for code size or time limited versions).

The AVR is fast, cheap, in-circuit programmable, and development software can be had for FREE (really free, not crippled or limited in any way). I’ve paid thousands of dollars for development boards, programming boards, and C compilers for the other devices, but never again -- I like free. The hardware used in this text, the ATMEL Butterfly Evaluation Board can be modified with a few components to turn it into a decent development system and the Butterfly and

needed components can be had for less than $40.00 (See Appendix 1 Project Kits). You can't get a better development system for 10 times this price and you can pay 100 times this and not get as good.

Okay, maybe I am a zealot.

Goals

What I hope to accomplish is to help you learn **some** C programming on **a specific** microcontroller and provide you with enough foundation knowledge that you can go off on your own somewhat prepared to tackle the plethora (don't you just love that word, say it 10 times real quick) of microcontrollers and C programming systems that infest the planet.

Both C programming and microcontrollers are best learned while doing projects. I've tried to provide projects that are both useful and enhance the learning process, but I've got to admit that many of the early projects are pretty lame and are put in mainly to help you learn C syntax and methods.

Suggested Prerequisites:

- You should be able to use Windows applications.
- You should have an elementary knowledge of electronics, or at least be willing to study some tutorials as you go along so that you'll know things like why you need to use a resistor when you light up an LED.
- I've received lots of suggestions about what needs to be in this book. Some folks are adamant that one must first learn assembly language and microcrocontroller architecture and basic electronics and digital logic and bla bla bla before even attempting C on microcontrollers. I politely disagree and say that you should just jump right in and learn what's fun for you. You'll run across lots of stuff that you will want to learn about, but I won't cover in the book so you should be able to bracket your ignorance (and mine) making a note when you hit something you don't know but would like to. Then you can learn it later. I'm using lots of things that aren't directly relevant to C programming (like communicating with a microcontroller from a PC using a serial port or like what the heck is that transistor motor driver thingee…). If you get really curious, then GOOGLE for a tutorial on the topic.

By the time you complete the text and projects you will:

- Have an intermediate understanding of the C programming language.
- Have a elementary understanding microcontroller architecture.
- Be able to use the WinAVR and AVR Studio tools to build programs.
- Be able to use C to develop microcontroller functions such as:
    - Port Inputs and Outputs
    - Read a joystick
    - Use timers
    - Program a Real Time Clock
    - Communicate with a PC
    - Conduct analog to digital and digital to analog conversions
    - Measure temperature, light, and voltage
    - Control motors
    - Make music
    - Control the LCD
    - Flash LEDs like crazy

On the CD you will find the ATMEL ATMEGA169 data book. At 364 pages, it is the comprehensive source of information for the microcontroller used on the AVR Butterfly board. Open it on your PC with Adobe Acrobat and look around a bit: intimidating isn't it? But don't worry; one of the purposes of this text is to give you enough knowledge so that you can winnow the wheat from the chaff in the data book and pull out what you need for your C based control applications.

I know how easy it is to get bogged down in all the detail and lose momentum on this journey, so we'll begin with the 'Quick Start' chapter by learning only enough to make something interesting happen: kind of a jet plane ride over the territory. Then we will proceed at a comfortable pace from the simple to the complex using as interesting examples as I can come up with. I'm partial to LEDs so you are going to see a lot of flashing lights before we are through, and hopefully the lights won't be from you passing out from boredom and boinking your head on the keyboard.

Chapter 2: Quick Start Guide

Software

AVRStudio – FREE and darn well worth it.

AVR Studio is provided free by the good folks at Atmel Corporation, who seem to understand that the more helpful free stuff they give developers, the more they will sell their microcontrollers. Actually, this software could cost hundreds and still be darn well worth it, but unless you just really like Norway, don't send them any money, they'll get theirs on the backend when you start buying thousands of AVRs for your next great invention. This guide is based on version 4.14 and if you use a newer version you may find differences in it and our discussions. You can find AVRStudio at www.atmel.com under the AVR Tools & Software section: http://atmel.com/dyn/products/tools_card.asp?tool_id=2725).

WinAVR – Oh, Whenever…

WinAVR is a GCC compiler toolset for Windows that we will use in AVRStudio. We will use this package under the AVR Studio IDE that has a GCC plug-in that finds the WinAVR installation and adapts it to the IDE. You can find WinAVR at: http://sourceforge.net/projects/winavr/ .

Developer Terminal

Figure 2: Developer Terminal shows the PC terminal we will use to communicate with the AVR in our workshop. You can get the terminal installer and related documents from www.smileymicros.com from the Downloads/Developer Terminal menu. The source code in C# and Visual Basic .NET for this terminal is discussed in detail in the book: *Virtual Serial Port Cookbook*, by Joe Pardue available from www.smileymicros.com, Amazon, and Nuts and Volts.

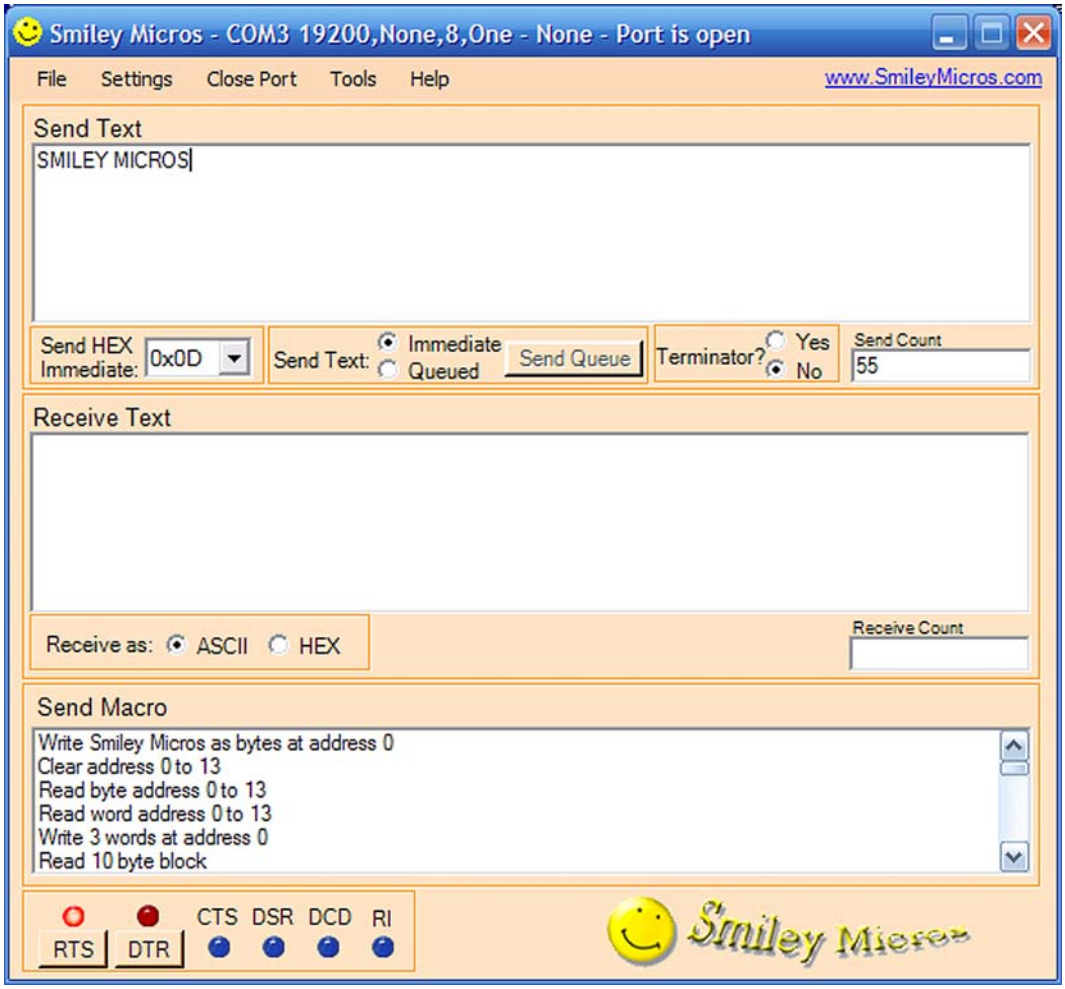


Figure 2: Developer Terminal

Hardware

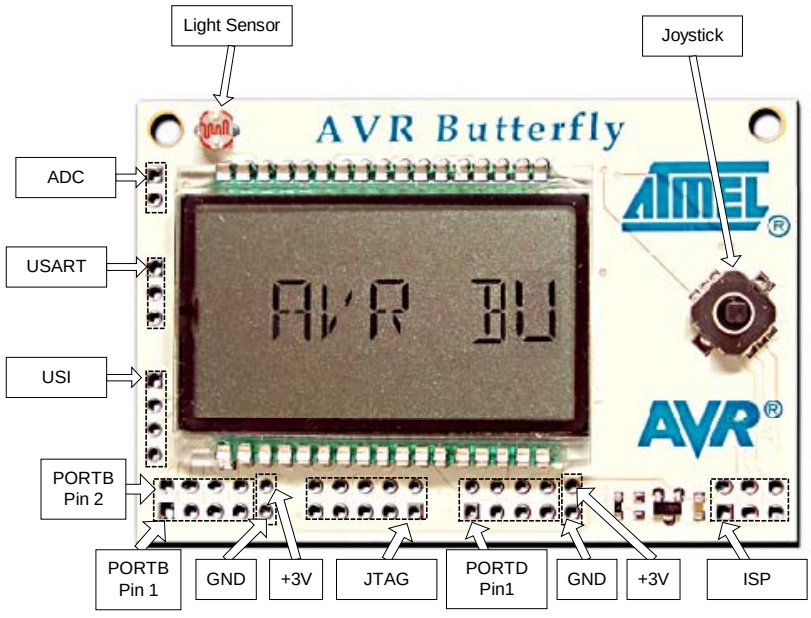


Figure 3: The Butterfly front

It is simply amazing what the Butterfly has built in:

- 100 segment LCD display
- 4 Mbit (that's 512,000 bytes!) dataflash memory
- Real Time Clock 32.768 kHz oscillator
- 4-way joystick, with center push button
- Light sensor
- Temperature sensor
- ADC voltage reading, 0-5V
- Piezo speaker for sound generation
- Header connector pads for access to peripherals
- RS-232 level converter for PC communications
- Bootloader for PC based programming without special hardware
- Pre-programmed demos with source code
- Built-in safety pin for hanging from your shirt (GEEK POWER!)
- Kitchen sink.

I mean this thing has everything (except a kitchen sink… sorry). If anyone can find a learning platform with anywhere near this much for this price, I want to hear about it. If I seem to be raving a bit, get used to it, I do that a lot.

The AVR Butterfly box has instructions to show you how to use the built-in functions. Play with it now before you risk destroying it in the next step. I shudder to think how many of these things will get burned up, blown up, stepped on, and drenched in coffee. And that's just me this morning. After you've seen how it works out of the box, remove the battery and prepare to add components for our learning platform.

Butterfly++ Mini-Kit Construction

Figure 4: Butterfly++ MiniKit from www.smileymicros.com

The Butterfly provides an excellent learning platform, but it can be even better with a few extra parts that Smiley Micros supplies in the Butterfly++ Mini-Kit. This kit includes a CdS Light Sensor, a DB9 female connector and wires, and a 2-AA battery holder with power on LED and resistor.

Adding a CdS LDR Light Sensor

Atmel's AVR Butterfly no longer has a light sensor due to European RoHS compliancy considerations (don't get me started, grrrr…). The Butterfly++

MiniKit provides a suitable substitute component that the user must solder to the Butterfly to use the light sensor function. This sensor works with the existing Butterfly software.

The CdS light sensor is device that has resistance proportional to the incident light. As a resistor, the device has no polarity so either leg can be inserted in the pads shown circled in red. Seat the sensor snug to the top of the Butterfly then solder the legs to the bottom and trim them just above the solder meniscus.

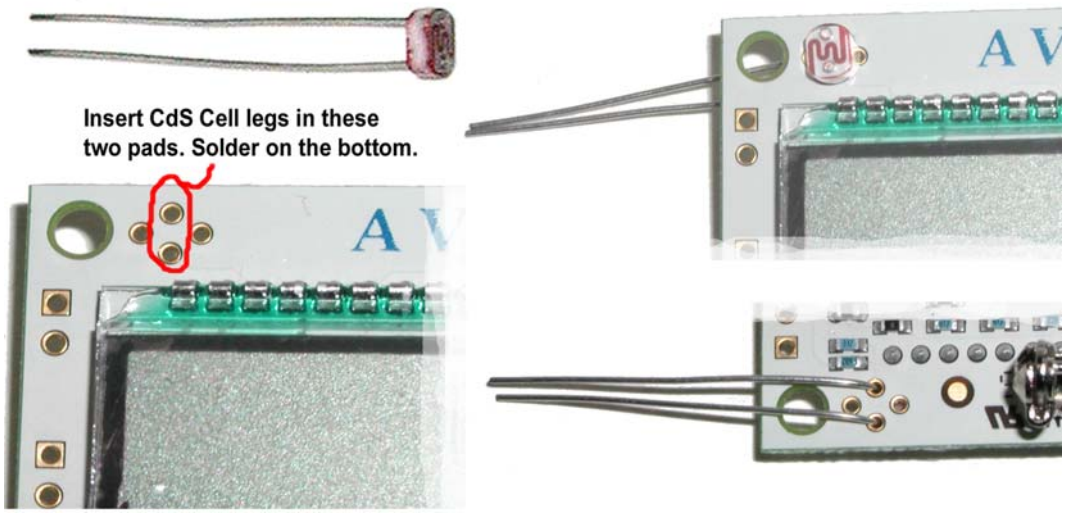


Figure 5: Butterfly CDS Light Meter

DB9 Female connector and wire

In order to communicate with a PC the Butterfly must connect to a serial cable. The mini-kit provides the connector and wire to make the connection to a serial cable.

The Butterfly has built-in RS232 converters for serial communication with a PC. Most serial cables will have a DB9 male connector on the device side that will mate with the provided DB9 Female connector (calm down – it's technical).

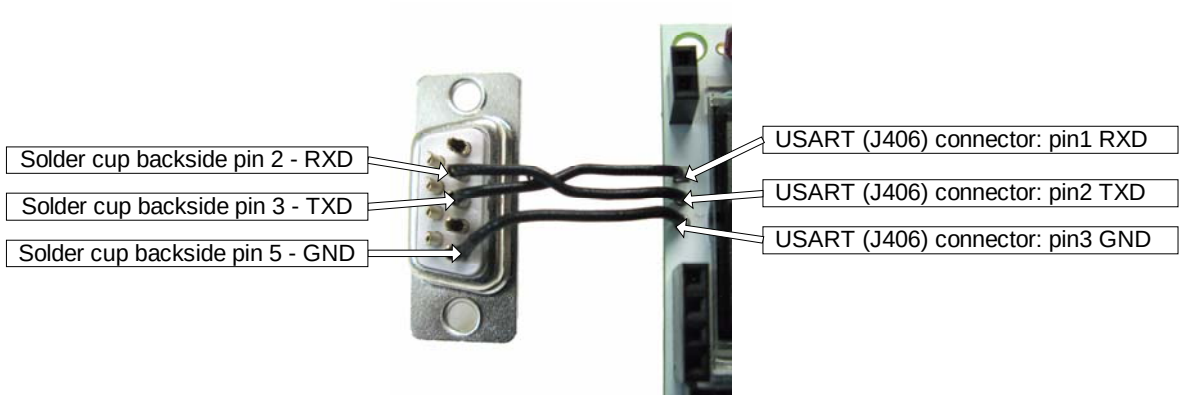


Figure 6: RS232 DB9 Connection

Strip about 1/8 inch from the ends of each wire and then carefully solder them to the Butterfly and the DB9 connector as shown. Notice that the upper wires in the picture cross. Be very careful to get this exactly according to Figure 6: RS232 Wiring; about half the emails I get for problems turn out to be related to either incorrect wiring or poor soldering this component.

Female Headers

Refer to Figure 3: The Butterfly Front to see the location of the ADC, USI, PORTB, and PORTD pads. Solder the 2-pin header to the ADC pads, the 4-pin header to the USI pads, and 2x5 headers to the PORTB and PORTD pads. Notice that Figure 7 shows a male header on the ISP pad, this is not included in the kit and won't be used for our work.

AVR Learning Base Board

The AVR Workshop Learning Platform is built on a foamcore board that lives in a protective foamcore box. Details for the construction can be found in: *Smiley's Workshop 1 Supplement: AVR Learning Platform Foamcore Base and Box* on www.smileymicros.com.

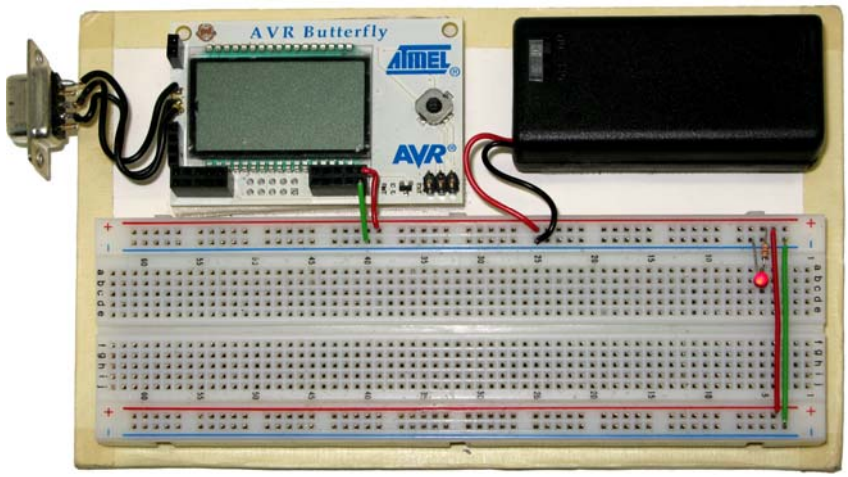


Figure 7: AVR Workshop Learning Platform

You will need to carefully twist the ends of the battery box wires until they are straight then soak them with solder so that it runs up under the insulation to make these wires strong enough to insert into the breadboard power bus. The red wire goes to the + red bus and the black wire goes to the – blue bus. Connect the two power busses with red and green wire and then put an LED with 2.2K-ohm resistor on the breadboard. The resistor goes to the + power, the LED short leg goes to the – power. The LED and resistor are then connected on a breadboard strip as shown.

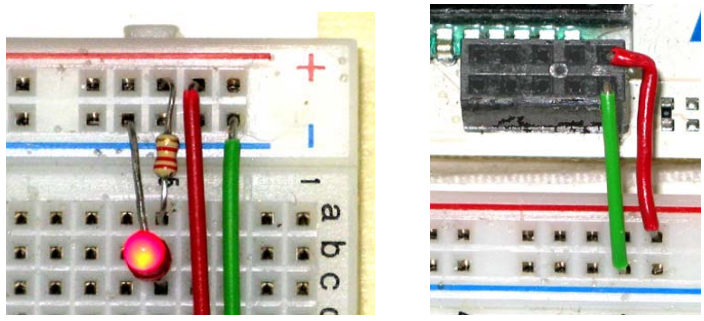


Figure 8: Close up of power connections

The Butterfly receives power from the bus as shown, the + red wire goes to the right most top of the header and the – green wire to the right most bottom of the header.

Test your Connection using Developer Terminal:

Hook your Butterfly DB9 connection to an RS232 cable from a PC. If you use a USB to Serial Converter cable you may have problems if the voltage levels are not robust, but I've used several and not had a problem.

Open Developer Terminal. You can find the user manual (I recommend: RTFM) by clicking the 'Help/Manual'. Click the 'Settings/Port' menu item to open the settings window. Select the RS232 COM port that the Butterfly is connected to. Set the baudrate to 19200, Data Bits to 8, Parity to None, Stop Bits to 1, and Handshaking to None.

You can test that your learning platform is working okay by two methods. A simple test is to turn the power off on the Butterfly and then WITH THE JOYSTICK BUTTON PRESSED TO THE CENTER turn the power back on. Now each time you press the joystick button to the center, you should see as series of question marks: ??????? in the Simple Terminal receive window. This is the Butterfly bootloader wondering what the heck is going on. Now reread this paragraph since another half of my emails are from folks who didn't quite grasp this step.

Another test is to turn the Butterfly on and click the joystick up to get the LCD scrolling. Move the joystick straight down three times till you see 'Name' then move the joystick to the right twice till you see 'Enter name' then move the joystick straight down once and you will see 'Download name' then push down the joystick center for a moment until you see 'Waiting for input'.

In Developer Terminal make sure the 'Send Text: Immediate' radio button is checked. Type in your name, then in the 'Send HEX Immediate' dropdown box, select and click on 0x0D that tells the Butterfly you are finished sending characters. Your name should appear on the LCD.

This isn't easy and there are many opportunities to mess up along the way. Many folks get this going right away, but others seem to have fits getting over this hurdle, so I've provided a Smiley's *Workshop 1 Supplement: Problems Communicating with the Butterfly* (on www.smileymicros.com) to help you get past this point. Trust me, if you see the ????? or get your name scrolling on the

LCD you are over a major hump and subsequent workshops will be easier to get going than this step.

Software

Let's write and compile our first C program using our AVR Learning Platform. This is not going to be the standard wimpy 'Hello World!" of yore, but a zippy software/hardware combination where we create some Cylon eyes. These aren't eyes of the cute sexy Cylons of the recent Battlestar Galactica, but the old fashioned '70s walking chrome toaster Cylons of the original series.

Figure 8: Cylon.

Getting Started With Free Stuff: AVR Studio and WinAVR

AVR Studio provides an IDE for writing, debugging, and simulating programs. We will use the WinAVR GCC C compiler toolset with AVR Studio via plug-in module.

You can find these at:
AVRStudio: http://atmel.com/dyn/products/tools_card.asp?tool_id=2725

WinAVR: http://sourceforge.net/projects/winavr/

Install WinAVR first and AVR Studio second (use the default locations so AVRStudio can find WinAVR – the third half of my emails come from folks who don't use the default locations and wonder why their code doesn't compile).

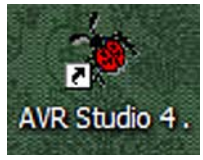


Figure 9: AVR Studio Desktop Icon.

Click on the AVR Studio desktop icon Figure 9. It opens with 'Welcome to AVR Studio 4' Figure 10. Click on the 'New Project' button.

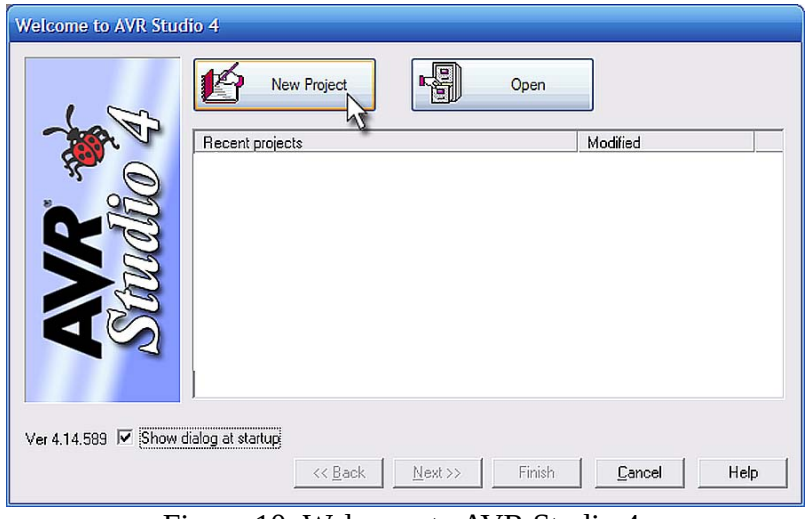


Figure 10: Welcome to AVR Studio 4.

In the 'Create new project' window Figure 11, click on AVR GCC, add the 'Project name': 'CylonEyes' and set the 'Location' to a convenient spot, then click next.

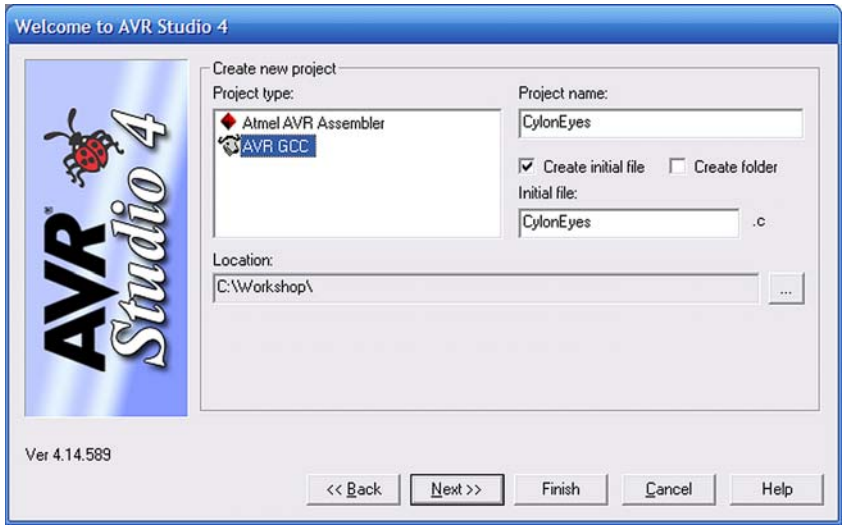


Figure 11: Create New Project.

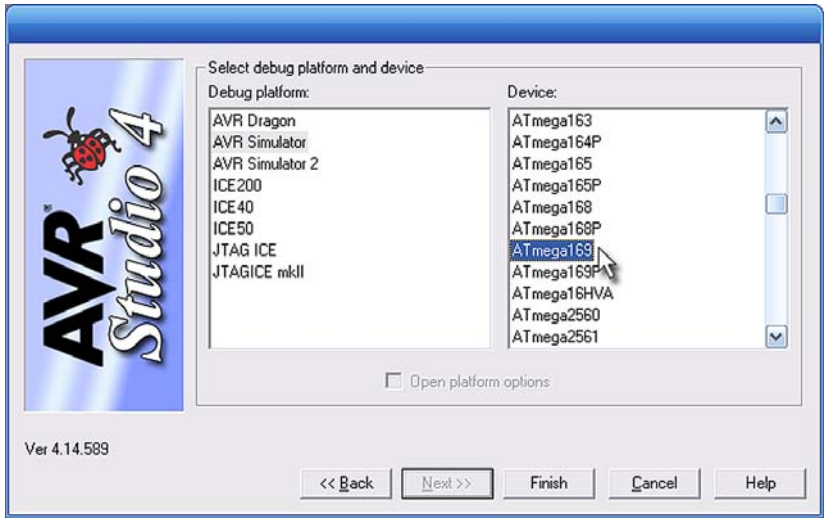


Figure 12: AVR Simulator and ATmega169

Selcect AVR Simulator and ATmega169 as shown in Figure 12.

Figure 13 shows the complex IDE with lots of tools that we won't be using just yet, so try not to have heart palpitations, it will mostly make sense eventually.

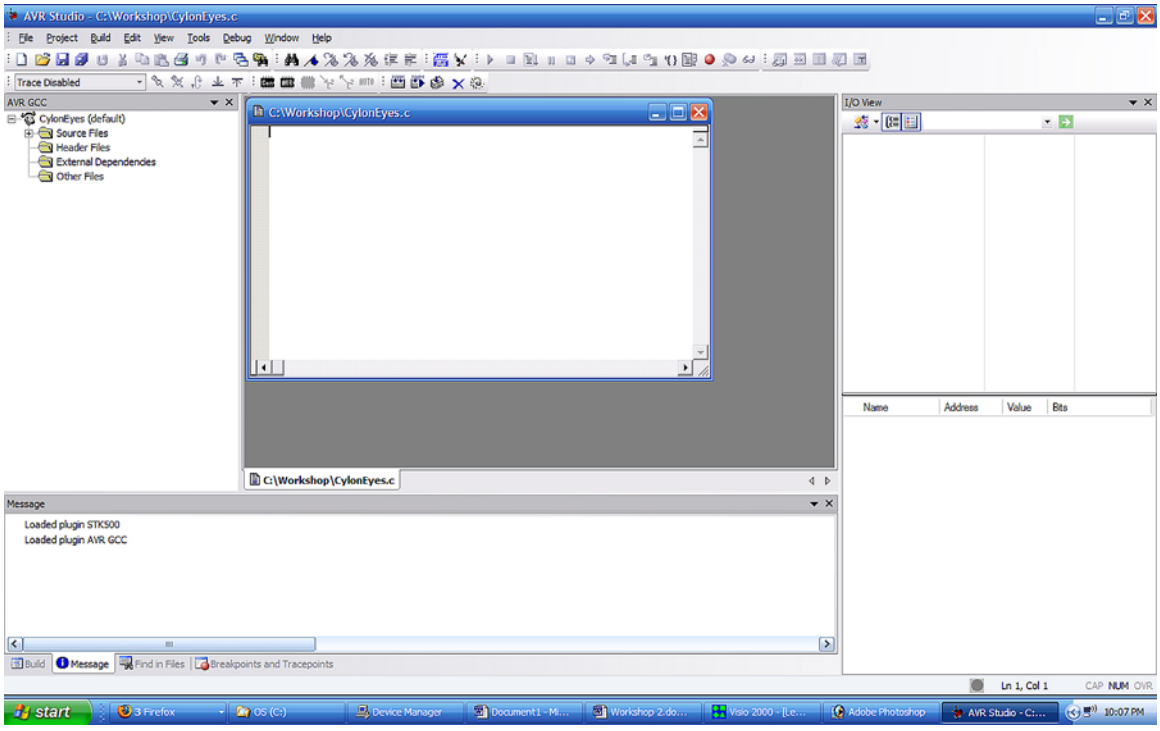


Figure 13: AVR Studio opened with CylonEyes.c.

Writing CylonEyes.c

You might wonder why blinking an LED is the first project, when traditional C programming texts start with a “Hello, world” program. The Butterfly has an LCD that can show the words so it should be easy, but controlling the LCD is much more complex than blinking an LED, so we’ll save the LCD for later when we’ve gotten a better handle on things. Actually, the main reason is that I’m partial to LEDs so you are going to see a lot of flashing lights before we are through, and hopefully the lights won’t be from you passing out from boredom and boinking your head on the keyboard.

You are going to use a lot of code in this series that will have stuff in it that you won’t understand (yet). My reasoning is that by jumping into the deep end, you get to do some interesting things now and you can learn how things work later.

Keep this in mind if you don't understand all of what we are doing here. Eventually you'll see an explanation or at least become more comfortable with mystery - a trait all programmers develop over time.

In the AVR Studio IDE center screen you'll see a text window titled: 'C:\Workshop\CylonEyes.c'. Type the following:

```
// CylonEyes.c

#include <avr/io.h>
#define F_CPU 1000000UL
#include <util/delay.h>

int main (void)
{
   int i = 0;

   // set PORTD for output
   DDRD = 0xFF;

      while(1) {
            for(i = 1; i < 128; i = i*2)
            {
               PORTD = i;
               _delay_loop_2(30000);
            }

            for(i = 128; i > 1; i -= i/2)
            {
               PORTD = i;
               _delay_loop_2(30000);
            }
      }
      return 1;
}
```

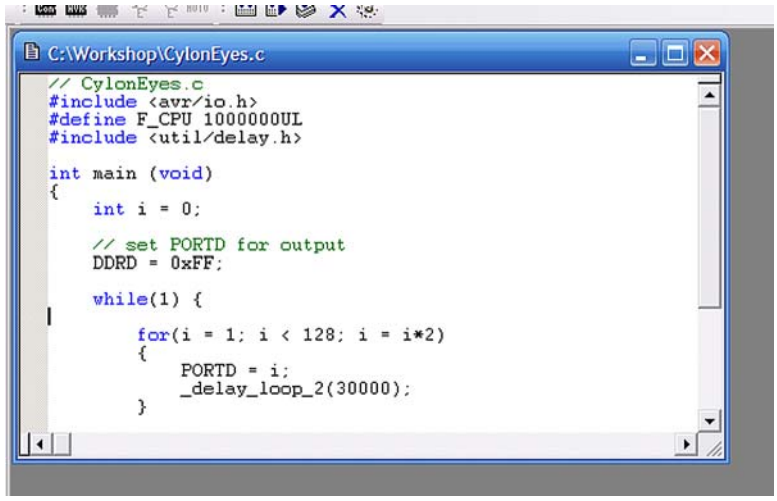


Figure 14: CylonEyes.c

Press the 'Build Active Configuration' button Figure 15 This will generate CylonEyes.hex.

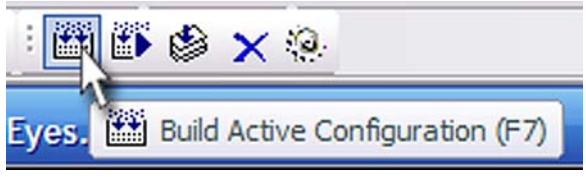


Figure 15: Build Active Configuration.

Download CylonEyes to the Butterfly.

Earlier you hooked the Butterfly to a RS232 cable and downloaded your name. Hook it up again and access the Butterfly bootloader by turning the Butterfly off and then pressing the joystick button to the center and holding it pressed while turning the Butterfly back on. (And remember that the 3rd half of my email comes from folks who mess up this step,)

Back to the AVR Studio, open the Tools menu and WHILE HOLDING JOYSTICK BUTTON PRESSED click the 'AVR Prog…' menu item (you may need to do this twice if you get the 'No Supported Board…' box). In the AVRprog window, browse to find the CylonEyes.hex file. Click on the 'Flash' 'Program'

button. You should see the progress bar zip along and AVR Prog will say: ‘Erasing Programming Verifying OK’.

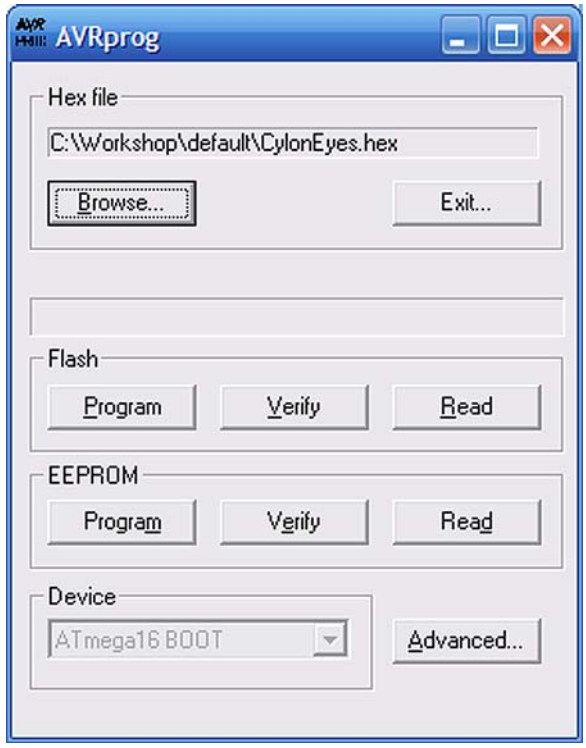


Figure 16: AVR Prog.

If instead of the window shown in Figure 16: AVRProg, you get the dreaded ‘No supported board found’ window shown in Figure 17 – try it again since sometimes it takes two times, but if it still doesn’t work, then you will need to look at ‘*Using AVRProg with the AVR Butterfly* pdf file in the Workshop 2 downloads on www.smileymicros.com. Don’t feel too bad, lots of folks have trouble getting over this hurdle, but once you get it working it is smooth sailing from here on out. Okay, that’s a lie; this stuff is always hard, so be careful, patient, and persistent.

Figure 17: No Supported Board Found.

Building CylonEyes Hardware.

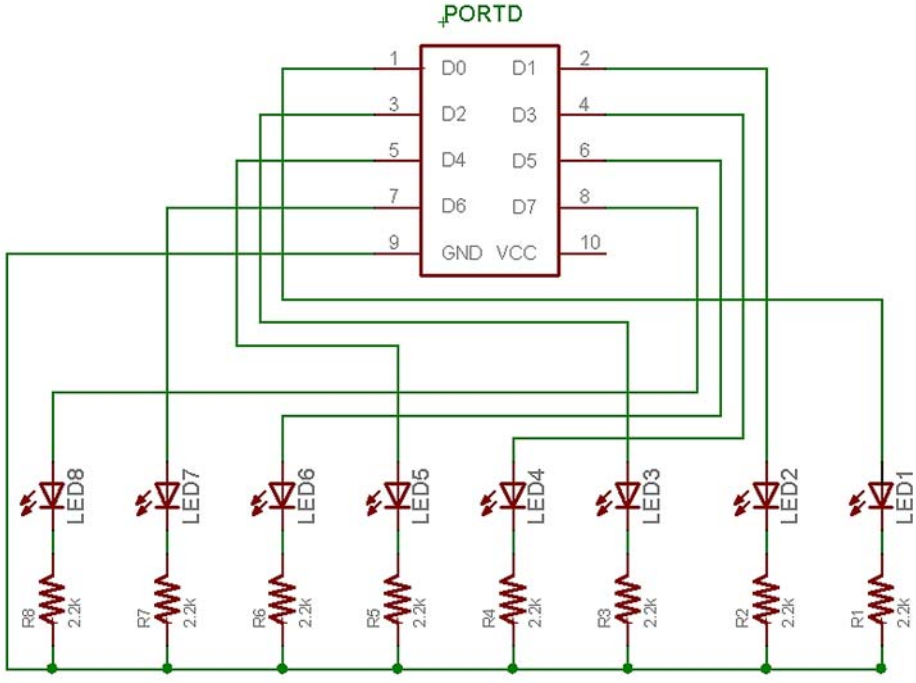


Figure 18: CylonEyes Schematic.

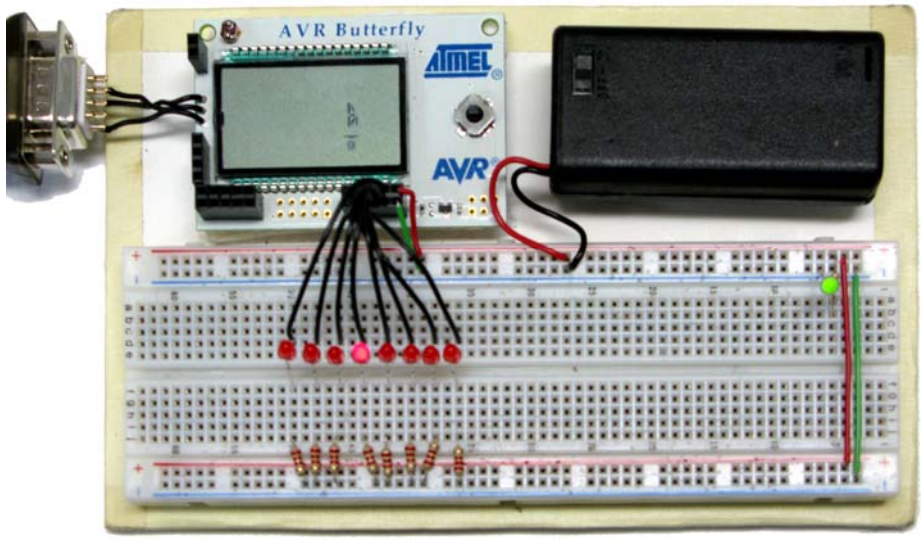


Figure 19: Base board with CylonEyes.

Follow the schematic shown in Figure 18 and as shown in Figure 19. If you haven't done this before, please refer to the supplement: Smiley's Workshop2 Supplement - Breadboards.pdf in the Smiley's Workshop download section of www.smileymicros.com.

Cycle the power, the LCD will be blank, click the joystick up and your LEDs should be making like a Cylon's eyes with the light bouncing back and forth. Cool, huh?

NOTE: the Butterfly LCD dances like crazy with each LED pass, because some of the Port D pins are also tied to the LCD. Will it harm the LCD? Probably not, but I don't know for sure, so don't leave CylonEyes running overnight.

When you compile CylonEyes.c you may suspect that a lot of stuff is going on in the background, and you would be right. Fortunately for us, we don't really need to know how it does what it does. We only need to know how to coax it to do what we need it to do: convert CylonEyes.c into CylonEyes.hex. If you raise the hood on WinAVR you would see a massively complex set of software that has been created over the years by folks involved in the open software movement.

When you compiled CylonEyes.c you may have suspected that a lot of stuff was going on in the background, and you would have been right. The compiler does a lot of things, and fortunately for us, we don't really need to know how it does what it does. We only need to know how to coax it to do what we need it to do, which in our case is convert CylonEyes.c into CylonEyes.hex that we can download to the Butterfly. If you raise the hood on WinAVR you would see a massively complex set of software that has been created over the years by folks involved in the open software movement. When you get a little extra time check out www.sourceforge.net.

When you have questions about WinAVR, and you will, check out the forums on www.AVRFreaks.net, especially the GCC forum, since WinAVR uses GCC to compile the C software. Try searching the forums before asking questions since someone has probably already asked your question and received good responses. Forum helpers tend to get annoyed with newbies who don't do sufficient background research before asking questions.

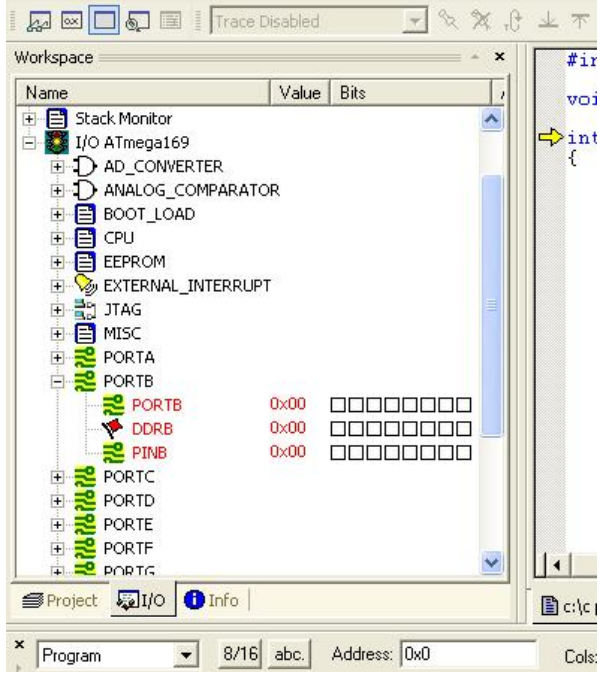
Simulation with AVRStudio

Now that you've gone to the trouble to construct the hardware, and have the burned fingers to prove it… guess what? You didn't need to do any of that to test CylonEyes.c or get an introduction to C programming for microcontrollers. With a minor modification you can run CylonEyes.c in the AVR Studio simulator and learn the introductory C programming ideas in the next chapter without any of the hardware. I decided to do things the hard way, ummm… hardware way because our goal is to control 'real' things like LEDs, not virtual things like little boxes on your PC screen. Theoretically, we could have a whole slew of virtual things to control, from LEDs to motors to full blown Cylon robots wreaking havoc on your screen, which actually sounds kind of fun, but not nearly so much fun as having a real Cylon robot stomping around your neighborhood scaring the noodles out of your enemies. Fun aside, it is often more practical to simulate software before running it in the real world. You wouldn't want your Cylon to mistake you, the Imperious Leader, for an enemy, would you?

The simulator runs your program in a virtual environment that is MUCH slower than the real microcontroller. Most of your code will run plenty fast to simulate,

but some things, such as the delay functions take too long to simulate. In Blinky we call _delay_loop_2(30000); We don't know yet how this function works, but we can guess that we are telling it to do something 30000 times. If we simulate the delay, the simulated LEDs will move at geologic speeds, making glaciers seem fast, so we remove the delay before simulation.

- Open CylonEyes.c in Programmers Notepad and save it to a new directory, SimCylon, as SimCylonc.
- Put comment lines in front of **both** of the _delay_loop_2() function calls in main():
- // _delay_loop_2(30000);
- Open the makefile in the SimCylon directory
- Change the target: TARGET = SimCylon
- Save the makefile to the SimCylon directory
- Run the Make All, then Make Extcoff.
- In the AVRStudio open the SimCylon.cof file.
- In the AVRStudio Workspace window click the I/O ATmega169, then the PORTD, you should see: (the following image shows PORTB instead of PORTD, -- live with it)



- In the toolbar click the AutoStep button:

- The simulator will run showing the LED scan as a scan of the PORTD and PIND items in the Workspace window:(this shows PORTB but you'll actually see PORTD)

- See, I told you it wasn't as much fun as watching real LEDs blink.
- Spend some time with the AVR Studio simulator and associated help files; you'll find the effort well worth it in the long run.

GOOD GRIEF!

That was a 'Quick Start'???? Well, maybe things would go quicker if you wanted to pay a fortune for a software and hardware development system, but for **FREE** software, and unbelievably cheap hardware, you've got to expect to do a little more of the work yourself. Besides, you couldn't pay for all the debugging education I bet you got just trying to follow what I was telling you. If you think the 'Quick Start' section was confusing, you should try reading all the stuff it's based on.

Chapter 3: A Brief Introduction to C – What Makes Blinky Blink?

This section takes a very brief look at Blinky.c to help begin understanding what each line means. Later, these items will be covered in greater detail in context of programs written specifically to aid in learning the C programming language as it is used for common microcontroller applications.

Comments

You can add comments (text the compiler ignores) to your code two ways.

For a single line of comments use double back slashes as in

```
// Blinky.c
```

For multiline comments, begin them with /* and end them with */ as in:

```
/*
Blinky.c is a really great first program for microcontrollers
it causes eight LEDs to scan back and forth like a Cylon’s eyes
*/
```

Include Files

[NOTE: This section is slightly modified from the book.]

```
#include <avr/io.h>
#define F_CPU 1000000l
#include <util/delay.h>
```

The ‘#include’ is a preprocessor directive that instructs the compiler to find the file in the <> brackets and tack it on at the head of the file you are about to compile. The io.h provides data for the port we use, and the delay.h provides the definitions for the delay function we call. The #define F_CPU 1000000l is placed before the delay.h include since that file requires a value for F_CPU in order to create a timed delay.

Operators

Operators are symbols that tell the compiler to do things such as set one variable equal to another, the '=' operator, as in 'DDRB = 0xFF' or the '++' operator for adding 1, as in 'counter++'.

Expressions, Statements, and Blocks

Expressions are combinations of variables, operators, and function calls that produce a single value. For example:

```
PORTD = 0xFF – counter++
```

This is an expression that sets the voltage on pins on Port D to +3V or 0V based on the value of the variable 'counter' subtracted from 0xFF (a hex number - we'll learn about these and ports later). Afterwards the counter is incremented.

Statements control the program flow and consist of keywords, expressions, and other statements. A semicolon ends a statement. For example:

```
TempInCelsius = 5 * (TempInFahrenheit-32)/9;
```

This is a statement that could prove useful if the Butterfly's temperature readings are derived in Fahrenheit, but the user wants to report them in Celsius.

Blocks are compound statements grouped by open and close braces: { }. For example:

```
for(i = 1; i < 128; i = i*2)
{
      PORTD = ~i;
      _delay_loop_2(30000);
}
```

This groups the two inner statements to be run depending on the condition of the 'for' statement which will be explained next.

Flow Control

Flow control statements dictate the order in which a series of actions are performed. For example: ‘for’ causes the program to repeat a block. In Blinky we have:

```
for(i = 1; i < 128; i = i*2)
{
      // Do something
}
```

On the first pass, the compiler evaluates the ‘for’ statement, notes that variable ‘i’ is equal to 1 which is less than 128, so it runs the block of ‘Do something’ code. After running the block the ‘for’ expression is reevaluated with ‘i’ now equal to the previous ‘i’ multiplied by 2 ‘i = i*2’ which is 2 and 2 < 128 is true, so the block is run again. Next loop, i = 4, and so on till i = 128, and ‘128 < 128’ is no longer true, so the program stops running the loop and goes to the next statement following the closing bracket.

Quick now, how many times does this loop run? The series of ‘i’ values evaluated against the ‘< 128’ is ‘1,2,4,8,16,32,64,128’ and since it takes the 128 as the cue to quit, the loop runs 8 times.

The while(‘expression’) statement tests the ‘expression’ to see if it is true (meaning that it is not equal to 0, which is defined as false) and allows the block to run if it is true, then, after running through the loop it retests the ‘expression’, looping thru the block each time it finds the ‘expression’ true. The program skips the block and proceeds to the next statement when the expression becomes false.

```
Int 1 = 0;
while(i < 10)
{
      // load the x array with the y array
      x[i] = y[i++];
}
```

In the example shown, the ‘while’ loop runs 10 times, since the ‘i’ is incremented (has 1 added to the current value) 10 times, this would put the first 10 values of the y array into the x array – more on arrays later.

A while(1) will run the loop forever because '1' is the definition of true (false is defined as 0). In Blinky.c we use this to keep the 'main' function from exiting.

Functions

A function encapsulates a computation. Think of them as building material for C programming. A house might be built of studs, nails, and panels. The architect is assured that all 2x4 studs are the same, as are each of the nails and each of the panels, so there is no need to worry about how to make a 2x4 or a nail or a panel, you just stick them where needed and don't worry how they were made. In the Blinky program, the main() function twice uses the _delay_loop_2() function. The writer of the main() function doesn't need to know **how** the _delay_loop_2(30000) function does its job, he only needs to know **what** it does and what parameters to use, in this case 30000, will cause a delay of about 1/8 second.

The _delay_loop_2() function is declared in the header delay.h and the makefile is set up so that the compiler knows where to look for it.

Encapsulation of code in functions is a key idea in C programming and helps make chunks of code more convenient to use. And just as important, it provides a way to make tested code reusable without having to rewrite it. The idea of function encapsulation is so important in software engineering that the C++ language was developed primarily to formalize these and related concepts and force their use.

The Main() Thing

All C programs must have a 'main' function that contains the code that is first run when the program begins.

```
int main (void)
{
   // Do something
}
```

Blinky has:

```
int main (void)
{
      int i = 0;
      // set PORTD for output
      DDRD
      = 0xFF;

      while(1)
      {
            for(i = 1; i < 128; i = i*2)
            {
               PORTD = i;
               _delay_loop_2(30000);
            }

            for(i = 128; i > 1; i -= i/2)
            {
               PORTD = i;
               _delay_loop_2(30000);
            }
      }
}
```

In this function we leave C for a moment and look at things that are specific to the AVR microcontroller. The line:

```
      DDRD = 0xFF;
```

Sets the microcontroller Data Direction Register D to equal 255. This tells the microcontroller that Port D pins, which are hooked up to our LEDs, are to be used to output voltage states (which we use to turn the LEDs on and off). We use the hexadecimal version, 0xFF, of 255 here because it is easier to understand what's happening. You disagree? Well, by the time you finish this text, you'll be using hexadecimal numbers like a pro and understand they do make working with microcontrollers easier, but for now, just humor me.

The program tests the while(1) and finding it true, proceeds to the 'for' statement, which is also true and passes to the line:

```
            PORTD = i;
```

Which causes the microcontroller to set the Port D pins to light up the LEDs with the value of i.

Say what? Okay, ‘i’ starts off equal to 1, which in binary is 00000001 (like hexadecimal, you’ll grow to love binary). This provides +3V on the rightmost LED, lighting it up and leaves the other LEDs unlit at 0V.

The first ‘for’ loop runs eight times, each time moving the lit LED to the left, then it exits. In the next ‘for’ loop the -= operator subtracts i/2 from i and sets i equal to the results causing the LED to move to the right. When it is finished the loop runs again… for how long? Right… forever. Or at least until either the universe ends or you unplug the Butterfly.

NOTE: the Butterfly LCD dances like crazy with each LED pass, because some of the Port D pins are also tied to the LCD. It’s a bug in our design, but in the world of marketing it would be called a free bonus feature available exclusively to you for an unheard of low price if you act immediately. Will it harm the LCD? Probably not, but I don’t know for sure, so don’t leave Blinky running overnight.

That’s enough for a quickie introduction. We skimmed over a lot that you’ll see in detail later. You now know just enough to be dangerous and I hope the learning process hasn’t caused your forehead to do too much damage to your keyboard.

www.SmileyMicros.com.