Your First AVR C Program

6/6/08 Joe Pardue © 2008

Last month in the first article in this series we:

- Learned where to get the software and hardware you’ll need,
- Built the AVR Butterfly based AVR Learning Platform.
- Tested it with Developer Terminal

This month we will 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 1: 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).

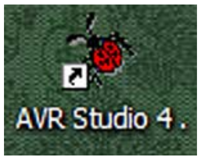


Figure 2: AVR Studio Desktop Icon.

Click on the AVR Studio desktop icon Figure 2. It opens with 'Welcome to AVR Studio 4' Figure 3. Click on the 'New Project' button.

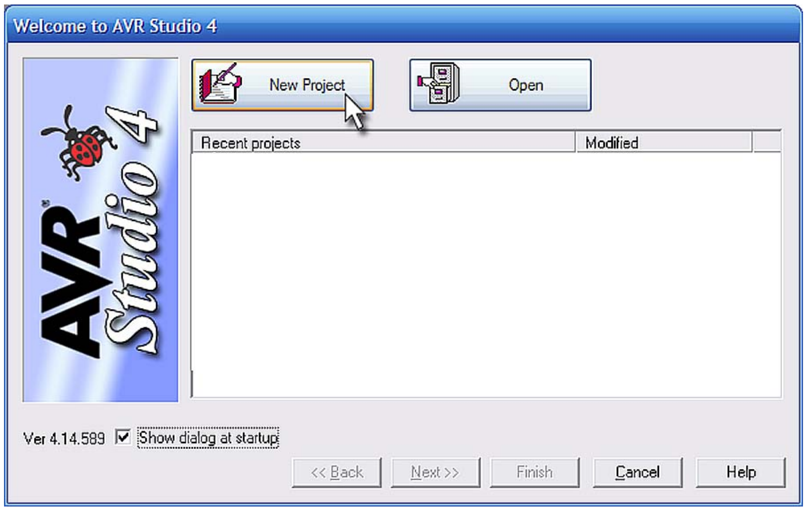
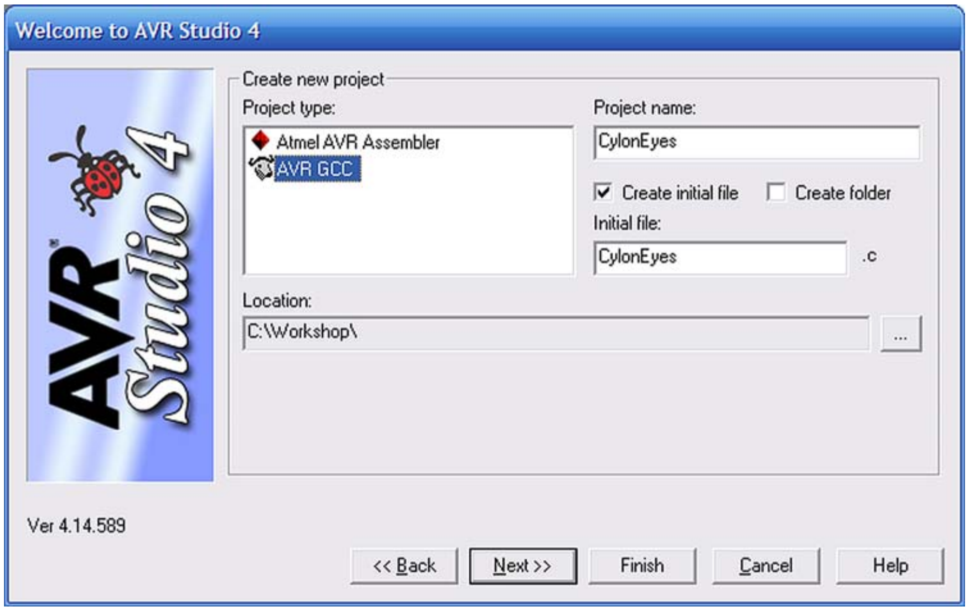


Figure 3: Welcome to AVR Studio 4.

In the 'Create new project' window Figure 4, click on AVR GCC, add the 'Project name': 'CylonEyes' and set the 'Location' to a convenient spot, then click finish.



Create new project

Project type:
- Atmel AVR Assembler
- AVR GCC

Project name: CylonEyes

[✓] Create initial file   [ ] Create folder

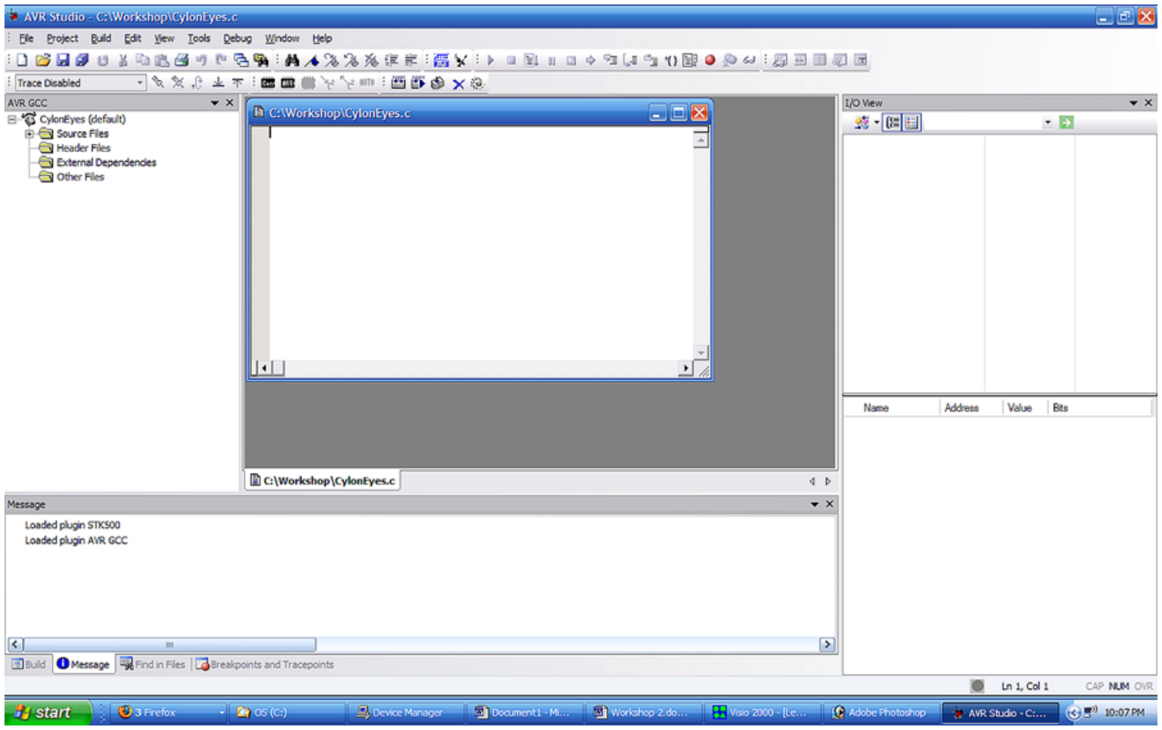
Initial file: CylonEyes .c

Location: C:\Workshop\

Ver 4.14.589

Figure 4: Create New Project.

Figure 5 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.



```
Loaded plugin STK500
Loaded plugin AVR GCC
```

Figure 5: AVR Studio opened with CylonEyes.c.

Setting Up the Project Configuration Options

From the 'Project' menu item select 'Configuration Options'.

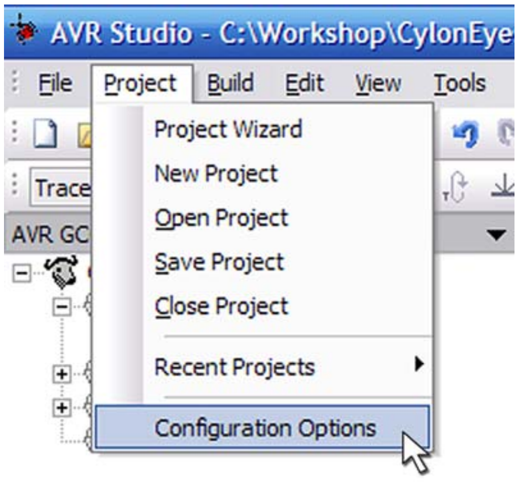
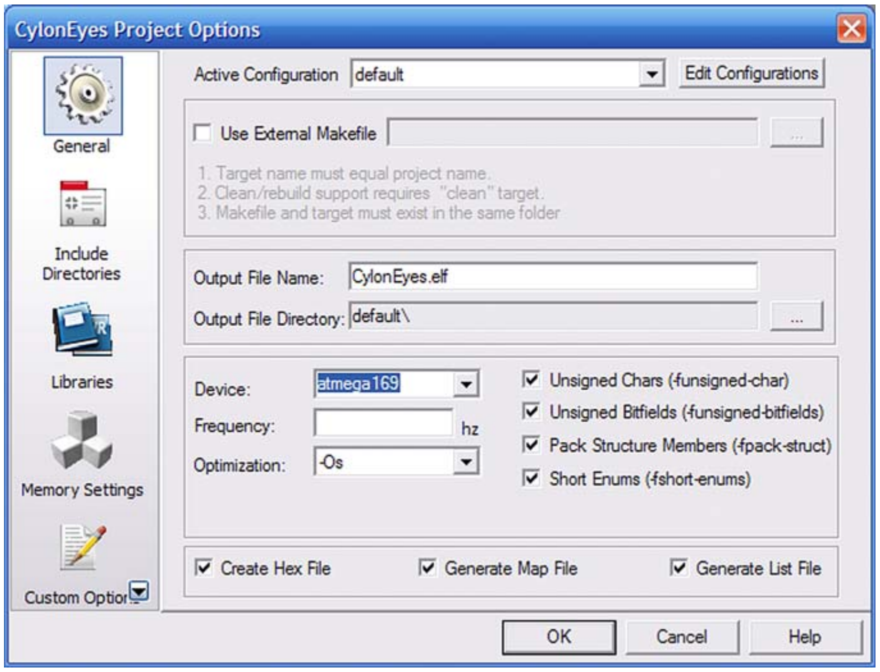


Figure 6: Project Configuration Options.



Active Configuration: default

☐ Use External Makefile

1. Target name must equal project name.
2. Clean/rebuild support requires "clean" target.
3. Makefile and target must exist in the same folder

Output File Name: CylonEyes.elf
Output File Directory: default\

Device: atmega169
Frequency: hz
Optimization: -Os

☑ Unsigned Chars (-funsigned-char)
☑ Unsigned Bitfields (-funsigned-bitfields)
☑ Pack Structure Members (-fpack-struct)
☑ Short Enums (-fshort-enums)

☑ Create Hex File ☑ Generate Map File ☑ Generate List File

Figure 7: Project Options.

In the 'Project Options' window select the atmega169 from the 'Device:' dropdown box.

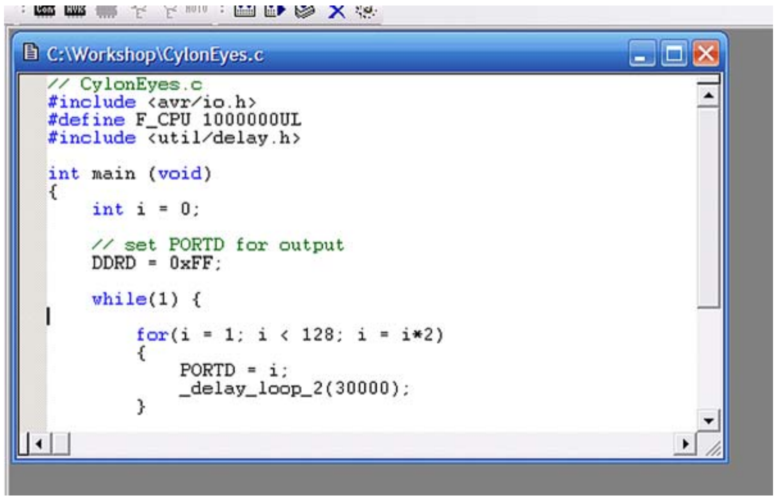
CylonEyes.c

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>

// The last character is a lower case 'L' not a 1 (one)
#define F_CPU 1000000l

#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;
}
```



C:\Workshop\CylonEyes.c

```
// 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);
        }
```

Figure 8: CylonEyes.c

Press the 'Build Active Configuration' button Figure 9. This will generate CylonEyes.hex.

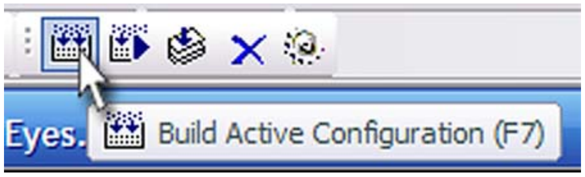
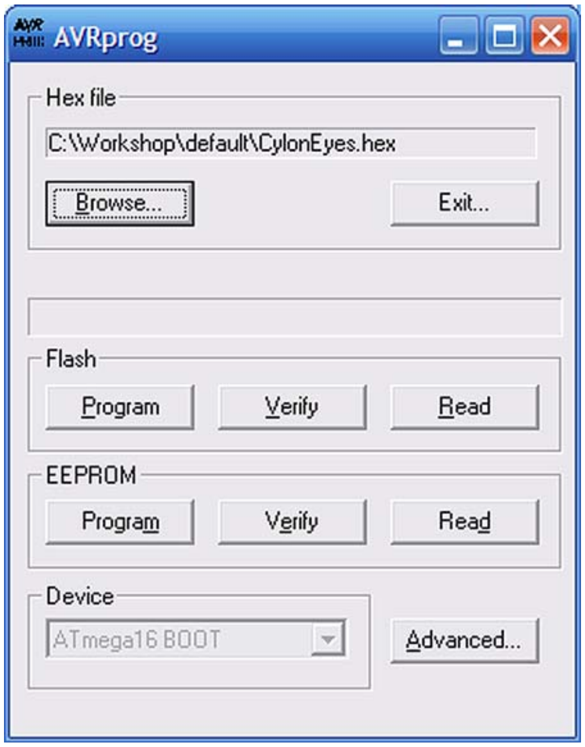


Figure 9: Build Active Configuration.

Download CylonEyes to the Butterfly.

In Workshop 1: AVR C Programming Learning Platform, 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.

Back to the AVR Studio, open the Tools menu and WHILE HOLDING JOYSTICK BUTTON PRESSED click the 'AVR Prog…' menu item. 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'.

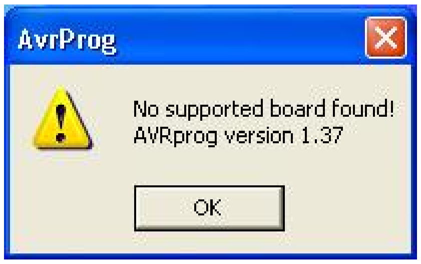


C:\Workshop\default\CylonEyes.hex

ATmega16 BOOT

Figure 10: AVR Prog.

If instead of the window shown in Figure 10: AVRProg, you get the dreaded 'No supported board found' window shown in Figure 11, 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.



No supported board found!
AVRprog version 1.37

Figure 11: No Supported Board Found.

Building CylonEyes Hardware.

We built the AVR Workshop Learning Platform is built in the last workshop. Details for the construction can be found in: *Smiley's Workshop 1 Supplement: AVR Learning Platform Foamcore Base and Box* on www.smileymicros.com.

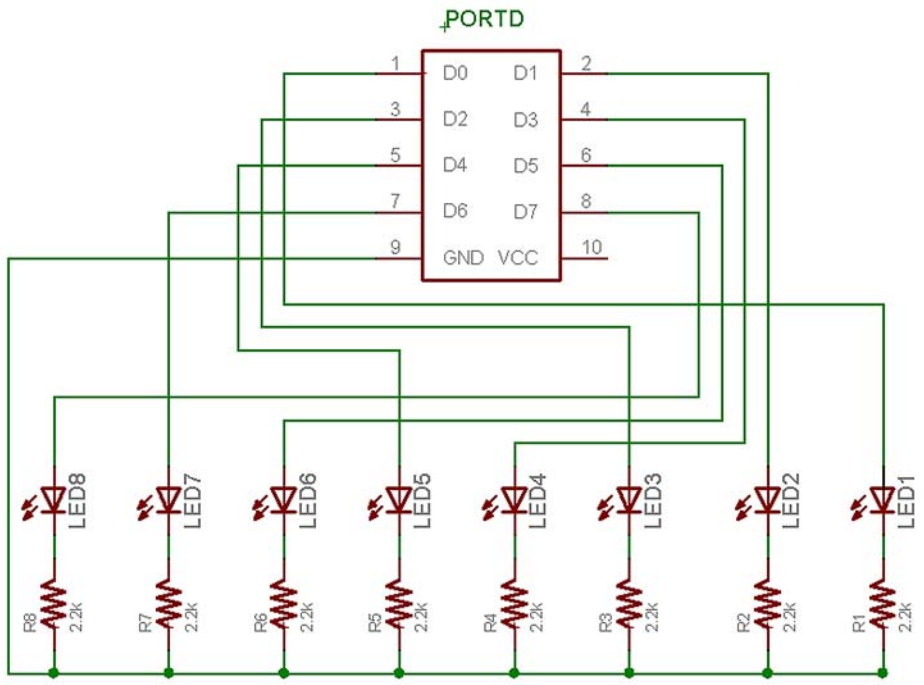


Figure 12: CylonEyes Schematic.

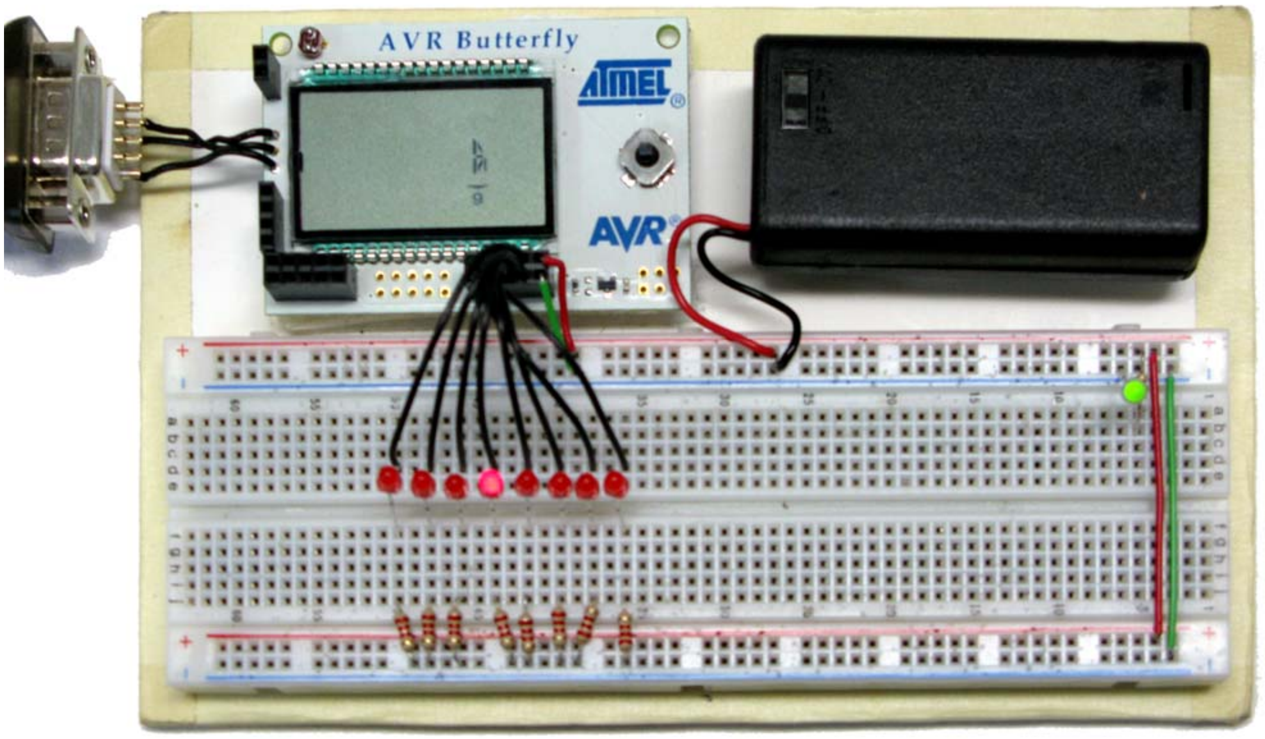


Figure 13: Base board with CylonEyes.

Follow the schematic shown in Figure 12 and as shown in Figure 13. If you haven’t done this before, please refer to the supplement: *Using a breadboard* from 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 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.

A Brief Introduction to C – What Makes CylonEyes Blink those LEDs?

This section takes a very brief look at CylonEyes.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

```
// CylonEyes.c
```

For multiline comments, begin them with /* and end them with */ .

Include Files

```
   #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 CylonEyes we have:

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

On the first pass, the code evaluates the ‘for’ statement, notes that variable ‘i’ is equal to 1 which is less than 128, and runs the block of ‘Do something’ code. Next the ‘for’ expression is reevaluated with ‘i’ now multiplied by 2 ‘i = i*2’ which is 2 and 2 < 128 is true, so the block runs again. Next, i = 4, and so on till i = 128, and ‘128 < 128’ is false. 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 true. After running through the loop it retests the 'expression', looping thru the block each time it is true. The program 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++];
}
```

The 'while' loop runs 10 times, since the 'i' is incremented (has 1 added to the current value) 10 times, putting the first 10 values of the y array into the x array – more on arrays later.

A while(1) runs the loop forever because '1' is true (false is 0). We use this to keep the 'main' function from exiting.

Functions

A function encapsulates a computation. Think of them as building material for C programs. A house might be built of studs, nails, and panels. The architect knows 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 CylonEyes 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 AVRStudio 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
}
```

CylonEyes 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 0xFF. 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, if you persist with these workshops, 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;
```

This causes the microcontroller to set the Port D pins to light up the LEDs with the pattern made by the value of i.

Say what? Okay, ‘i’ starts off equal to 1, which in binary is 00000001. This provides +3V on the rightmost LED, 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.

We skimmed over a lot that you’ll see in detail in later workshops. 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.

This workshop is the second of a series that will introduce C programming for the AVR microcontroller based on the book *C Programming for Microcontrollers* by Joe Pardue available from www.smileymicros.com or Nuts and Volts.