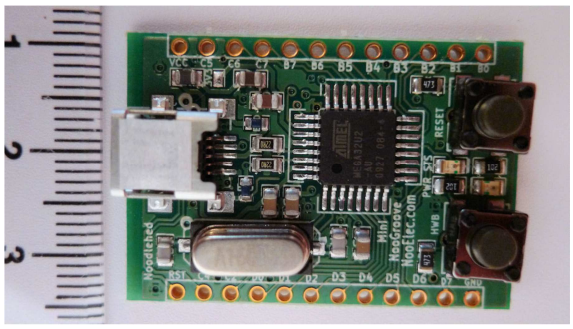
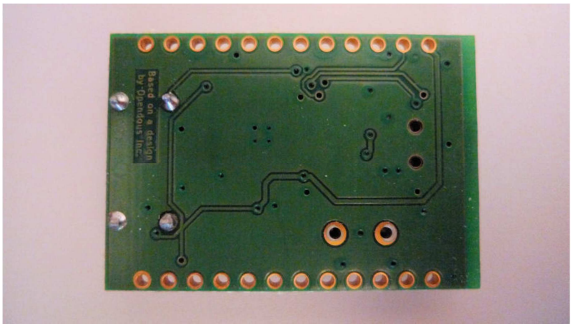
Writing Code and Programming Microcontrollers

This document shows how to develop and program software into microcontrollers. It uses the example of an Atmel ATmega32U2 device and free software.

The ATmega32U2 is ideal, because it doesn't need any programmer, just a USB cable. There are quite a few vendors that sell a ready-made board for $20 or less.





The particular board used in the example is available from http://www.nooelec.com (it is cheaper from eBay; it is not worth spending more than $20 on it). Another board that may be suitable (not purchased) is available from http://www.mattairtech.com

If a board is not available, it is easily constructed, it has very few parts, and the microcontroller is available for less than $5 in a single quantity, from suppliers like Farnell.com. Most boards will not come with a USB cable, so that will need to be obtained.

The steps to using the board are:

- Set up the software development environment
- Write the software (the example in this document will just light an LED)
- Install the programmer software, and use it to program the device

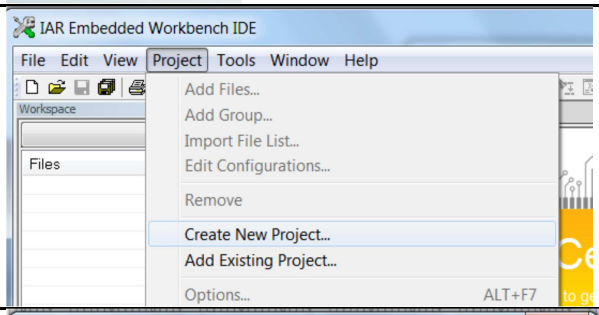
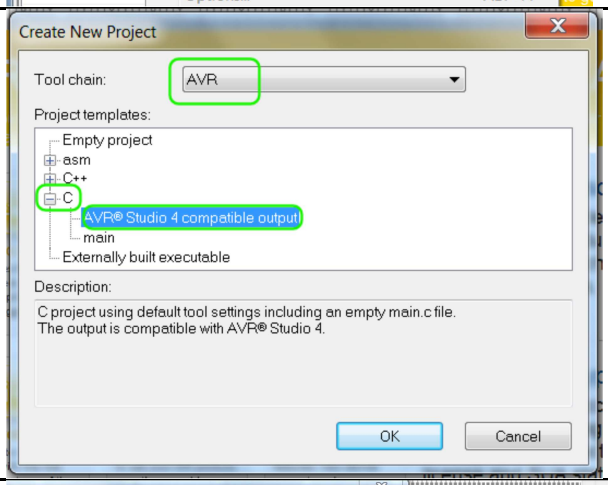
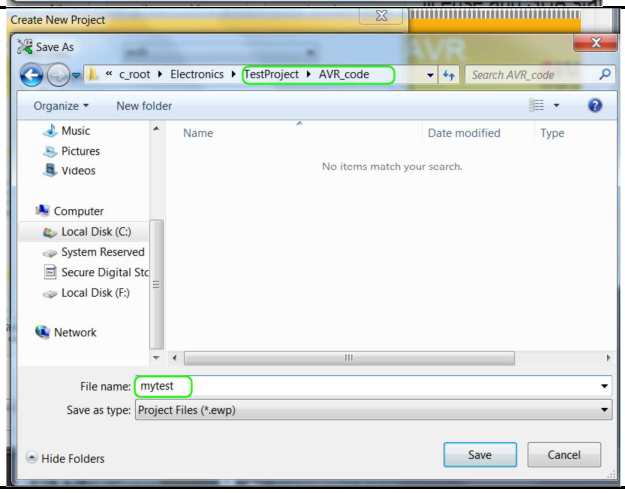
1. Set up the software development environment

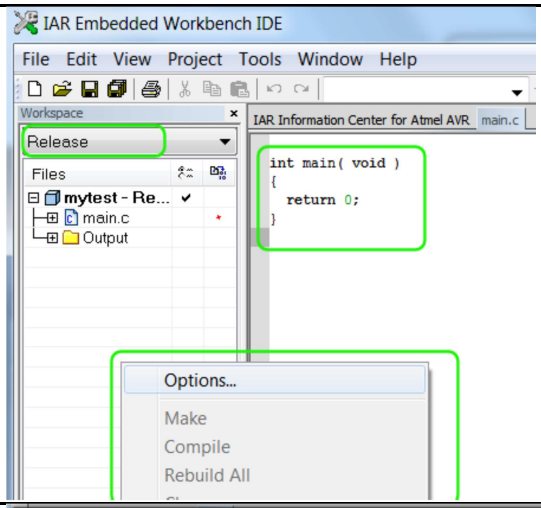
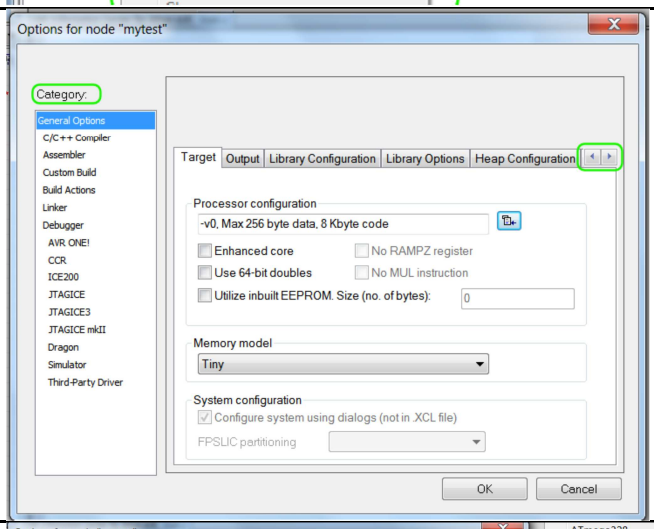
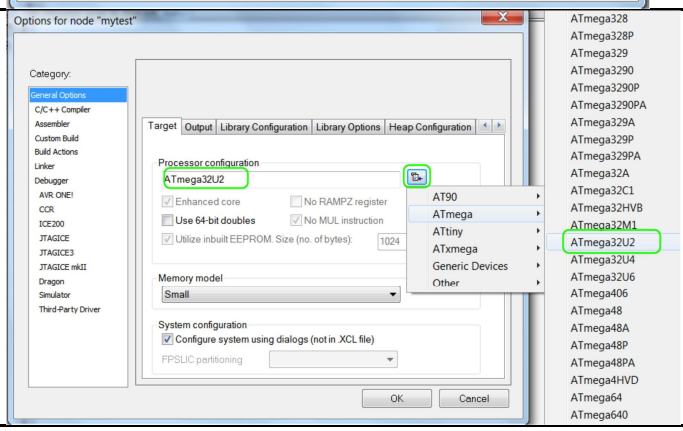
The IAR Embedded Workbench contains a text editor (aka IDE), compiler, assembler and linker. The procedure is:

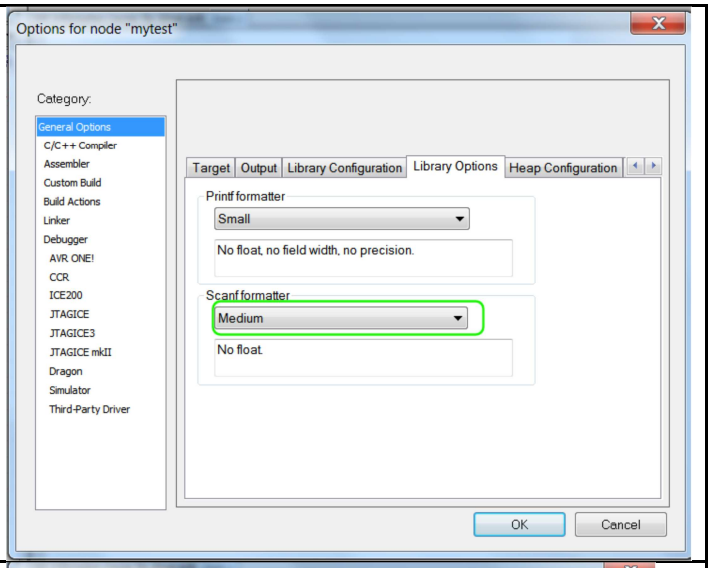
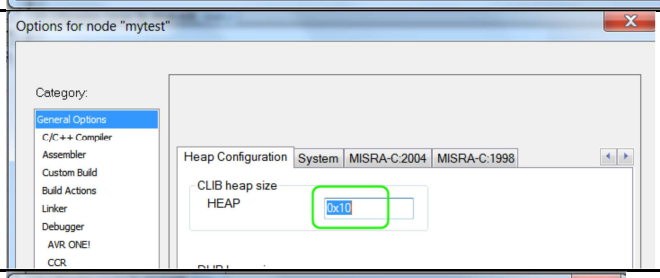
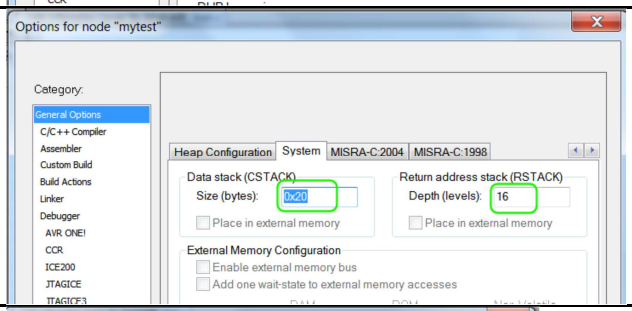
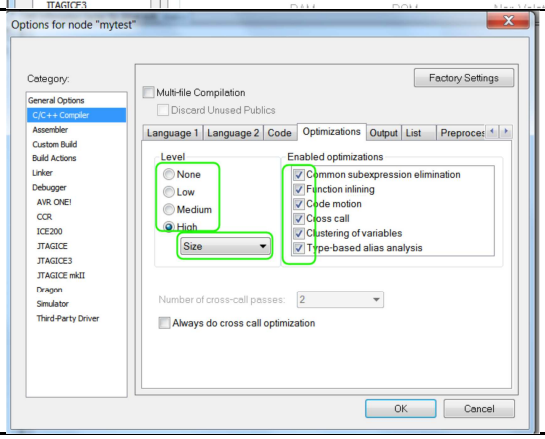
a) Install IAR EW for the microcontroller that you're intending to use
b) Create a project and adjust some settings
c) Write some code
d) Click 'make' to compile, assemble and link the code
e) After this, you're ready to download the code into the device. It is discussed in a later section.

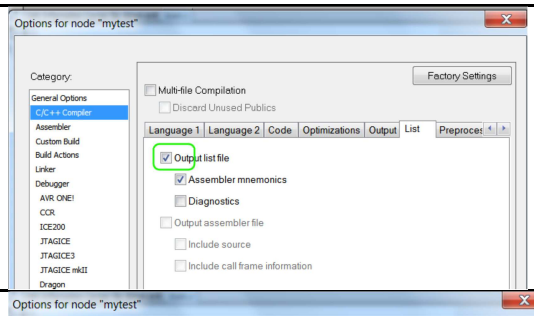
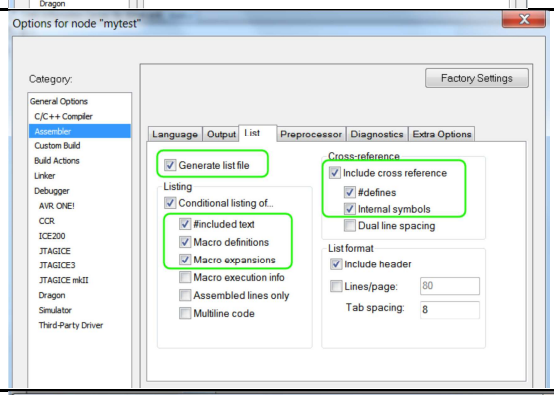
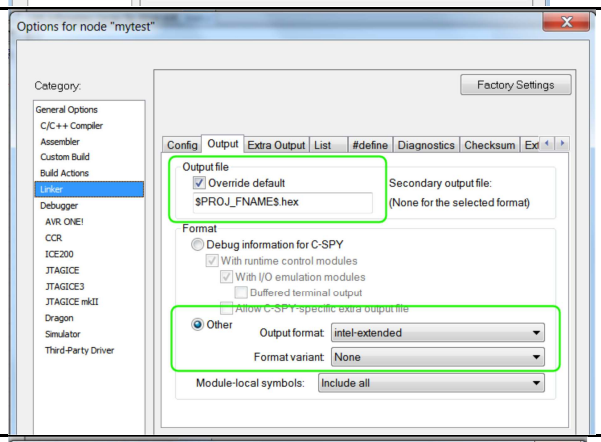
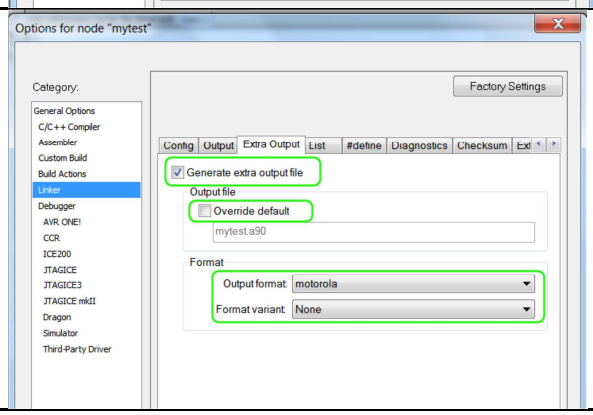
The detailed steps for items a) to d) are all below.

Some people prefer to use the GNU compiler (in the form of the WinAVR software suite) for the AVR devices. It is an excellent compiler. However the IAR compiler is very good too, and I prefer it for small embedded microcontrollers. The free version of the IAR compiler is limited to 4kbytes of executable code however.

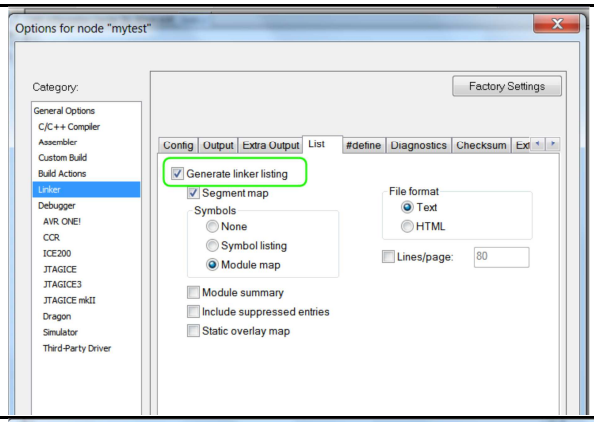
	Steps	Illustrations
1	Download IAR EW Kickstart Edition for AVR, from iar.com and then install it.	
2	Select Project->Create New Project	
3	Ensure AVR is selected, then select C->"AVR Studio 4 compatible output"	
4	Select the project file location. Best to create a folder for the entire project (circuit diagrams, PDFs, etc.) called TestProject in this case, and then a subfolder to contain all the source code and compiled code for the AVR – called AVR_code in this example. The file name was just called mytest but could be called the entire project name, or whatever.	

5	You'll notice that the start of the code has been auto-generated. Select 'Release' (it will initially say 'Debug') and then right-click in the files pane, and then select Options in the pop-up menu.	
6	There are a lot of options. A lot of the categories need to be traversed, and the tabs too, including the ones that are hidden and need to be brought into view by using the right/left buttons. The settings are now described.	
7	Select the correct processor by clicking the little button shown here. In this case, the ATmega -> ATmega32U2 device was selected.	

8	Under Library Options, the scanf formatter may be set to medium by default. You can leave it here, or set it to 'none' if you like. Chances are you will not use it.	
9	The Heap is set to something like 0x10 (16 bytes) by default. This is probably ok. Usually the heap is not used much for small embedded programs in C (the stack is used more; see later). Weird memory problems when the microcontroller is running might be a symptom of too low a heap setting.	
10	The CSTACK and RSTACK values shown here are important. The defaults are ok here, but you may need to increase them if there is a memory related error during linking.	
11	In the "C/C++ Compiler" category, under the Optimizations tab, observe the default settings. They are probably ok. Bear in mind that with some settings, if you have a 'for' loop in your code that does nothing (e.g. to create a delay) the compiler may optimize it out. 'for' delay loops are a bad idea in general anyway, but are sometimes are useful when prototyping.	

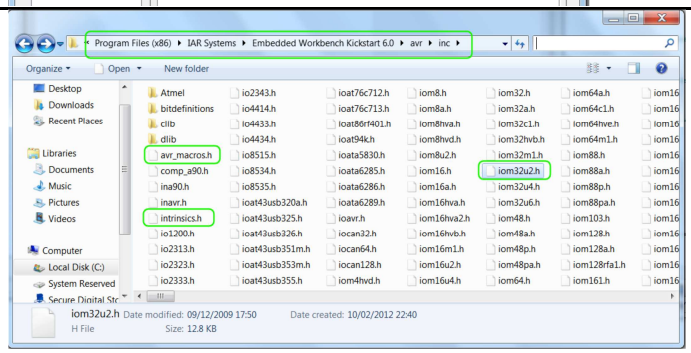
12	In the 'List' tab, ensure "Output list file" is checked. It is not essential, but it does help for debugging sometimes to have the assembler listing, so it's best to always enable it.	
13	In the Assembler category 'List' tab, use the settings shown here. They are extra bits of information that are helpful when debugging directly from the assembler listings. For simple C programs, you will be curious to see the assembler code, so this is helpful. It is best to always enable these.	
14	In the Linker category, under the Output tab, observe the default settings. These should be fine for your programmer.	
15	If you think the programmer might not accept the output, then you can generate another file too. Here it is selected to generate a Motorola format file, which is a common file format for some programmers. There is no harm in generating an extra output file even if you don't need to use it.	

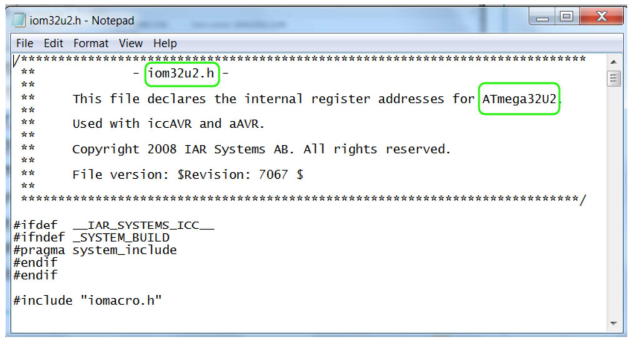
16 It is always a good idea to generate a linker listing, so you can see how big your code is.



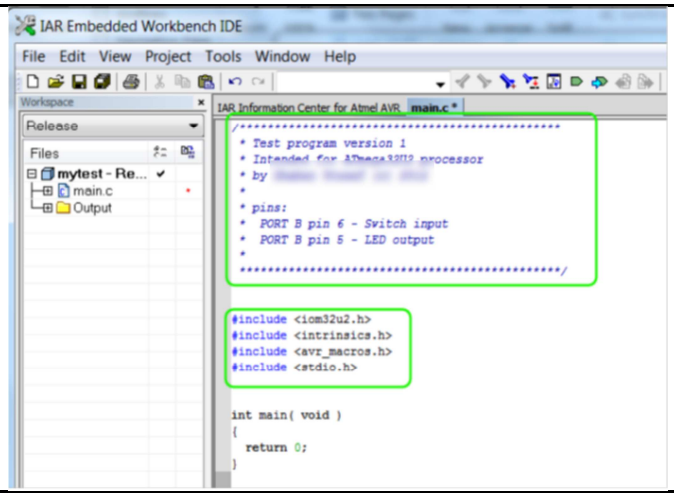
17 Time to write some code. First step is to include some files. In a Windows Explorer window, traverse to where the IAR software got installed (C:\Program Files (x86) in this example) and then navigate deeper into the 'inc' folder as shown here. There are three file names that you are looking for. Note the file names down.

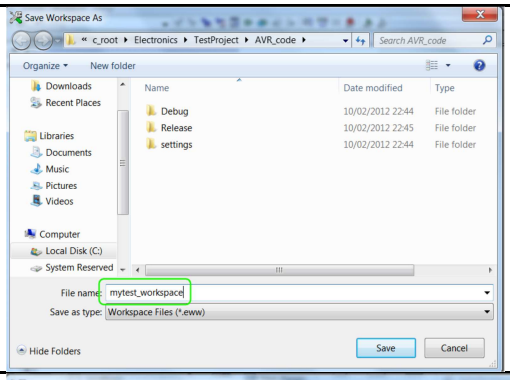
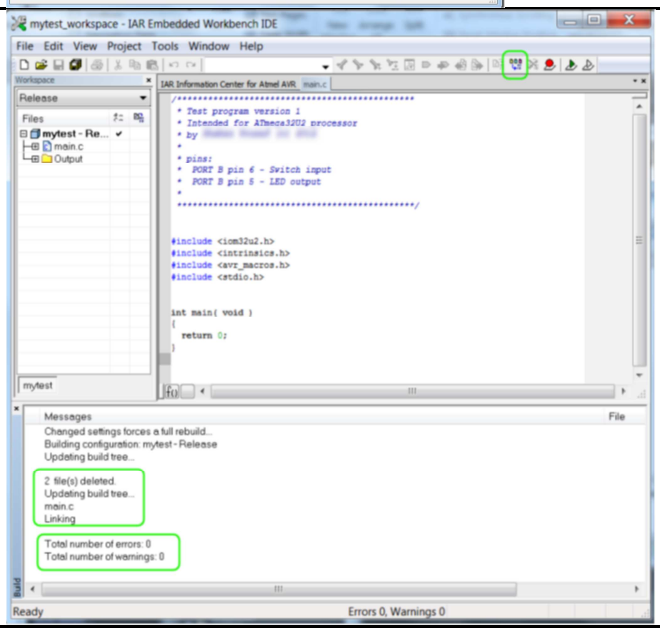
Depending on the microcontroller, they will be different. In this case, the three filenames are:
avr_macros.h
intrinsics.h
iom32u2.h
If you are unsure of the correct filename for the last file, then you can open the file in a text editor (e.g. Notepad) and check what it says. In this case, it looked like this. It matches the microcontroller name (ATmega32U2).





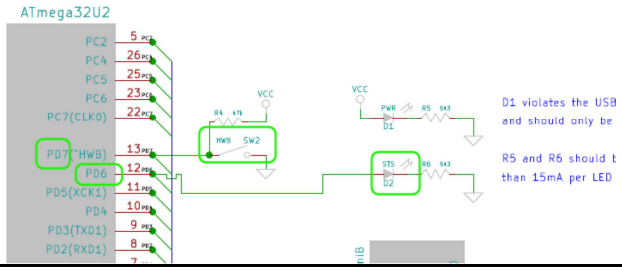
18 The code can now be added to include these file names as shown here. Also a good idea to insert some comments as shown here. I tend to record the microcontroller variant here, and the pins that are being used.



19	Click File -> 'Save All'. When prompted to save the workspace, I tend to give it the same name as the project, but with '_workspace' appended so that the file in this case got saved as mytest_workspace.eww Now, the project file, the workspace file and the first code file (main.c) have all been saved.	
20	As a quick test, try to compile and make the final linked file that can be used by the programmer. Click on the 'make' icon on the toolbar. It looks like this:  (don't click on the 'compile' icon, because that merely does half of what is required). If it is successful, the output will look as shown in the screenshot here. The important things to see are the errors and warnings of course, but for information reasons you can also see that it compiled one file (main.c) and then linked it.	

2. Write a software program

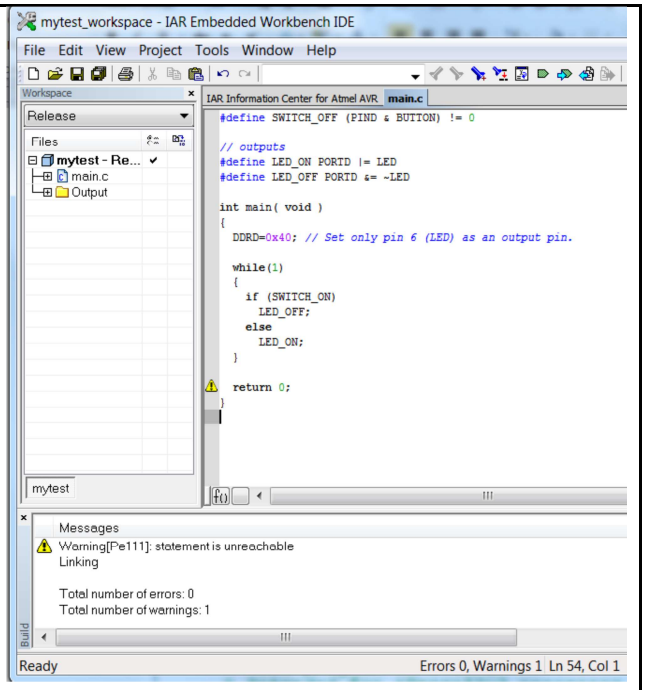
The microcontroller board used contained a couple of LEDs and switches. A simple program can be used to control one on the LEDs with a switch.

	Steps	Illustrations
1	From the circuit diagram of the microcontroller board, you can find some input/outputs. You can see here that the example microcontroller board has an LED connected to pin 12 (PD6) and a switch connected between ground and pin 13 (PD7)	

2 Some code will be written to make the LED turn on, and to turn off the LED when the switch is pressed.

You could copy and paste this code into the IAR EW main.c program file:

```
/*****************************************
 * Test program version 1
 * Intended for ATmega32U2 processor
 * by S. Yousaf
 *
 * pins:
 *  PORT D  6 - LED output
 *  PORT D  7 - Switch output
 *

****************************************/

#include <iom32u2.h>
#include <intrinsics.h>
#include <avr_macros.h>
#include <stdio.h>

// bits
#define BIT0 0x01
#define BIT1 0x02
#define BIT2 0x04
#define BIT3 0x08
#define BIT4 0x10
#define BIT5 0x20
#define BIT6 0x40
#define BIT7 0x80

// port D stuff
#define LED        BIT6
#define BUTTON     BIT7

// inputs
#define SWITCH_ON (PIND & BUTTON) == 0
#define SWITCH_OFF (PIND & BUTTON) != 0

// outputs
#define LED_ON PORTD |= LED
#define LED_OFF PORTD &= ~LED

int main( void )
{
  // Set only pin6 (LED) as an output pin
  DDRD=0x40;

  while(1)
  {
    if (SWITCH_ON)
      LED_OFF;
    else
      LED_ON;
  }

  return 0;
}
```



Save the file, and then click on the 'Make' button. It should state that the number of errors is zero, but there is one warning. The warning can be safely ignored (it is warning that the program will never end, and hence never execute the 'return 0;' statement).

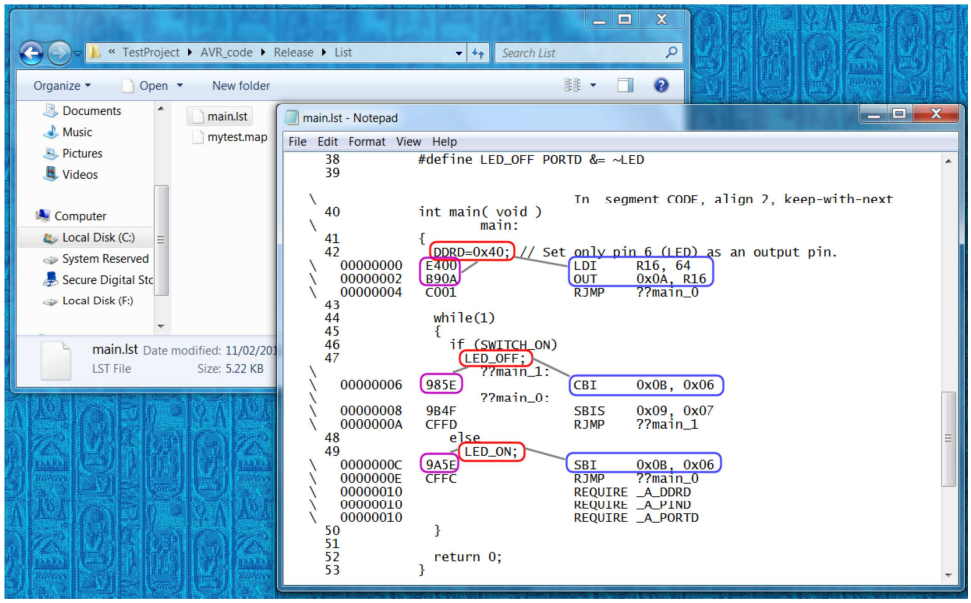
Always get in the habit of saving before clicking on 'Make'!

You are now ready to move on to the next section, to program the microcontroller board. However, you may be curious what happened when you clicked on the 'Make' button.

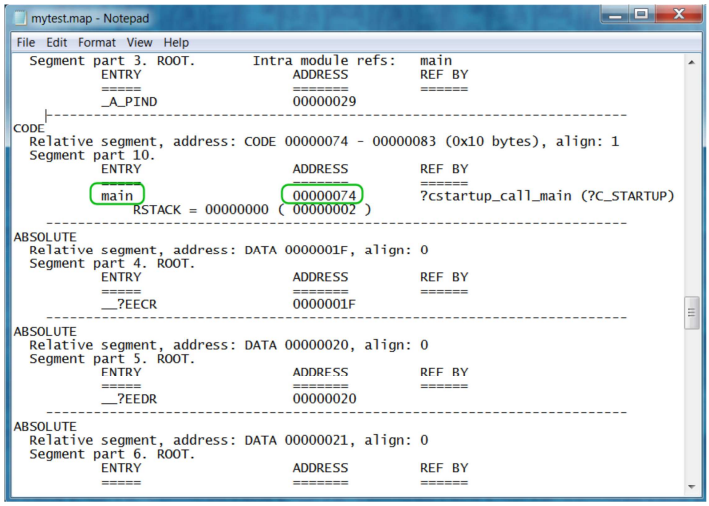
Basically the IAR EW software ran three things; a compiler, an assembler and a linker.

The compiler converted the C code into assembler code. The assembler converted the assembler code into machine code.

If you are curious, then you can go into the correct folder using Windows Explorer as shown here, then you can look at the converted file using Notepad; it will be called 'main.lst'. It shows the result of the compiler and assembler; it shows the C source code interspersed with the assembler code and the corresponding machine instructions (machine code). The screenshot below shows these for three of the C source code lines. You can see that the 'DDRD=0x40;' C code was compiled into two assembler codes (LDI R16, 64 and OUT 0x0A, R16). Each of the two assembler codes was translated into two hex bytes that are the machine code.



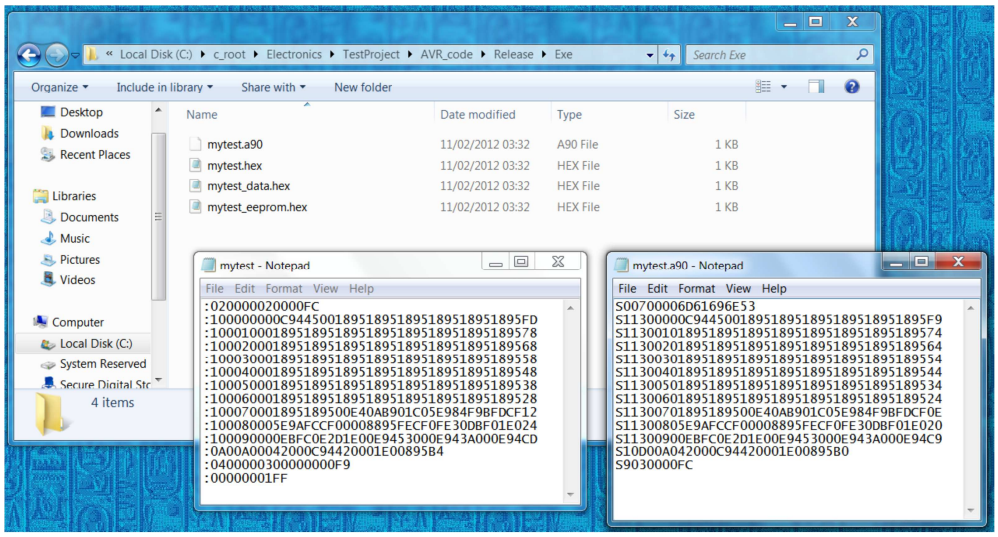
The linker took the assembled code, and took any library files (in our case, we didn't really use any library functions, but there is a so-called 'C run-time' library which all C programs will rely on. It handles how to launch the main program, and how to clean up after the program has run (the example program runs forever). A library file is a type of assembled machine code file. The linker was responsible for deciding where in the microcontroller memory each assembled module should reside. You can see the debug generated by the linker here (this file is in the same folder). It shows that the main function was placed at address 0x74 in hex, and that it is called by the 'cstartup' function which is part of the C run-time library.



```
mytest.map - Notepad
File  Edit  Format  View  Help
  Segment part 3. ROOT.        Intra module refs:   main
            ENTRY                  ADDRESS          REF BY
            =====                  =======          ======
            _A_PIND                00000029
    ├-------------------------------------------------------------------------
CODE
  Relative segment, address: CODE 00000074 - 00000083 (0x10 bytes), align: 1
  Segment part 10.
            ENTRY                  ADDRESS          REF BY
            =====                  =======          ======
            main                   00000074         ?cstartup_call_main (?C_STARTUP)
                RSTACK = 00000000 ( 00000002 )
    -------------------------------------------------------------------------
ABSOLUTE
  Relative segment, address: DATA 0000001F, align: 0
  Segment part 4. ROOT.
            ENTRY                  ADDRESS          REF BY
            =====                  =======          ======
            __?EECR                0000001F
    -------------------------------------------------------------------------
ABSOLUTE
  Relative segment, address: DATA 00000020, align: 0
  Segment part 5. ROOT.
            ENTRY                  ADDRESS          REF BY
            =====                  =======          ======
            __?EEDR                00000020
    -------------------------------------------------------------------------
ABSOLUTE
  Relative segment, address: DATA 00000021, align: 0
  Segment part 6. ROOT.
            ENTRY                  ADDRESS          REF BY
            =====                  =======          ======
```

The actual file that the programmer will use is stored in the Exe folder, which is one folder back. There you will find a mytest.hex file (an Intel programmer format that we will use), and the optional mytest.a90 file (a Motorola format which is quite popular but isn't used in this example).

Notepad will reveal the files to look like this. The digits are hexadecimal, and are composed of a memory address location, a number of digits counter, and the machine instructions, and a checksum usually.



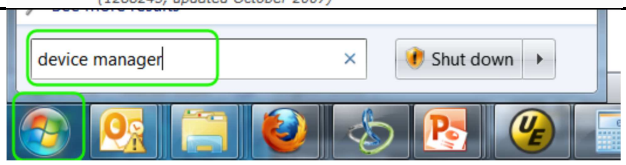
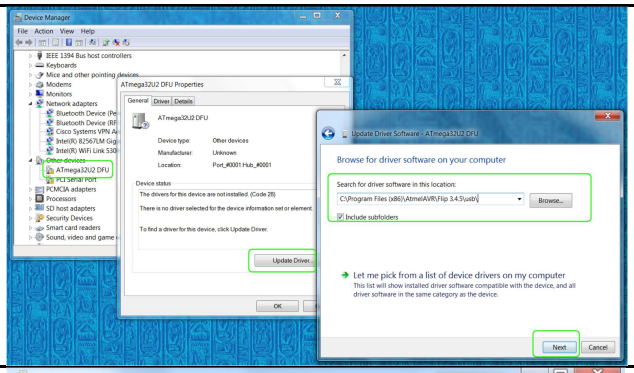
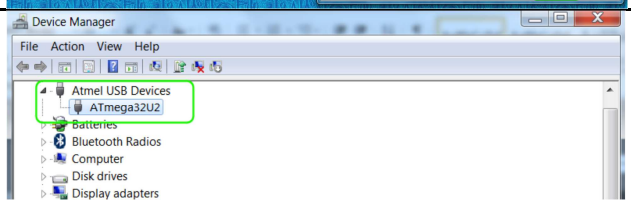
3. Program the device

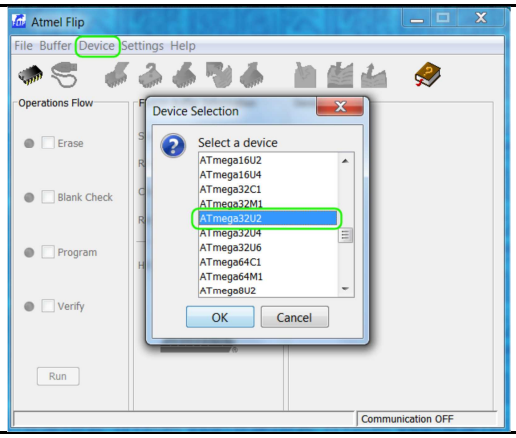
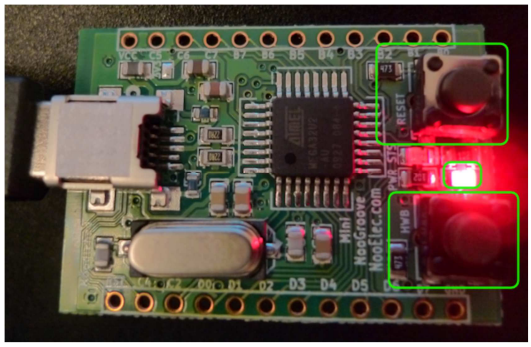
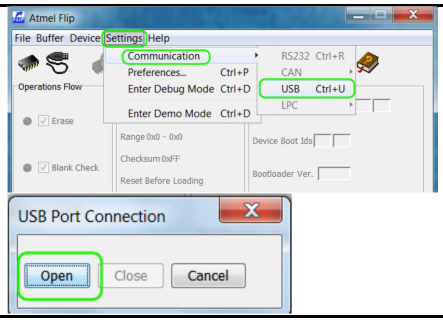
Now that the code is compiled, assembled and linked, it is possible to program it into the device. The process involves:

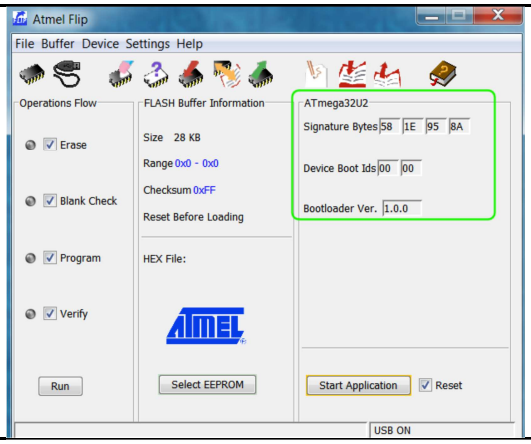
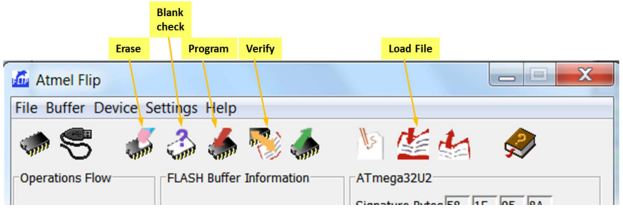
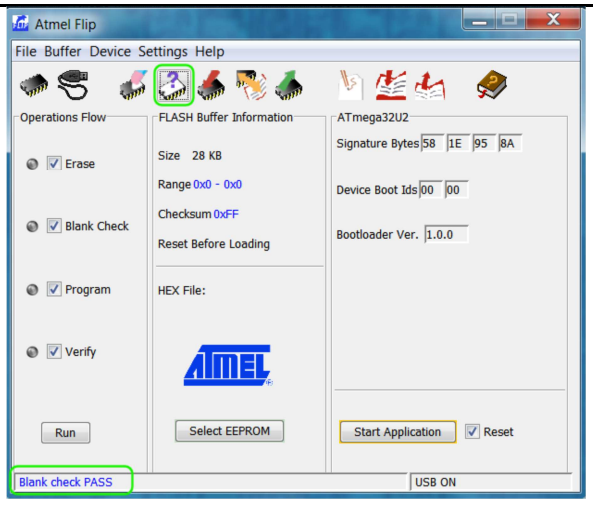
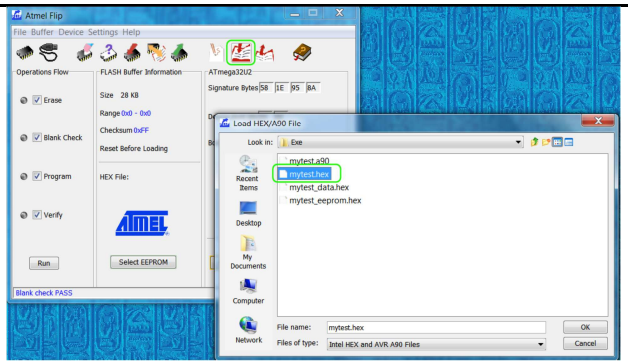
a) Download and install the programmer software
b) Connect the microcontroller board via USB to the PC

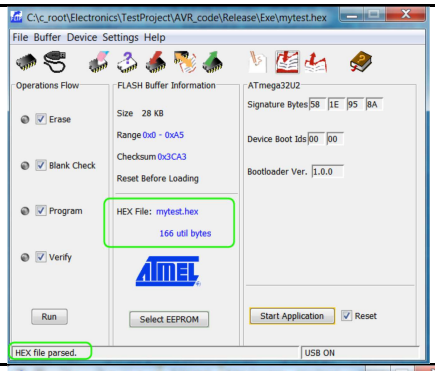
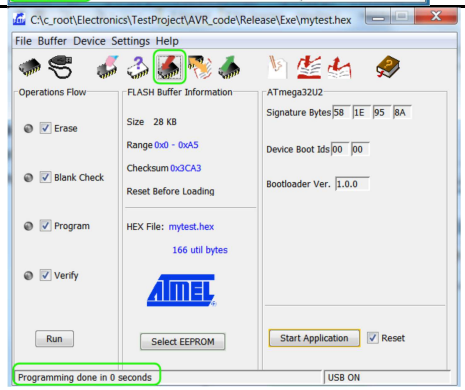
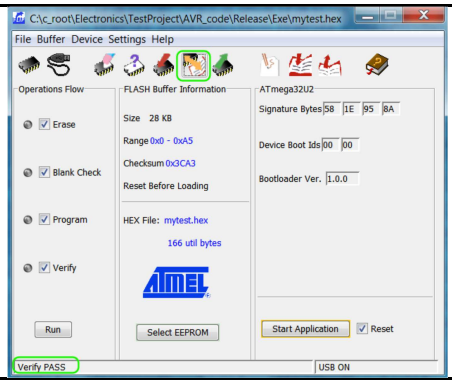
c) Set the microcontroller board into the programming mode by pressing some buttons on the board in a special pattern
d) Select the mytest.hex file in the programmer software
e) Erase the microcontroller, then hit the program button in the software
f) Once complete, press the Reset button on the microcontroller board, and enjoy the software running

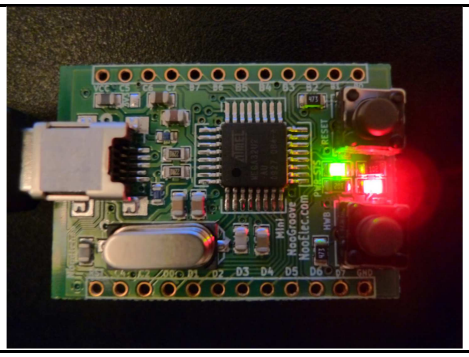
The detailed steps are shown here.

	Steps	Illustrations
1	From the Atmel website, download the 'Flip' software. The direct link is http://www.atmel.com/tools/FLIP.aspx If you already have Java installed, it is a small download. If you don't have Java installed, download the larger one. Once downloaded, install the software.	
2	Plug in the USB cable into the computer and the microcontroller board. (Most boards won't be supplied with a cable, so you may need to buy one). You will get a message saying that the device is not successfully installed. To resolve it, a driver needs to be installed as described in this and the next step. Start up Device Manager. On Windows 7, you can start it up just by typing it as shown here and hitting enter. Otherwise, you can find it in the Windows Control Panel.	
3	In the Device Manager, find the device (it will have a warning sign against it), right-click it and select Properties. Then, click the 'Update Driver' button as shown here. Locate the 'usb' folder within the Flip software installed location (it is most likely to be in C:\Program Files (x86)\Atmel\Flip 3.4.5\usb) and click Next to install it. Ignore the warning, and continue to install the driver.	
4	Once the driver is successfully installed, you'll see that there is no warning sign in the Device Manager for that device. You can now close the Device Manager. Unplugging and re-plugging the microcontroller board will always work ok now. (Test it now).	

5	With the microcontroller board plugged in, start up the Flip software. Click on Device->Select and then choose the correct device (ATmega32U2 in this example).	
6	It is most likely that the microcontroller has a special procedure to get it ready for programming. On the example board, the procedure is: **ATmega32U2 Programming mode procedure** 1. Hold down the 'HWB' button 2. While held down, now press the 'RESET' button 3. Release the 'RESET' button 4. After a few seconds, release the 'HWB' button The photo here shows the HWB button at the bottom, and the RESET button at the top. The red light is the power light. Once you have executed the reset procedure, move on to the next step.	
7	Click on Settings->Communication and select USB as shown here. In the small window that appears, click on Open.	

8	If all is well, then the Flip programmer software will have queried the microcontroller via USB, and will have indicated some 'Signature Bytes' as shown here. If it doesn't, then reattempt the programming mode procedure on the microcontroller board, and then reattempt the Settings->Communication and then select USB again and click Open. Now, the Signature Bytes should be populated.	
9	Familiarise yourself with these menu options. Blank check – Check if the device is erased Erase – Erase the device Load File – Prepare the programmer with the file that is going to be programmed Program – Program the microcontroller Verify – Confirm that the microcontroller has successfully been programmed	
10	Click on 'Blank check'. If the device is blank, it will print the information at the bottom-left, as shown here. Even if the device is already blank, you should click on 'Erase' now. If you don't, then later when you come to program the device, you may get a 'Device protection is set' error. If that occurs, click on Erase. In general, if things go wrong at any stage, try the programming mode procedure (HWB and RESET buttons on the microcontroller board in this example) and the reattempt clicking on Settings->Communication -> USB and then click Open again.	
11	Click on 'Load File' and then find where the file you wish to program is located. It will be in the TestProject\AVR_code\Release\Exe folder for this example. Select the mytest.hex file. Once the file is loaded (almost instantly), you will see 'HEX file parsed' printed in the lower-left section as shown in the screenshot here. Don't move on to the next step unless you can see the Hex File details as printed in the	

	screenshot here.	
12	If the Hex File is displayed successfully as shown in the previous step, then you can now click on the Program button. If all goes well, it will (almost instantly) display 'Programming done in 0 seconds' in the lower-left as shown here. If things go wrong, just do the programming mode procedure again on the Microcontroller board, and then in the Flip software click on Settings->Communication -> USB and click on Open again, and then redo the Erase and Verify steps and then reattempt the Program button. If it still doesn't work, then most likely the Flip software has got stuck. It happens quite often. If so, then close the Flip software and restart it. Redo all the steps.	
13	Now you can confirm if the programming was successful, by clicking on the Verify button. The lower-left will display 'VERIFY pass'.	
14	Now that the device is successfully programmed, you can run the program by just pressing the reset button on the microcontroller board once, to get it out of the programming mode.	

15	If you recall, the program was very simple; it just turns on a particular LED on the board, but turns it off while the HWB button is pressed. You can see it running in the photo here. If you press the HWB button, the green LED will turn off.	

5. Next steps

The microcontroller has a lot of functions, so it is worthwhile to examine the data sheets for it. The microcontroller contains many internal features and functions, such as an EEPROM, serial interface, counters and so on, which you'll want to use.

As the software grows, you will want to split out some functions into separate files. To do this, you need to make use of header files that you will create yourself. It is worth reading a short book on C programming.

Some people 'swear by' having an operating system installed in the microcontroller; however for small applications it really is not necessary. Many commercial applications that use small microcontrollers will not use an operating system. As your software grows more complex, then an OS becomes increasingly more useful and necessary.