ARM Cross Development with Eclipse

By: James P. Lynch

1 Introduction

I credit my interest in science and electronics to science fiction movies in the fifties. Robbie the Robot in the movie “Forbidden Planet” especially enthralled me and I watched every episode of Rocky Jones, Space Ranger on television. In high school, I built a robot and even received a ham radio operator license at age 13.

Electronic kits were popular then and I built many Heath kits and Knight kits, everything from ham radio gear to televisions, personal computers and robots. These kits not only saved money at the time, but the extensive instruction manuals taught the basics of electronics.

Unfortunately, surface mount technology and pick-and-place machines obliterated any cost advantage to “building it yourself” and Heath and Allied Radio all dropped out of the kit business.

What of our children today? They have home computers to play with, don’t they? Do you learn anything by playing a Star Wars game or downloading music? I think not, while these pastimes may be fun they are certainly not intellectually creative.

A couple years ago, there were 5 billion microcomputer chips manufactured planet-wide. Only 300 million of these went into desktop computers. The rest went into toasters, cars, fighter jets and Roomba vacuum cleaners. This is where the real action is in the field of computer science and engineering.

Can today’s young student or home hobbyist tired of watching Reality Television dabble in microcomputer electronics? The answer is an unequivocal YES!

Most people start out with projects involving the Microchip **PIC** series of microcontrollers. You may have seen these in Nuts and Volts magazine or visited the plethora of web sites devoted to **PIC** computing. **PIC** microcomputer chips are very cheap (a couple of dollars) and you can get an IDE (Integrated Development Environment), compilers and emulators from Microchip and others for a very reasonable price.

Another inexpensive microcontroller for the hobbyist to work with is the **Rabbit** microcomputer. The **Rabbit** line is an 8-bit microcontroller with development packages (board and software) costing less that $140.

I've longed for a real, state-of-the-art microcomputer to play with. One that can do 32-bit arithmetic as fast as a speeding bullet and has all the on-board RAM and EPROM needed to build sophisticated applications. My prayers have been answered recently as big players such as Texas Instruments, Philips and Atmel have been selling inexpensive microcontroller chips based on the 32-bit ARM architecture. These chips have integrated RAM and FLASH memory, a rich set of peripherals such as serial I/O, PWM, I2C, SSI, Timers etc. and high performance at low power consumption.

A very good example from this group is the Philips LPC2000 family of microcontrollers. The LPC2106 has the following features, all enclosed in a 48-pin package costing about $11.80 (latest price from Digikey for one LPC2106).

Key features

- 16/32-bit ARM7TDMI-S processor.
- 64 kB on-chip Static RAM.
- 128 kB on-chip Flash Program Memory. In-System Programming (ISP) and In-Application Programming (IAP) via on-chip boot-loader software.
- Vectored Interrupt Controller with configurable priorities and vector addresses.
- JTAG interface enables breakpoints and watch points.
- Multiple serial interfaces including two UARTs (16C550), Fast I²C (400 kbits/s) and SPI™.
- Two 32-bit timers (7 capture/compare channels), PWM unit (6 outputs), Real Time Clock and Watchdog.
- Up to thirty-two 5 V tolerant general-purpose I/O pins in a tiny LQFP48 (7 x 7 mm^2) package.
- 60 MHz maximum CPU clock available from programmable on-chip Phase-Locked Loop with settling time of 100 us.
- On-chip crystal oscillator with an operating range of 1 MHz to 30 MHz.
- Two low power modes: Idle and Power-down.
- Processor wake-up from Power-down mode via external interrupt.
- Individual enable/disable of peripheral functions for power optimization.
- Dual power supply:
    - CPU operating voltage range of 1.65 V to 1.95 V (1.8 V +- 8.3 pct.).
    - I/O power supply range of 3.0 V to 3.6 V (3.3 V +- 10 pct.) with 5 V tolerant I/O pads.

Several companies have come forward with the LPC2000 microcontroller chips placed on modern surface-mount boards, ready to use. Olimex and New Micros have a nice catalog of inexpensive boards using the Philips ARM family. I wrote a similar tutorial for the New Micros **TiniARM** nine months ago and you can see it on their web site www.newmicros.com.

Olimex, an up-and-coming electronics company in Bulgaria, offers a family of Philips LPC2100 boards. Specifically they offer three versions with the LPC2106 CPU. You can buy these from Spark Fun Electronics in Colorado; their web site is www.sparkfun.com The Olimex boards are also carried by Microcontroller Pros in California, their web site is www.microcontrollershop.com

This is the Olimex LPC-H2106 header board. You can literally solder this tiny board onto Radio Shack perfboard, attach a power supply and serial cable and start programming. It costs about $49.95 Obviously, it requires some soldering to get started.

This is the Olimex LPC-P2106 prototype board. Everything is done for you. There's a power connector for a wall-wart power supply, a DB-9 serial connector and a JTAG port. It costs about $59.95 plus $2.95 for the wall-wart power supply.

This is the Olimex LPT-MT development board; it has everything the prototype board above includes plus a LCD display and four pushbuttons to experiment with. It costs about $79.95 plus $2.95 for the wall-wart power supply.

For starting out, I would recommend the **LPC-P2106** prototype board since it has an open prototype area for adding I2C chips and the like for advanced experimentation.

When you do design and develop something really clever, you could use the LPC-H2106 header board soldered into a nice Jameco or Digikey prototype board and know that the CPU end of your project will work straight away. If you need to build multiple copies of your design, Spark Fun can get small runs of blank circuit boards built for $5.00 per square inch. You can acquire the Eagle-Lite software from CadSoft for free to design the schematic and PCB masks.

So the hardware to experiment with 32-bit ARM microprocessors is available and affordable. What about the software required for editing, compiling, linking and downloading applications for the LPC2106 board?

Embedded microcomputer development software has always been considered "professional" and priced accordingly. It's very common for an engineer in a technical company to spend $1000 to $5000 for a professional development package. I once ordered $18,000 of compilers and emulators for a single project. In the professional engineering world, time is money. The commercial software development packages for the ARM architecture install easily, are well supported and rarely have bugs. In fact, most of them can load your program into either RAM or FLASH and you can set breakpoints in either. The professional compiler packages are also quite efficient; they generate compact and speedy code.

The Rowley CrossWorks recommended by Olimex is $904.00, clearly out of the range for the student or hobby experimenter. I've seen other packages going up as high as $3000. A professional would not bat an eyelash about paying this – time is money.

There is a low cost alternative to the high priced professional software development packages, the GNU toolset. GNU is the cornerstone of the open-source software movement. It was used to build the LINUX operating system. The GNU Toolset includes compilers, linkers, utilities for all the major microprocessor platforms, including the ARM architecture. The GNU toolset is free.

The editor of choice these days is the Eclipse open-source Integrated Development Environment (IDE). By adding the CDT plugin (C/C++ Development Toolkit), you can edit and build C programs using the GNU compiler toolkit. Eclipse is also free.

Philips provides a Windows flash programming utility that allows you to transfer the hex file created by the GNU compiler/linker into the onboard flash EPROM on the LPC2106 microprocessor chip. The Philips tool is also free.

Macraigor has made available a free Windows utility called OCDremote that allows the Eclipse/GDB (GNU Debugger) to access the Philips LPC2106 microprocessor via the JTAG port using an expensive device called the "**wiggler**". I've had more success running the open-source **Insight** debugger than the debugger included with Eclipse; however it's only usable for programs running from RAM memory.

At this point, you're probably saying "this is great – all these tools and they're FREE!" In the interest of honesty and openness, let's delineate the downside of the free open software GNU tools.

- The GNU tools do not currently generate as efficient code as the professional compilers.

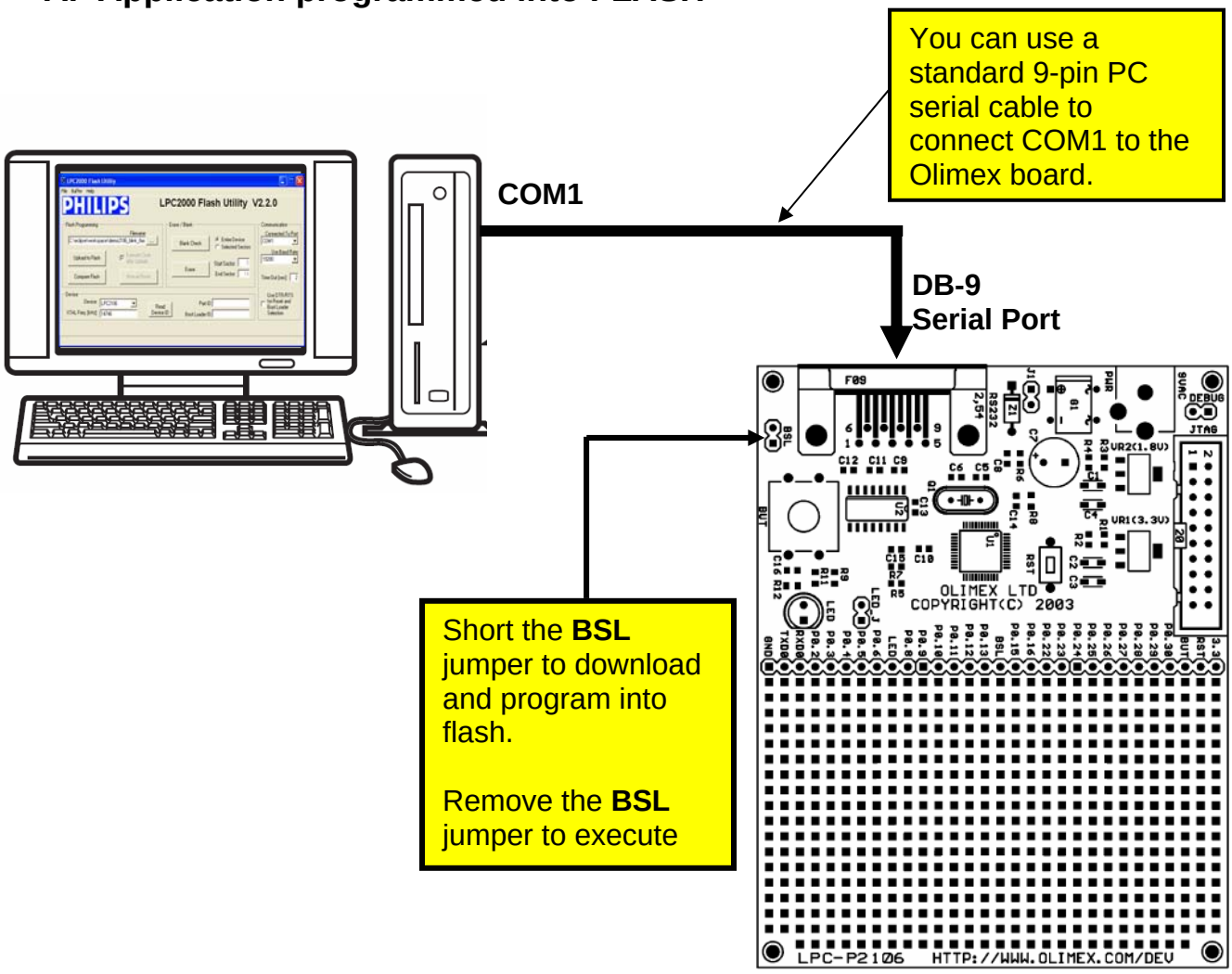
- The Insight Debugger cannot set a software breakpoint in FLASH since it can't erase and reprogram the FLASH.

- The OCDRemote JTAG utility does not support hardware breakpoints.

If you were a professional programmer, you would not accept these limitations. For the student or hobbyist, the Eclipse/GNU toolset still gives fantastic capabilities for zero cost.

The Eclipse/GNU Compiler toolset we will be creating in this tutorial operates in two modes.

A. Application programmed into FLASH



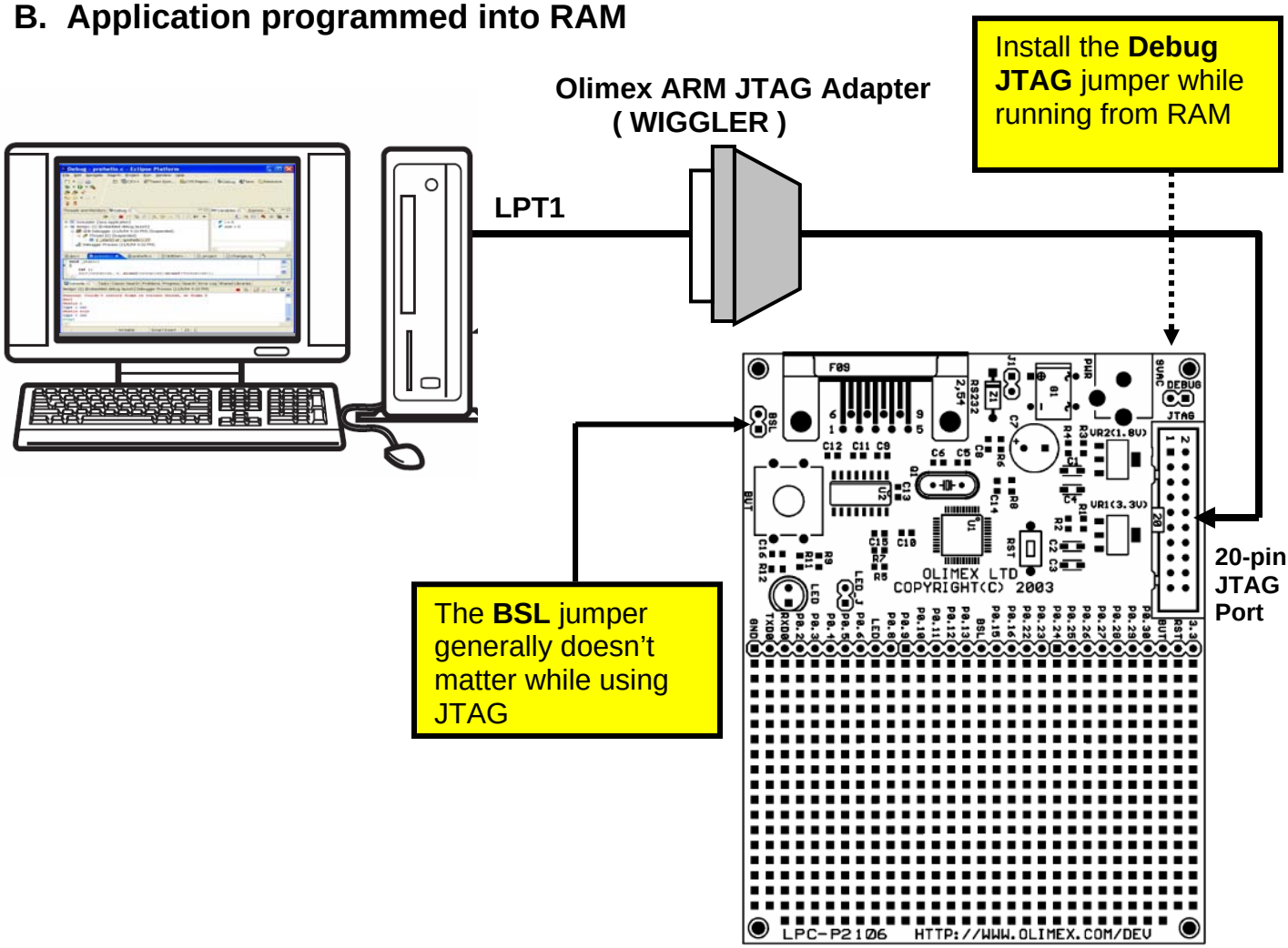
In this mode, the Eclipse/GNU development system assembles, compiles and links your application for loading into FLASH memory. The output of the compiler/linker suite is an Intel hex file, e.g. **main.hex**.

The Philips In-System Programming (**ISP**) utility is started within Eclipse and will download your hex file and program the flash memory through the standard COM1 serial cable. The Boot Strap Loader (BSL) jumper must be shorted (installed) to run the **ISP** flash programming utility.

To execute the application, you remove the BSL jumper and push the RESET button to start the application.

Unfortunately, the Insight debugger cannot set a software breakpoint (it can't program FLASH) and it also doesn't support hardware breakpoints. This effectively renders the debugger useless in this mode.

B. Application programmed into RAM



In this mode, the Eclipse/GNU development system assembles, compiles and links your application for loading into RAM memory. The output of the compiler/linker suite is a GNU **main.out** file.

The PC is connected from the PC's printer port LPT1 to the JTAG port through the **Olimex ARM JTAG** interface (costs about $19.95 from Spark Fun Electronics). The Olimex **ARM JTAG** is a clone of the Macraigor **Wiggler**.

You can run the **OCDRemote** program as an external tool from within Eclipse. The **Insight** debugger (started from within Eclipse) communicates with the Macraigor **OCDRemote** program that operates the JTAG port using the **Wiggler**. With the **Insight** debugger, you can connect to the **Wiggler** and load the GNU **main.out** file into RAM. From this point on, you can set software breakpoints, view variables and structures and, of course, run the application.

The drawback is that the application must fit within RAM memory on the LPC2106, which is 64 Kbytes. Still, it's better than nothing.

My purpose in this tutorial is to guide the student or hobbyist through the myriad of documentation and web sites containing the necessary component parts of a working ARM software development environment. I've devised a simple sample program that blinks an LED that is compatible in every way with the GNU assembler, compiler and linker. There are two variants of this program; a FLASH-based version and a RAM-based version. I've substituted the GNU **Insight** graphical debugger instead of the Eclipse debugger because it is simpler to use and currently more reliable.

If you get this to work, you are well on your way to the fascinating world of embedded software development. Take a deep breath and HERE WE GO!

If you are very new to ARM microcomputers, there's no better introductory book than "**The Insider's Guide to the Philips ARM7-Based Microcontrollers**" by Trevor Martin. Martin is an executive of Hitex, a UK vendor of embedded microcomputer development software and hardware and he obviously understands his material.

You can download this e-book for free from the Hitex web site.

http://www.hitex.co.uk/arm/lpc2000book/index.html

There is a controversial section in Chapter 2 with benchmarks showing that the GNU toolset is 4 times slower in execution performance and 3.5 times larger in code size than other professional compiler suites for the ARM microprocessors. Already Mr. Martin has been challenged about these benchmarks on the internet message boards; see "The Dhrystone benchmark, the LPC2106 and GNU GCC" at this web address:

http://www.compuphase.com/dhrystone.htm

Well, we can't fault Trevor Martin for tooting his own horn! In any case, Martin's book is a magnificent work and it would behoove you to download and spend a couple hours reading it. I've used Hitex tools professionally and can vouch for their quality and value. Read his book! Better yet, it's required reading.

2 Installing the Necessary Components

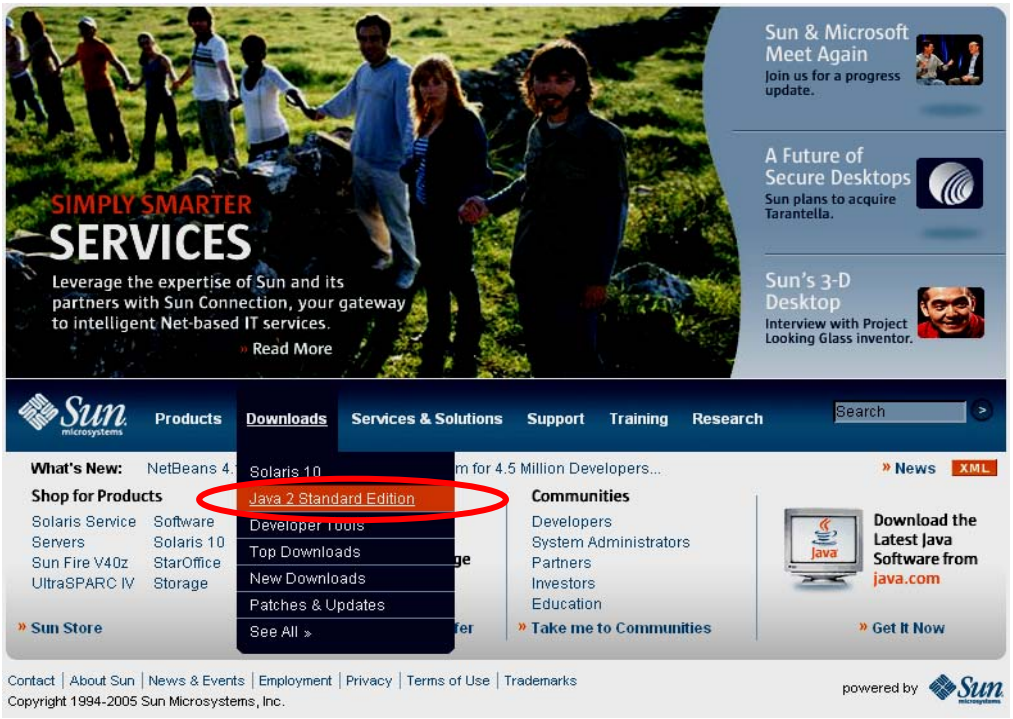
To set up an ARM cross-development environment using Eclipse, you need to download and install several components. The required parts of the Eclipse/ARM cross development system are:

1. **SUN Java Runtime**
2. **Eclipse IDE**
3. **Eclipse CDT Plug-in for C++/C Development**
4. **CYGWIN GNU C++/C Compiler and Toolset for Windows**
5. **GNUARM GNU C++/C Compiler for ARM Targets**
6. **GNUARM Insight Debugger**
7. **Philips Flash Programmer for LPC2100 Family CPUs**
8. **Macraigor OCDremote for JTAG debugging**

3 JAVA Runtime

The Eclipse IDE was written entirely in JAVA. Therefore, you must have the JAVA runtime installed on your Windows computer to run Eclipse. Most people already have JAVA set up in their Windows system, but just in case you don't have JAVA installed, here's how to do it.

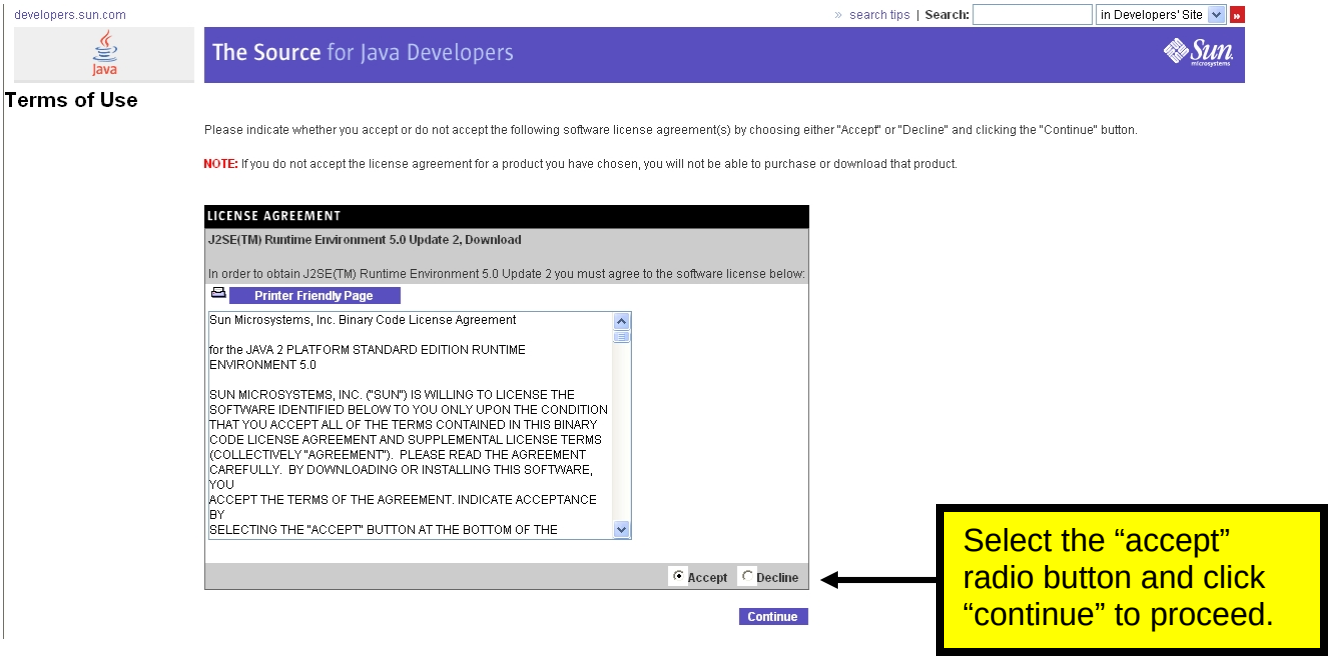
The JAVA runtime is available free at www.sun.com. The following screen will appear. Click on "**Downloads – Java 2 Standard Edition**" to continue.



Select the “latest and greatest” Java runtime system by clicking on **J2SE 5.0**.

Specifically, we need only the Java Runtime Environment (JRE). Click on “**Download JRE 5.0 Update 3.**”

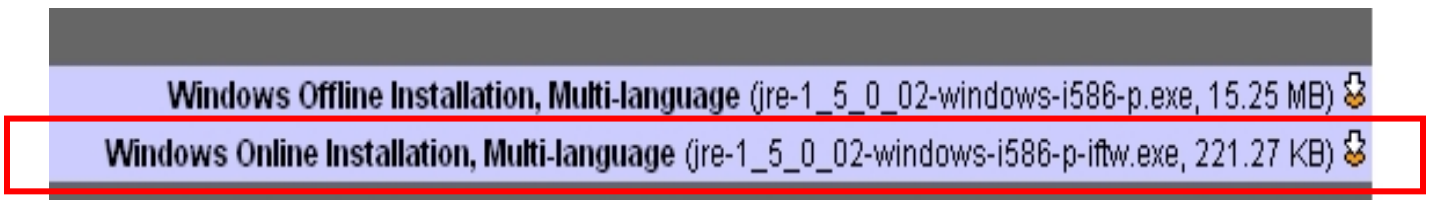
The Sun “Terms of Use” screen appears first. You have to accept the Sun binary code license to proceed. If you develop a commercial product using the Sun JAVA tools, you will have to pay royalties to them.



Select the “accept” radio button and click “continue” to proceed.

One more choice to decide on – we want the “**online**” installation for Windows.

Here’s a blow-up of the line we must click on. We select “**online**” so we can install immediately.

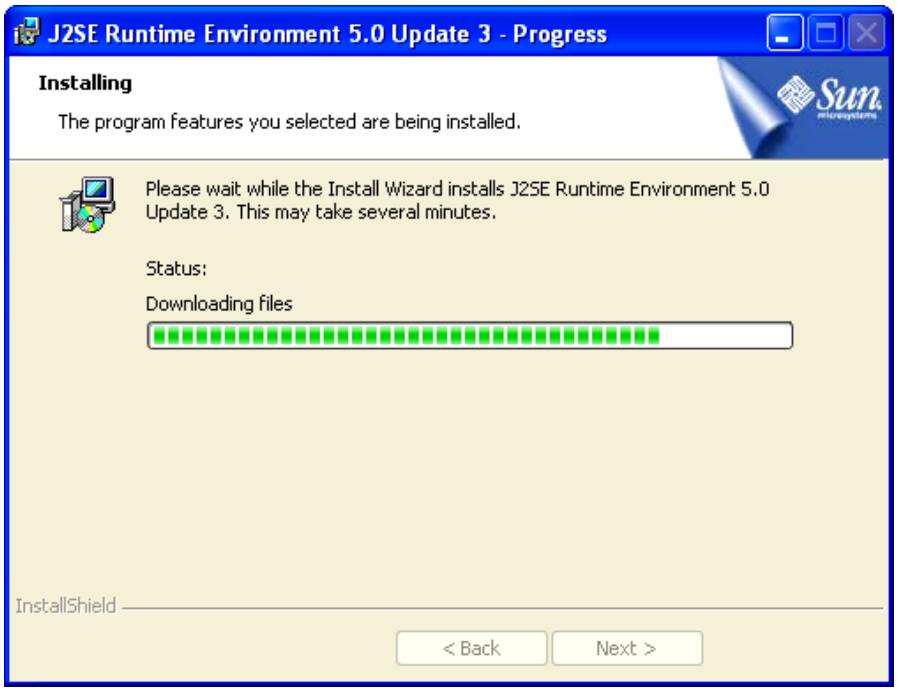


Finally the “file download” window appears. Click on “**Run**” to download and run the installation.

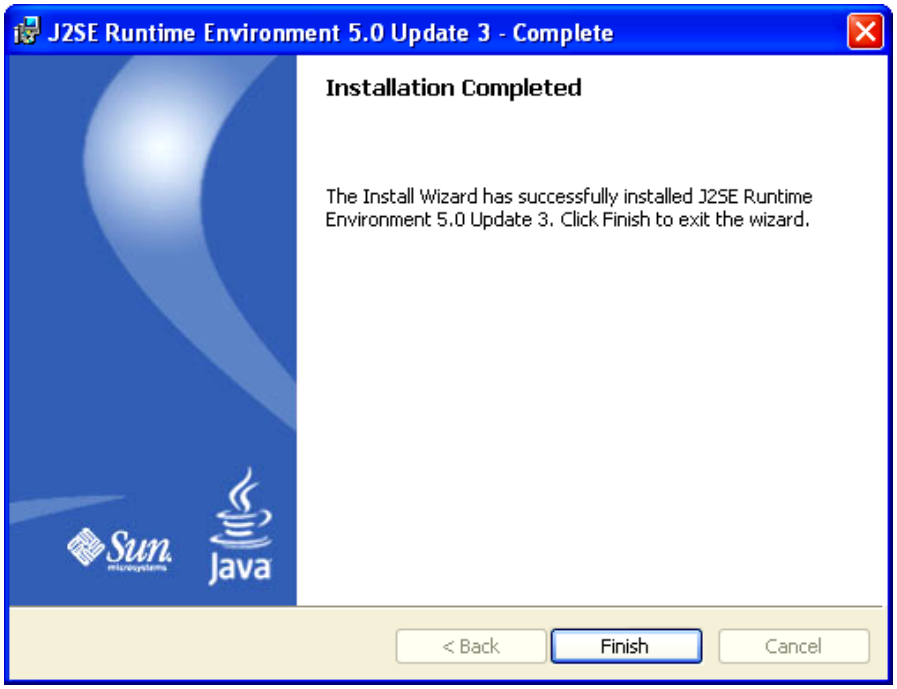
`

Now the downloading will start.

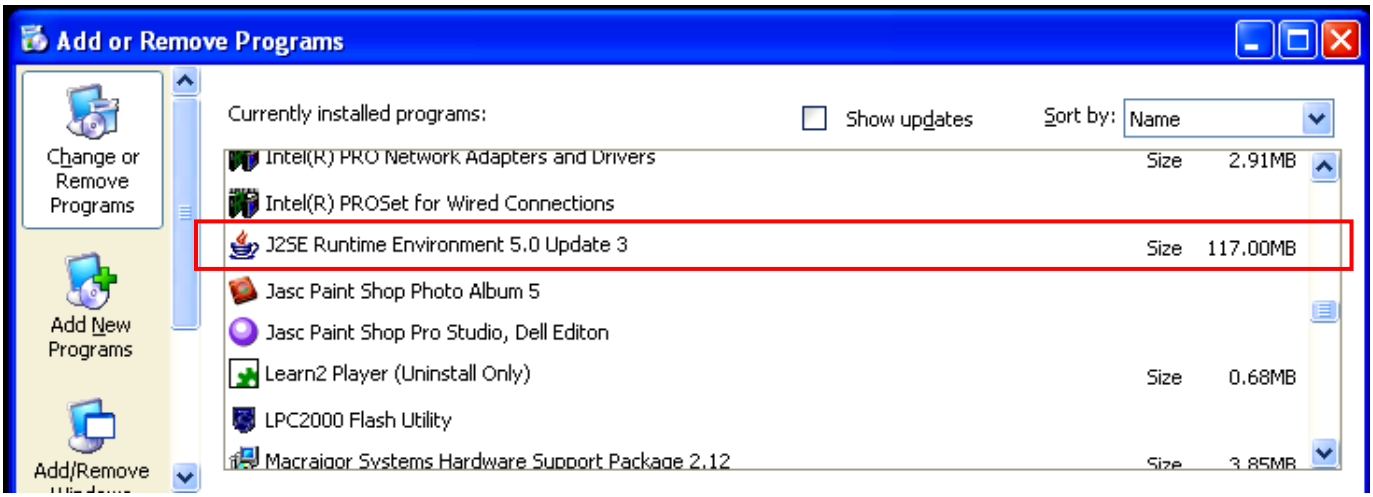
After downloading, the installation will proceed automatically.



When the Java Runtime Environment installation completes, you will see this display. Click on "**Finish**."



As a quick check, go to the Windows **Start** menu and select "**Start – Control Panel – Add or Remove Programs**." Scroll down the list of installed programs and see if the Java J2SE Runtime Environment was indeed installed!



The Sun Microsystems web site is very dynamic, changing all the time. Don't be surprised if some of the example displays shown here are a bit different.

4 Eclipse IDE

The Eclipse IDE is a complete integrated development platform similar to Microsoft's Visual Studio. Originally developed by IBM, it has been donated to the Open-Source community and is now a massive world-wide Open-Source development project.

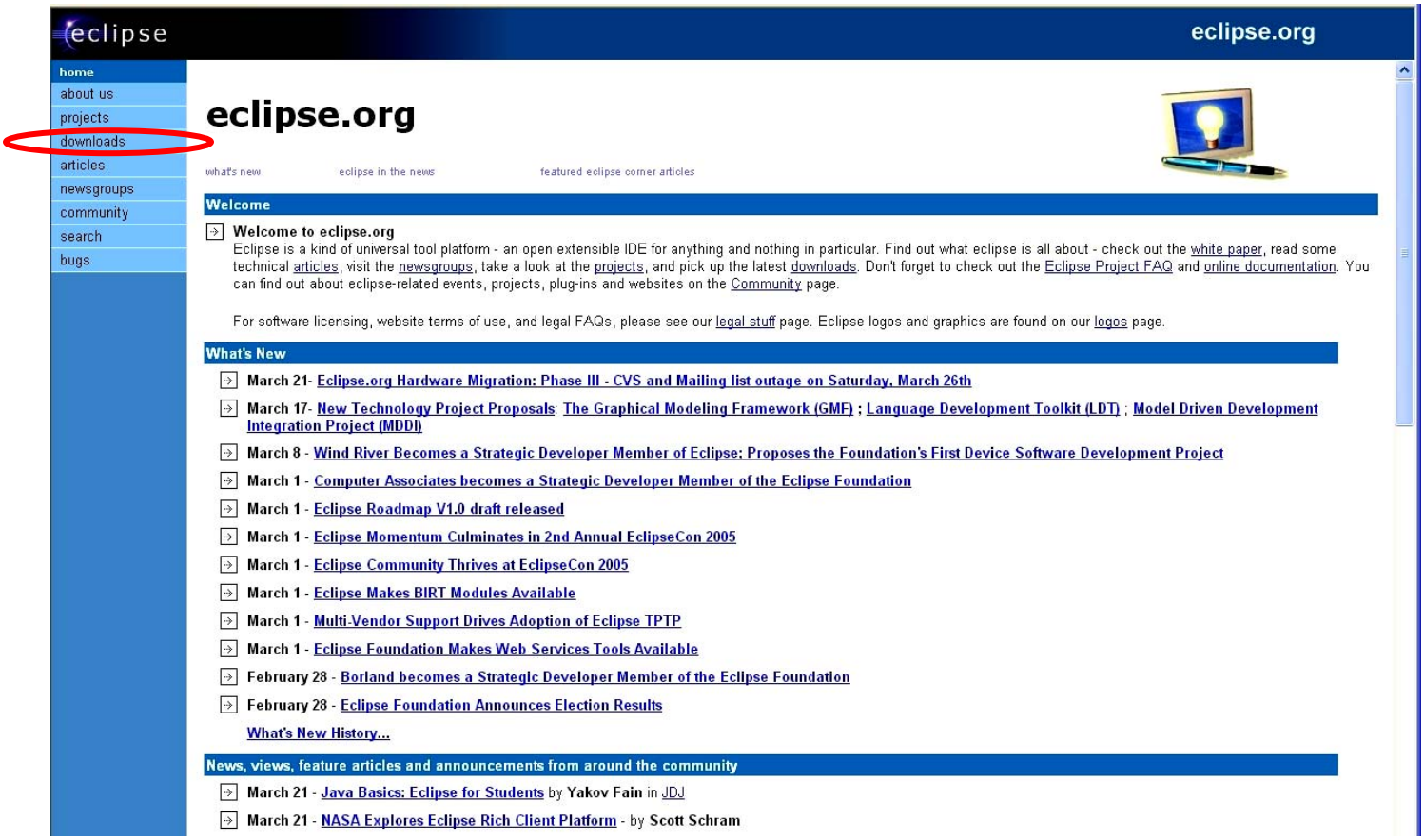
Eclipse, by itself, is configured to edit and debug JAVA programs. By installing the CDT plug-ins, you can use Eclipse to edit and debug C/C++ programs (more on that later).

When properly setup, you will have a sophisticated programmer's editor, compilers and debugger sufficient to design, build and debug ARM applications.

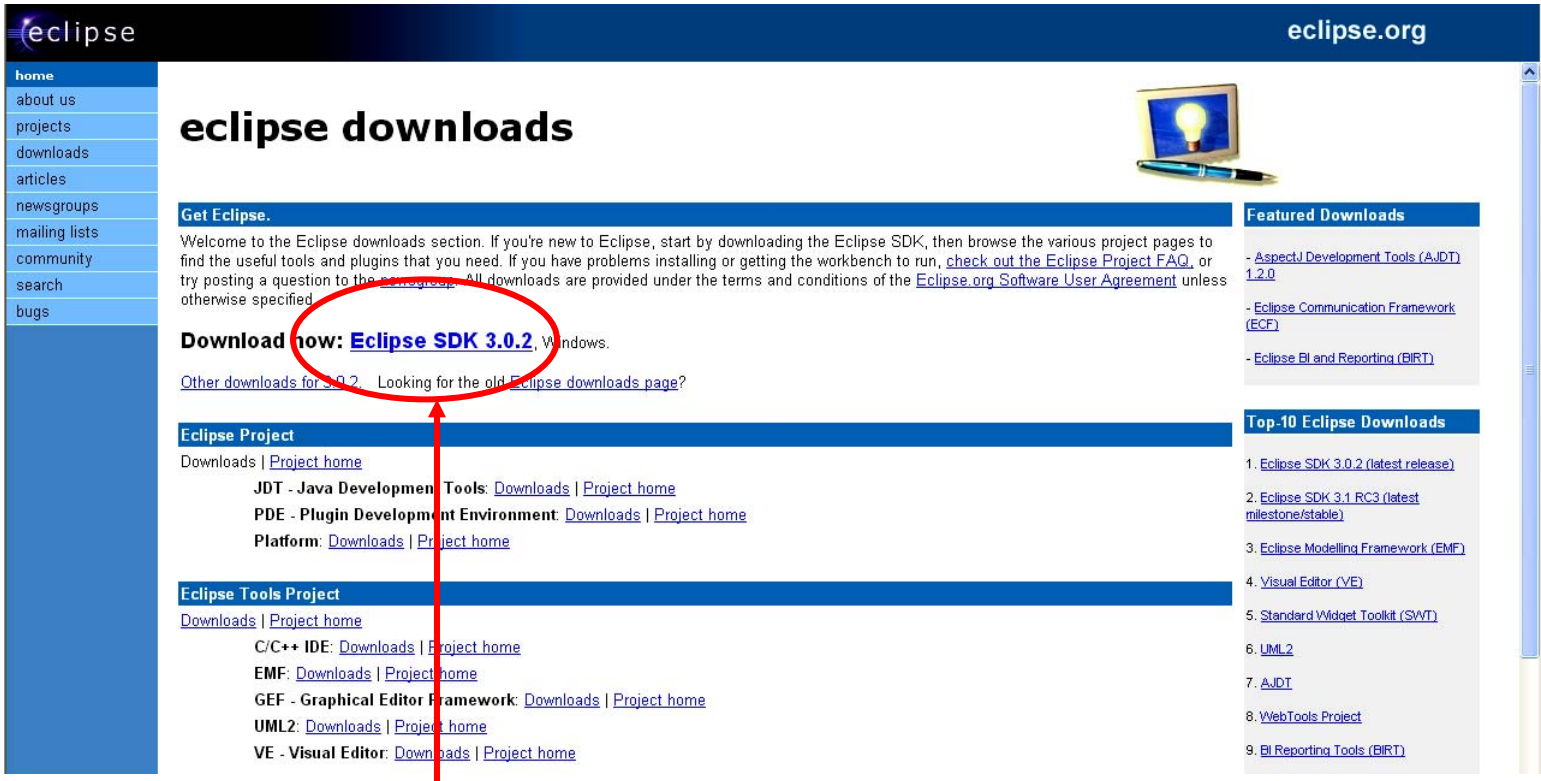
You can download Eclipse for free at the following web site.

www.eclipse.org

The following Eclipse welcome page will display. Expect some differences from my example below since the Eclipse web site is very dynamic.



Click on "**Downloads**" to get things started.



Click on Eclipse SDK 3.0.2 to get the latest stable release for Windows

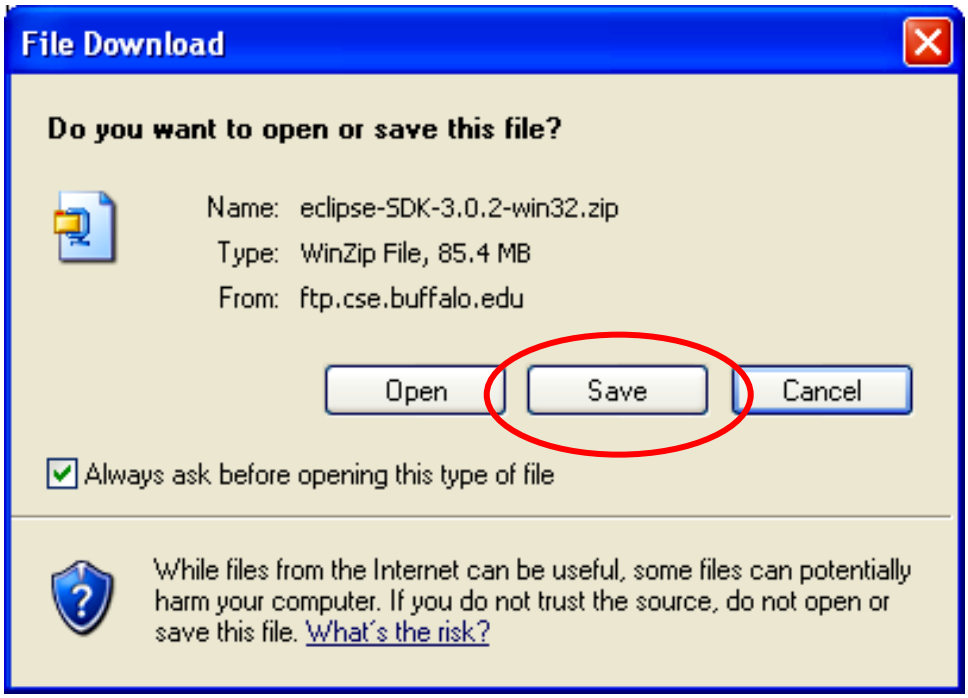
When working with the Eclipse and CDT, it's important to be sure that the CDT plugin you've selected is compatible with the Eclipse revision you also selected. Be sure to study the Eclipse web sites to be sure that you have compatible revisions selected.

If you click on **Eclipse SDK 3.0.2** where it says "Download Now:" shown above, this is the **Windows** version of the download.

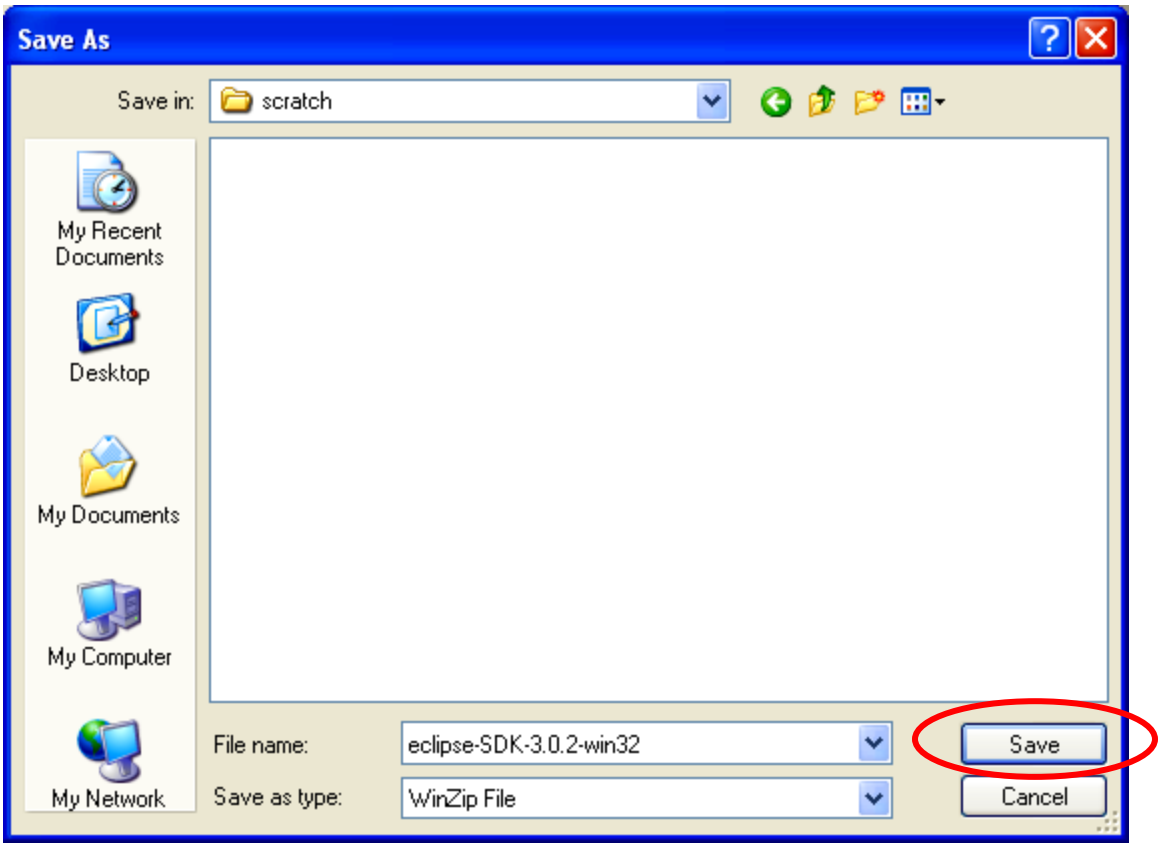
What appears next is a list of download mirror sites that host the Eclipse components. I selected the **University of Buffalo** in my home town (and where I got my Master's degree).

When the mirror site starts the download process, you have to select a destination directory to place the Eclipse zip file. In my case, I created an empty **C:/scratch** directory on one of my hard drives (you could use any other drive as well).

First click on **Save** below.

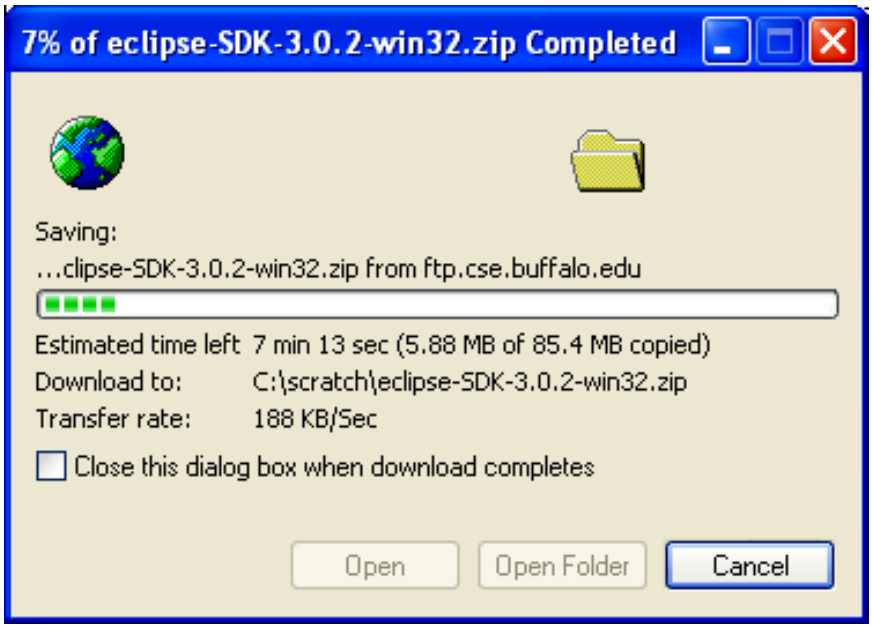


Now browse to the **c:/scratch** directory that you created previously.

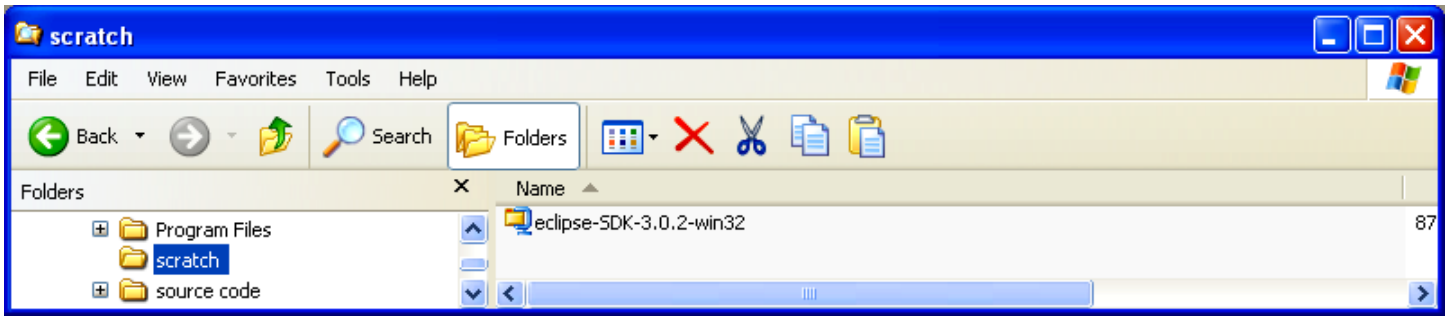


Click on **Save** to start the download.

Now the download will start. Eclipse is delivered as a ZIP file. It's 85 **megabytes** in length and takes 8 minutes and 20 seconds to download with my broadband cable modem. If you have a dialup internet connection, this will be excruciating. If you don't have a cable modem high-speed internet connection, I suggest you find somebody who does and go over there with a blank CDROM and a gift.



When the Eclipse download completes, you should see the following zip file in your scratch directory.



Contents of folder **c:/scratch/**

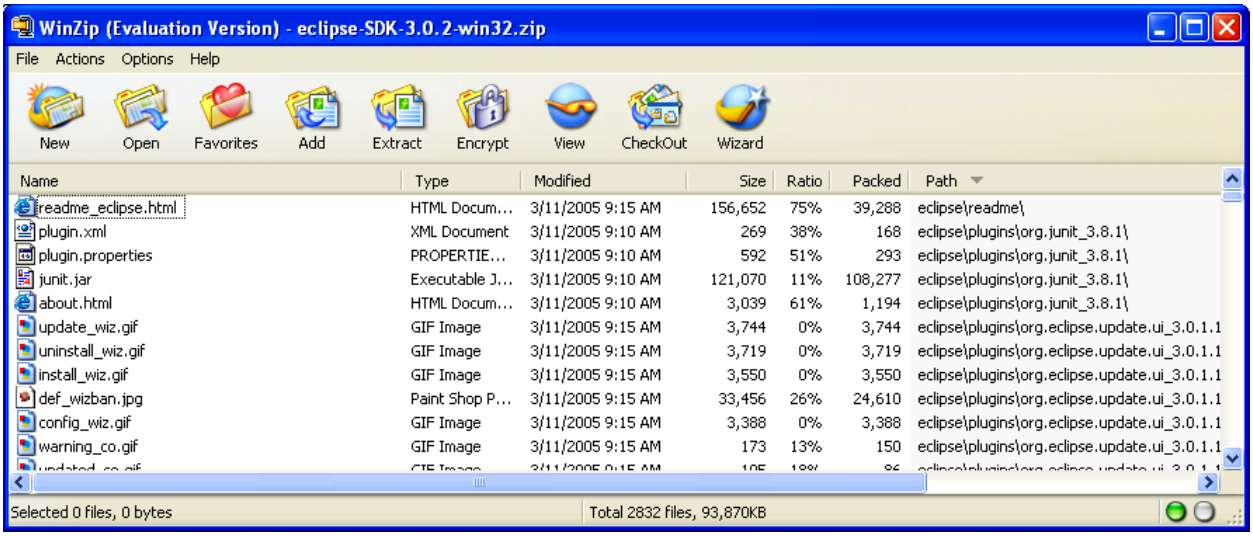
Eclipse is delivered as a ZIP file (**eclipse-SDK-3.0.2-win32.zip**). You can use WinZip to decompress this file and load its constituent parts on your hard drive.

If you don't have WinZip, you can get a free evaluation version from this address:

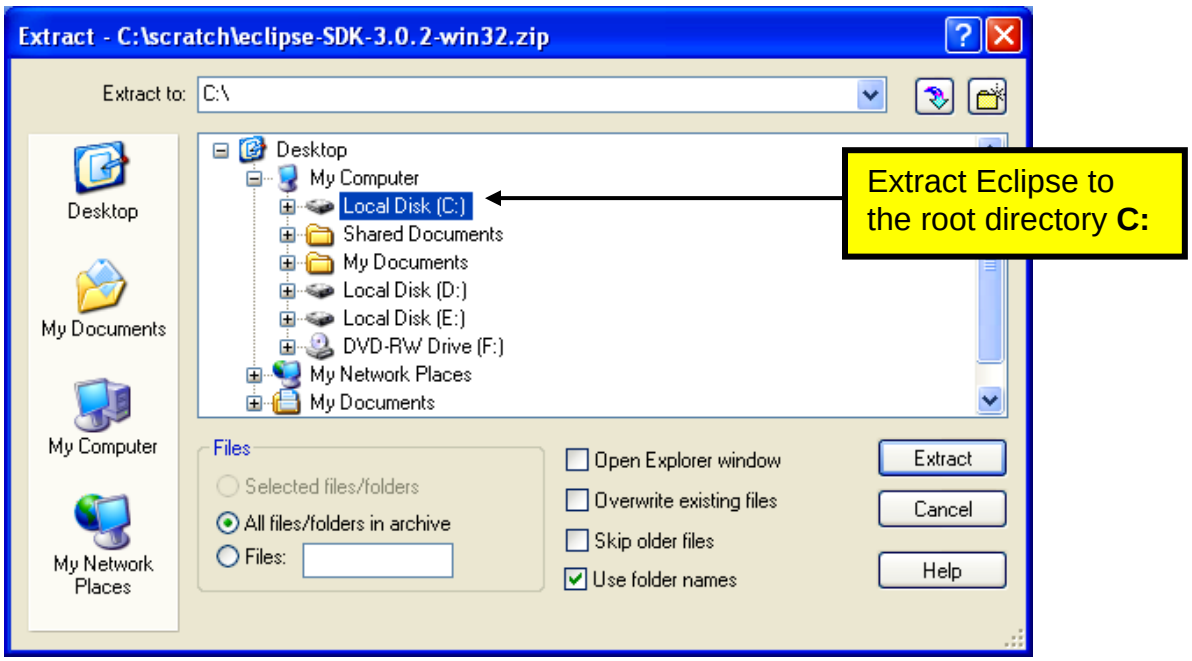
http://www.winzip.com/

There's a decent Help file supplied by WinZip. Therefore, we're going to assume that the reader is able to use a tool such as WinZip to extract from zip files.

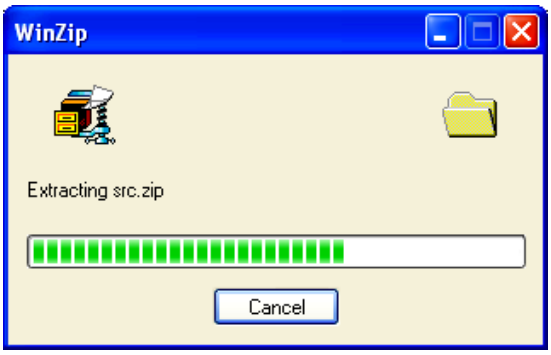
In my computer, with WinZip installed, double-clicking on the zip file name (**eclipse-SDK-3.1M7-win32.zip**) in the Windows Explorer display above will automatically start up WinZip.



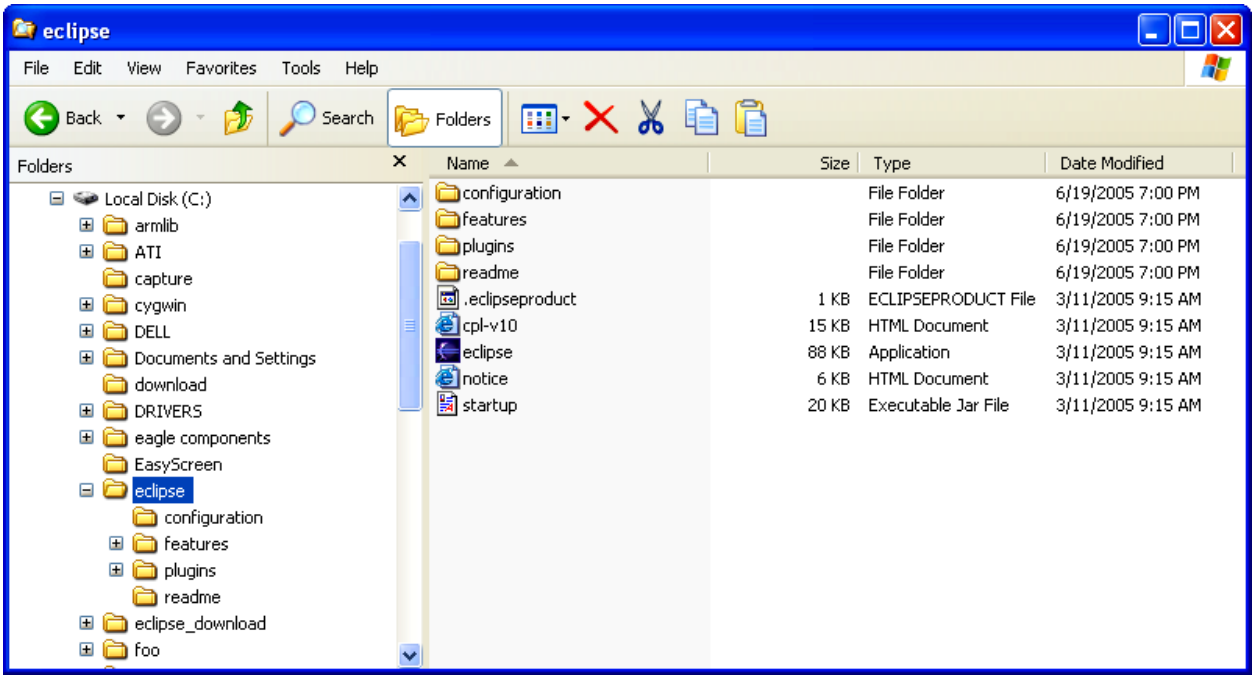
WinZip will ask you into what directory you wish to extract the contents of the zip file. In this case, you must specify the root drive **C:**



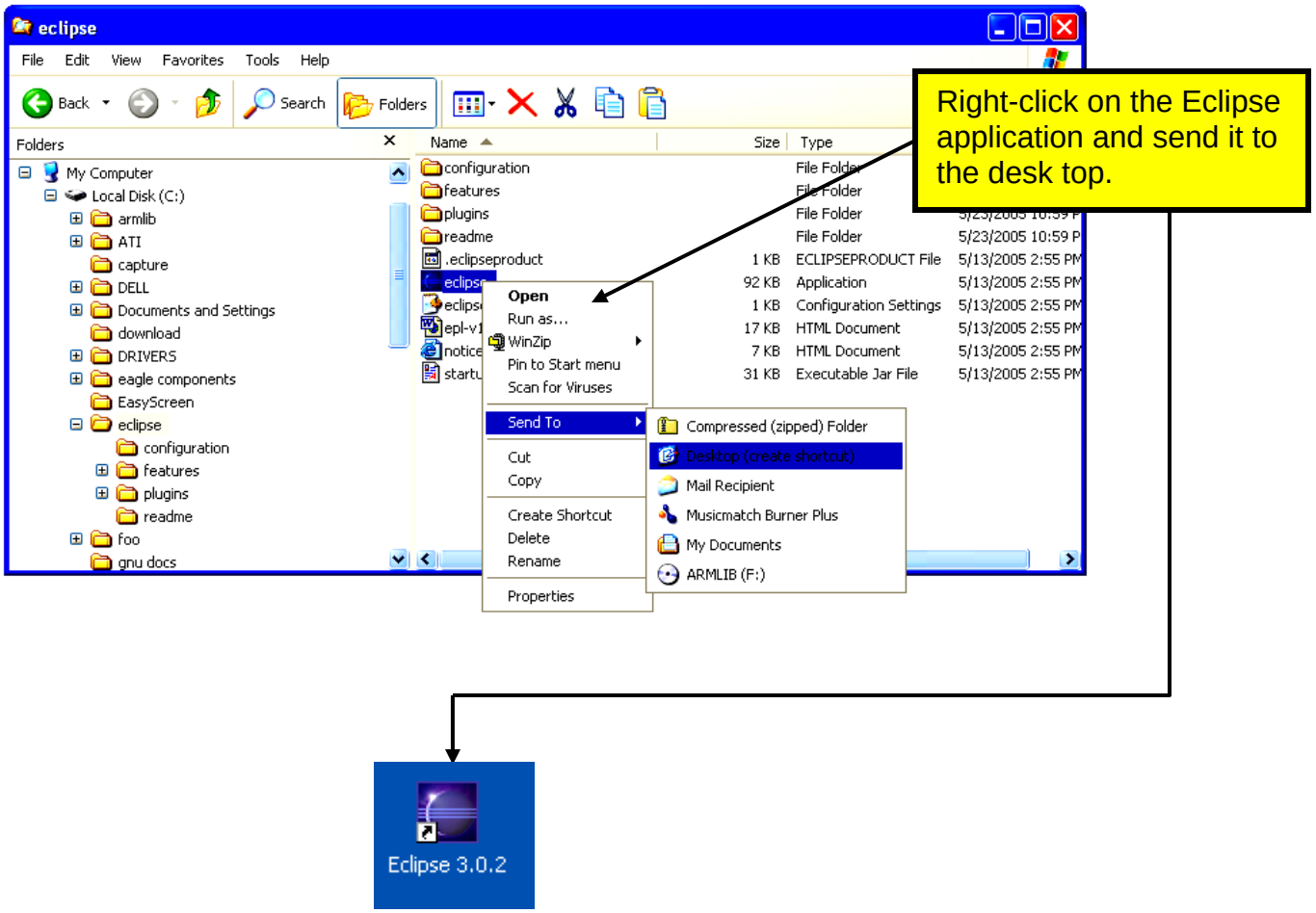
The WinZip Utility will start extracting all the Eclipse files and directories into a c:/eclipse directory on your root drive **C:**.



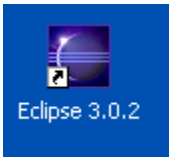
At this point, Eclipse is already installed (some things are done when you run it for the first time). The beauty of Eclipse is that there are no entries made into the Windows registry, Eclipse is just an ordinary executable file. Here's what the Eclipse directory looks like.



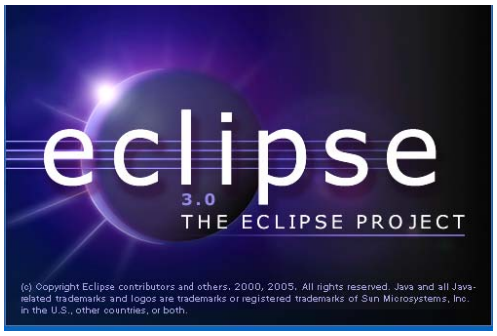
You can create a desktop icon for conveniently starting Eclipse by right-clicking on the Eclipse application above and sending it to the desk top.



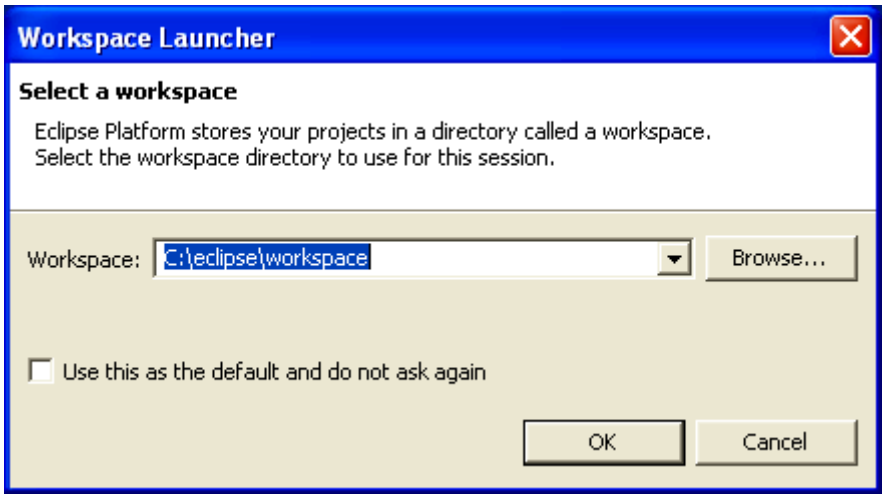
Now is a good time to test that Eclipse will actually run. Click on the desktop icon to start the Eclipse IDE.



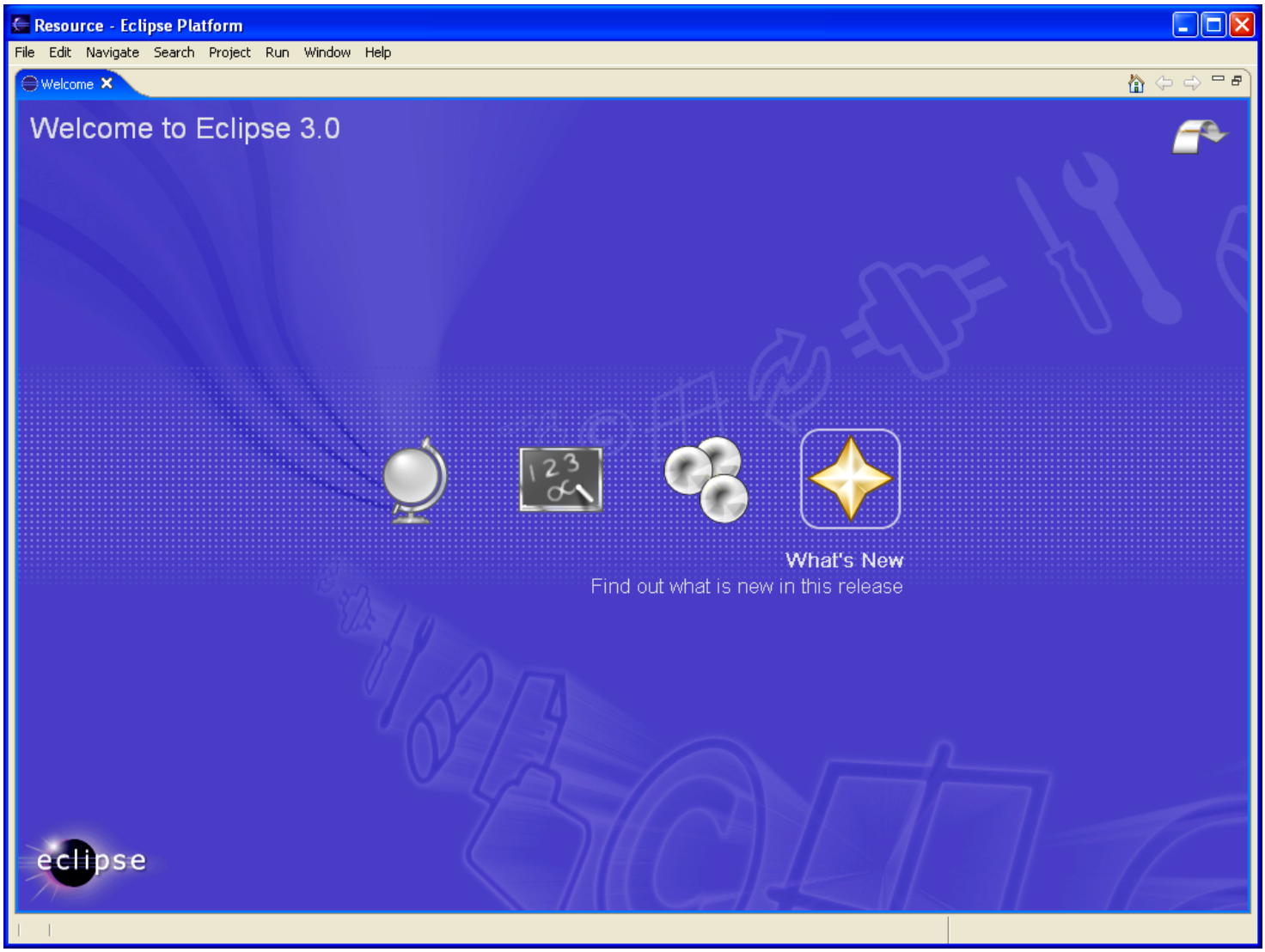
If the Eclipse Splash Screen appears, we have succeeded. If not, chances are that the Java Run Time Environment is not in place. Review and repeat the instructions on installing Java on your computer.



The first order of business is to specify the location of the Workspace. I choose to place the workspace within the Eclipse directory. You are free to place this anywhere; you can have multiple workspaces; here is where you make that choice.



When you click OK, the Eclipse main screen will start up.



If you made it this far, you now have a complete Eclipse system capable of developing JAVA programs for the PC. There are a large number of JAVA books and some really good ones showing how to develop Windows applications with JAVA using the Eclipse toolkit.

Eclipse itself was written entirely in JAVA and this shows you just how sophisticated a program can be developed with the Eclipse JAVA IDE.

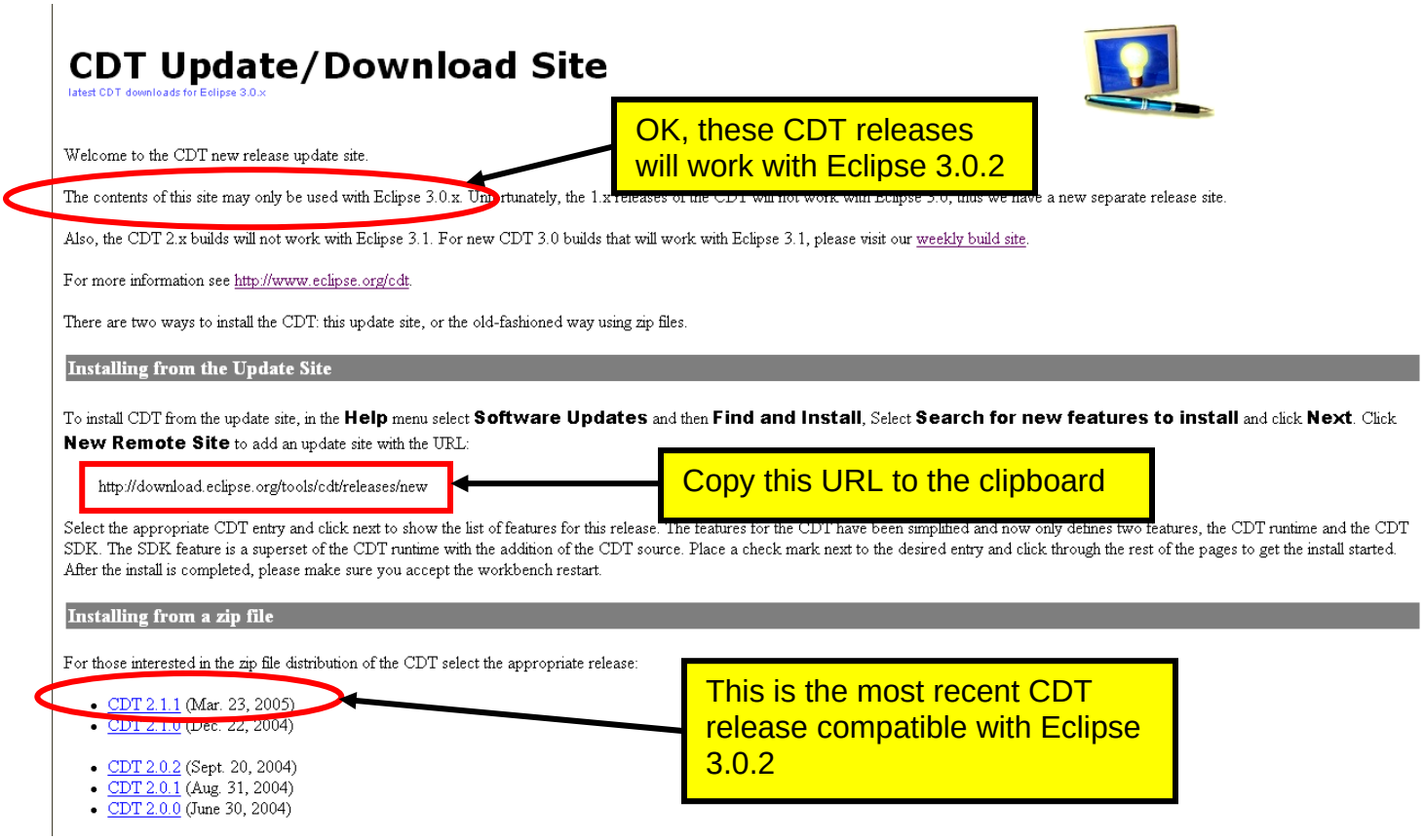
However, the point of this tutorial is to show how the Eclipse platform with the CDT plugins can be used to develop embedded software in C language for the ARM microcomputers.

5 Eclipse CDT

Eclipse, just by itself, is designed to edit and debug JAVA programs. To equip it to handle C and C++ programs, you need to download the **CDT** (C Development Toolkit) plug-in. The CDT plug-in is simply a collection of files that are inserted into two Eclipse directories.

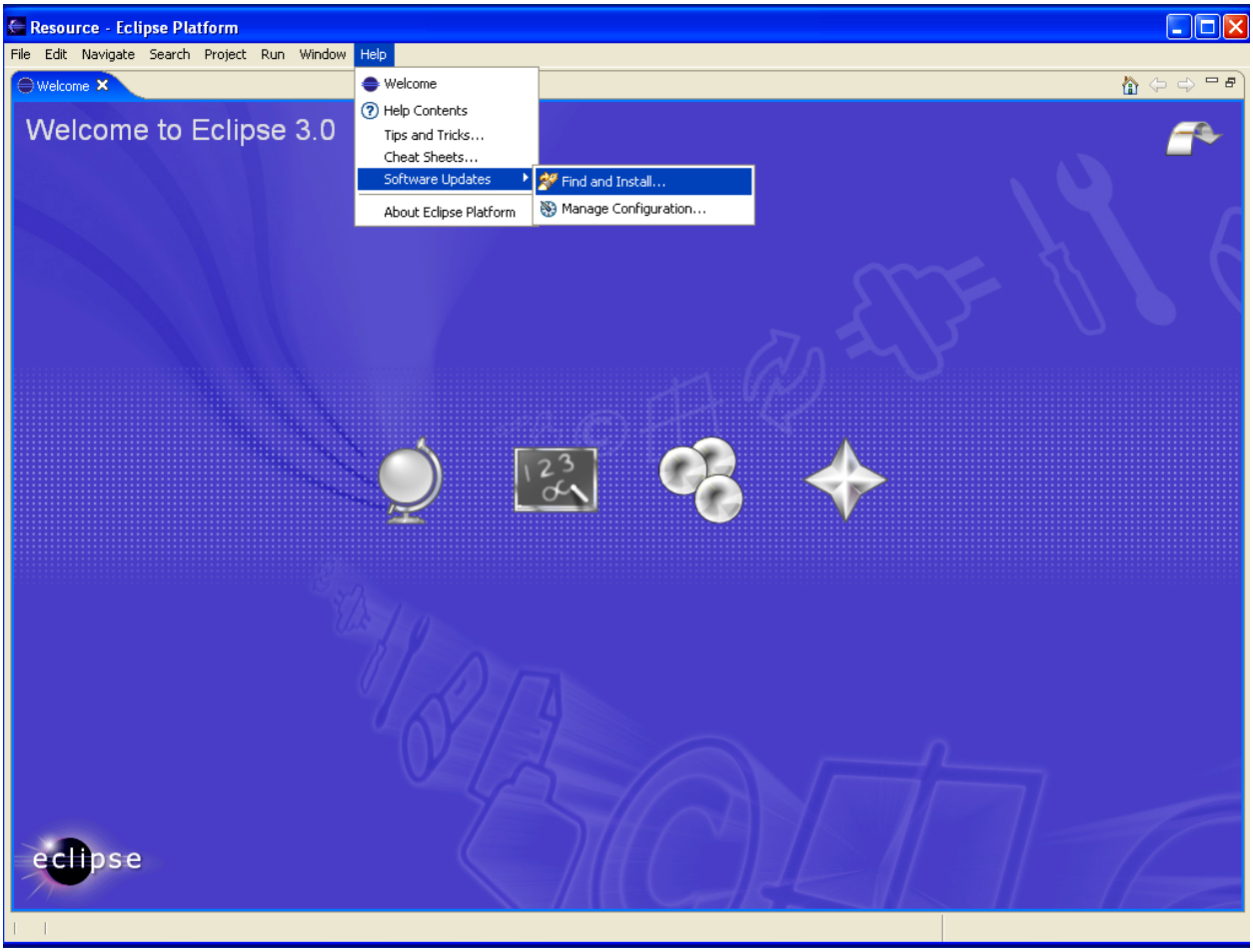
The CDT plugin selected must be compatible with the **Eclipse SDK 3.0.2** release we just installed! To determine this, click on the following CDT link.

http://update.eclipse.org/tools/cdt/releases/new

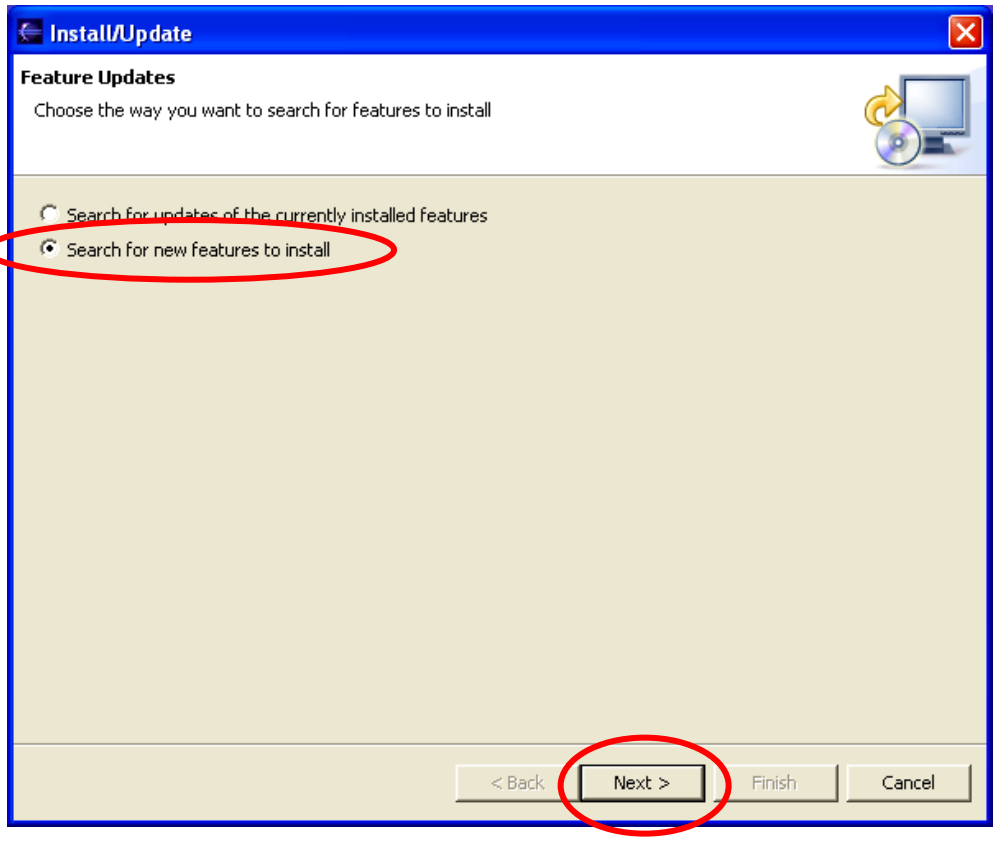


We can let Eclipse install the CDT plugins for us.

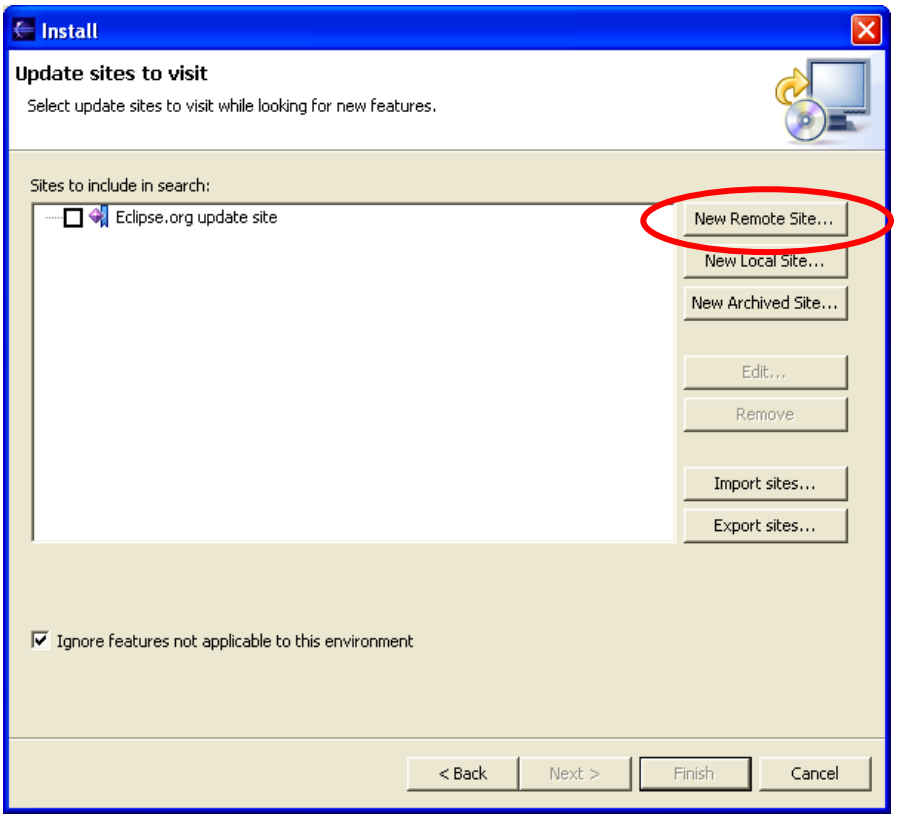
Start up Eclipse and click on "**Help – Software Updates – Find and Install …** "



Select the "**Search for new features to install**" radio button and click "**Next**"



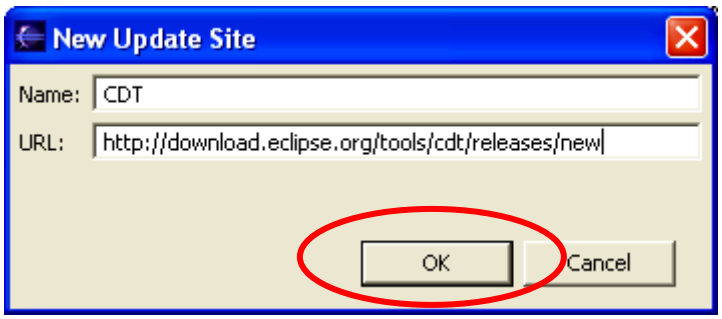
We need to specify the CDT update site, so click on "**New Remote Site**."



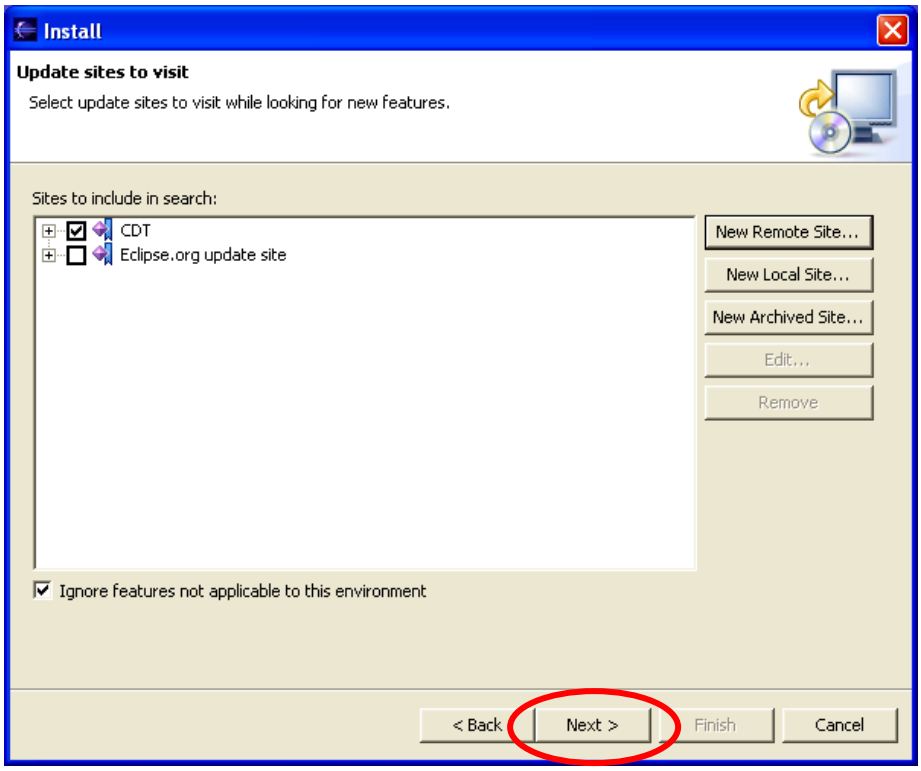
If you still have the CDT update URL saved above, paste it into the URL text box below.

http://download.eclipse.org/tools/cdt/releases/new

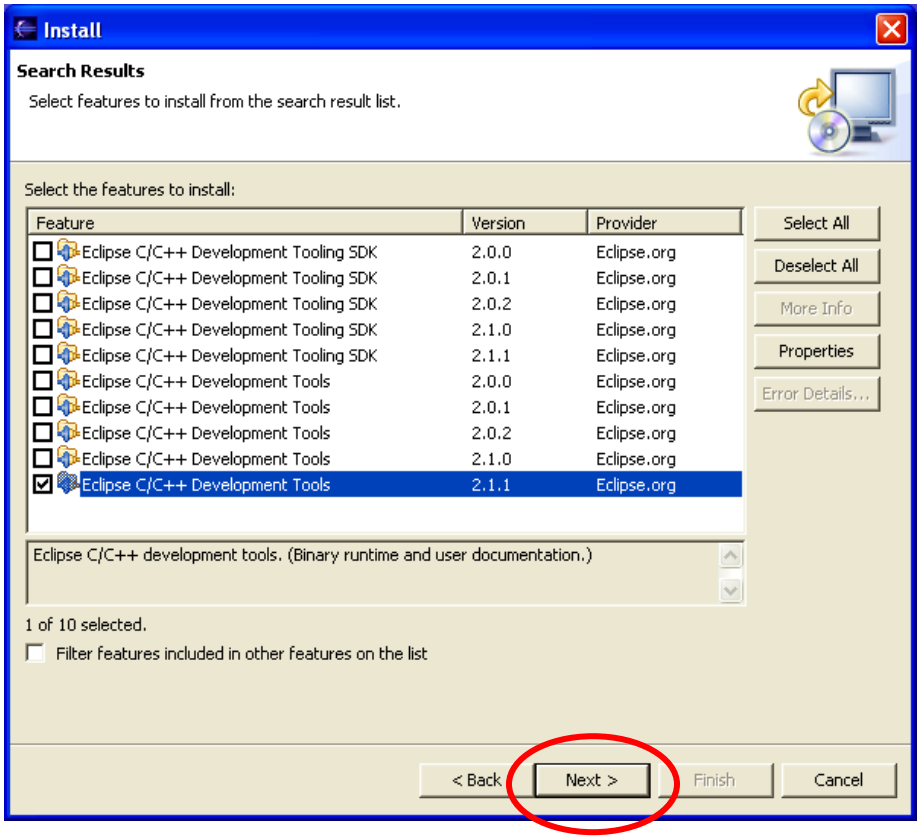
Enter "**CDT**" into the Name text box and click **OK**.



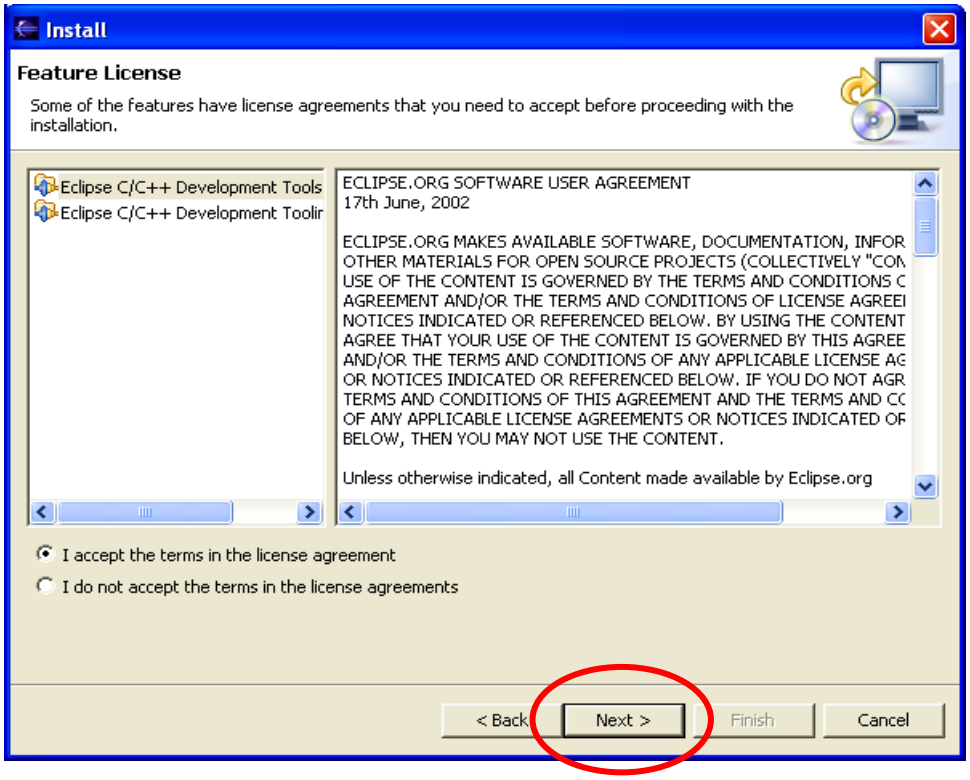
Now check the “**CDT**” box and click “**Next**” to continue the CDT installation.



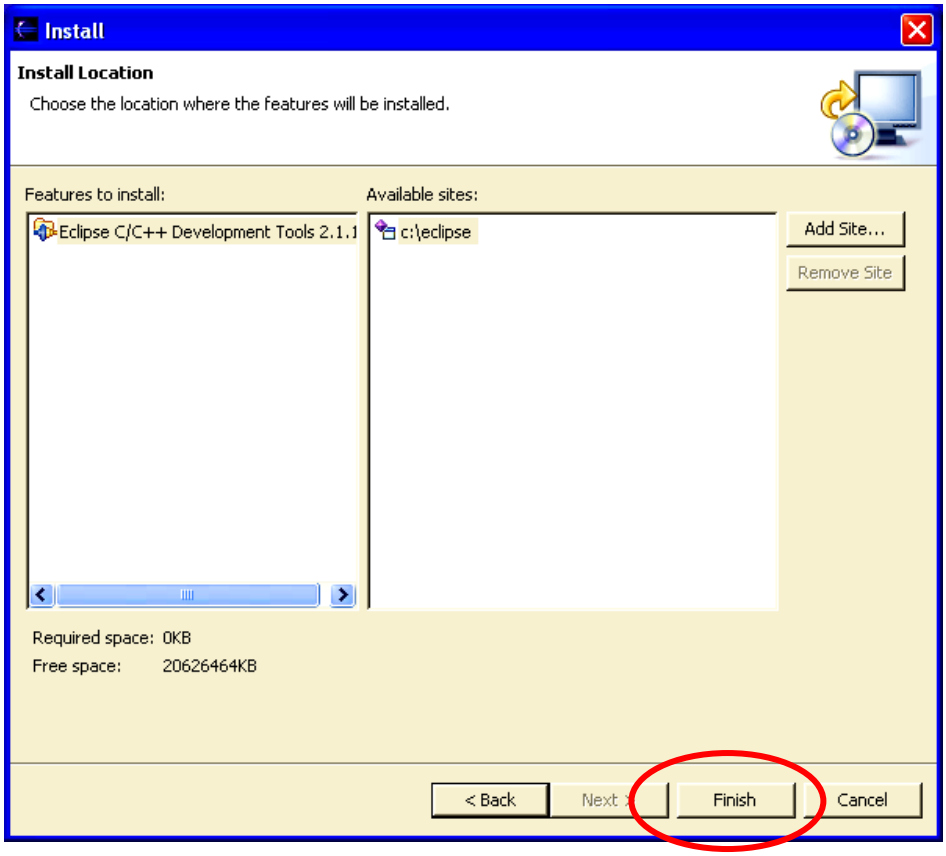
Check the box for “**Eclipse C/C++ Development Tools 2.1.1**” in the window below and click “**Next**” to continue the installation.



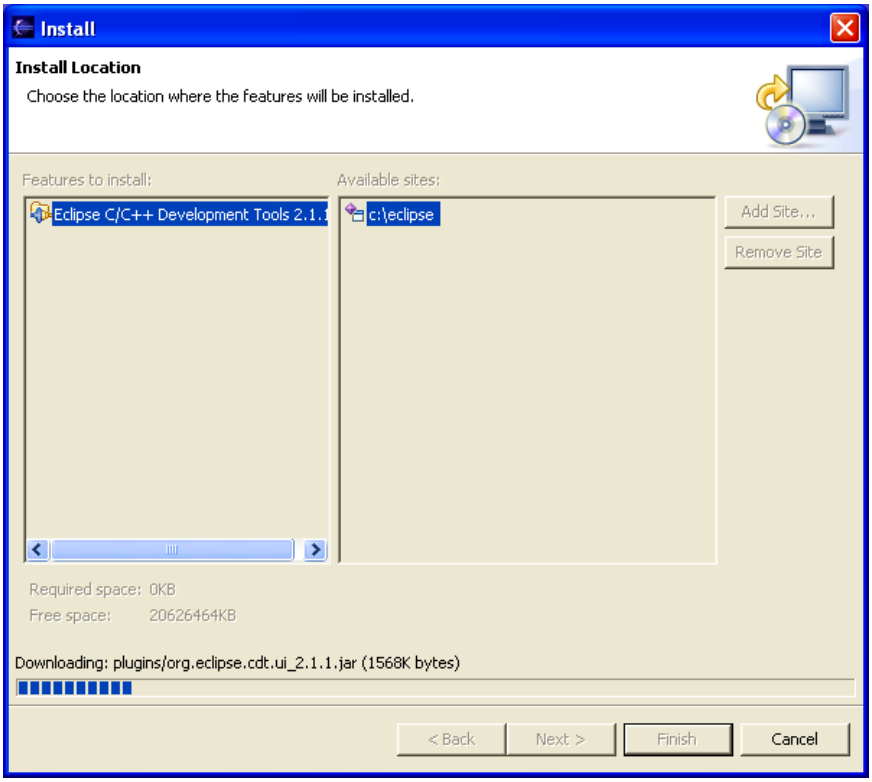
Accept the license agreement and click “**Next**.”



Take the defaults on the next screen and click “**Finish**” to start the CDT installation.

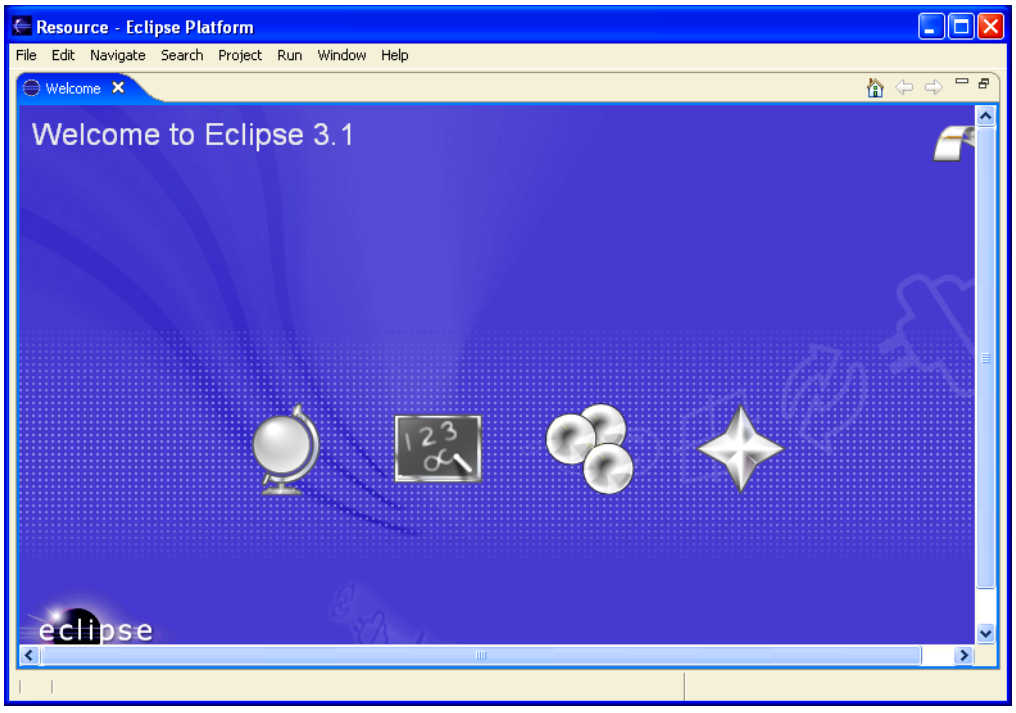


The CDT installation will start up.

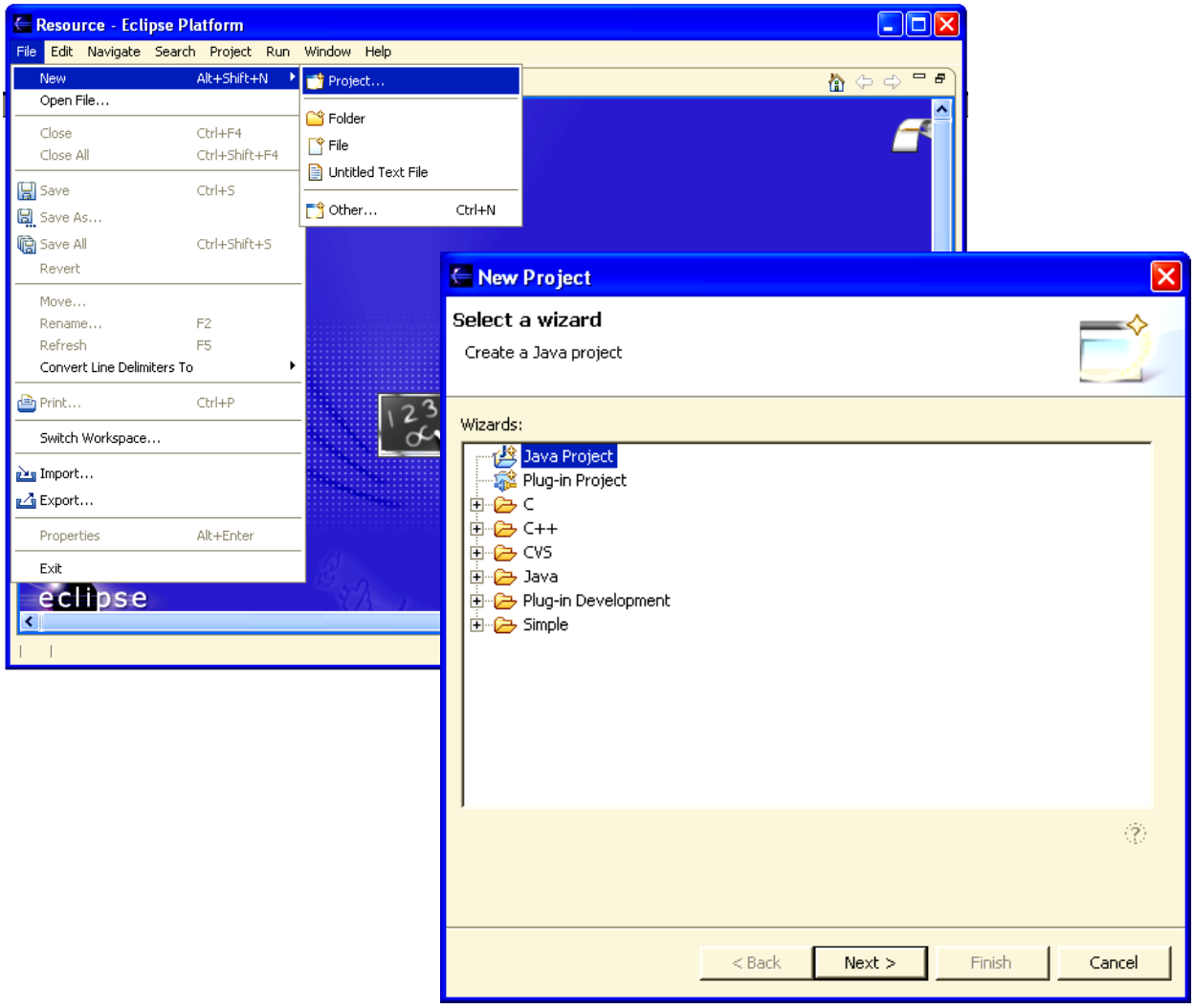


When the CDT download and installation completes, a pop-up window will advise you to restart Eclipse. Answer yes and when Eclipse restarts, it will have the CDT plugins properly installed. Eclipse must be restarted for the CDT plugins to take effect.

Let's now test if the CDT Plugin installation was successful. If Eclipse is not already running, click on the desktop Eclipse icon to start up the Eclipse IDE (Integrated Development Environment).



Now click on “**File – New – Project**”. This should display the “new project” window.



If you see **C** and **C++** as possible projects to select, then you know that the CDT has been installed successfully. Exit from Eclipse and we’ll proceed to downloading the Cygwin GNU Toolkit.

6 CYGWIN GNU Toolset for Windows

The GNU toolset is an open-source implementation of a universal compiler suite; it provides C, C++, ADA, FORTRAN, JAVA, and Objective C. All these language compilers can be targeted to most of the modern microcomputer platforms (such as the ARM 32-bit RISC microcontrollers) as well as the ubiquitous Intel/Microsoft PC platforms. By the way, GNU stands for “GNU, not Unix”, really – I’m serious!

Unfortunately for all of us that have desktop Intel/Microsoft PC platforms, the GNU toolset was originally developed and implemented with the Linux operating system. To the rescue came Cygwin, a company that created a set of Windows dynamic link libraries that trick the GNU compiler toolset into thinking that it’s running on a Linux platform. If you install the GNU compiler toolset using the Cygwin system, you can literally open up a DOS command window on your screen and type in a DOS command like this:

```
>arm-elf-gcc   –g  –c     main.c
```

The above will compile the source file **main.c** into an object file **main.o** for the ARM microcontroller architecture. In other words, if you install the Cygwin GNU toolset properly, you can forget that the GNU compiler system is Linux-based.

Normally, the Cygwin installation gives you a compiler toolset whose target architecture is the Windows/Intel PC. It does not include a compiler toolset for the ARM microprocessors, the MIPS microprocessors, and so forth.

It is possible to build a compiler toolset for the ARM processors using the generic Cygwin GNU toolkit. In his book “**Embedded System Design on a Shoestring**”, Lewin A.R.W. Edwards gives detailed instructions on just how to do that. Fortunately, there are quite a few pre-built tool chains on the internet that simplify the process. One such tool chain is GNUARM which gives you a complete set of ARM compilers, assemblers and linkers. This will be done in the next section of this tutorial.

It’s worth mentioning that the GNUARM tool chain doesn’t include the crucial MAKE utility, it’s in the Cygwin tool kit we’re about to install. This is why you have to add two path specifications to your Windows environment; one for the **c:/cygwin/bin** folder and one for the **c:/programfiles/gnuarm/bin**.

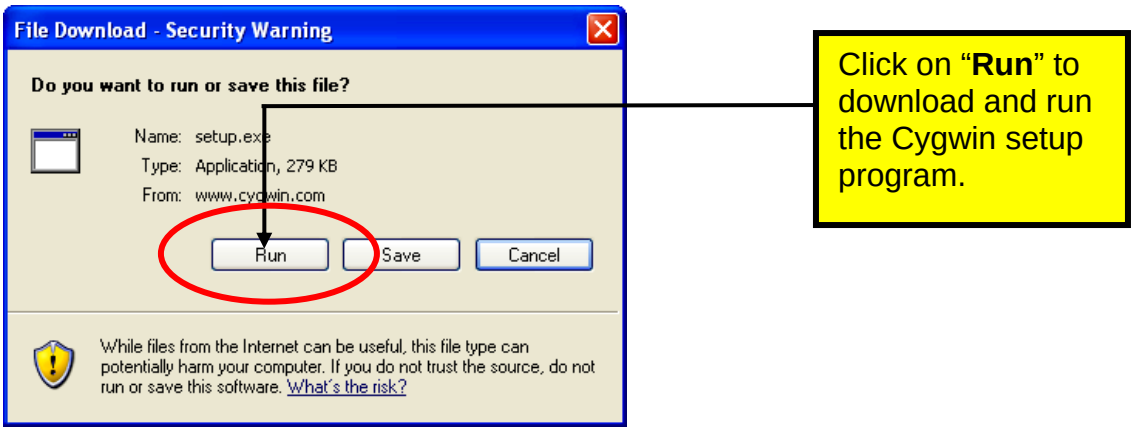
The Cygwin site that has the GNU toolset for Windows is:

www.cygwin.com

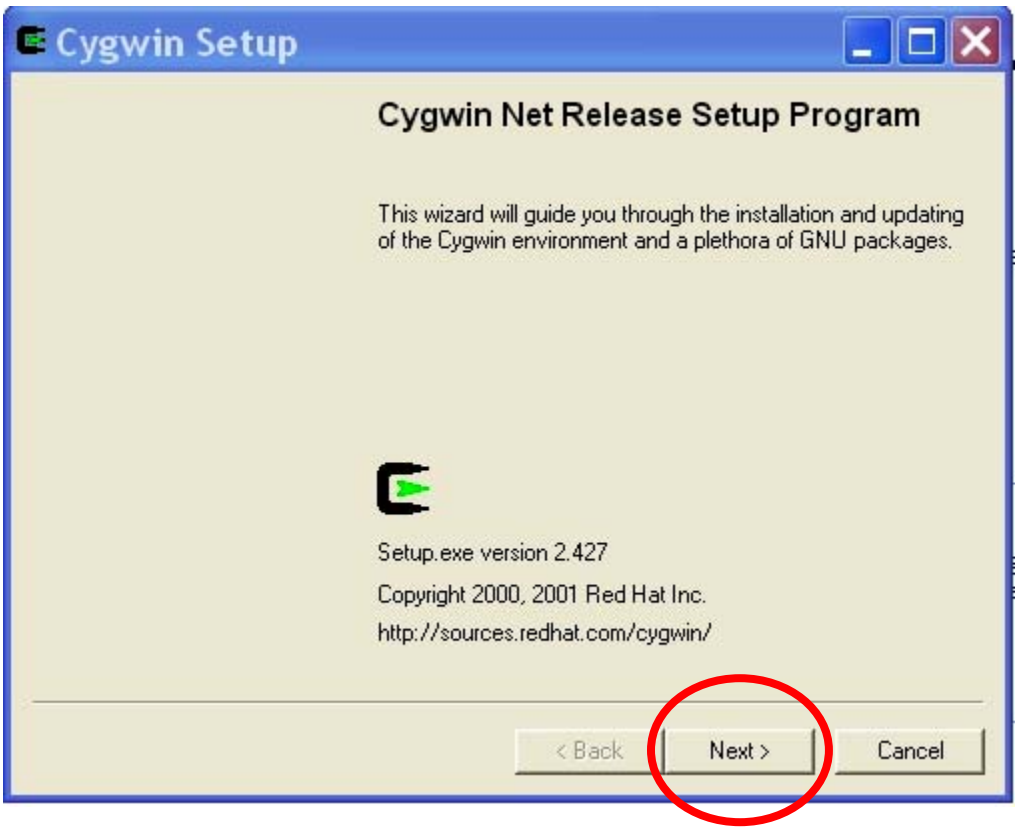
The Cygwin web site opens as follows:

The first thing to do is to click on the install icon:

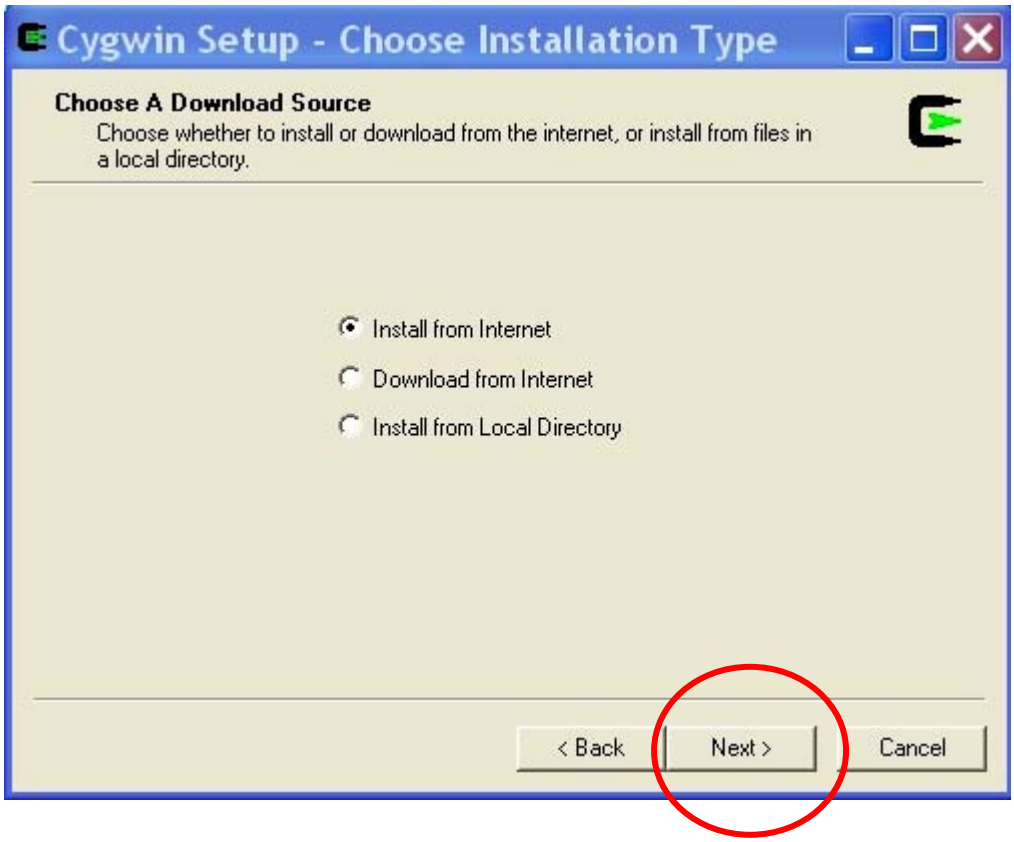
We need to download the setup executable and automatically run it.



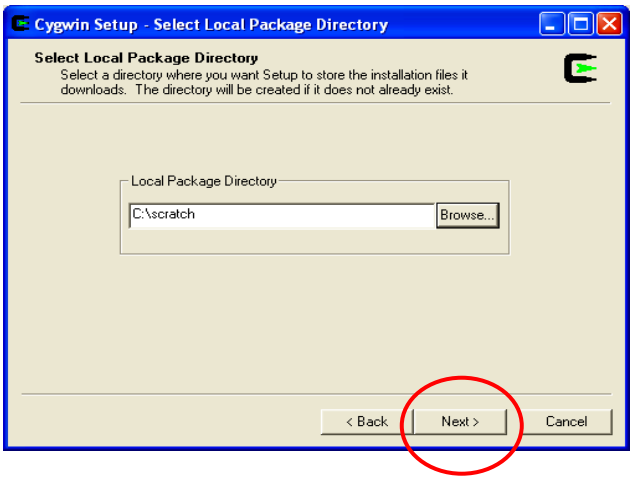
Now the Cygwin wizard will start up. Select “**Next**” to continue.



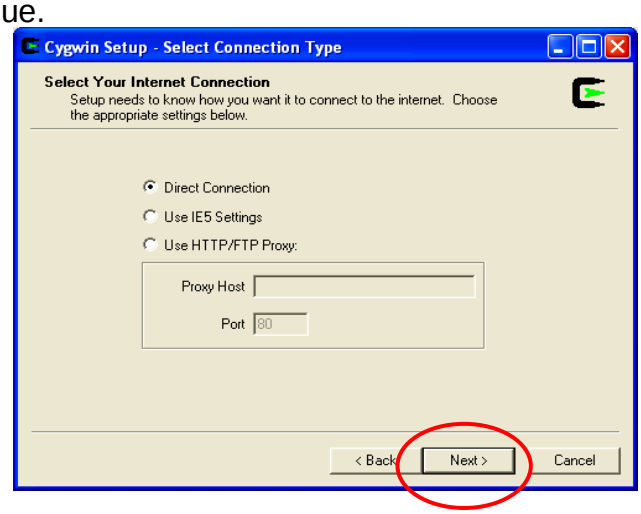
Choose “**Install from Internet**” and then click “**Next**.”



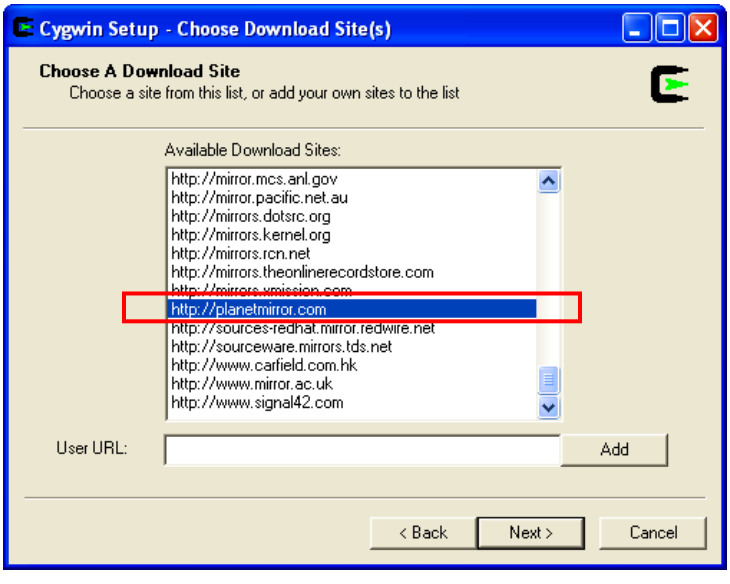
Now we specify a directory where all the downloaded components go, our **c:/scratch** folder will do just fine.



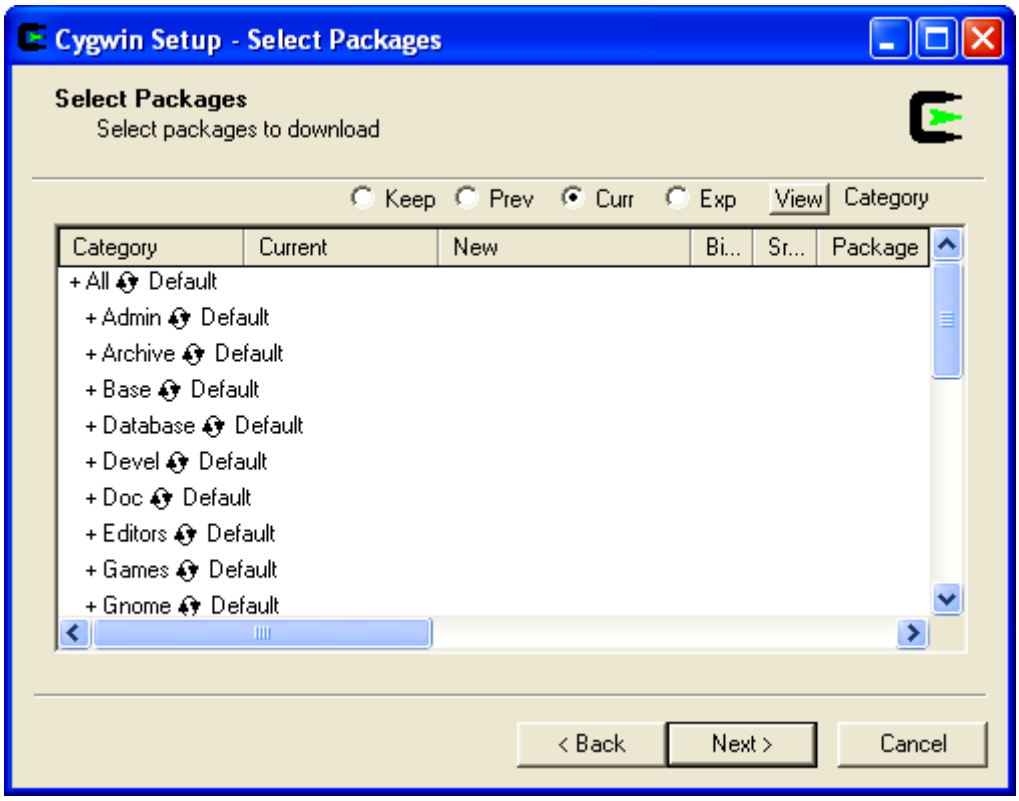
Since I have a high speed internet connection, I always select "**Direct Connection**." Click "**Next**" to continue.



Now the Cygwin Installer presents you with a list of mirror sites that can deliver the Cygwin GNU Toolkit. It's a bit of a mystery which one to choose; I picked http://planetmirror.com because it sounds cool. You may have to experiment to find one that downloads the fastest. Click "**Next**" to continue.



Cygwin will download a few bits for a couple of seconds and then display this “Select Packages” list allowing you to tailor exactly what is included in the down load.



The screen above allows you to specify what GNU packages you wish to install.

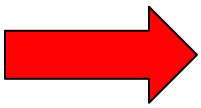
Basically, we want an installation that will allow us to compile for the Windows XP / Intel platform. This will allow us to use Eclipse to build Windows applications (not covered in this document). Remember that we’ll be installing the GNUARM suite of compilers, linkers etc. for the ARM processor family shortly.

If you look at the Cygwin “Select Packages” screen below, you’ll see the following line.

You must click on the little circle with the two arrowheads until the line changes to this:

This will force installation of the default GNU compiler suite for Windows/Intel targets. Here's the "**Select Packages**" screen before clicking on the circle with arrowheads. The following four packages must be selected and changed from "**default**" to "**install**."

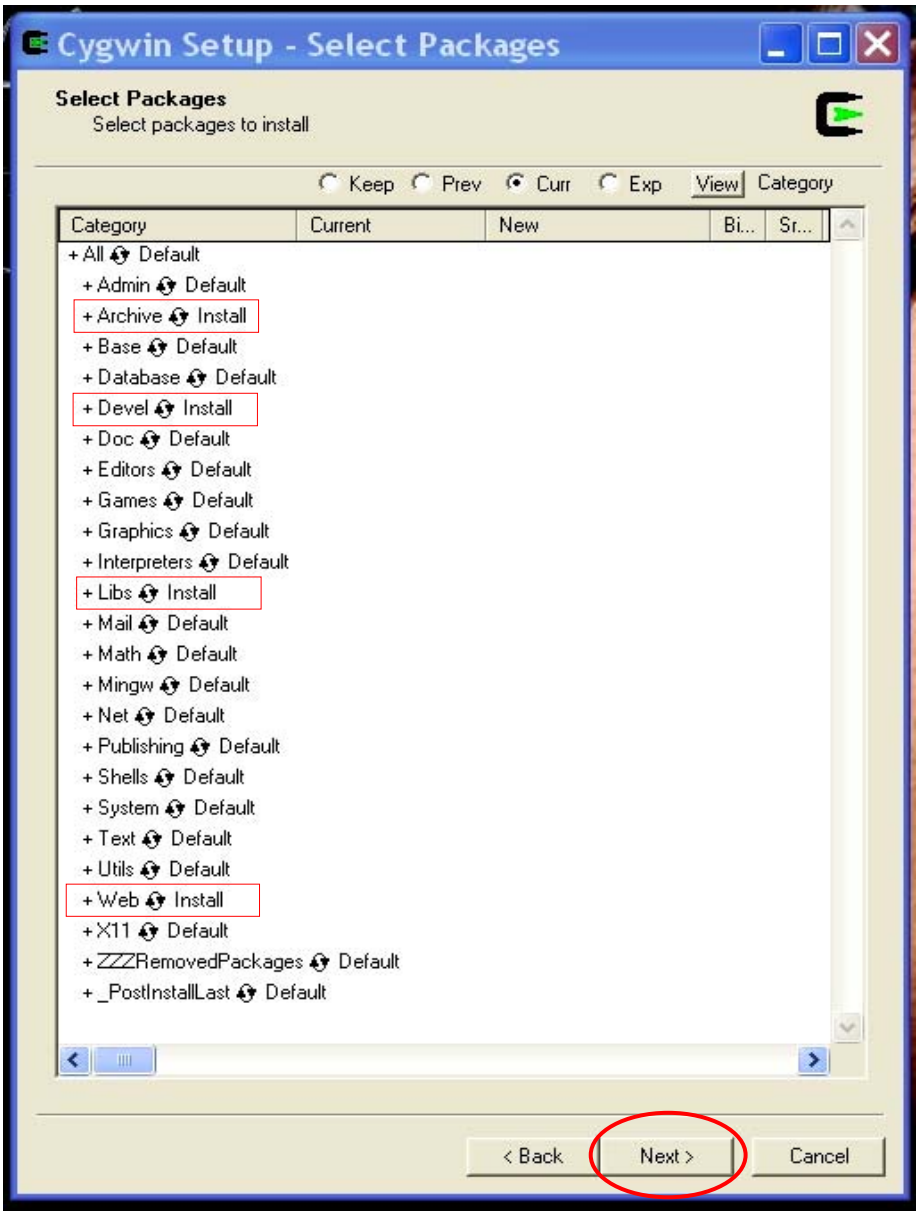
Archive ↻ Default
Devel ↻ Default
Libs ↻ Default
Web ↻ Default



Archive ↻ Install
Devel ↻ Install
Libs ↻ Install
Web ↻ Install

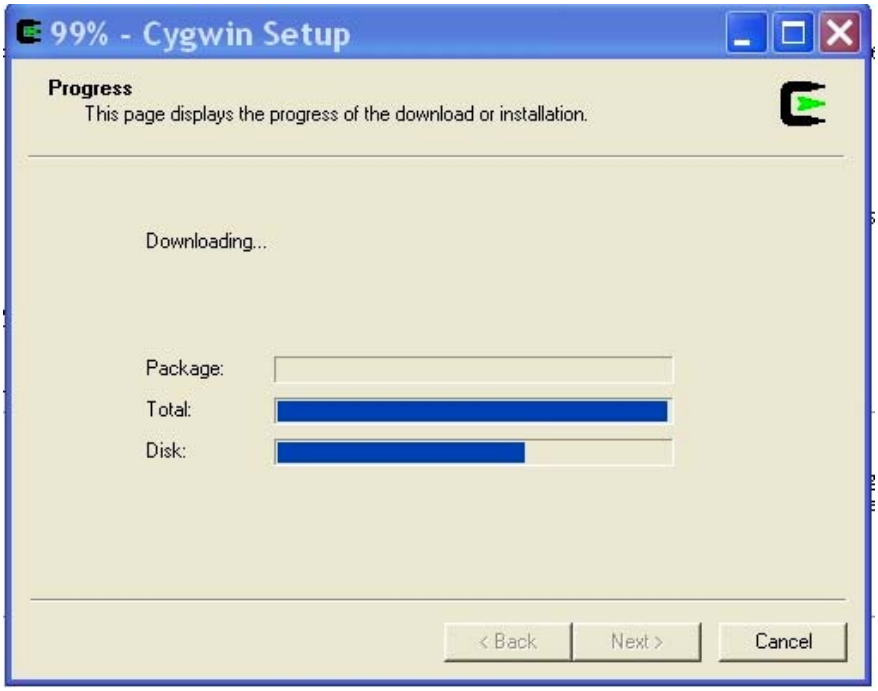
Click on the little circle with the arrowheads until you change the four packages listed above from "**default**" to "**install**."

You should see the screen displayed directly below. Note that the Archive, Devel, Libs and Web components are selected for "Install". Everything else is left as "default."



Click "**Next'** to start the download.

Now the Cygwin will start downloading. This creates a huge 700 Megabyte directory on your hard drive and takes 30 minutes to download and install using a cable modem.



When the installation completes, Cygwin will ask you if you want any desktop icons and start menu entries set up. Say "**No**" to both. These icons allow you to bring up the BASH shell emulator (like the command prompt window in Windows XP). This would allow you do some Linux operations, but this capability is not necessary for our purposes here. Click on "Finish" to complete the installation.

Now the Cygwin installation manager completes and shows the following result.

The directory **c:\cygwin\bin** must be added to the **Windows XP** path environment variable. This allows Eclipse to easily find the Make utility, etc.

Using the **Start Menu**, go to the **Control Panel** and click on the "**System**" icon.

Then click on the "**Advanced**" tab and select the "**Environment Variables**" icon. Highlight the "**Path**" line and hit the "**Edit**" button. Add the addition to the path as shown in the dialog box shown below (don't forget the semicolon separator). The Cygwin FAQ advises putting this path specification before all the others, but it worked for me sitting at the end of the list.

We are now finished with the CYGWIN installation. It runs silently in the background and you should never have to think about it again.

7 Downloading the GNUARM Compiler Suite

At this point, we have all the GNU tools needed to compile and link software for Windows/Intel computers. It is possible to use all this to build a custom GNU compiler suite for the ARM processor family. The very informative book "**Embedded System Design on a Shoestring**" by Lewin A.R.W. Edwards ©2003 describes how to do this and it is rather involved.

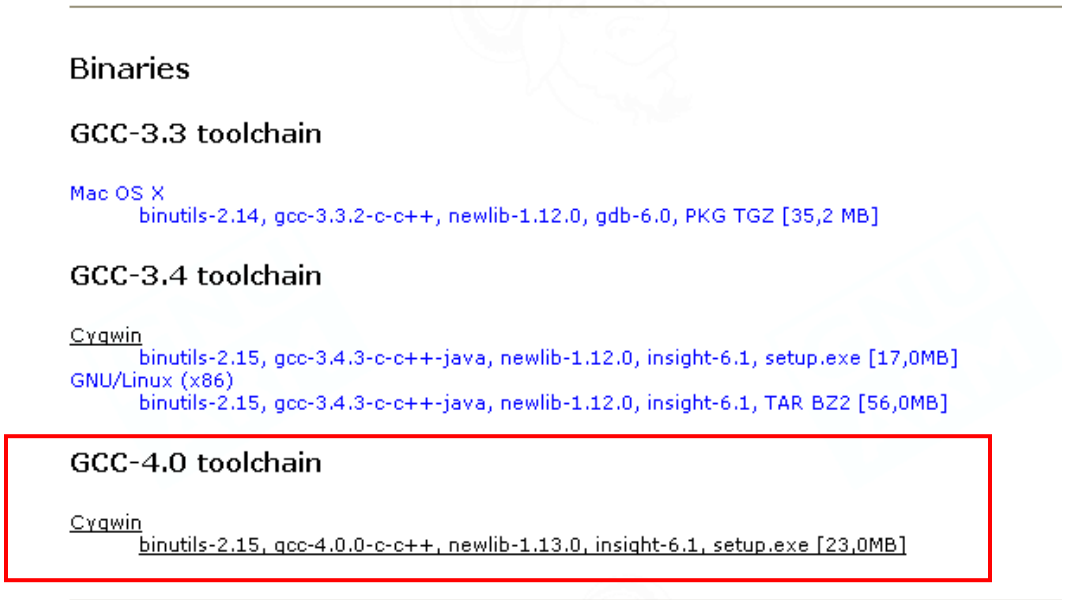
Fortunately, Pablo Bleyer Kocik and the people at **gnuarm.com** have come to the rescue with pre-built GNU compiler suite for the ARM processors. Just download it with the included installer and you're ready to go.

Click on the following link to download the GNUARM package.

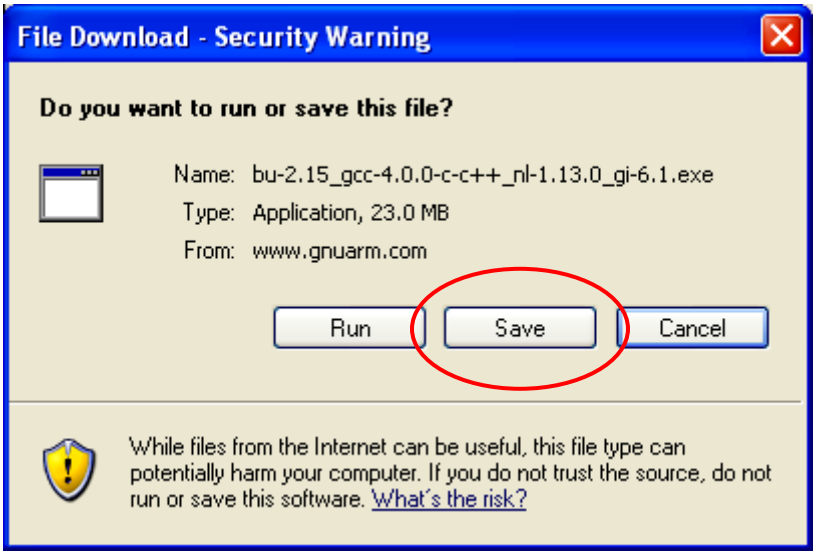
www.gnuarm.com

The GNUARM web site will display and you should click on the "**Files**" tab.

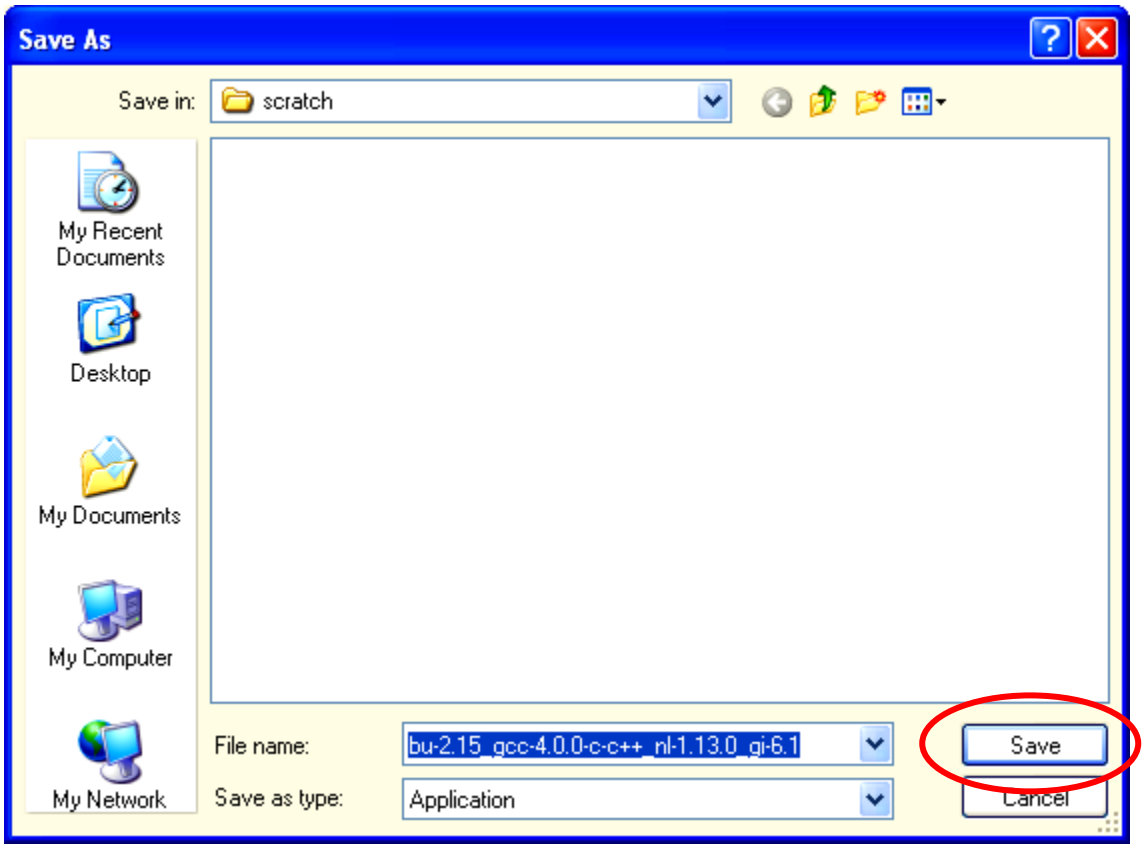
The correct package to download is **Binaries Cygwin – GCC- 4.0 toolchain**



Just like all the other downloads we've done, we direct this one to our empty download directory on the hard drive. Here we click "**Save**" and then specify the download destination.

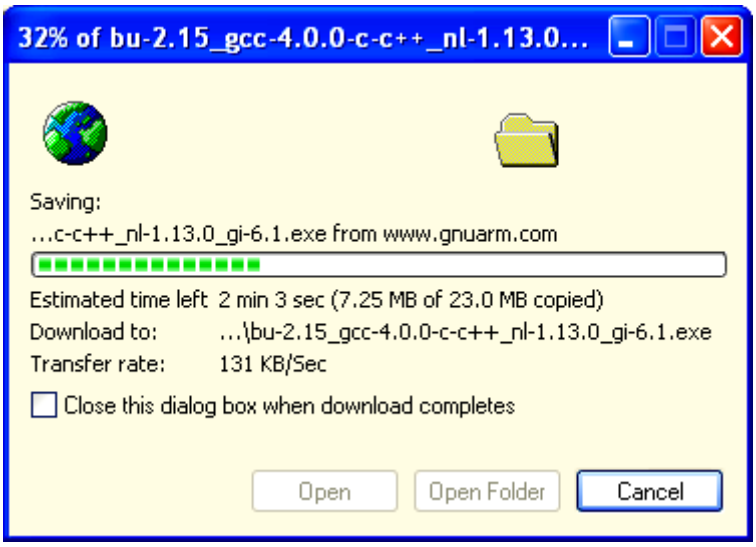


Once again, our **c:/scratch** directory will suffice.



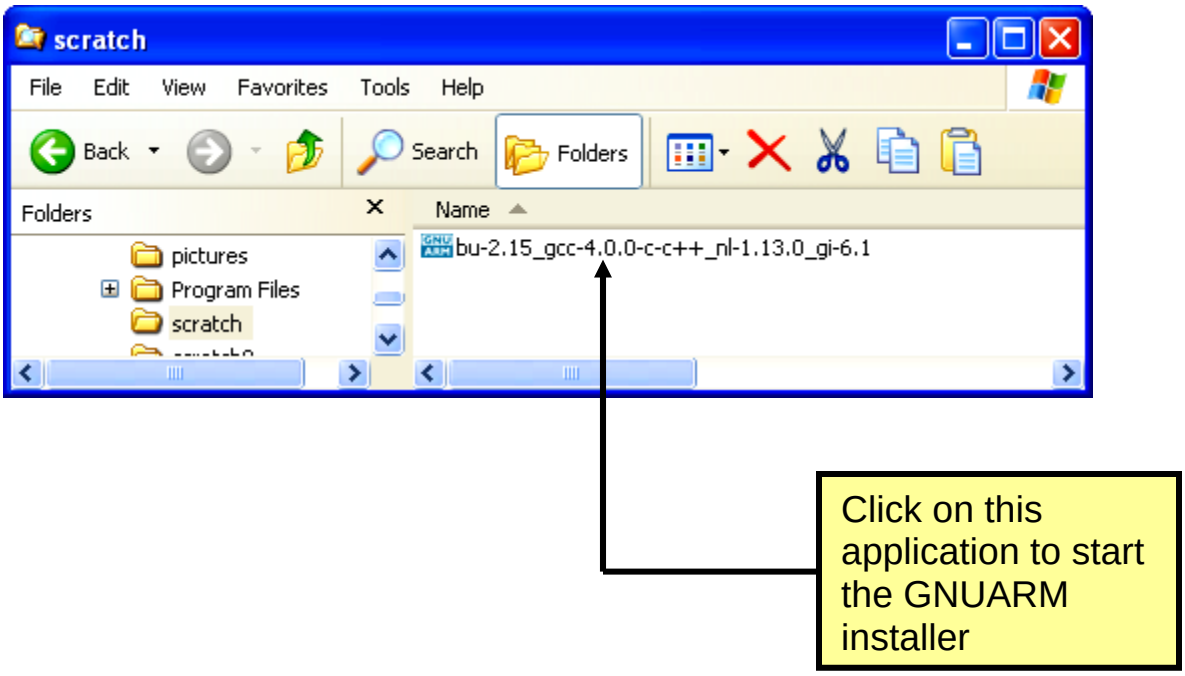
As you can see, this download has a very long name!

This download is a 18 megabyte file and takes 30 seconds on a cable modem.

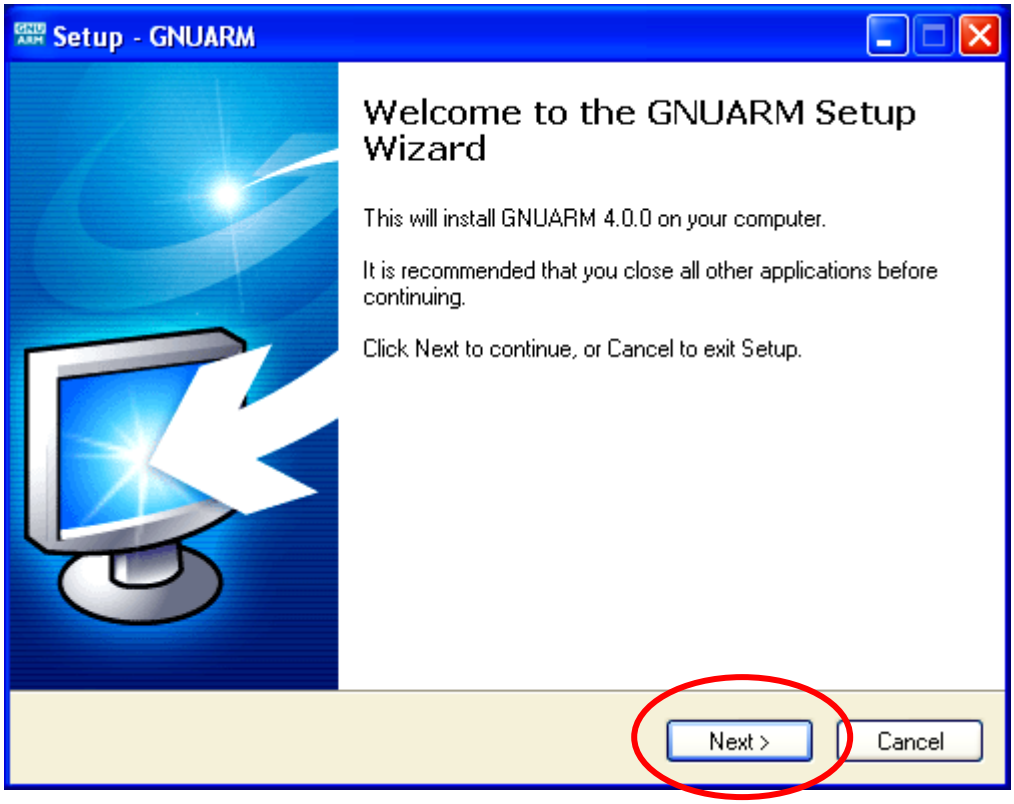


The download directory now has the following setup application with the following unintelligible filename: **bu-2.15_gcc-3.4.1-c-c++-java_nl-1.12.0_gi-6.0.exe**

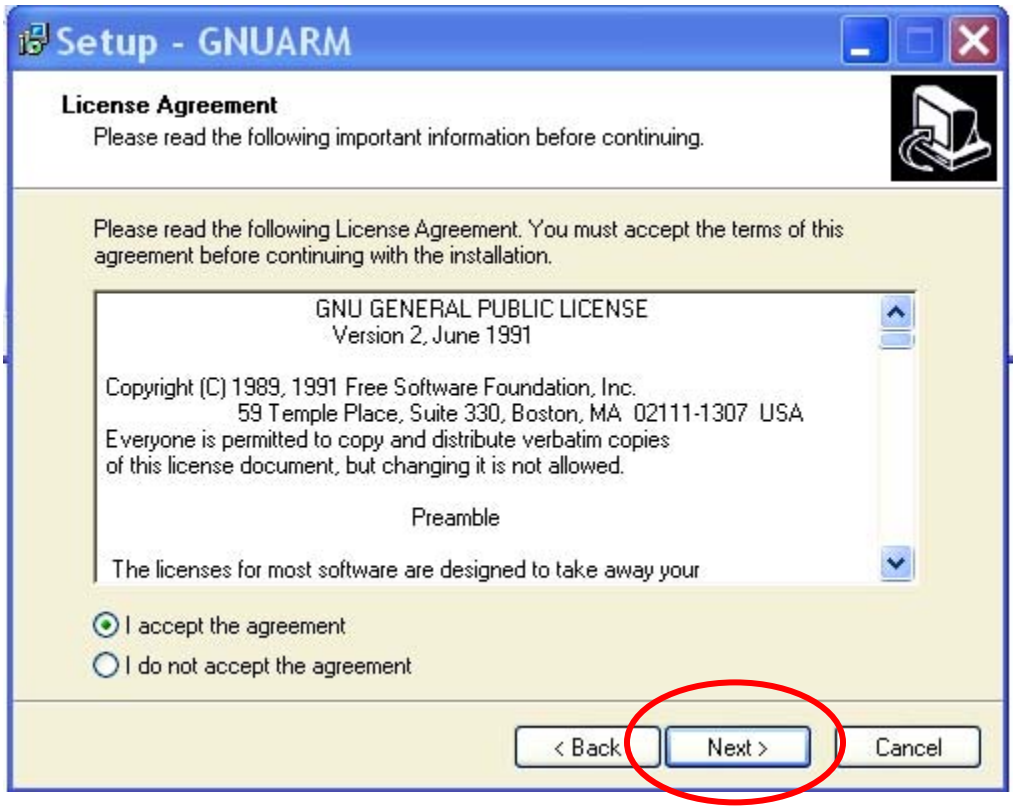
Click on that filename to start the installer.



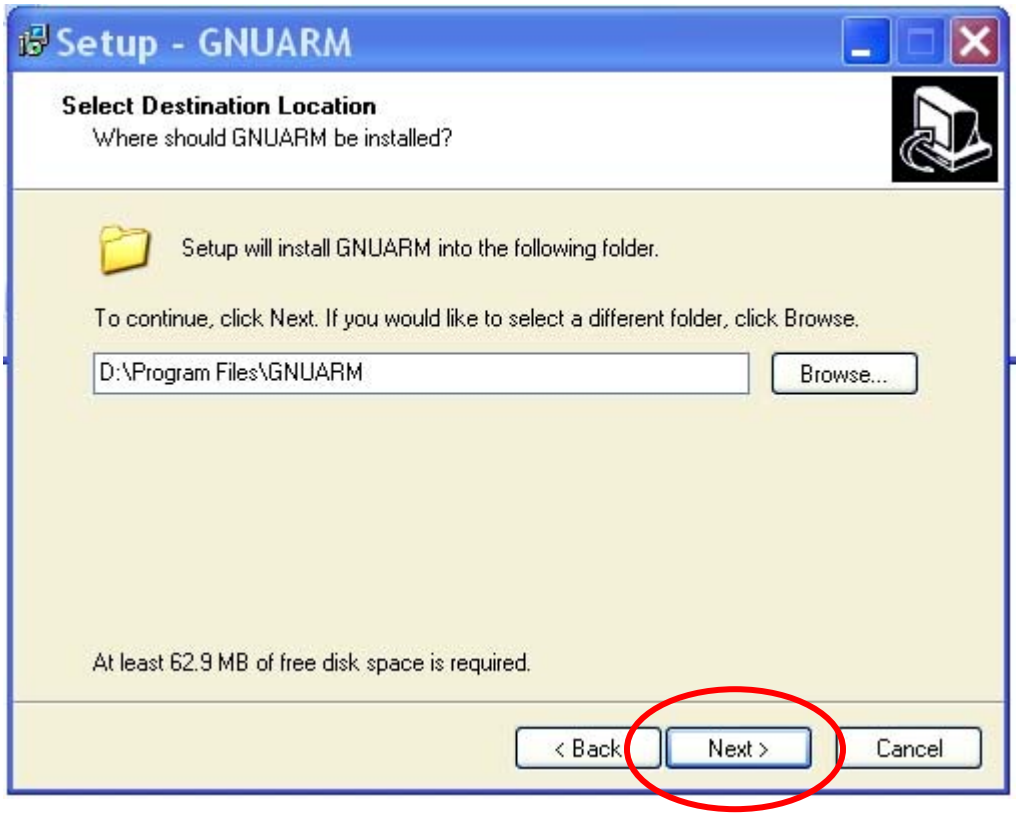
The GNUARM installer will now start. Click **“Next”** to continue.



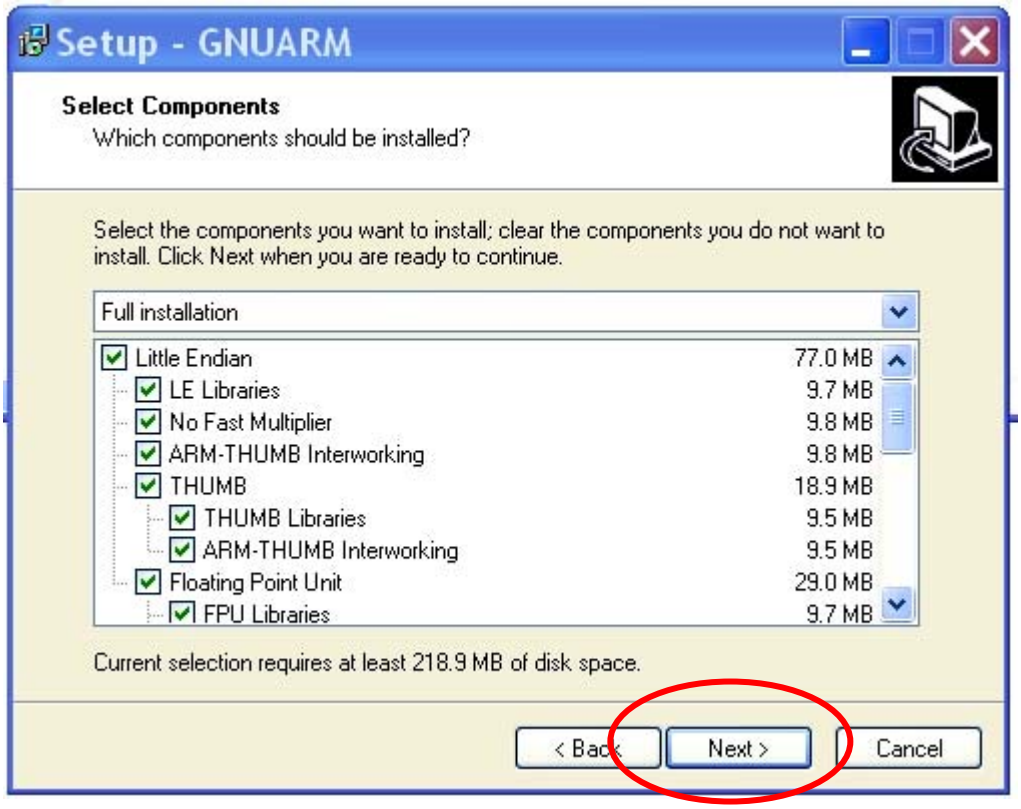
Accept the GNU license agreement – don’t worry, it’s still free. Click **“Next”** to continue.



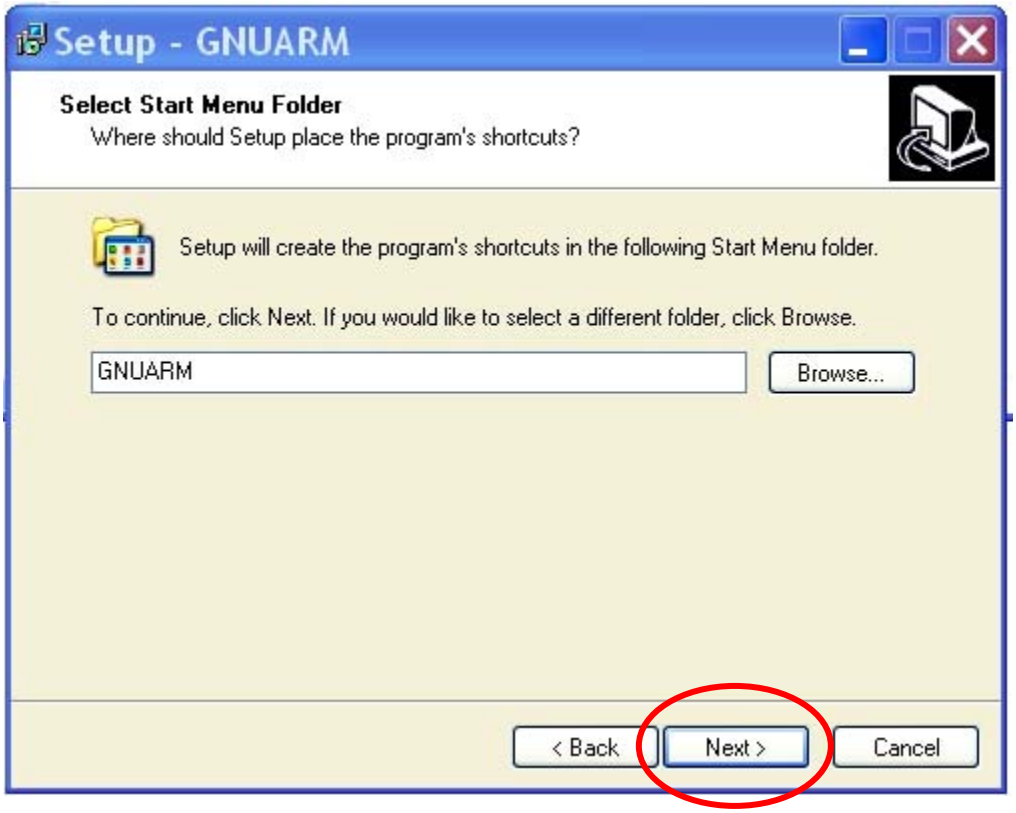
We'll take the default and let it install into the "**Program Files**" directory. Click **"Next"** to continue.



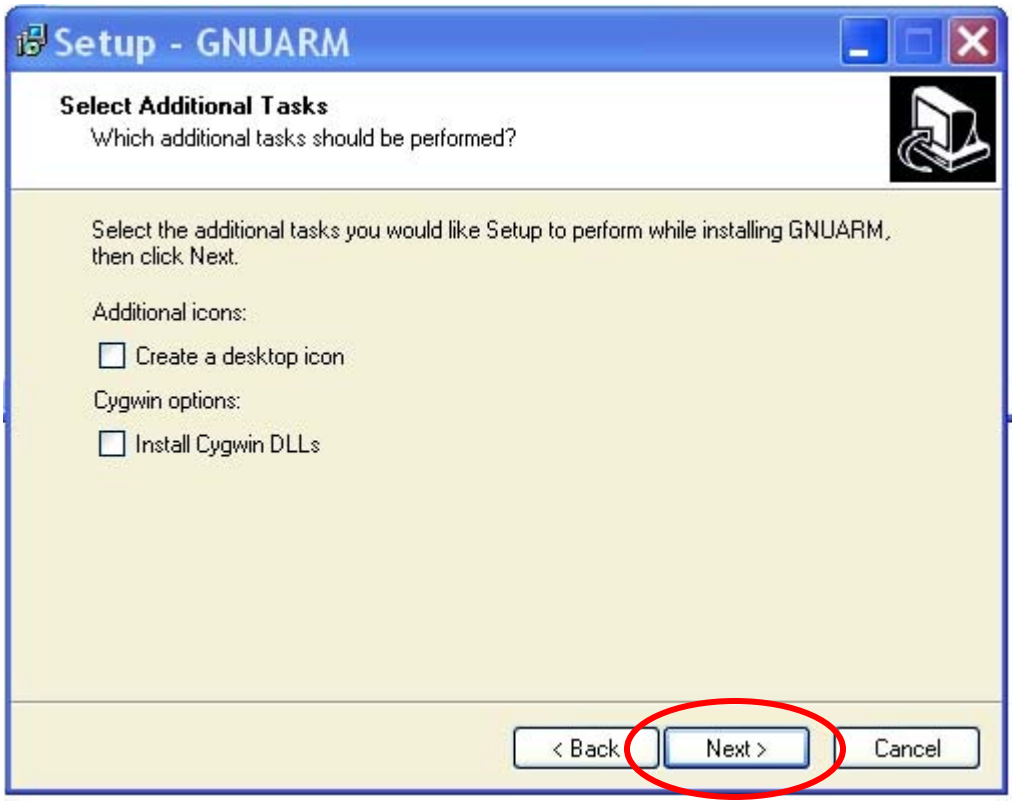
We'll also take the defaults on the "Select Components" window. Click **"Next"** to continue.



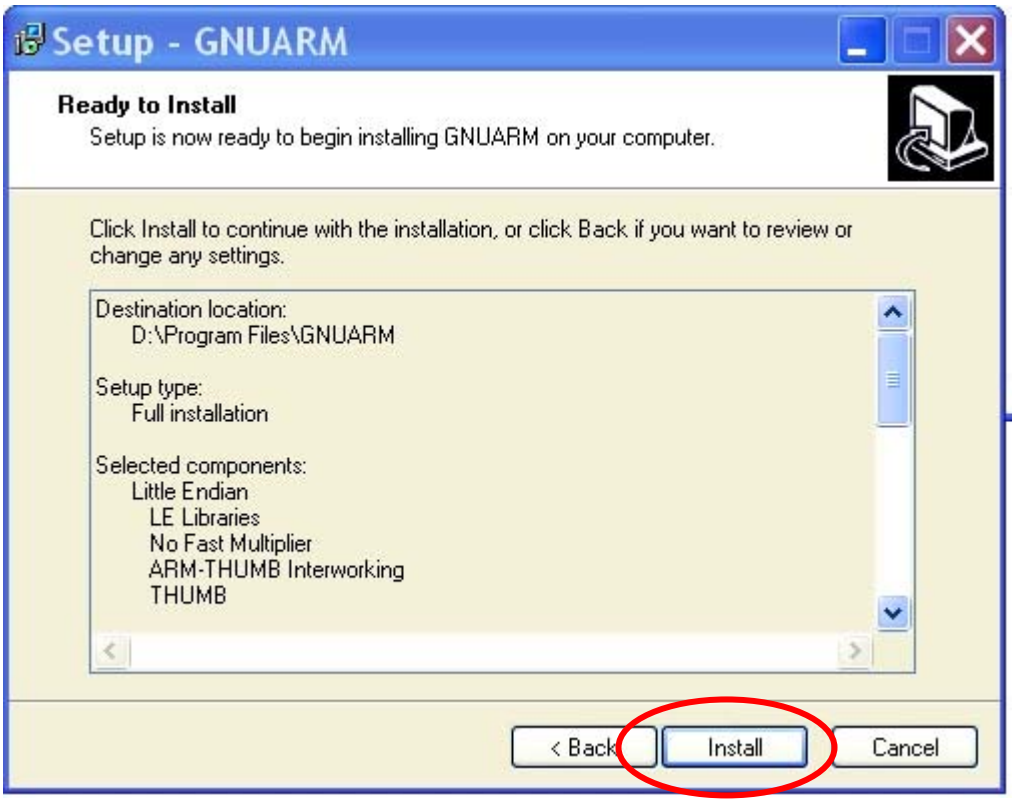
Take the default on this screen. Click **“Next”** to continue.



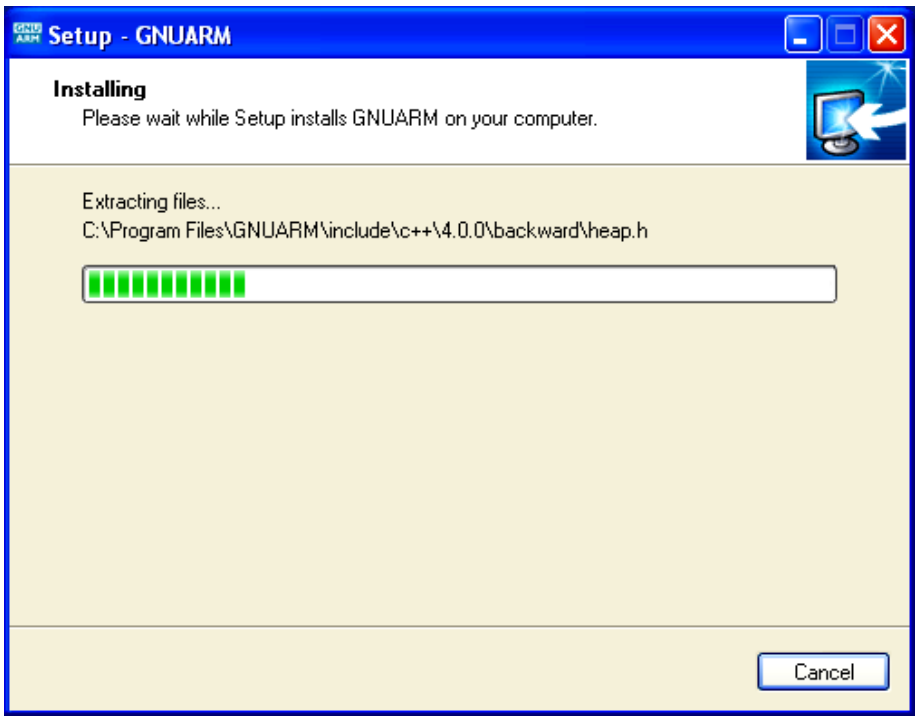
It’s very important that you don’t check **“Install Cygwin DLLs**”. We already have the Cygwin DLL installed from our Cygwin environment installation. Since all operations are called from within Eclipse, we don’t need a “**desktop icon**” either. Click **“Next”** to continue.



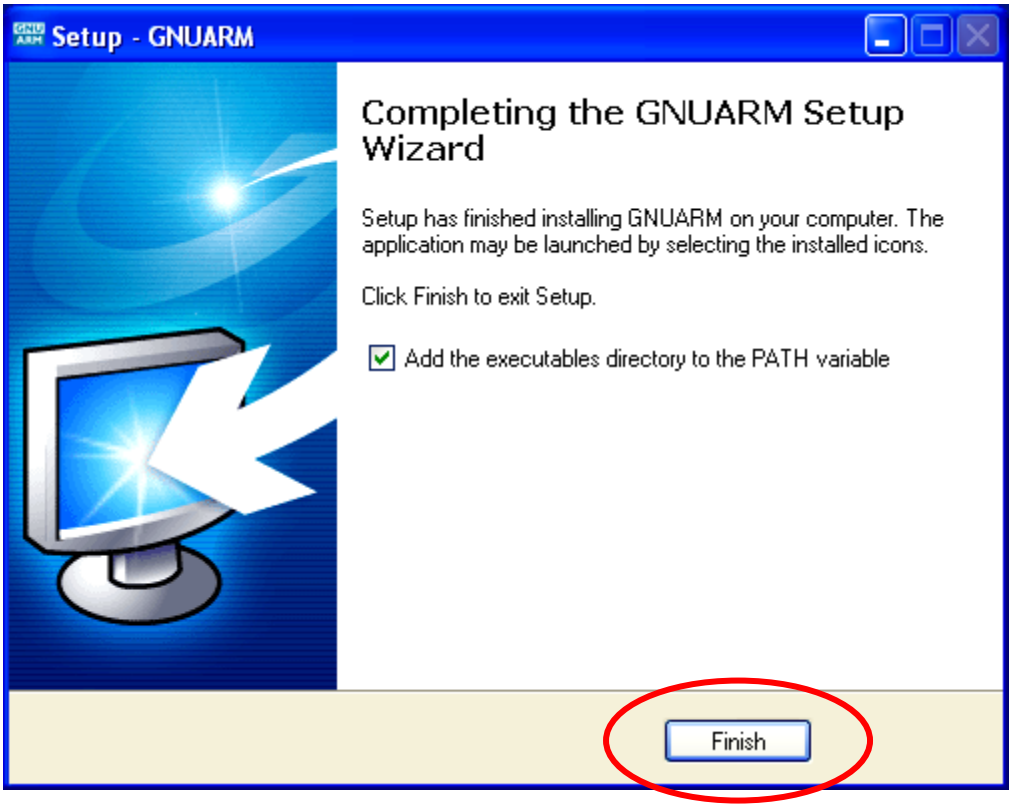
Click on “**Install**” to start the GNUARM installation.



Sit back and watch the GNUARM compiler suite install itself.



When it completes, the following screen is presented. Make sure that "**Add the executables directory to the PATH variable**" is checked. This is crucial.



This completes the installation of the compiler suites. Since Eclipse will call these components via the make file, you won't have to think about it again.

It's worth mentioning that the GNUARM web site has a nice Yahoo user group with other users posing and answering questions about GNUARM. Pay them a visit. The GNUARM web site also has links to all the ARM documentation you'll ever need.

8 Installing the Philips LPC2000 Flash Utility into Eclipse

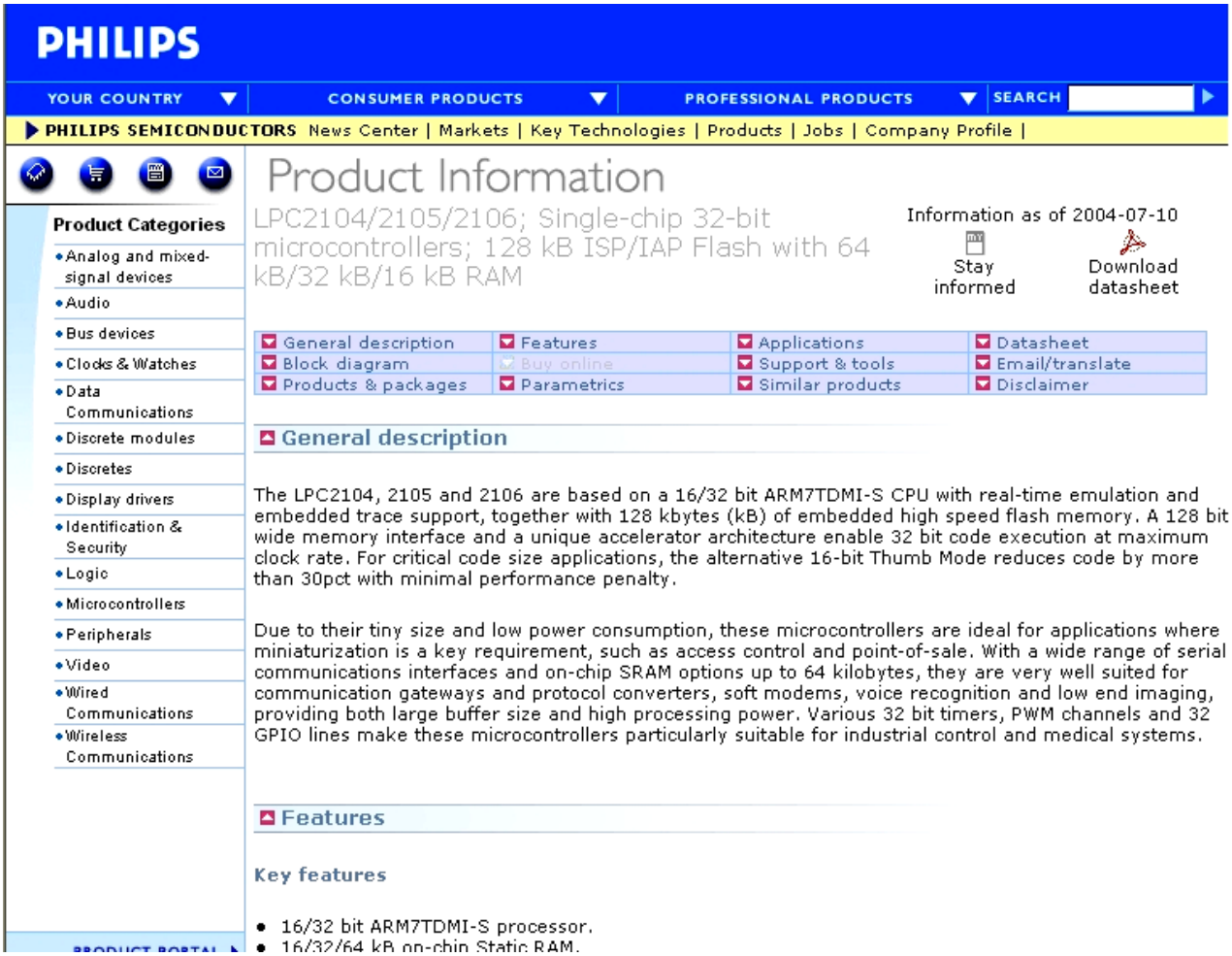
The Philips LPC2000 Flash Utility allows downloading of hex files from the COM1 port of the desktop computer to the **Olimex LPC-P2106** board's flash (or RAM) memory.

We need to download the latest version of this program from the Philips web site and unzip and install it into the **program files** directory. Then we will start Eclipse and add the LPC2000 Flash Utility as an external tool to be invoked.

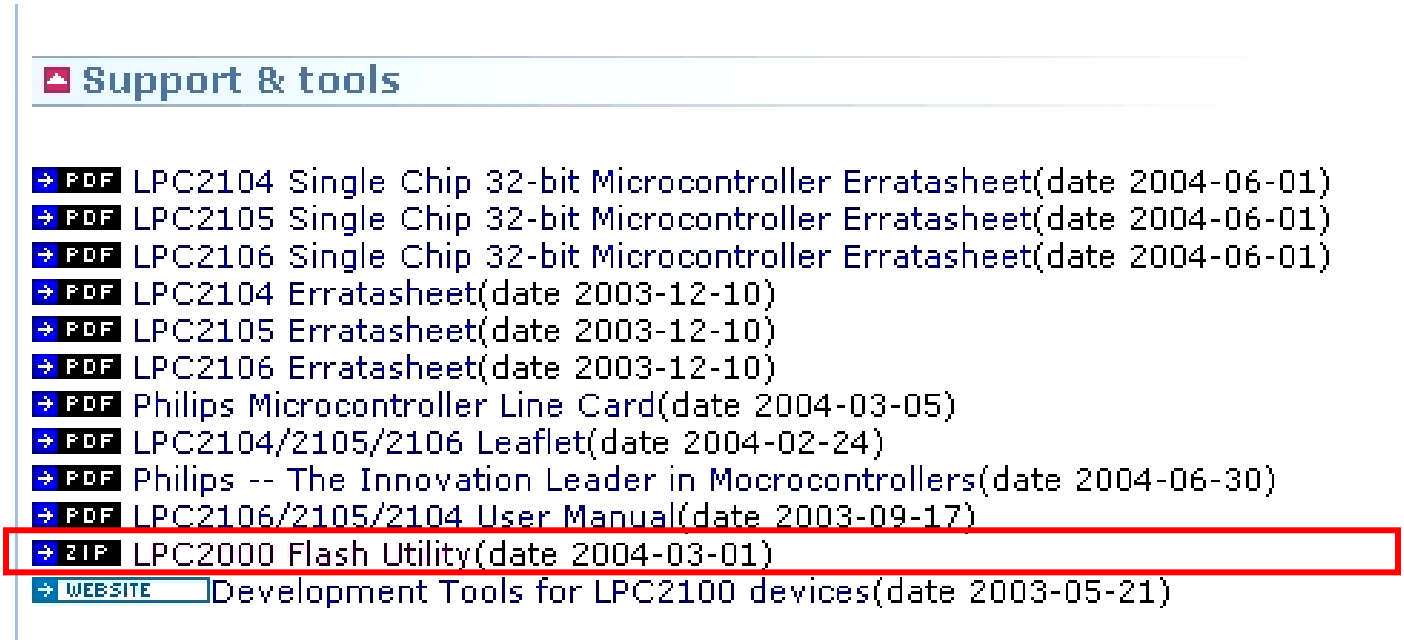
Click on the following link to access the Philips LPC2106 web page.

www.semiconductors.philips.com/pip/LPC2106.html

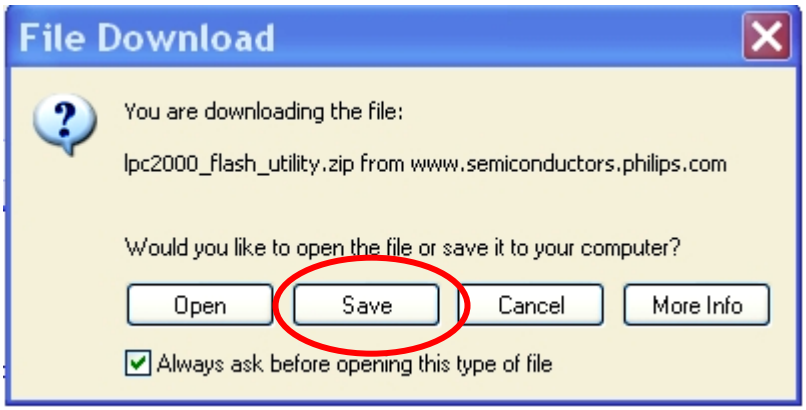
The following web page for the LPC2106 should open.

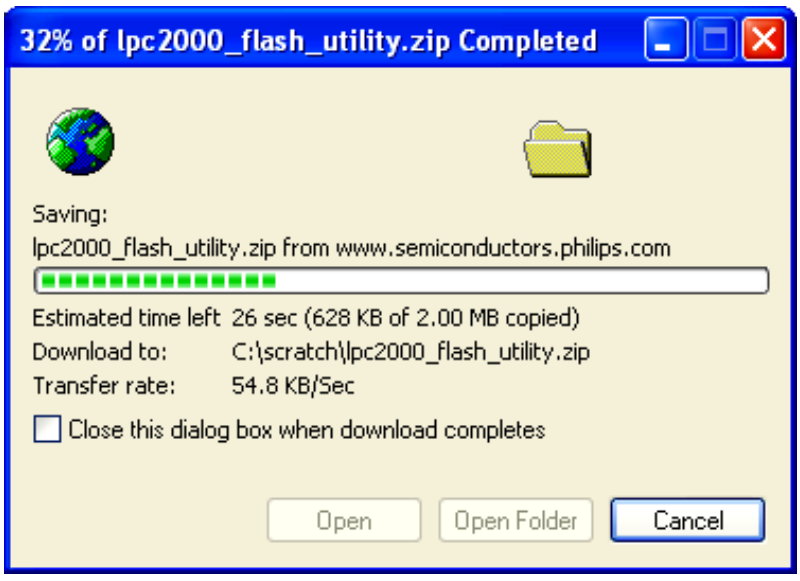


If you scroll down this page, you will see a link to the LPC2000 Flash Utility download. Click on the ZIP file LPC2000 Flash Utility (date 2004-03-01)

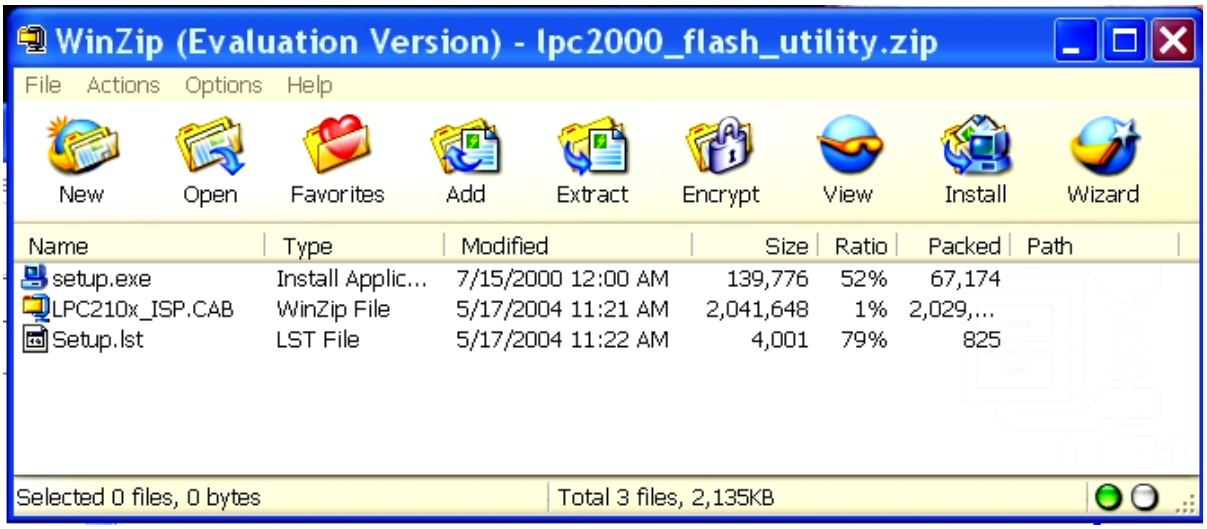


As before, we'll save the downloaded zip file in our empty **c:/scratch** directory. This is a fairly short download, only about 2 megabytes.

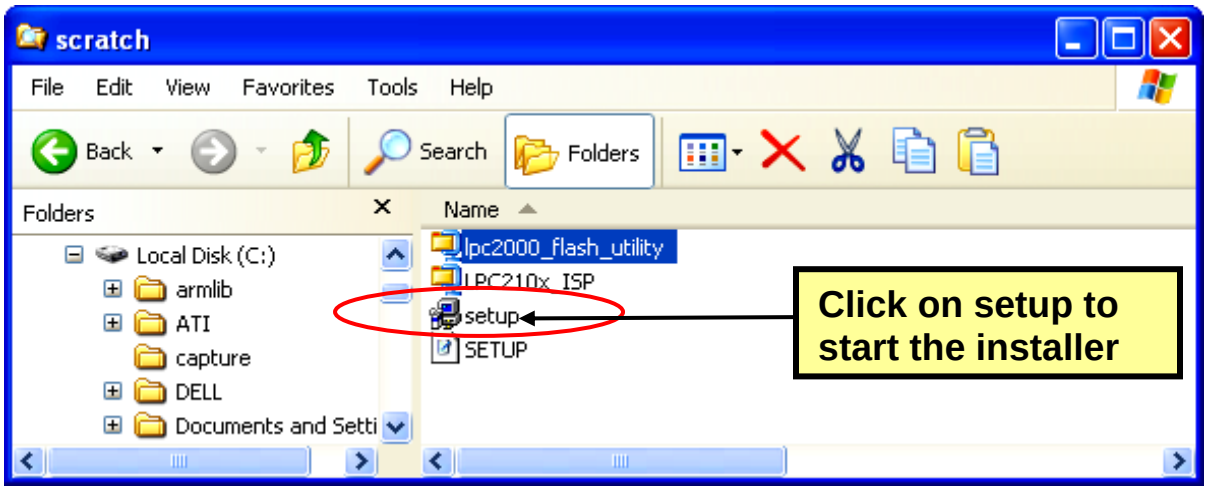




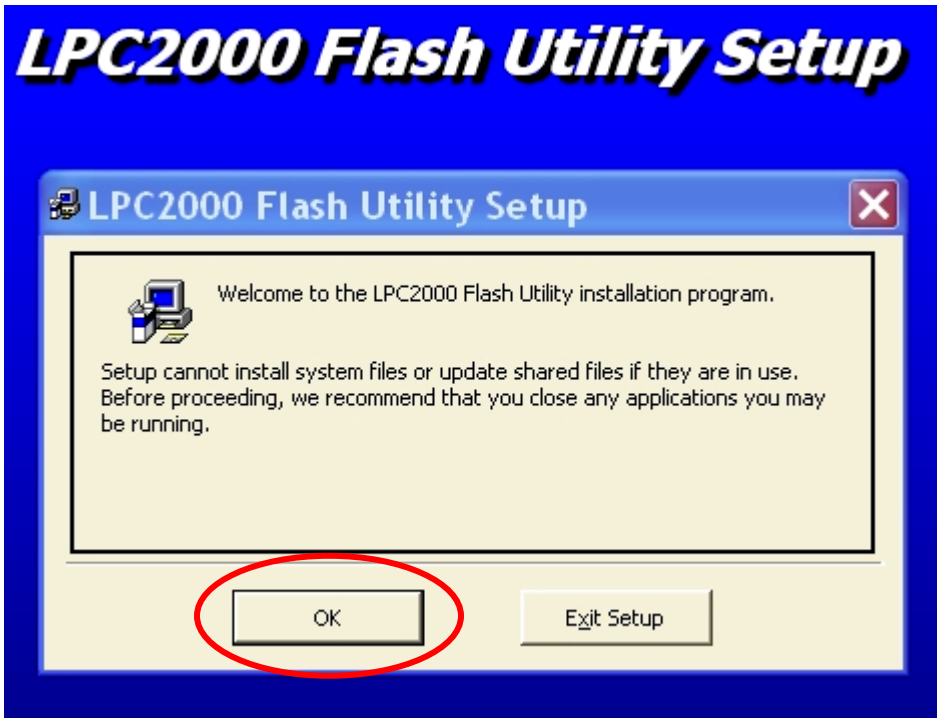
We'll use WinZip to unzip this into the **c:/scratch** directory.



Now you can see that the download directory has a setup utility and another zip file containing the LPC2000 Hex Utility. Click on the **setup.exe** application to start the installer.



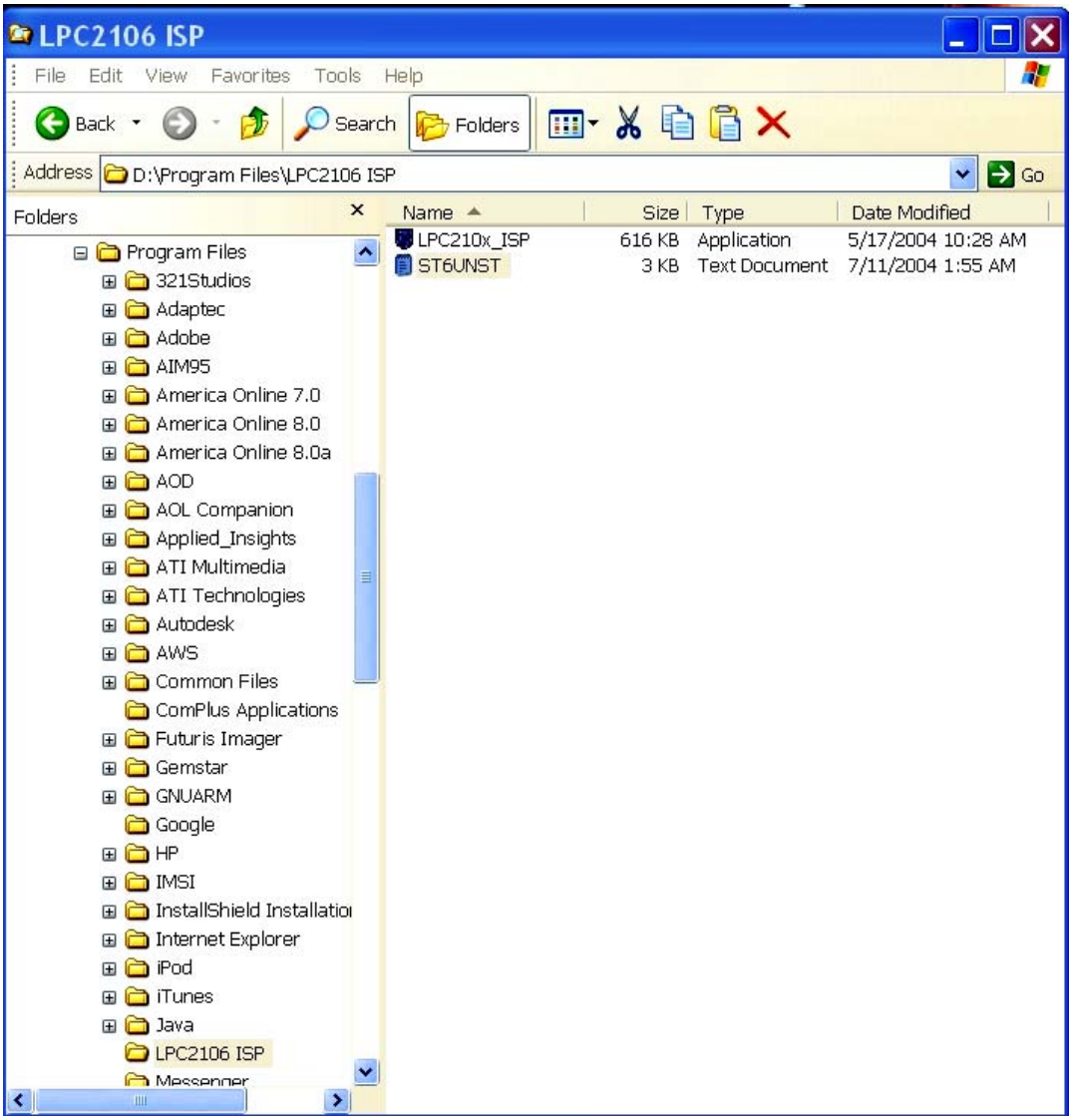
The LPC2000 Flash Utility setup now starts. Click on **OK** to proceed.



Take the default on this screen below and let it install the LPC2000 Flash Utility into the Program Files directory.

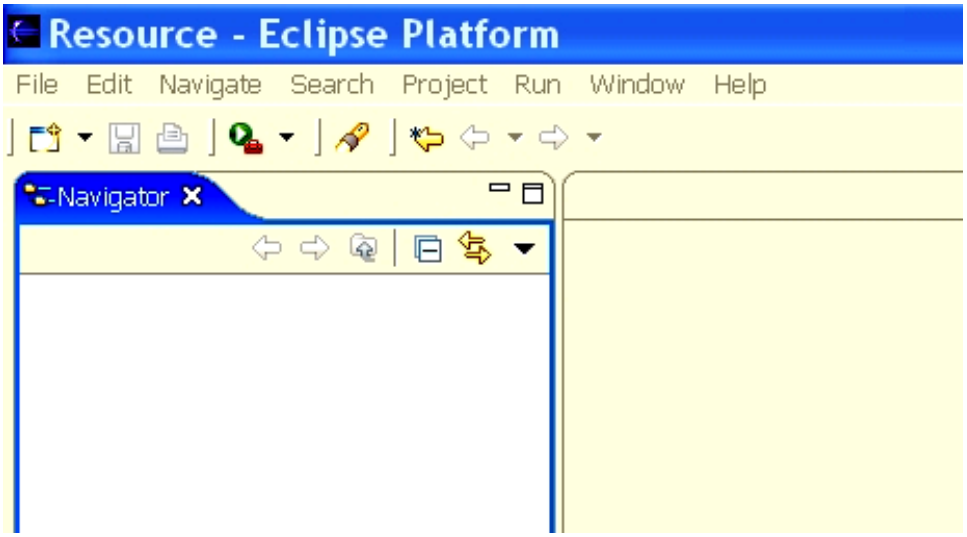
In a very few seconds, the installer will complete and you should see this screen.

Here we see the utility residing in the Program Files directory, just as promised.

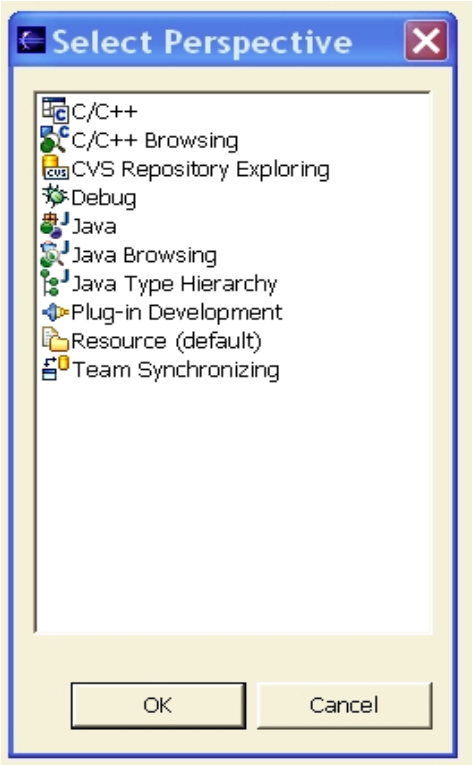


Now that the Philips LPC2000 Flash Utility is properly installed on our computer, we'd like to install it into Eclipse so that it can be invoked from the RUN pull-down menu under the "**external tools**" option. Start Eclipse by clicking on the desktop icon.

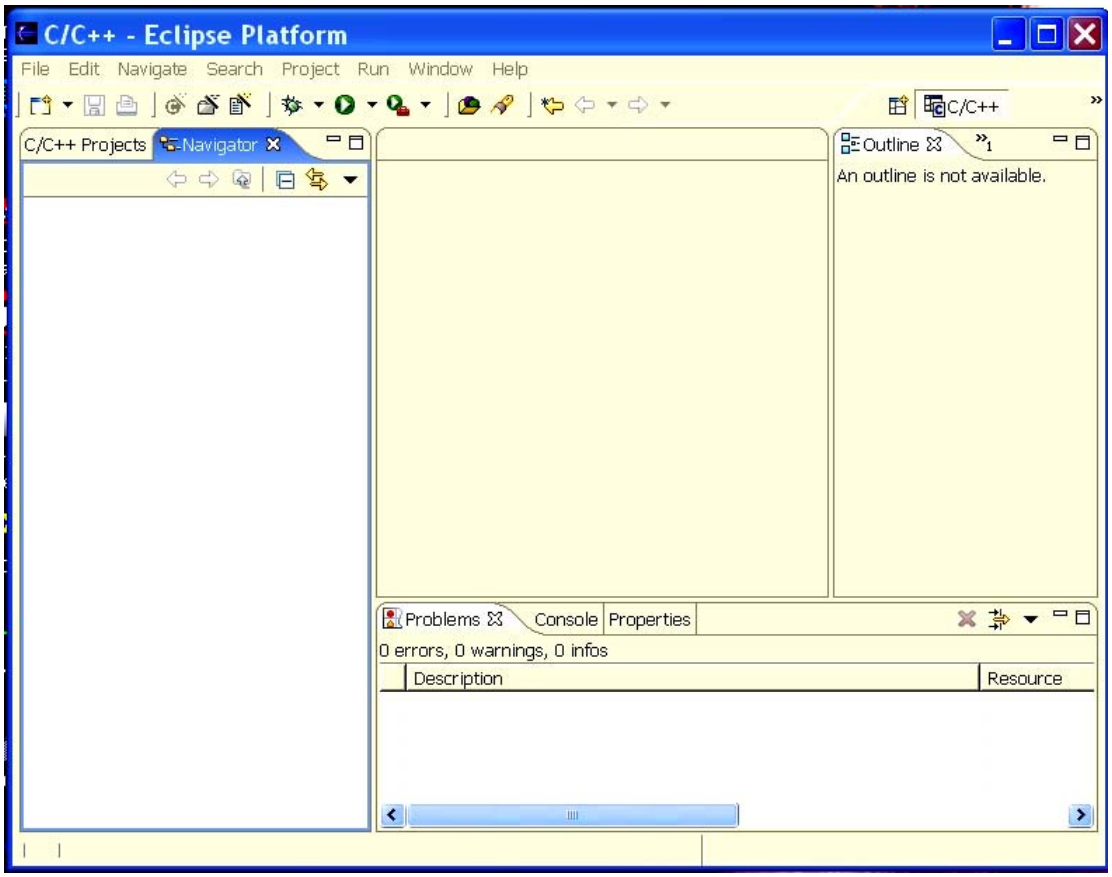
The layout of the Eclipse screen is called a "perspective." The default perspective is the "resource" perspective, as shown below.



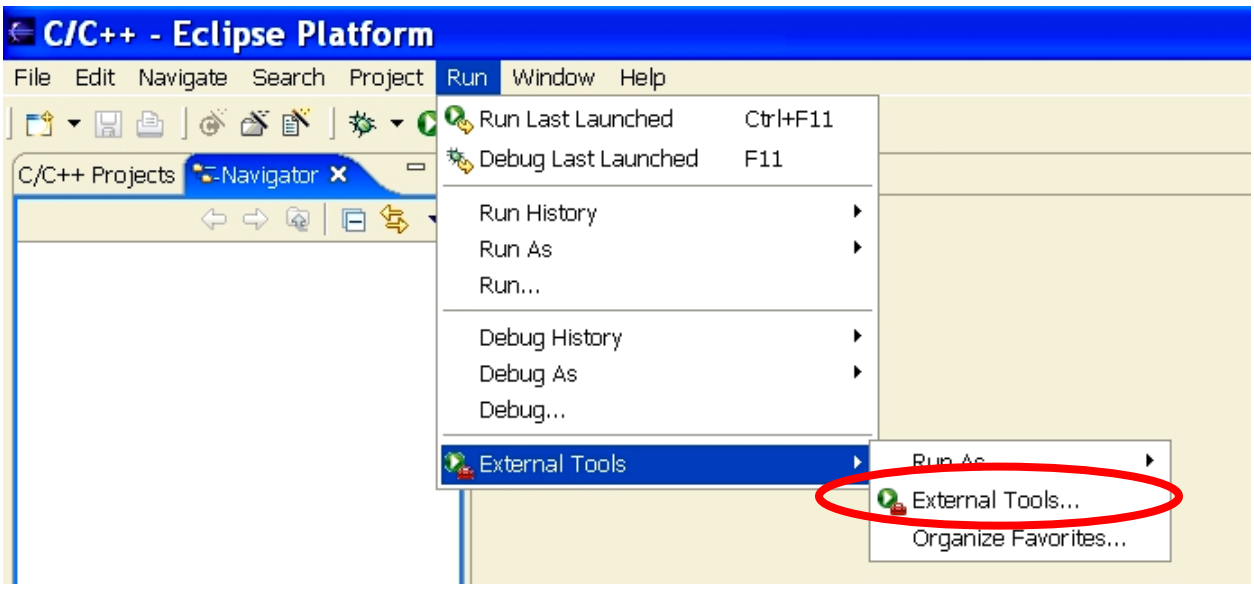
We need to change it into the C/C++ perspective. In the **Window** pull-down menu, select **Window – Open Perspective – Other – C/C++** and then click **OK**.



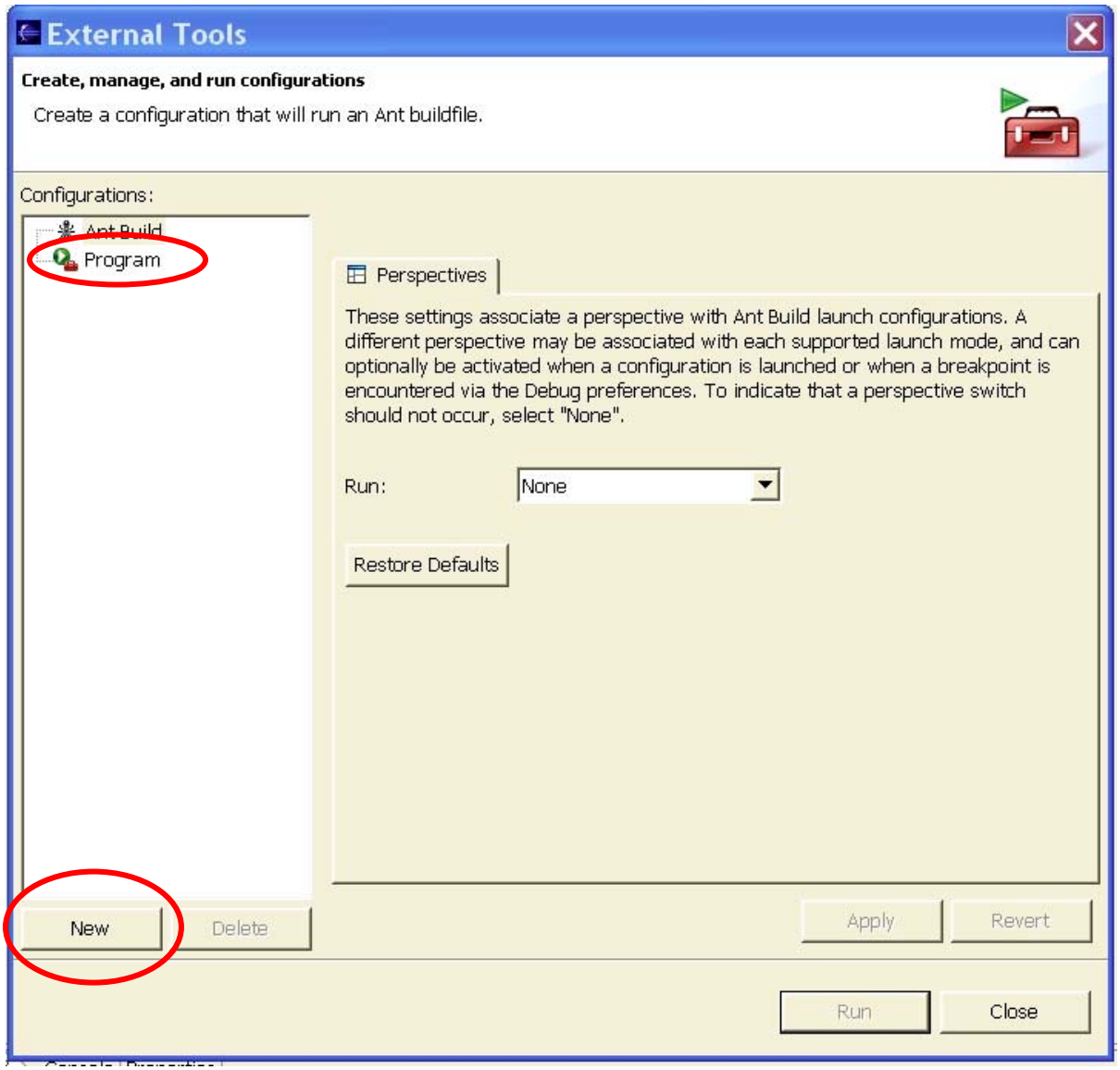
Eclipse will now switch to the **C/C++** perspective shown below and will remember it when you exit.



Now we want to add the Philips LPC2000 Flash Utility to the "**External Tools**" part of the **Run** pull-down menu. Select **RUN – External Tools – External Tools**.



We want to add a new program to the External Tools list, so click on **Program** and then **New**.

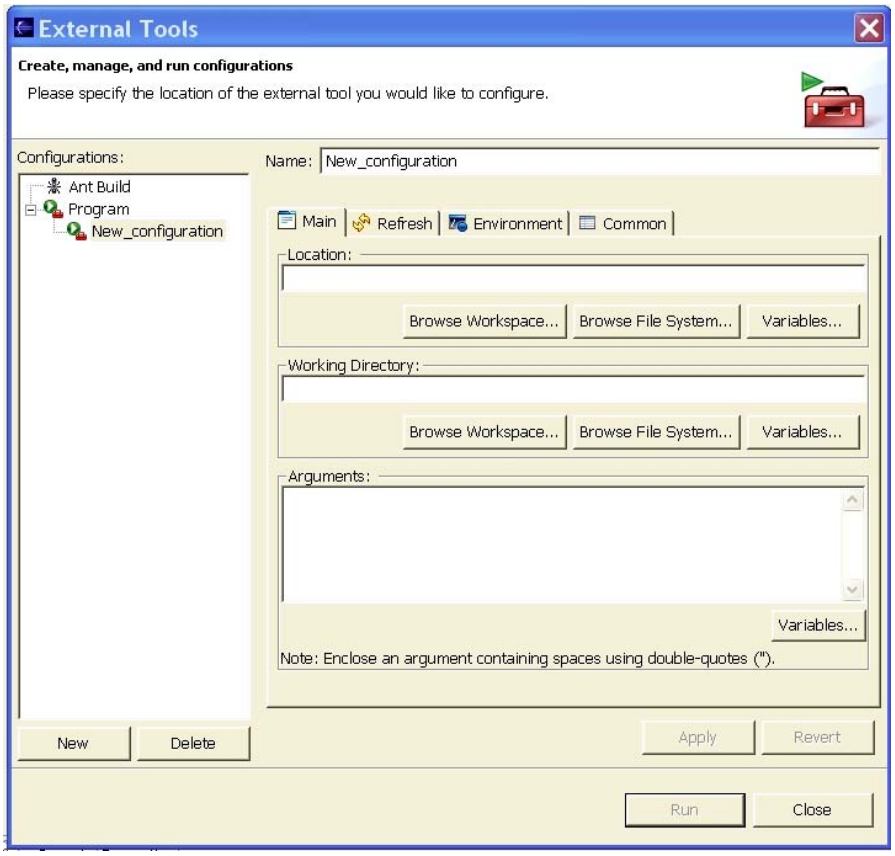


Note below that there's a new program under the "program" tree with the name **New_configuration** and there's no specifications as to what it is.

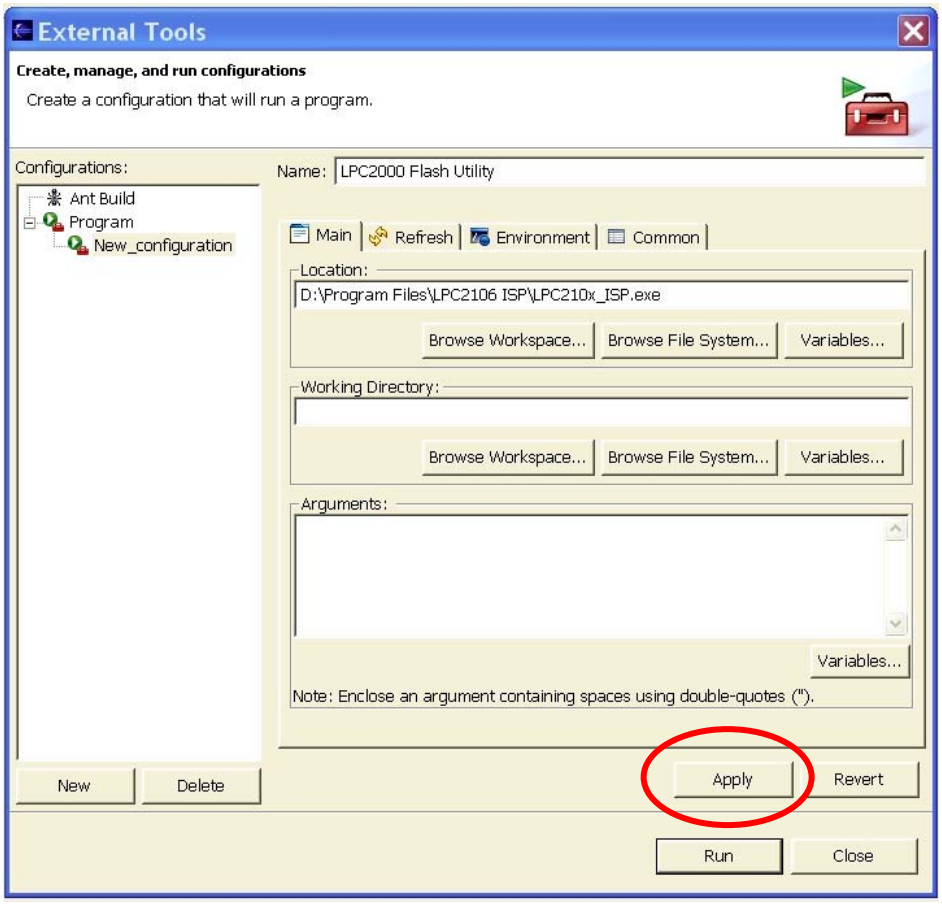
In the **Name** text box, replace **New-configuration** with **LPC2000 Flash Utility**.

In the **Location** text box, use the "**Browse File System**" tool to find the Philips LPC2000 Flash Utility in the Program Files directory. Its name is **LPC210x_IPC.exe**.

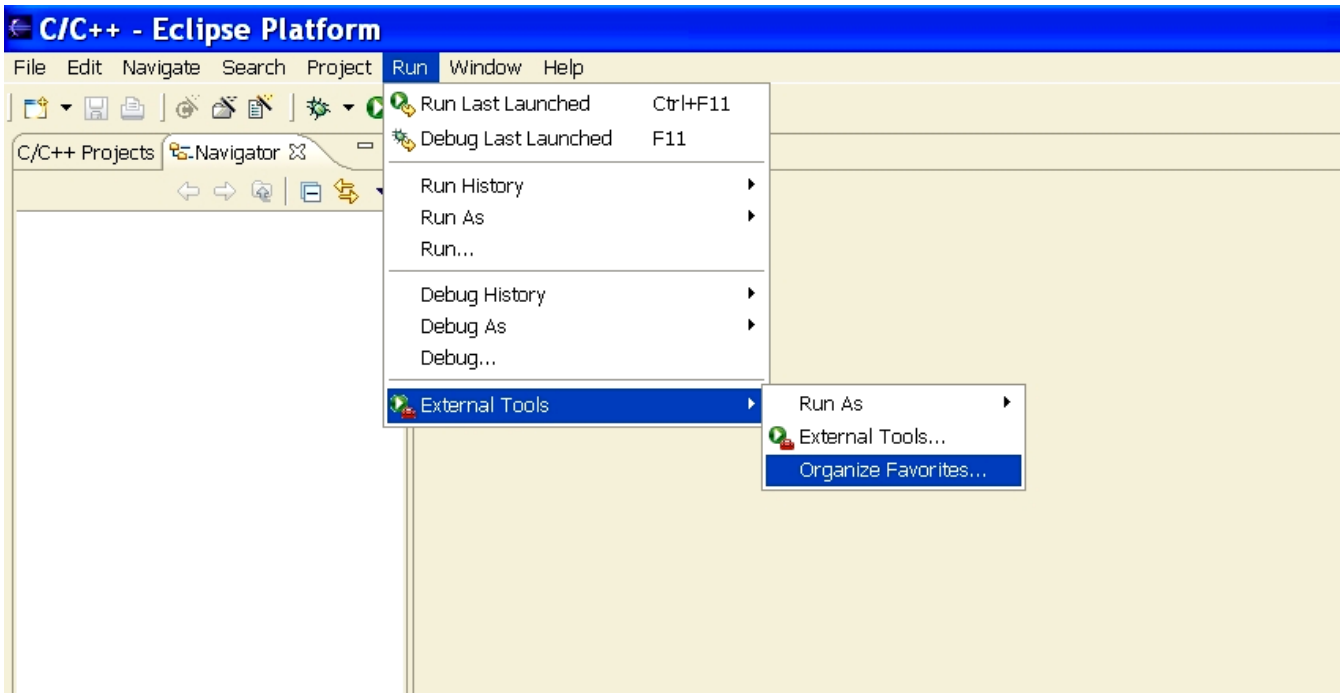
Here’s the External Tools window before editing.



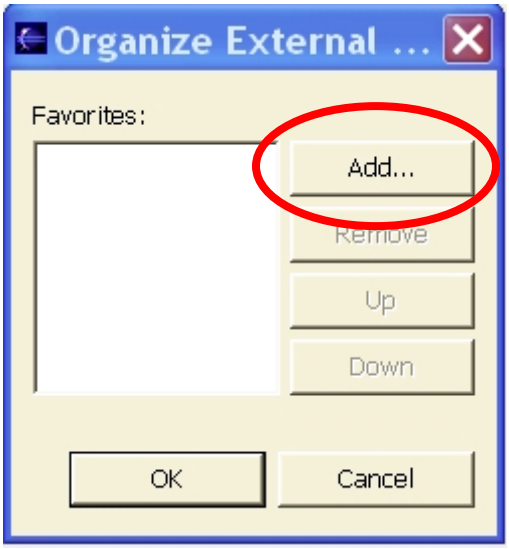
Here’s the External Tools window after our modifications. Click on **Apply** to accept.



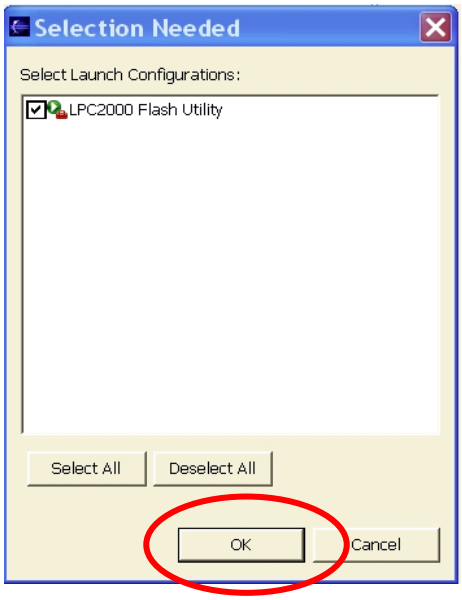
Close everything out and return to the **Run** pull-down menu. Select **Run – External Tools – Organize Favorites**.



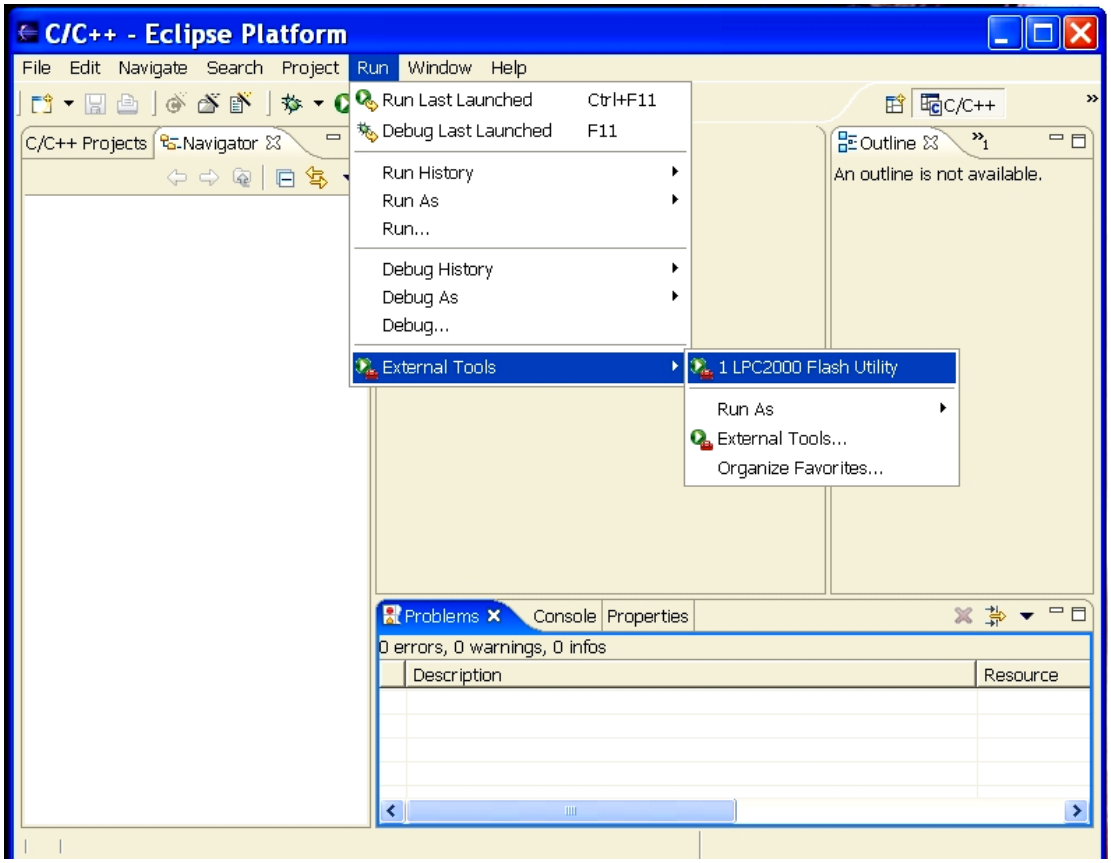
We're now going to put the Philips PLC2000 Flash Utility into the "favorites" list. Click on "**Add**" in the window below.



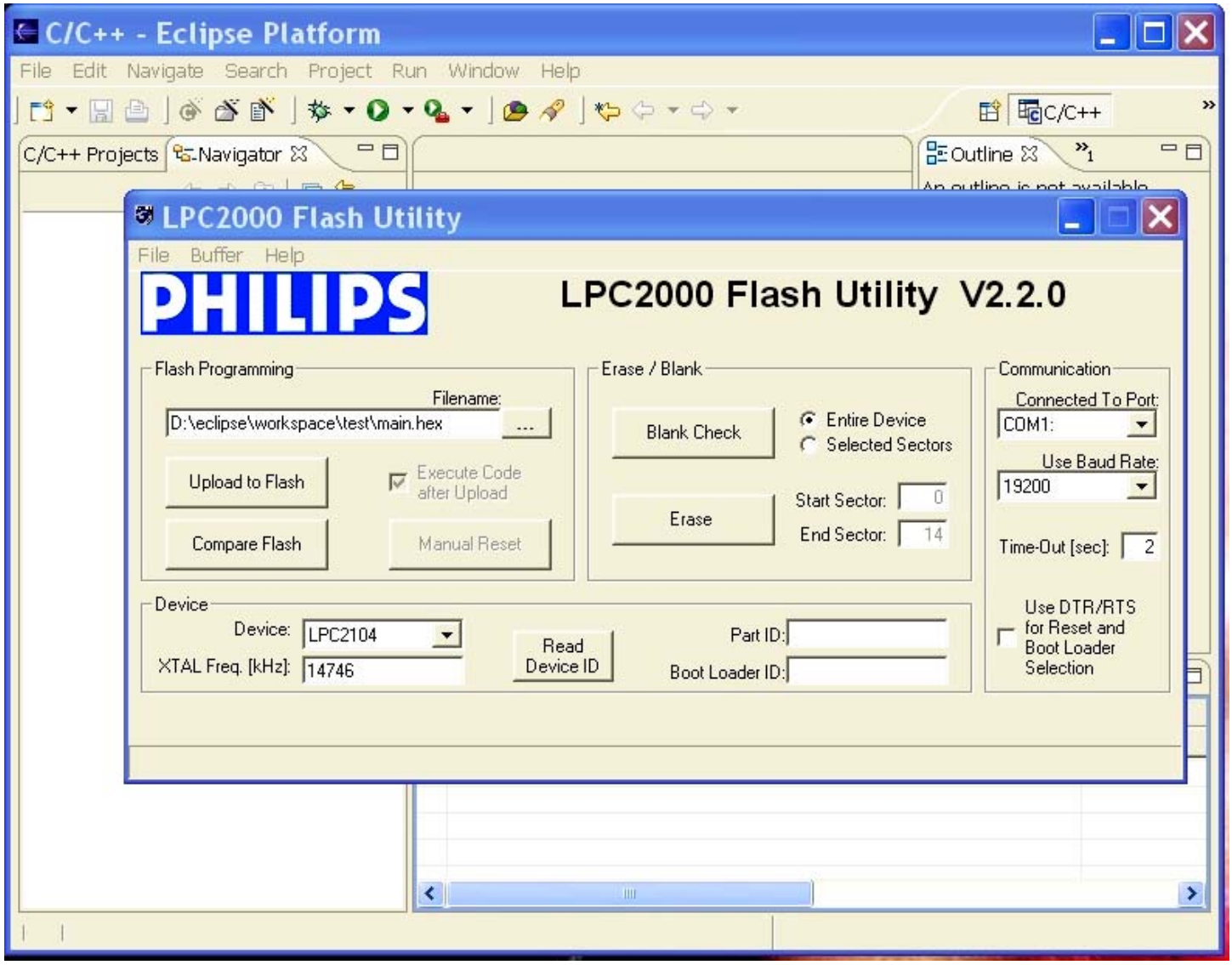
Click the selection box for LPC2000 Flash Utility. This will add it to the favorites list.



Now when we click on the **Run** pull-down menu and select **“**External Tools,” we see the **LPC2000 Flash Utility** at the top of the list.



Click on LPC2000 Flash Utility to verify that it runs.



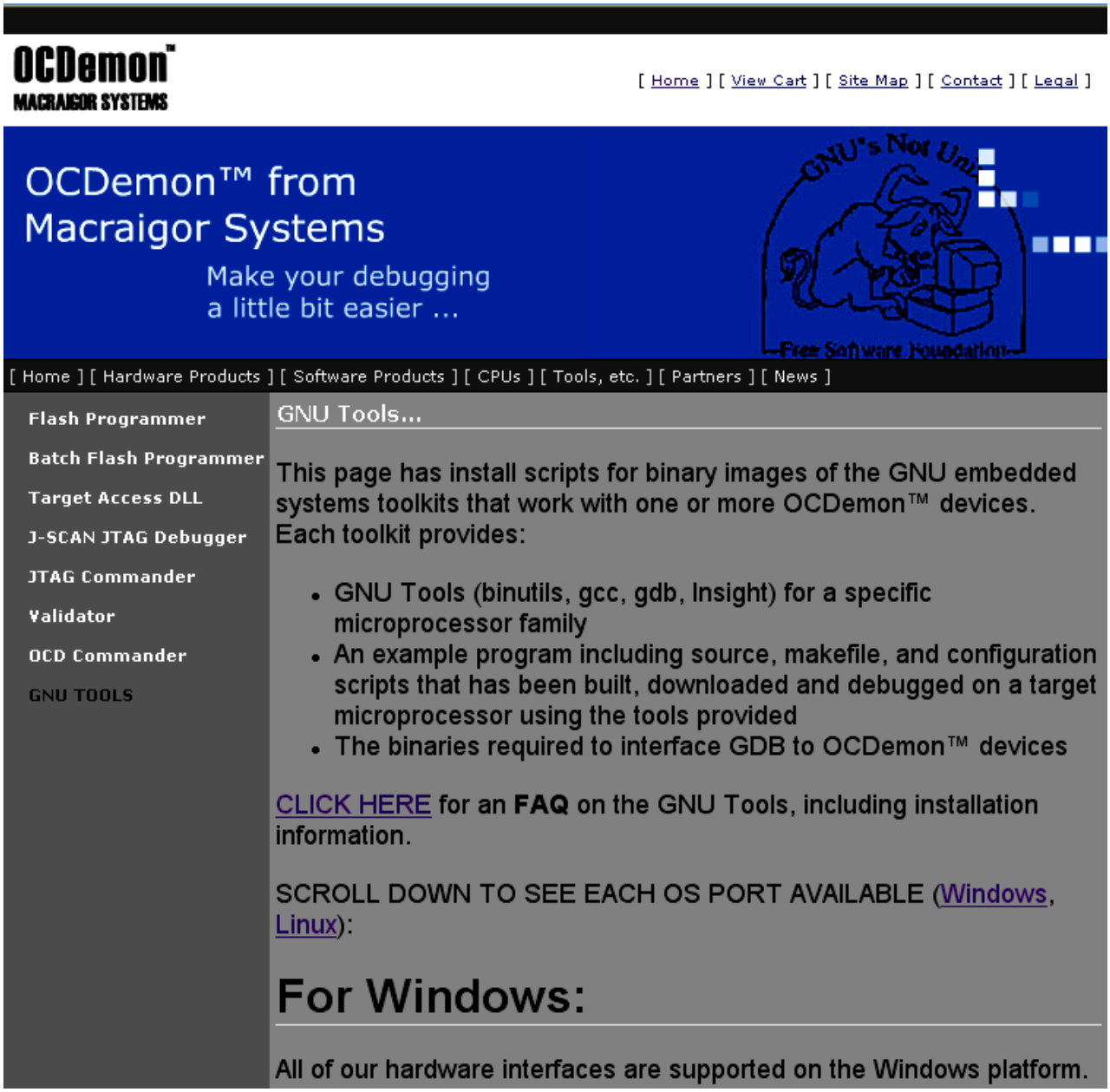
Now cancel the LPC2000 Flash Utility and quit Eclipse.

9 Installing the Macraigor OCDremote Utility

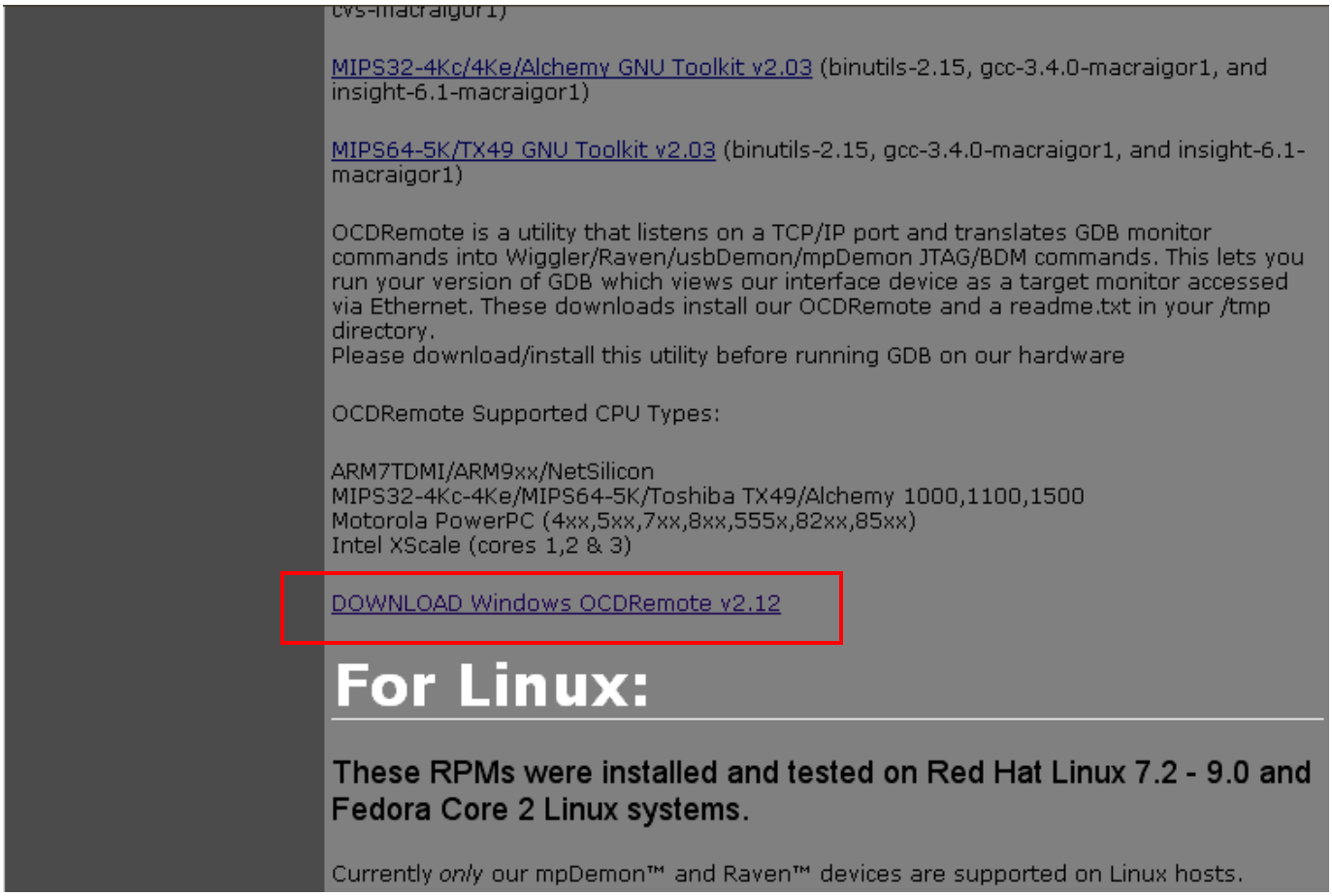
OCDRemote is a utility that listens on a TCP/IP port and translates GDB monitor commands into **Wiggler** JTAG commands. This permits Eclipse/GDB to communicate to the Olimex LPC-P2100 board as a target monitor accessed via Ethernet. Macraigor has always made this utility available on the internet as "freeware." The **OCDRemote** utility can be downloaded at:

http://www.macraigor.com/full_gnu.htm

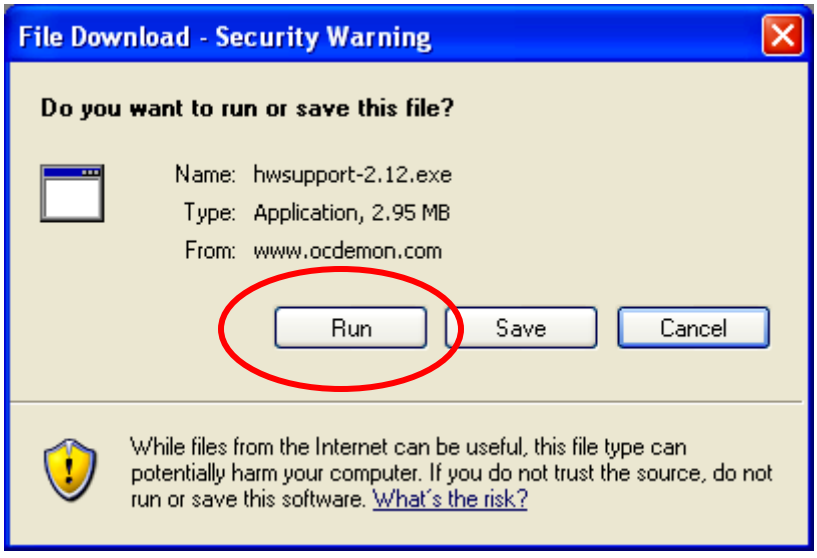
You should see the following screen open up.



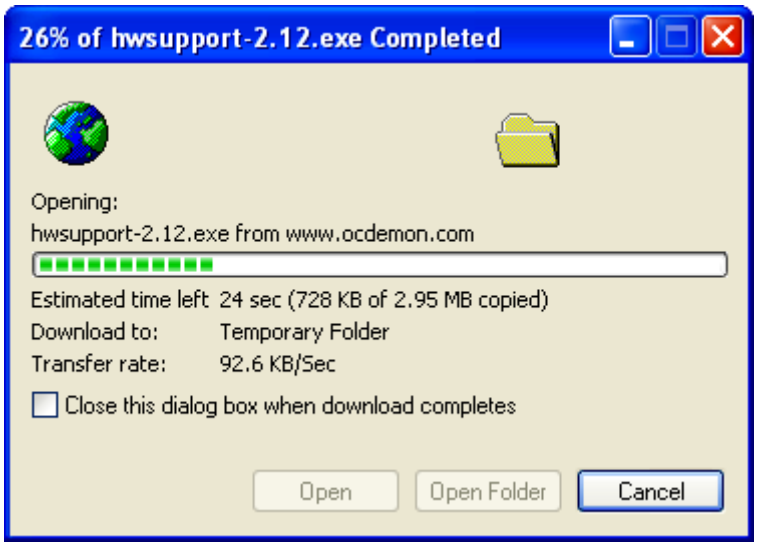
If you scroll the above screen down a bit, you should see the download for **OCDRemote**. Click on the link "**DOWNLOAD Windows OCDRemote v2.12**".



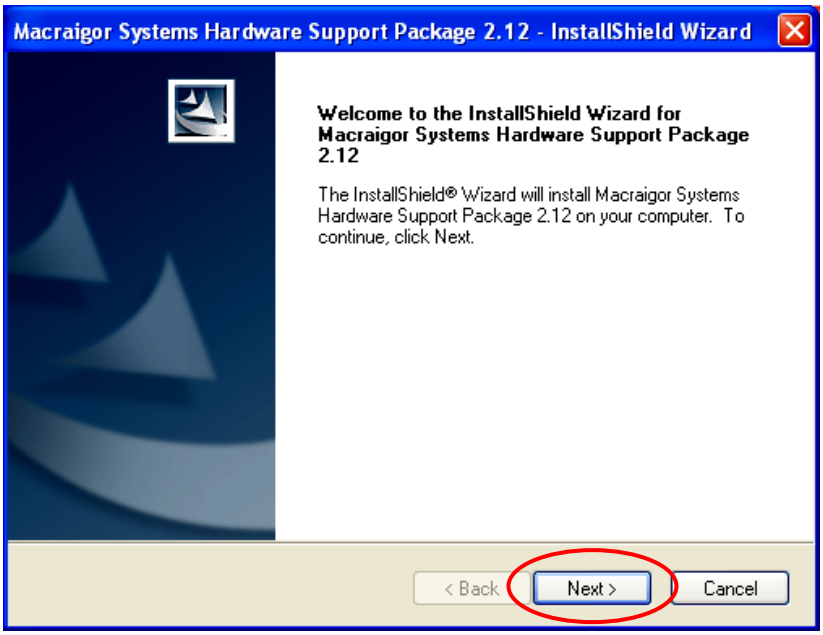
Click on "**Run**" so it will download and immediately install **OCDRemote**.



The download phase is quick since the **OCDRemote** is only a couple of megabytes.



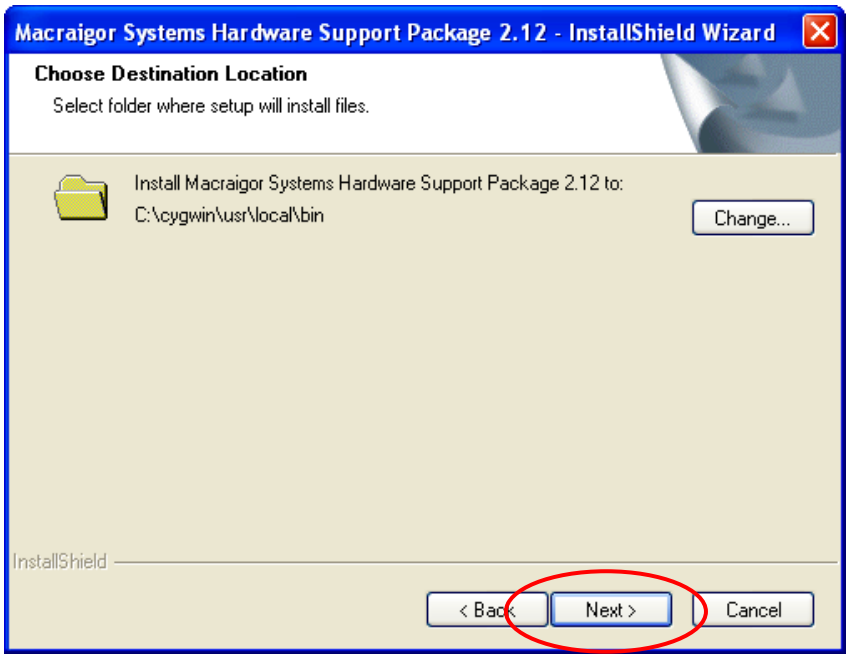
The Macraigor installer should start up; just click "**Next**" to continue.



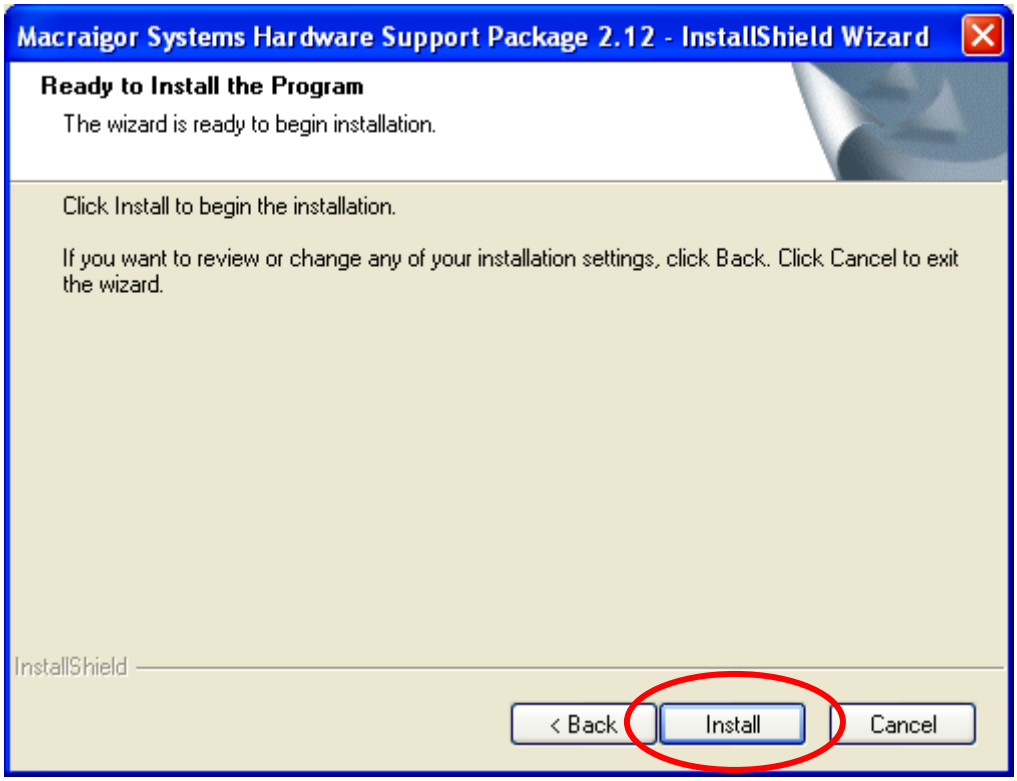
The next screen lets you choose where **OCDRemote** is installed. **OCDRemote** normally installs in **c:/cygwin/usr/local/bin**.

We'll have to make sure that this directory is on a Windows Path.

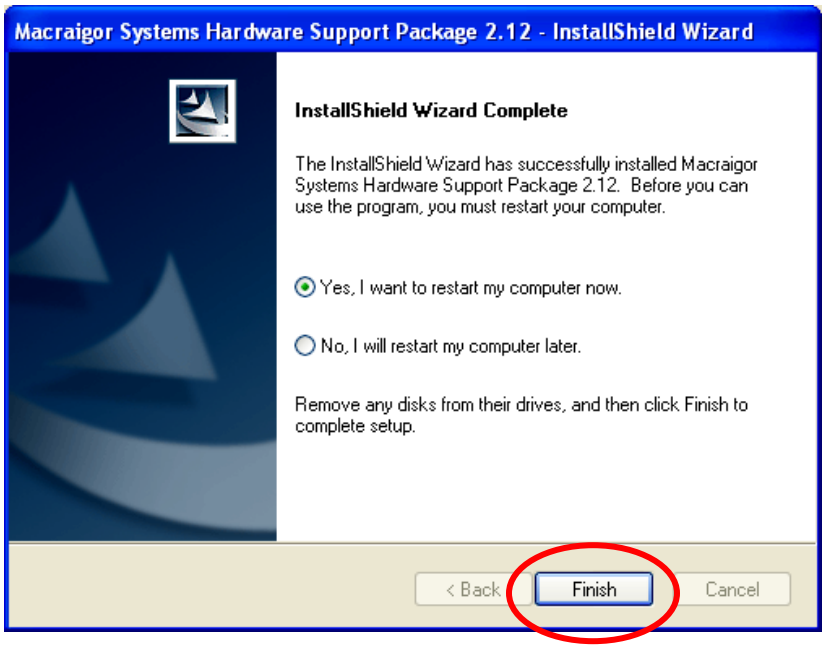
Click on “**Next**” to accept **c:/cygwin/usr/local/bin** as the **OCDRemote** installation directory.



Clicking on “**Install**” will complete the **OCDRemote** installation.

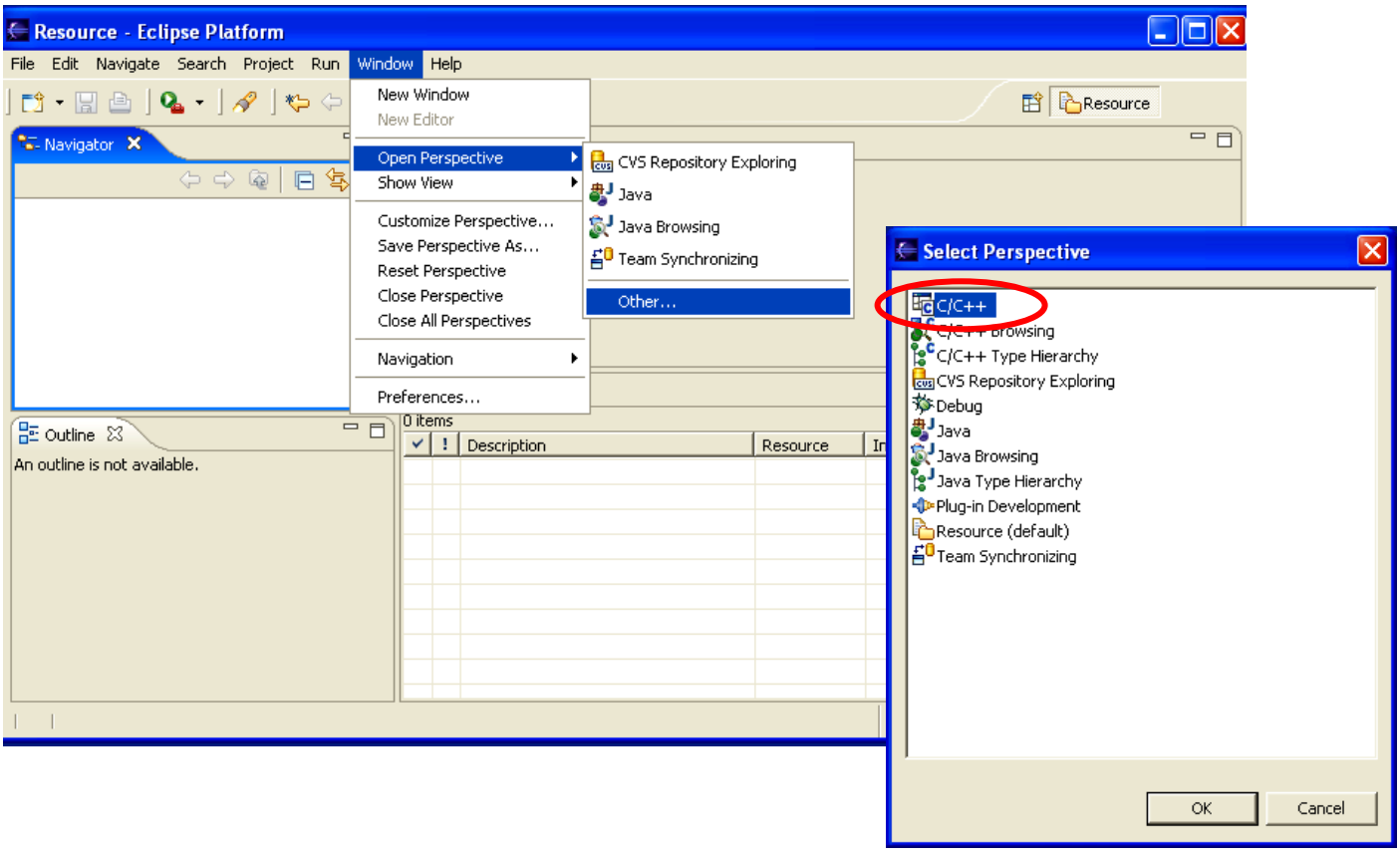


The Wizard completion screen lets you restart your computer to put **OCDRemote** into the Windows registry.

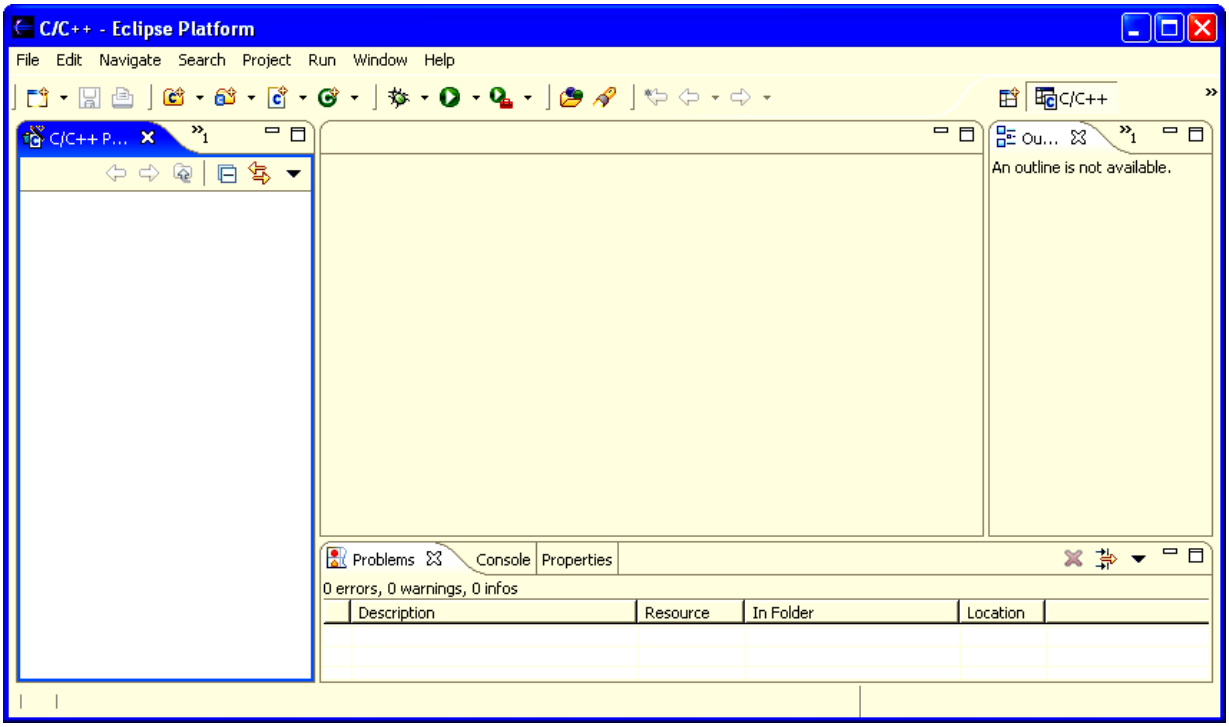


Just like the Philips ISP Flash Utility, we should install the Macraigor **OCDremote** utility as an “external tool” that can be accessed easily from the Eclipse CDT **RUN** pull-down menu.

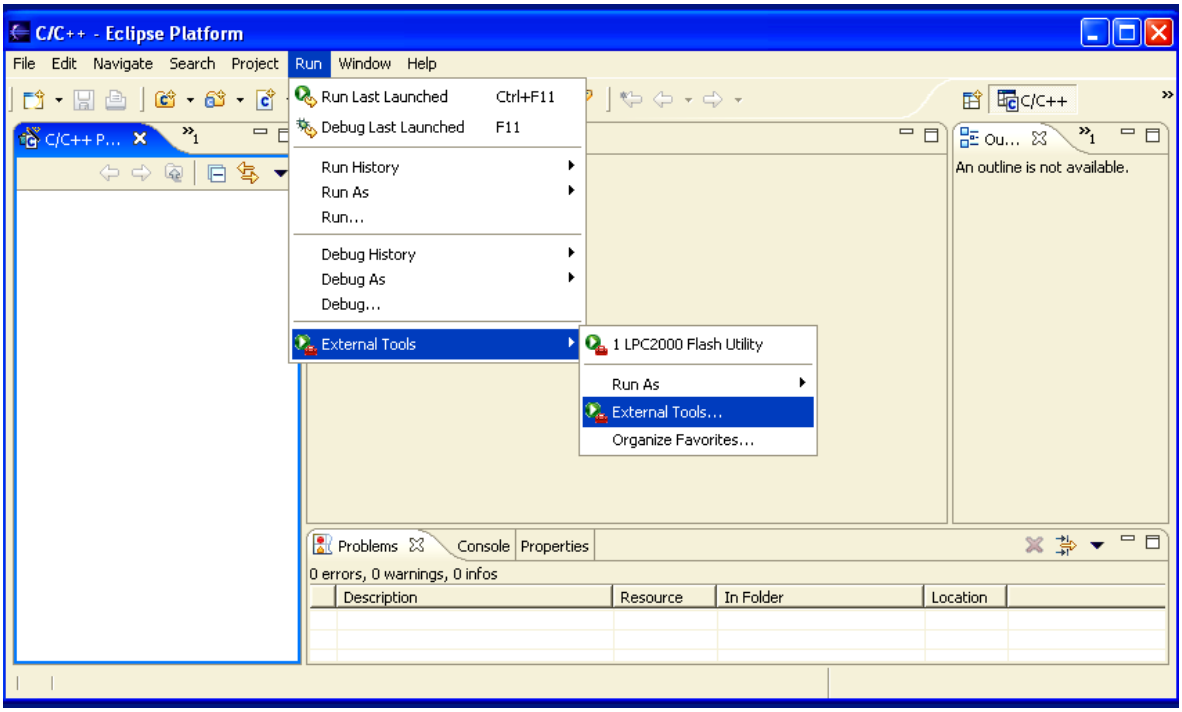
Start up Eclipse and, if necessary, switch to the C/C++ perspective by clicking “**Window – Open Perspective – Other – C/C++**.”



Switching perspectives brings up the C/C++ window (perspective) and this will be remembered when you re-enter Eclipse.



In a procedure similar to installing the Philips Flash Utility as an “External Tool”, click on “**Run – External Tools – External Tools …**” This will bring up the External Tools dialog.

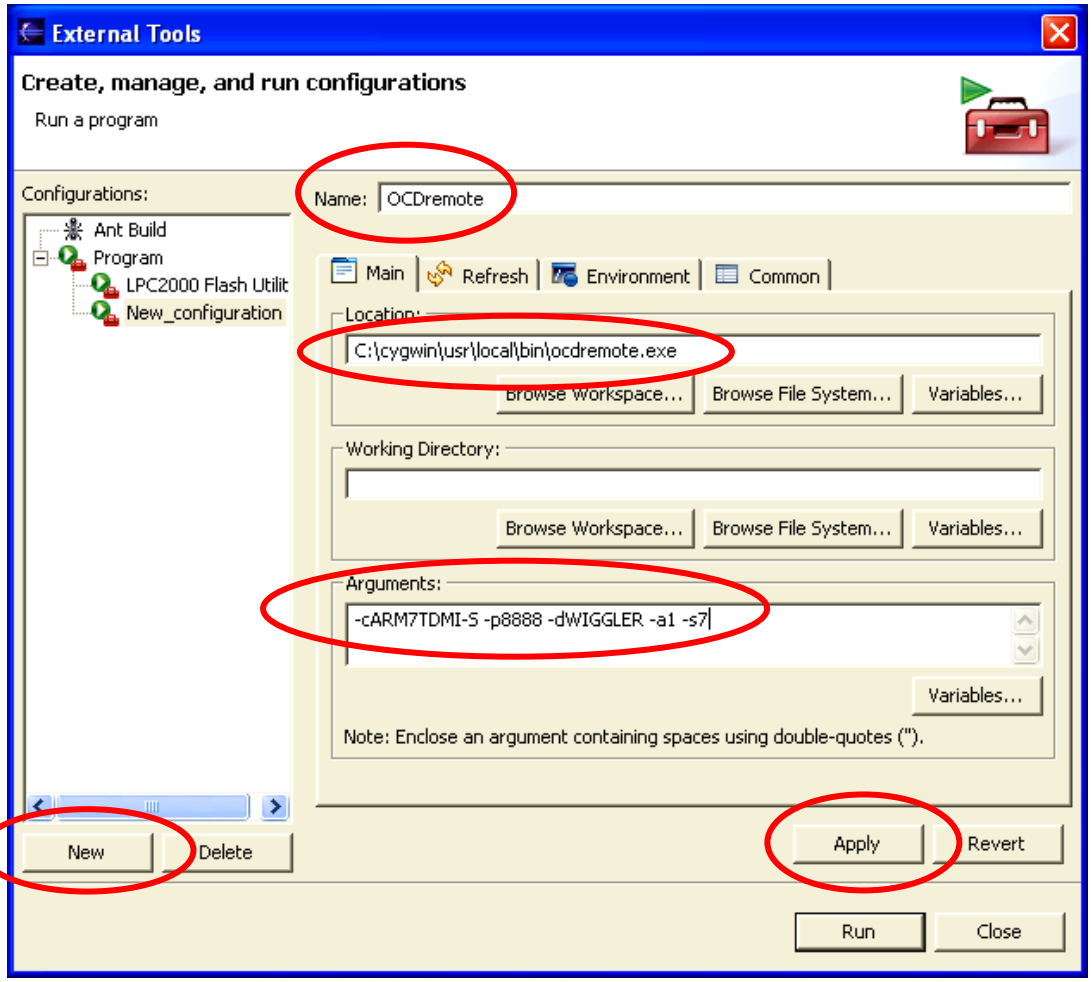


Click on “**New**” and replace the name with **OCDremote**. Use the “**browse file system**” to find it. It should be in the directory **c:/cygwin/usr/local/bin**.

The arguments needed to properly start the **OCDremote** are as follows:

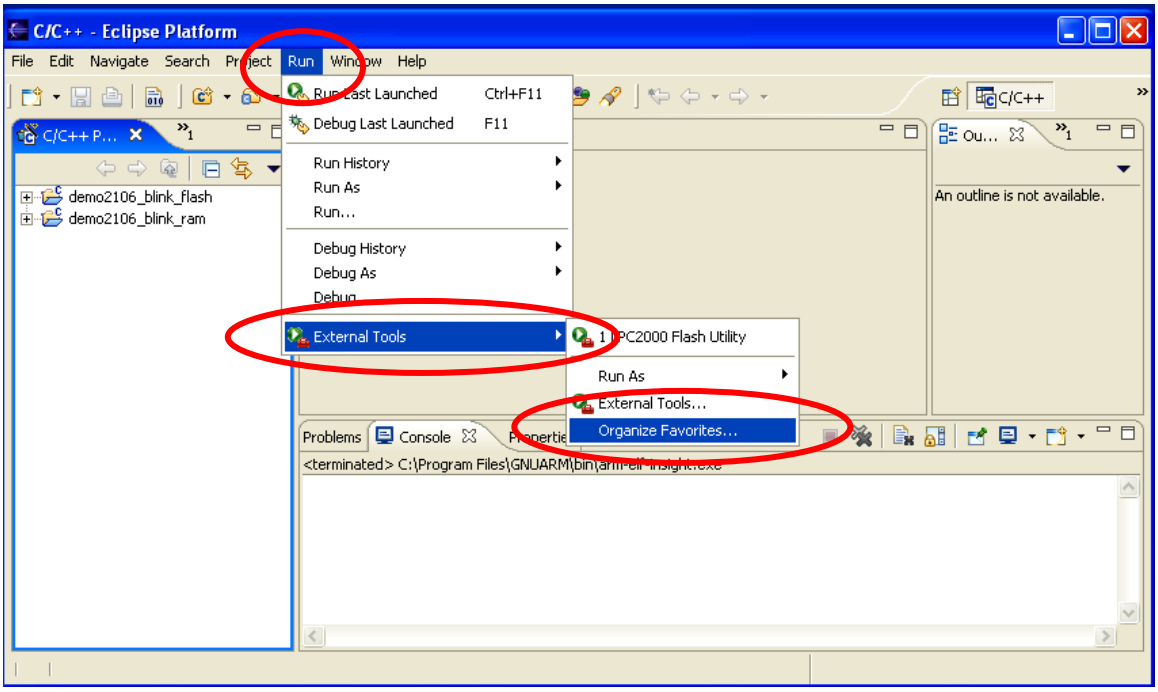
-cARM7TDMI-S	specifies the CPU being accessed
-p8888	specifies the pseudo TCP-IP port being used
-dWIGGLER	specifies the JTAG hardware being used
-a1	specifies LPT1 for the Wiggler
-s7	specifies next-to-slowest speed

It’s a good idea to not tamper with these values. Click on “**Apply**” to finish the setup.

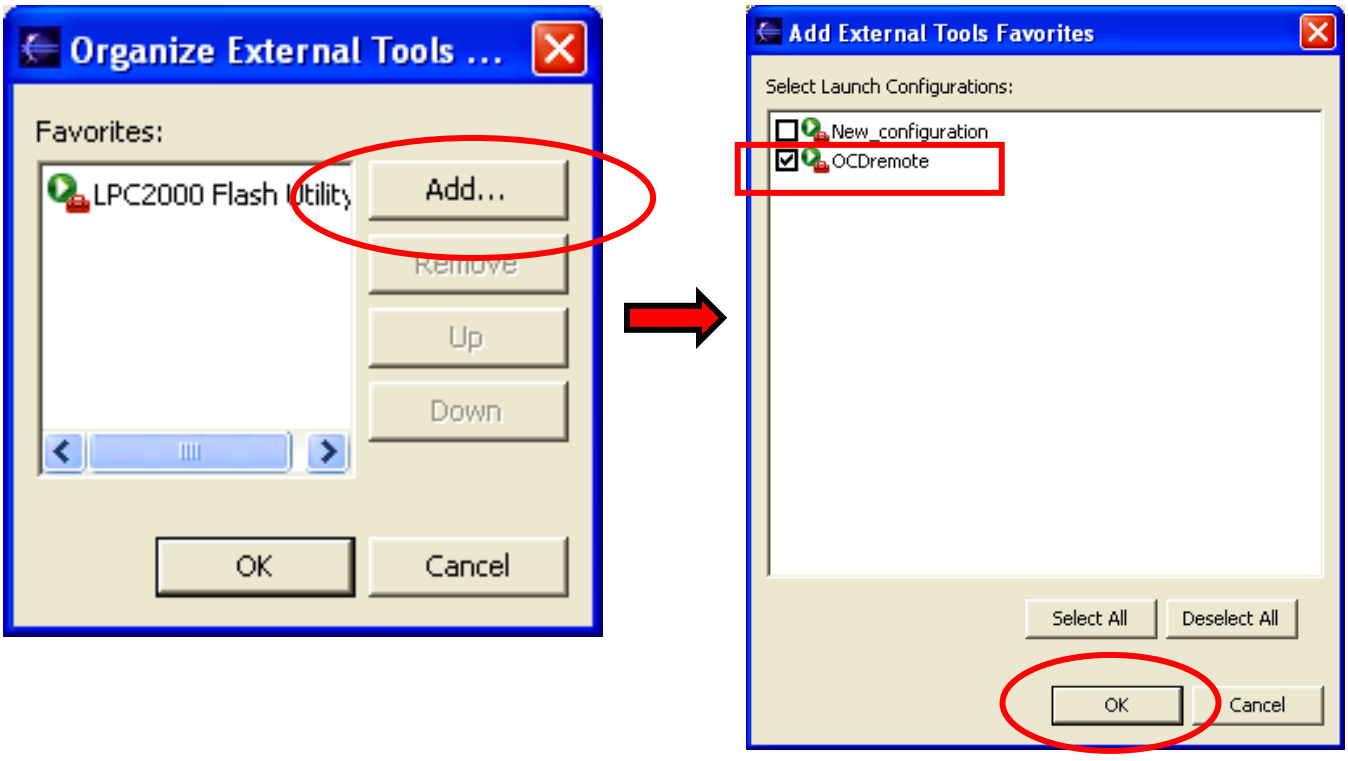


Just like the Philips LPC2000 Flash Utility, we’d like to include the **OCDremote** application in our list of “**favorite**” External Tools. This allows us to quickly start the **OCDremote** JTAG server from within Eclipse.

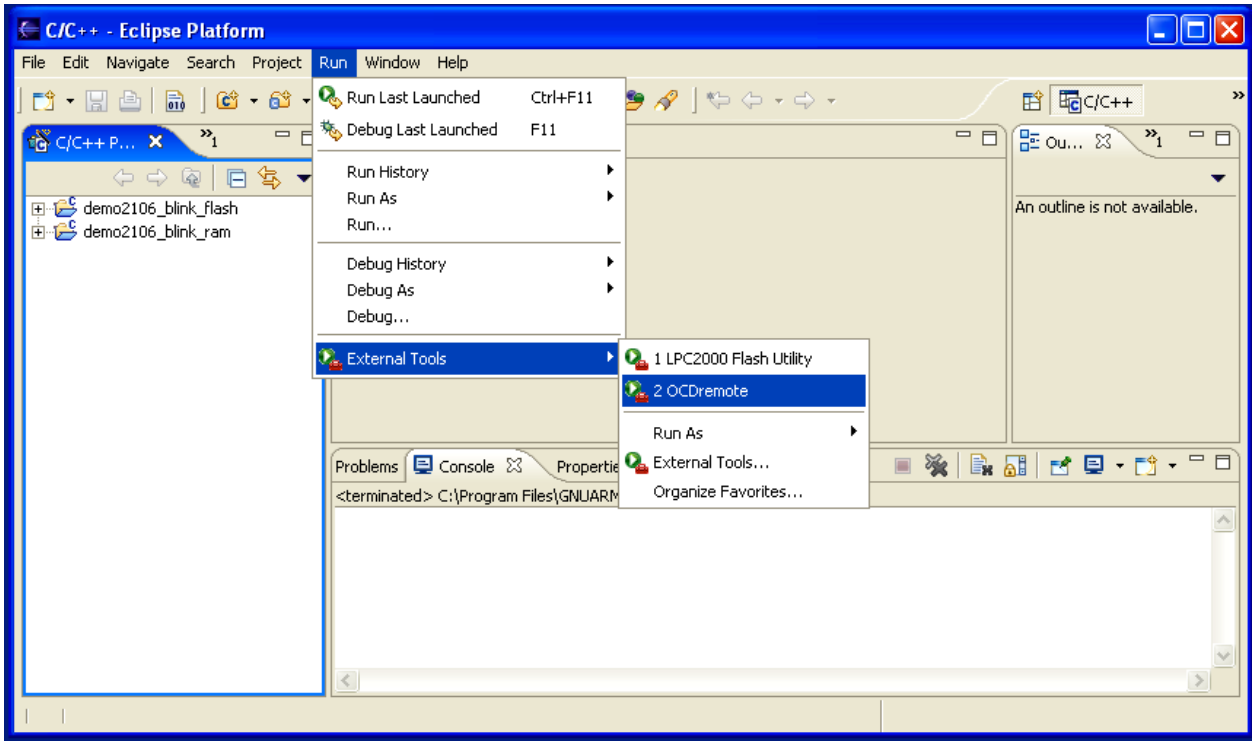
Click on “**Run – External Tools – Organize Favorites**”



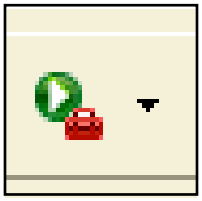
Now click on “**Add…**” in the Organize External Tools … window and follow that by checking “**OCDremote**” in the Add External Tools Configurations: window. Click on “**OK**” to add the OCDremote to the list of favorites.

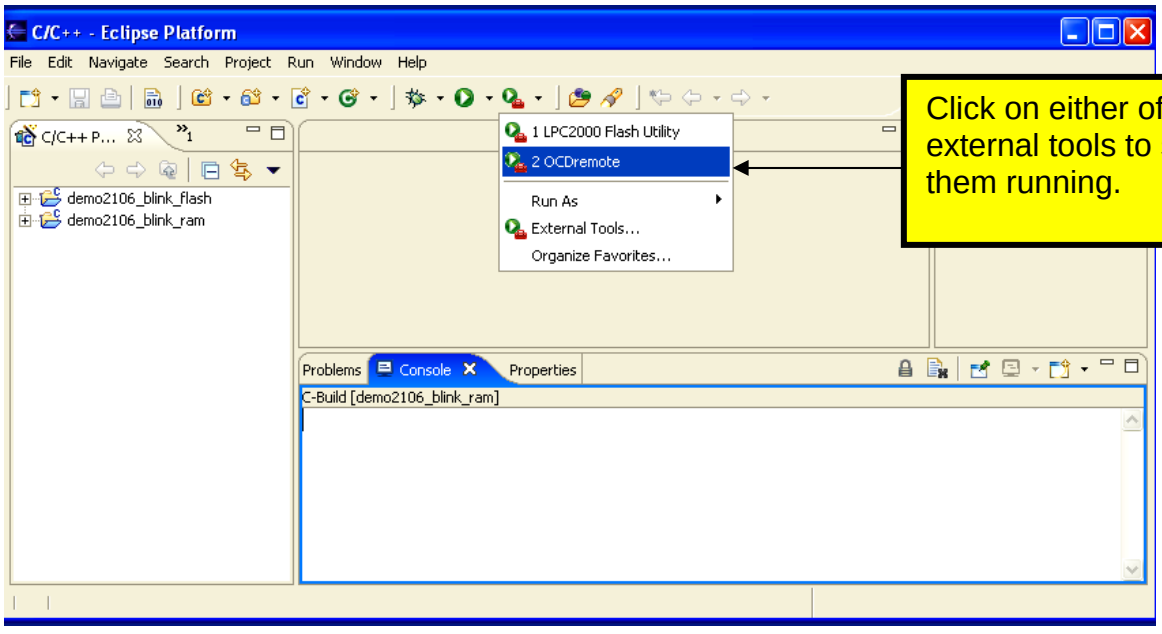


Now verify that the **OCDremote** is in the list of External Tools favorites. Click on "**Run – External Tools**" and see that it's now included in the list of favorites.



Now is a good time to point out that there's a handy shortcut button in Eclipse to run the External Tools. Click on the **External Tools** button's down arrow to expand the list of available tools.





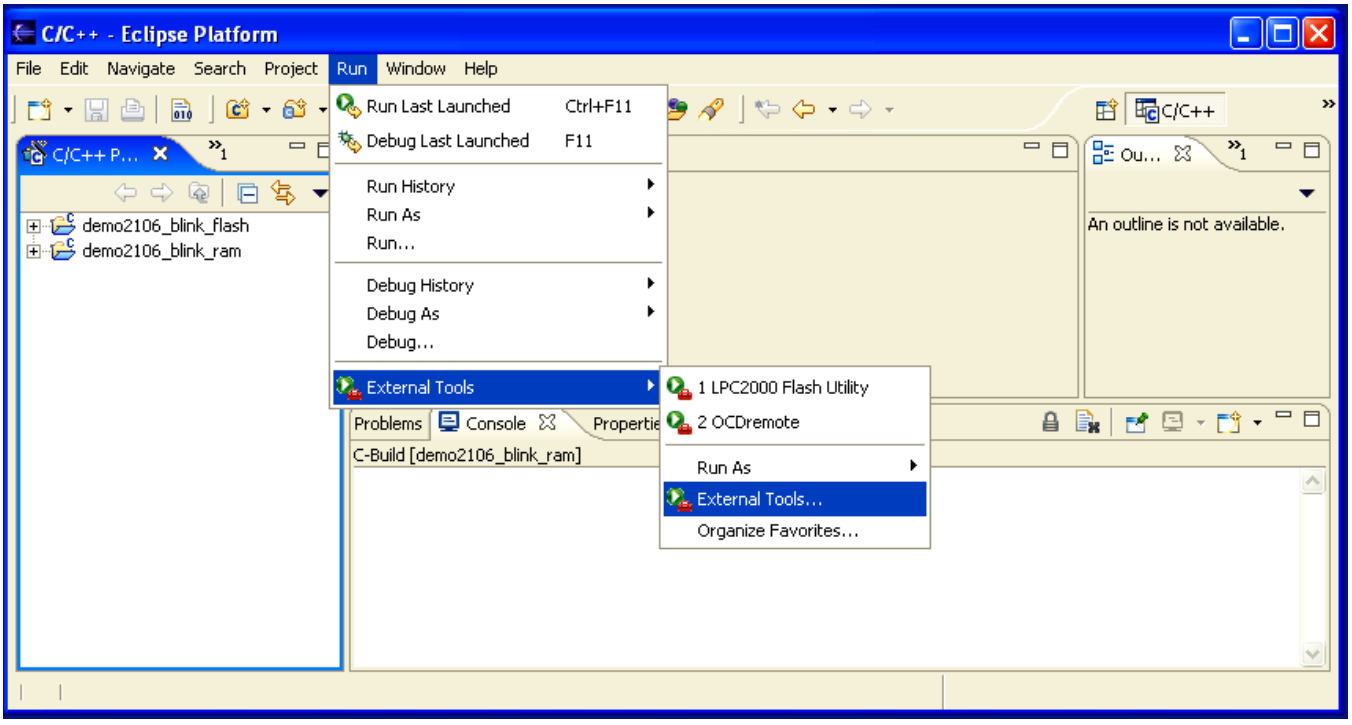
10 Installing the INSIGHT Graphical Debugger

Eclipse CDT has its own debugger, employing the GDB serial protocol. The truth is, I've been unable to get it to work reliably with the **Wiggler** using OCDremote and with the Segger J-Link USB JTAG interface using the J-Link server written by Nick Randell. As far as JTAG debugging goes, the Eclipse debugger is just not ready for prime time (they are working on it).

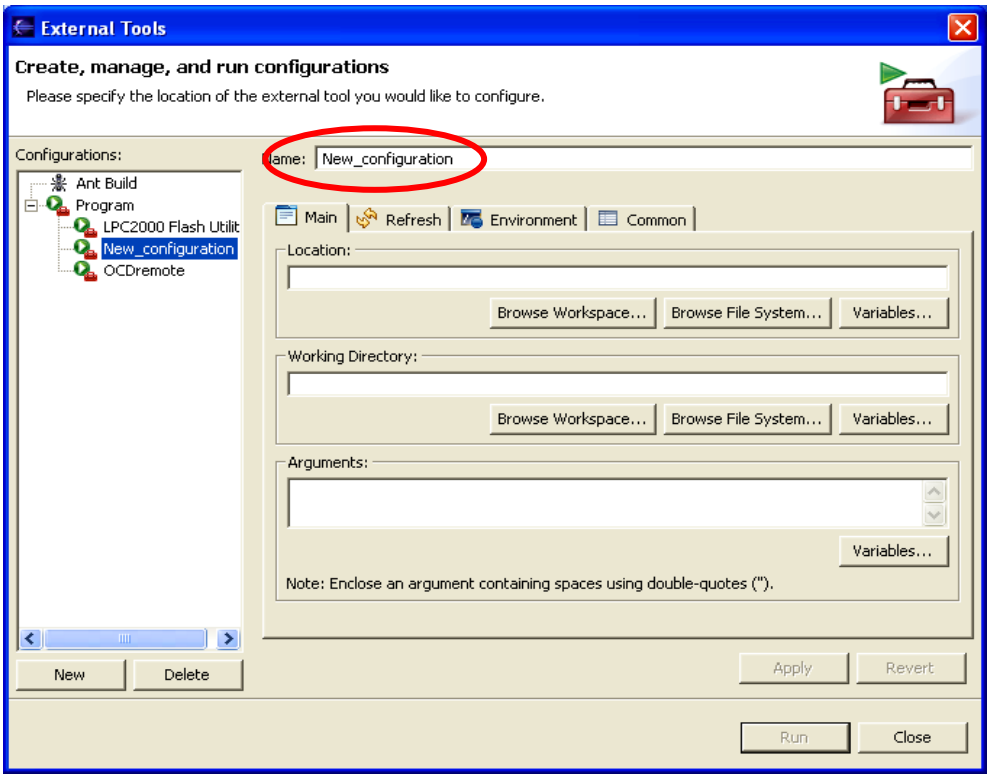
Insight is a pretty good open-source debugger that can be made to work with the Wiggler JTAG interface. It is already present in the GNUARM directory on C:/Program Files/ installed earlier.

Let's install the **Insight Debugger** as an Eclipse External Tool.

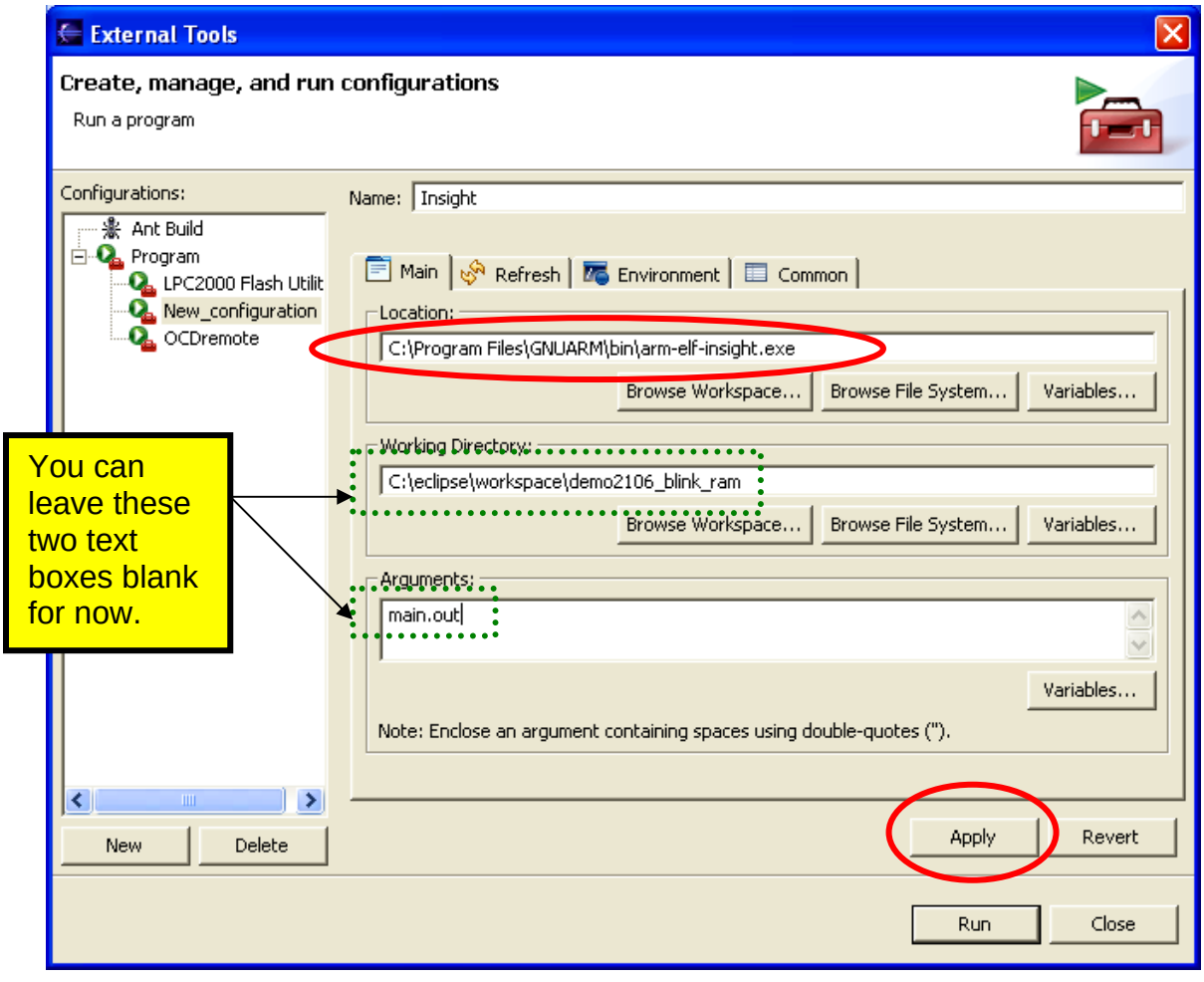
Click on "**Run – External Tools – External Tools …**"



In the **External Tools** dialog window below, click on "**New-configuration**" on the left.



Now fill this dialog in as shown below. The Insight Debugger is the executable file "**arm-elf-insight.exe**" and it's in the **c:/Program Files/GNUARM/bin** folder.

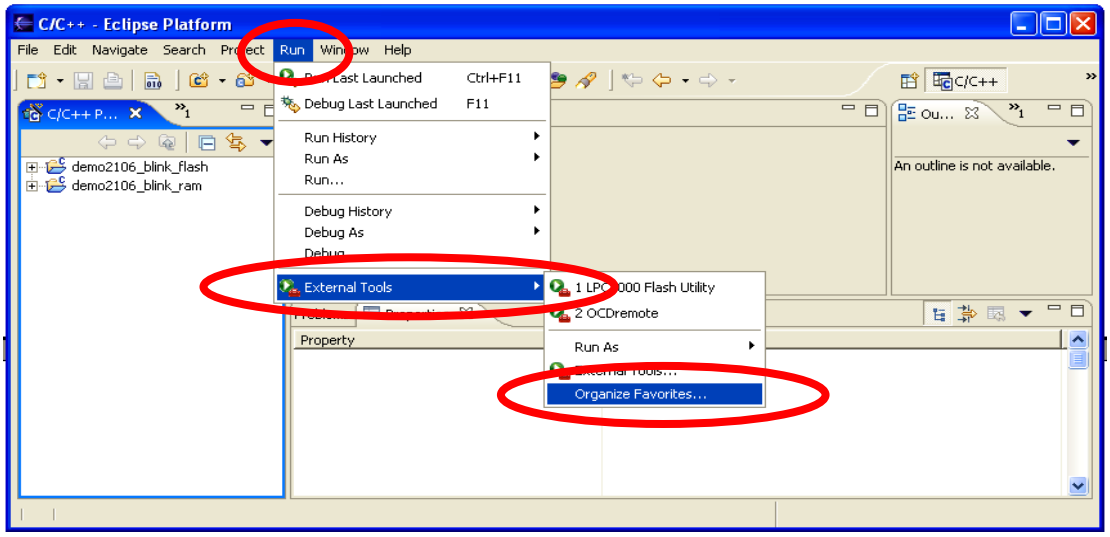


In the External Tools window above, the Insight executable is typed into the Location: text box. You can use the "**Browse File System**" button to hunt for it.

The "**Working Directory**" and "**Arguments**" text box will contain the project's workspace file folder and the project's arm-elf executable, in this example "**main.out**". For the moment, you can leave these two text boxes blank, we'll return to them later when we get ready to run the Insight debugger.

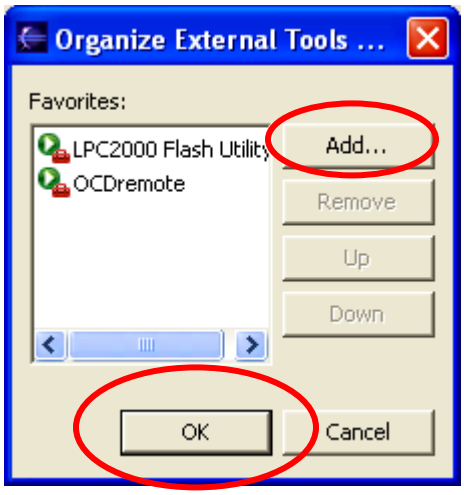
Similar to the Philips Flash Utility and the Macraigor OCDremote, we'd like to add the Insight debugger to our list of "favorite" External Tools.

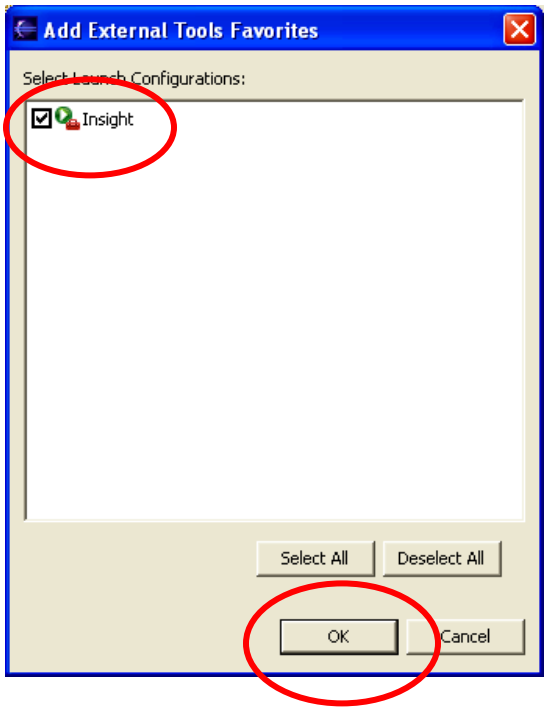
Click on "**Run – External Tools – Organize Favorites …**".



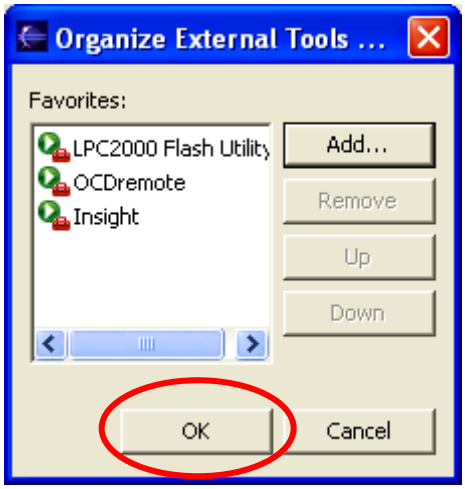
In the "Organize External Tools …" window on the left below, click on "**Add …**" and then "**OK**".

In the "Add External Tools Favorites" window on the right below, click and check-mark the Insight debugger select box and then click "**OK**" to add it as a favorite.



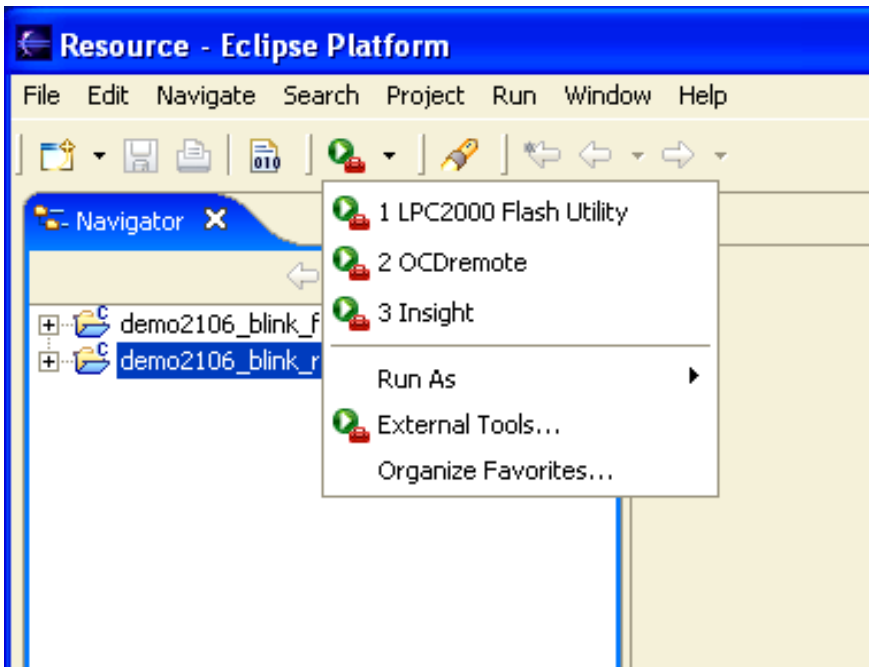


Finally, click on “OK” to officially add Insight as a favorite External Tool.



As one final confidence check, click on the “Run External Tools” button and verify that all three tools are properly installed.

It’s worth mentioning that to run the **Insight** Graphical Debugger; you will click on **OCDremote** first to get it running and then click on **Insight** to start the debugger.



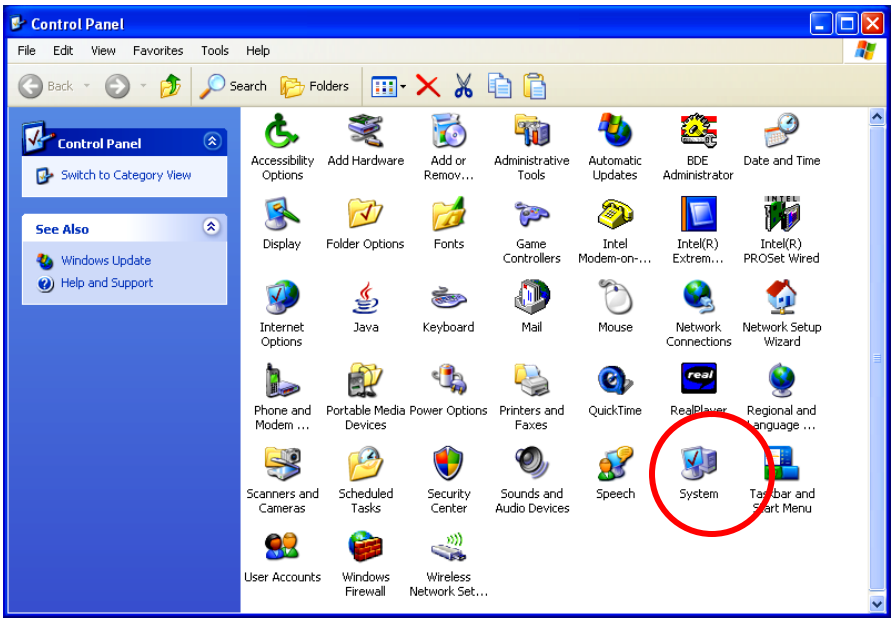
11 Verifying the PATH Settings

There is one final and very crucial step to make before we complete our tool building. We have to ensure that the Windows PATH environment variable has entries for the Cygwin toolset, the GNUARM toolset and the OCDremote JTAG server.

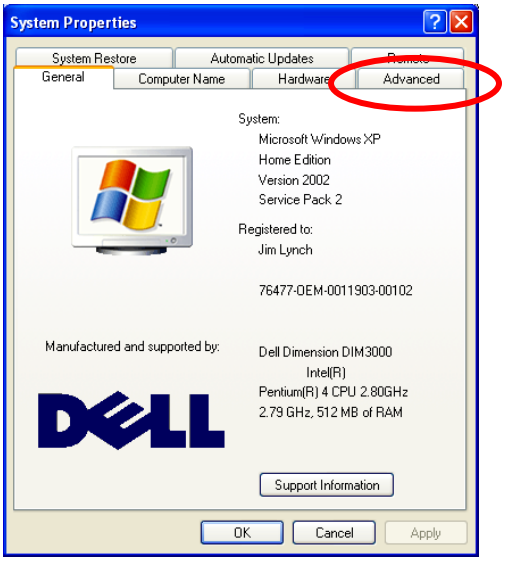
These are the three paths that **must** be present in the Windows environment:

c:\cygwin\bin
c:\program files\gnuarm\bin
c:\cygwin\usr\local\bin

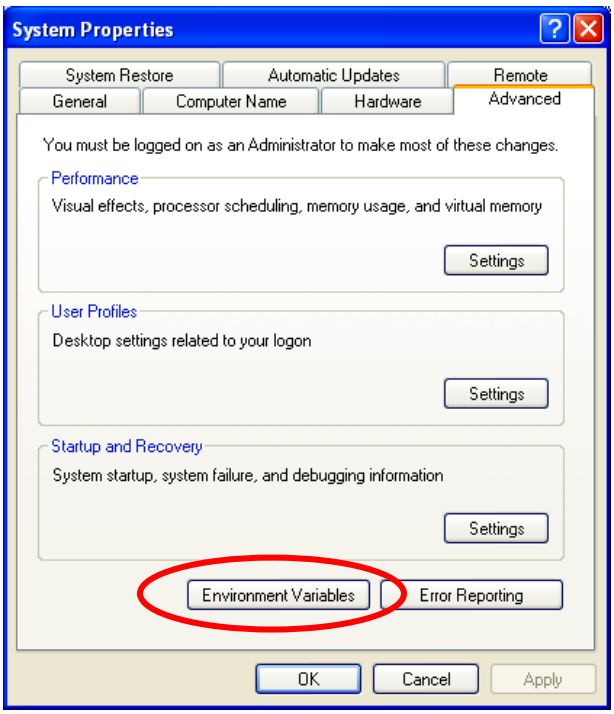
To verify that these paths are present in Windows and to make changes if required, start the Windows Control Panel by clicking "**Start – Control Panel**".



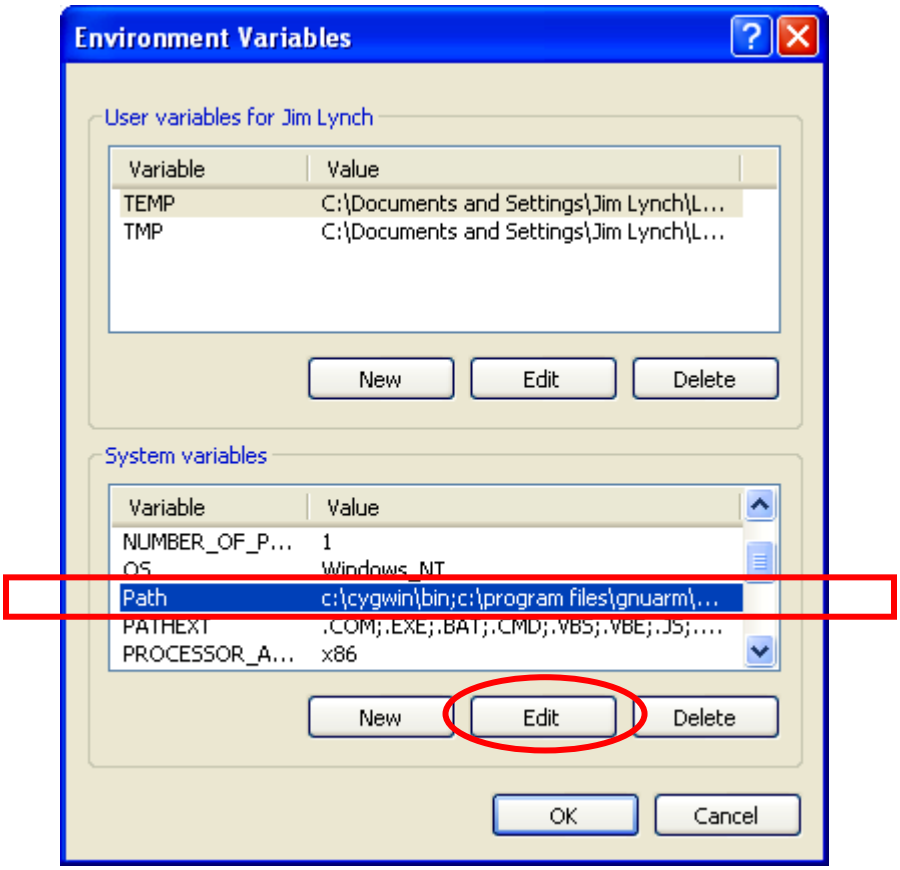
Now click on the "**Advanced**" tab below.



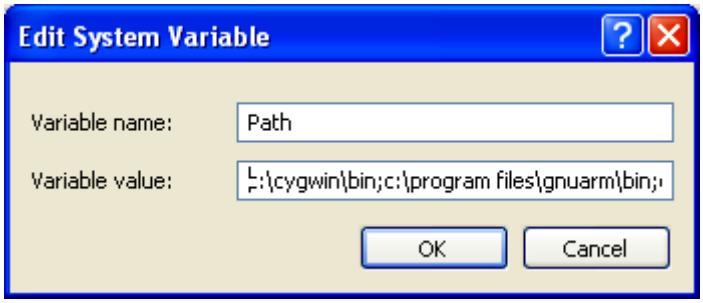
Now click on the “**Environment Variables**” button.



In the Environment Variables window, find the line for “**Path**” in the System Variables box on the bottom, click to select and highlight it and then click on “**Edit**”.



Take a very careful look at the “Edit System Variable” window (the Path Edit, in this case).



You should see the following paths specified, all separated by semicolons. The path is usually long and complex; you may find the bits and pieces for GNUARM interspersed throughout the path specification. I used cut and paste to place all my path specifications at the beginning of the specification (line); this is not really necessary.

You should see the following paths specified.

c:\cygwin\bin;c:\program files\gnuarm\bin;c:\cygwin\usr\local\bin

If any of the three is not present, now is the time to type them into the path specification.

I’ve found that not properly setting up the Path specification is the most common mistake made in configuring Eclipse to do cross-development.

This completes the setup of Eclipse and all the ancillary tools required to cross develop embedded software for the ARM microcomputer family (Philips LPC2000 family in specific).

If you stayed with me this far, your patience will soon be rewarded!

Or as Yoda would say, “***Rewarded soon, your patience will be!***”

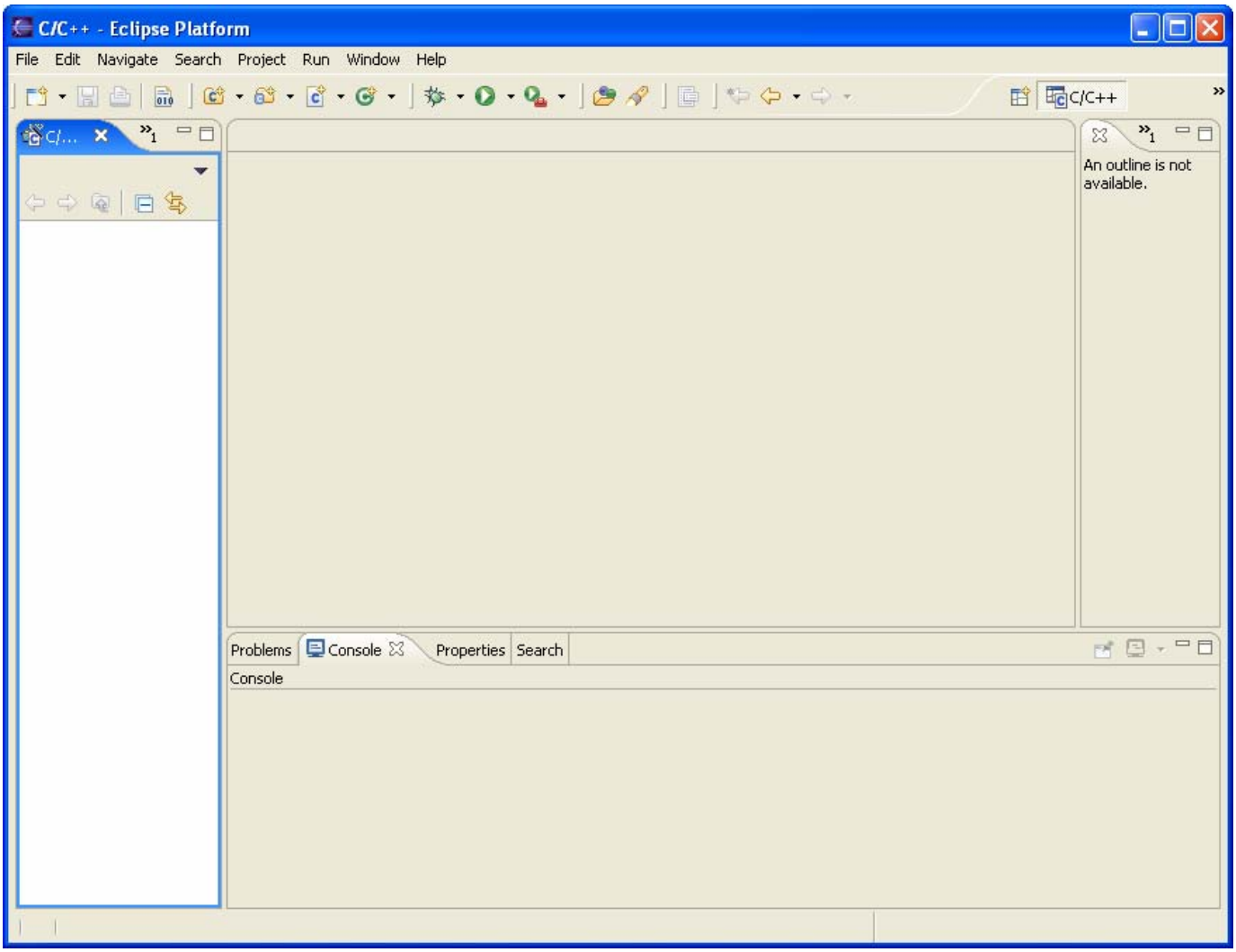
12 Creating a Simple Eclipse Project

At this point, we have a fully-functioning Eclipse IDE capable of building C/C++ programs for the ARM microprocessor (specifically for the Olimex LPC-P2106 prototype board).

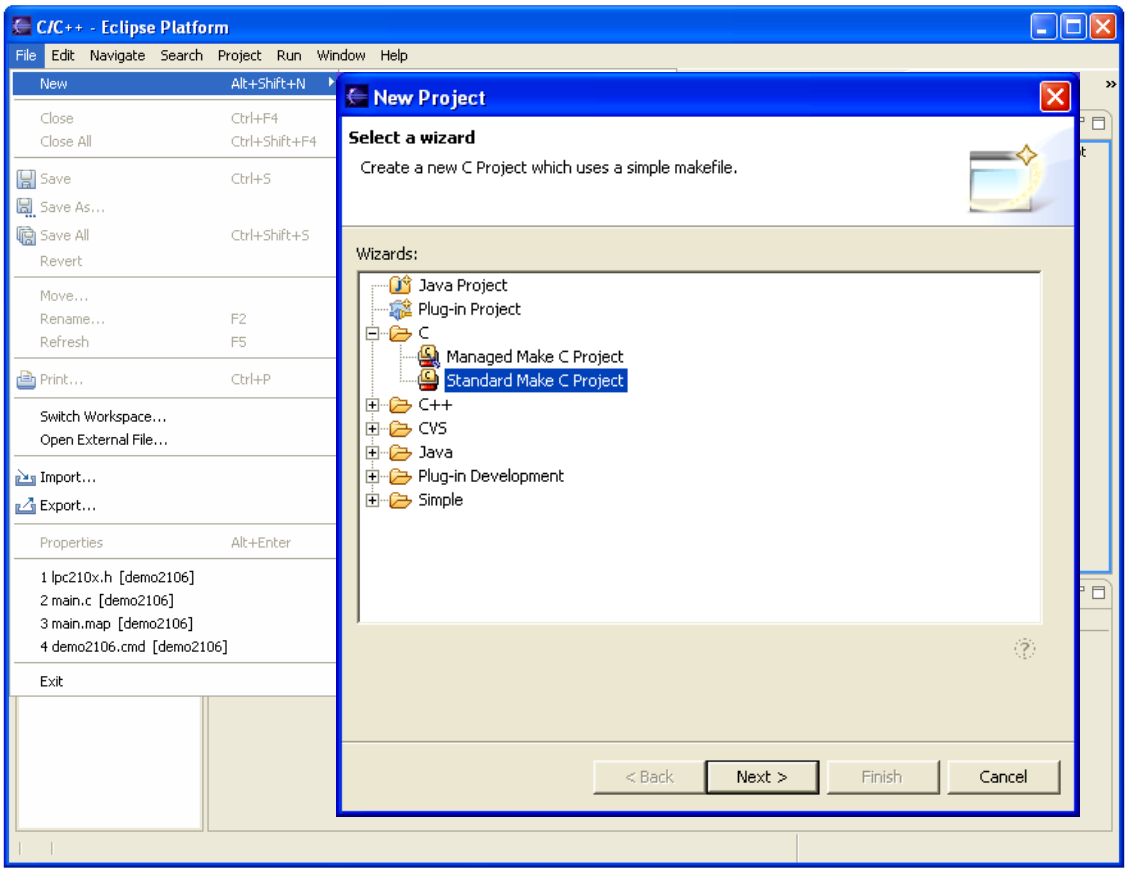
We will now create an Eclipse C project called "**demo2106_blink_flash**" that will blink the board's red LED_J which is I/O port P0.7. This demo uses no interrupts and runs totally out of onboard flash memory. It has been intentionally designed to be as simple and as straightforward as possible.

Click on our Eclipse desktop icon to start Eclipse.

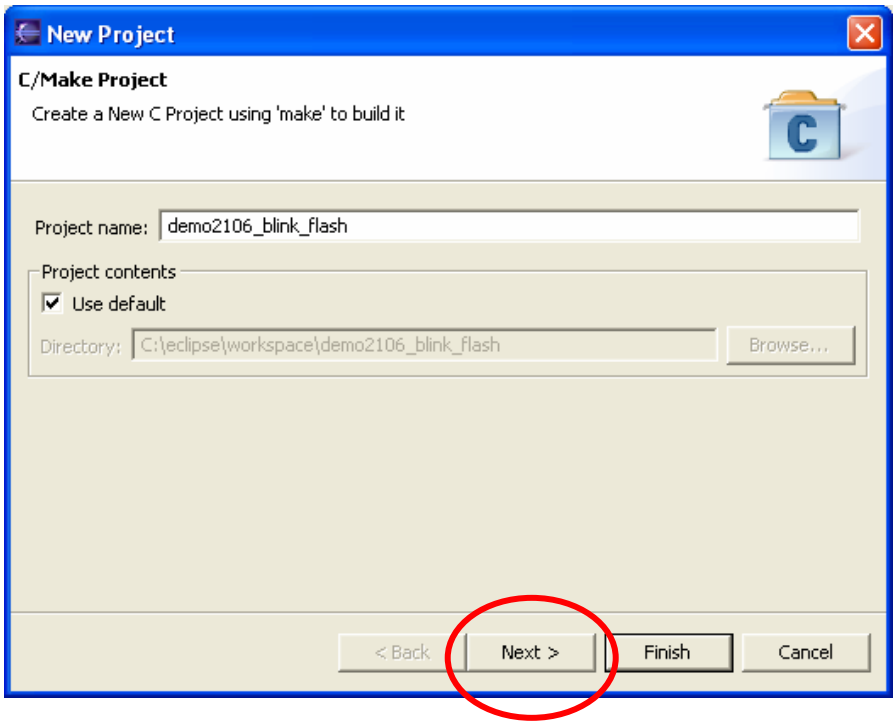
Eclipse should start and present the C/C++ perspective as shown below. If not, select "**Window - Open Perspective – Other - C/C++**" to change to the C++ perspective.



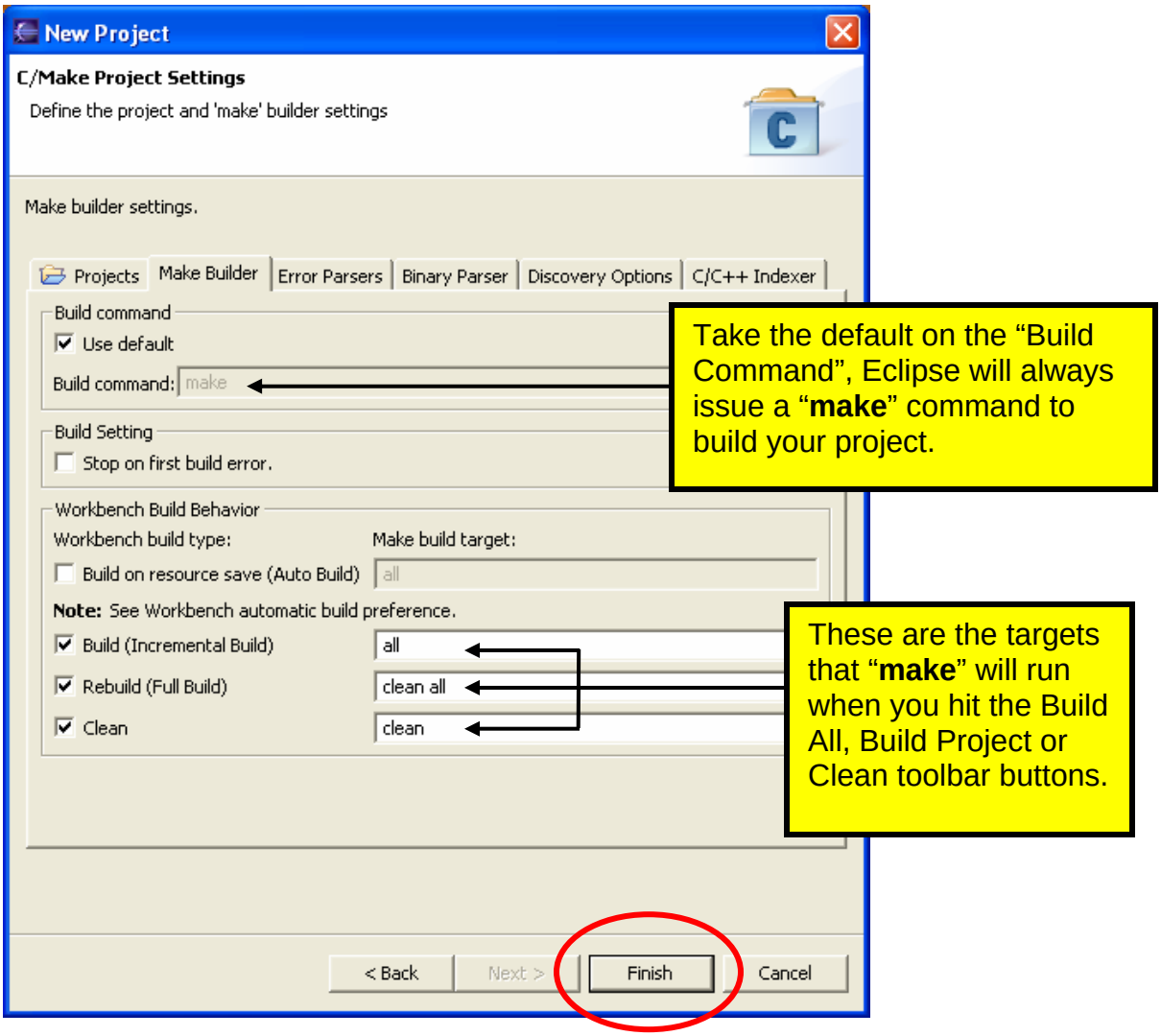
To create a project, select **File – New – New Project - Standard Make C Project** from the File pull-down menu and click "**Next**" to continue.



You should see the "New Project" dialog box and enter the project name (**demo2106_blink_flash**) in the box as shown below. Click on **Next** to continue.



The **New Project** dialog box appears next. If you click on the "**Make Builder**" tab, you'll notice that Eclipse build command is "**make**." Make is provided by the Cygwin GNU tools.



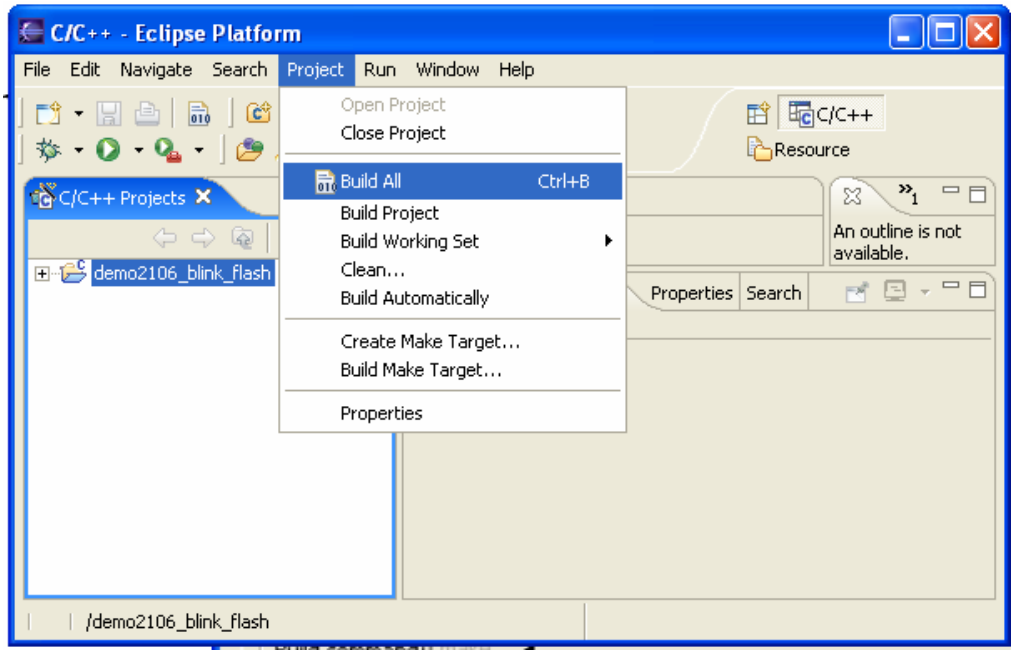
Let's remind ourselves that we installed the Cygwin GNU tools earlier in the tutorial and the Windows Explorer will show that the **make.exe** file is indeed in the directory **c:/cygwin/bin**, as shown below.

This is a good time to point out the differences between “Build All”, “Build Project” and “Clean.”

Build All Will execute the command “**make clean all**.”
It will first clean (delete) all object, list and output files.
Then it will rebuild everything, whether needed or not.

Build Project Will execute the command “**make all**.”
This will not clean (delete) anything.
It will only compile those source files that are “out-of-date.”

Clean Will execute the command “**make clean**.”
Will clean (delete) all object, list and output files.

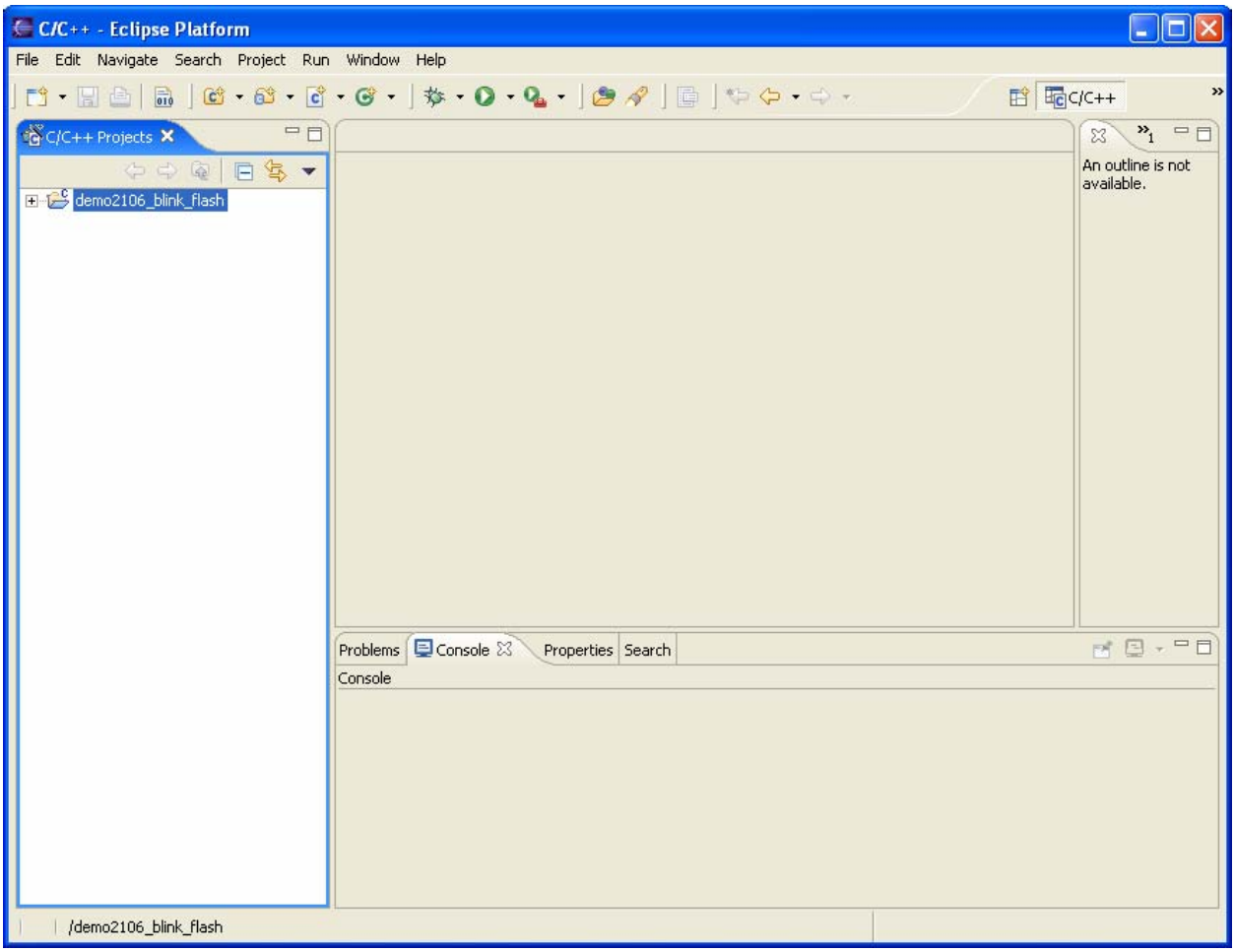


This is no different from opening up a DOS command window and typing the command in directly, such as.

```
> make clean all
```

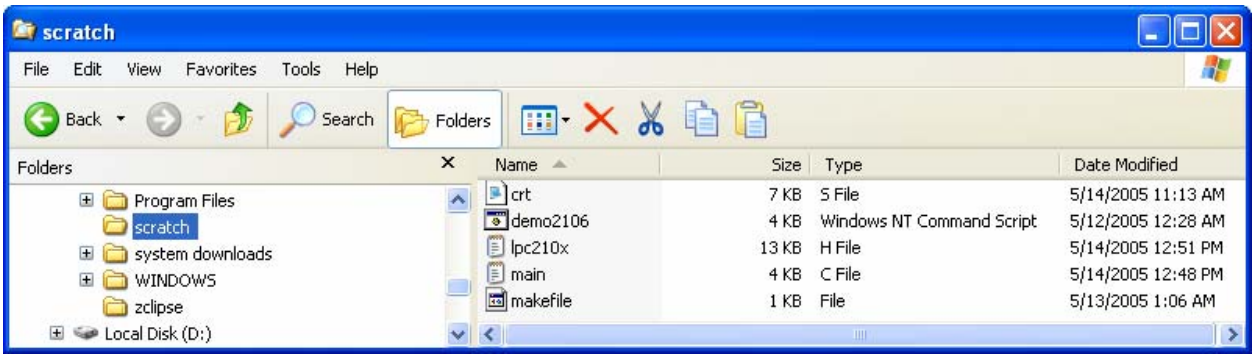
If you click “**Finish**” on the “New Project” dialog, Eclipse will return to the C/C++ Perspective.

Now the C/C++ perspective shows a bona fide project in the “C/C++ projects” box on the left. As of now, there are no source files created.

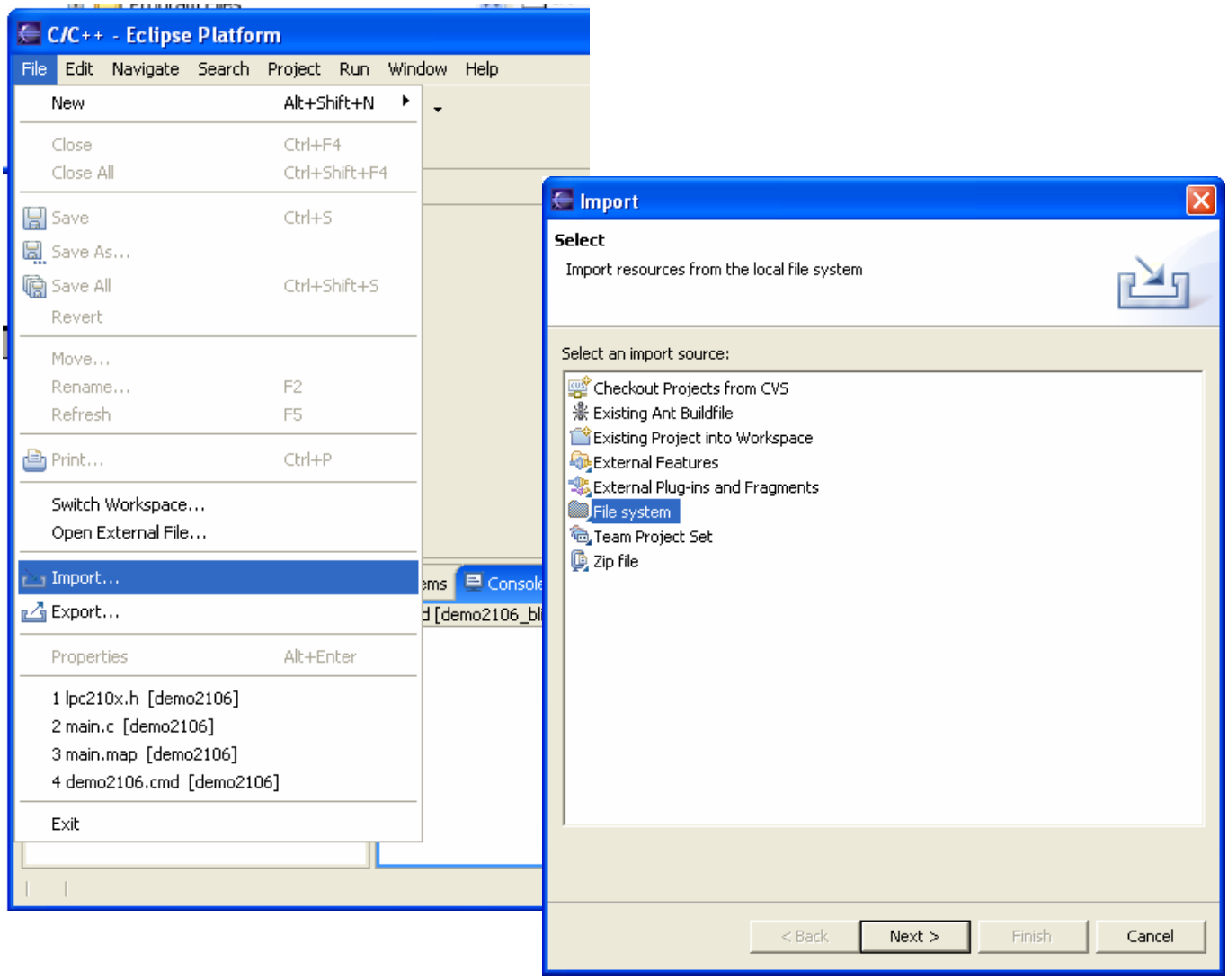


We can now use Eclipse/CDT’s **import** feature to copy the source files into the project.

Assuming that you successfully unzipped the “**demo2106_blink_flash.zip**” project files associated with this tutorial to an empty directory such as **c:/scratch**, you should have the following source and make files in that directory.

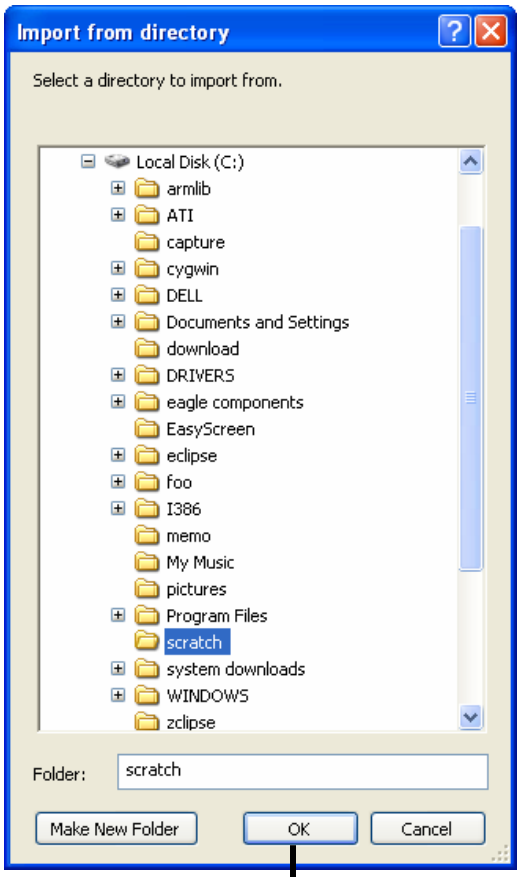


Click on the “**File**” pull-down menu and then click on “**Import.**” Then in the “**Import**” window, click on “**File System**.”

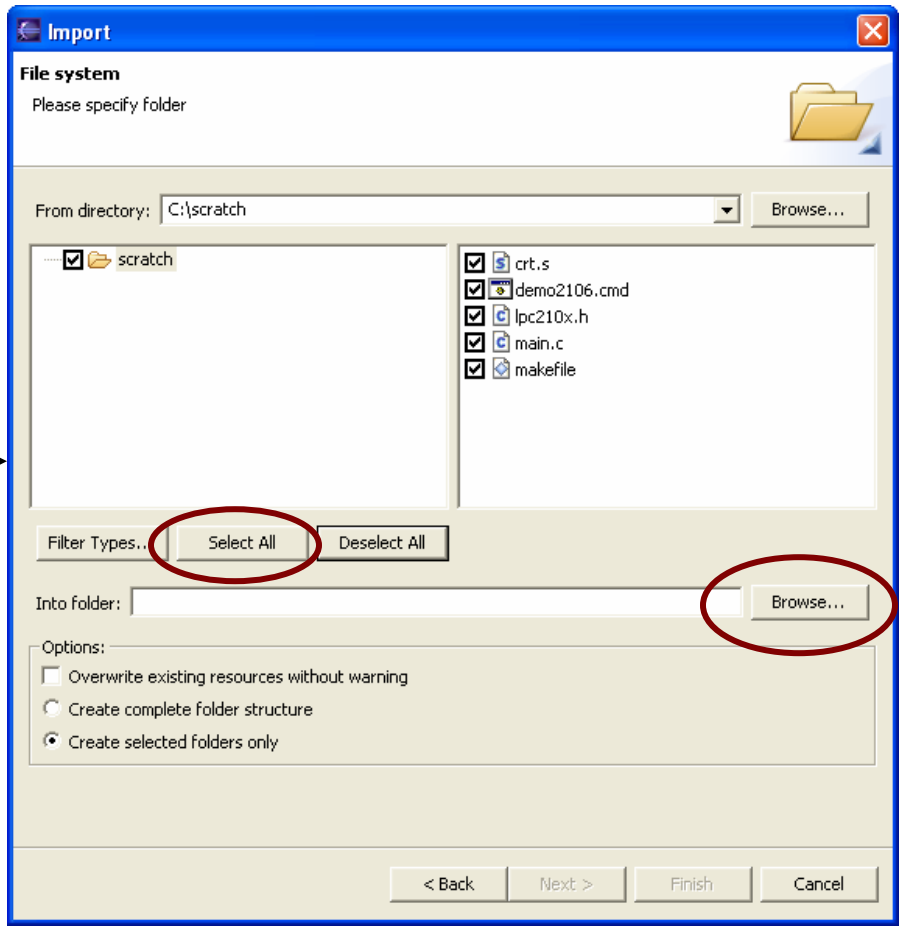


When the “**Import – File System**” window appears, click on the “**Browse**” button. Hunt for the sample project which is stored in the **c:/scratch/** directory.

Click on the directory “**scratch**” and hit the “**OK**” button in the “Import from directory” window on the left below.



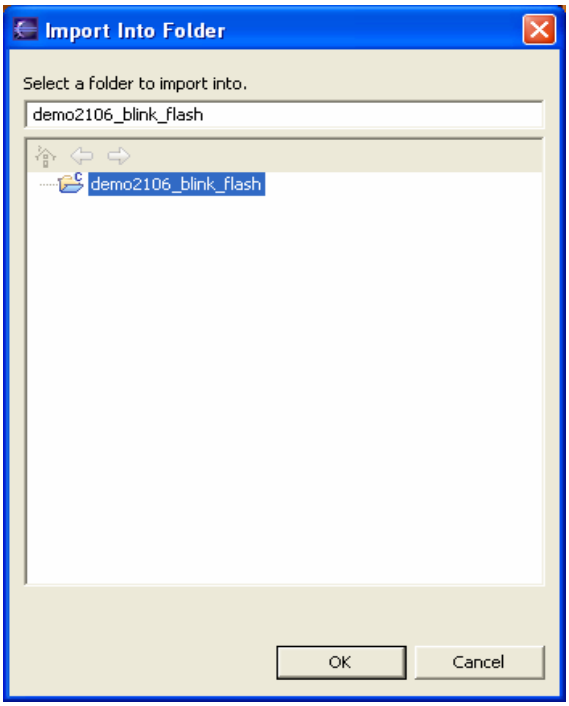
Click on “**Select All**” in the Import window below right to get the source files selected for import into our project.



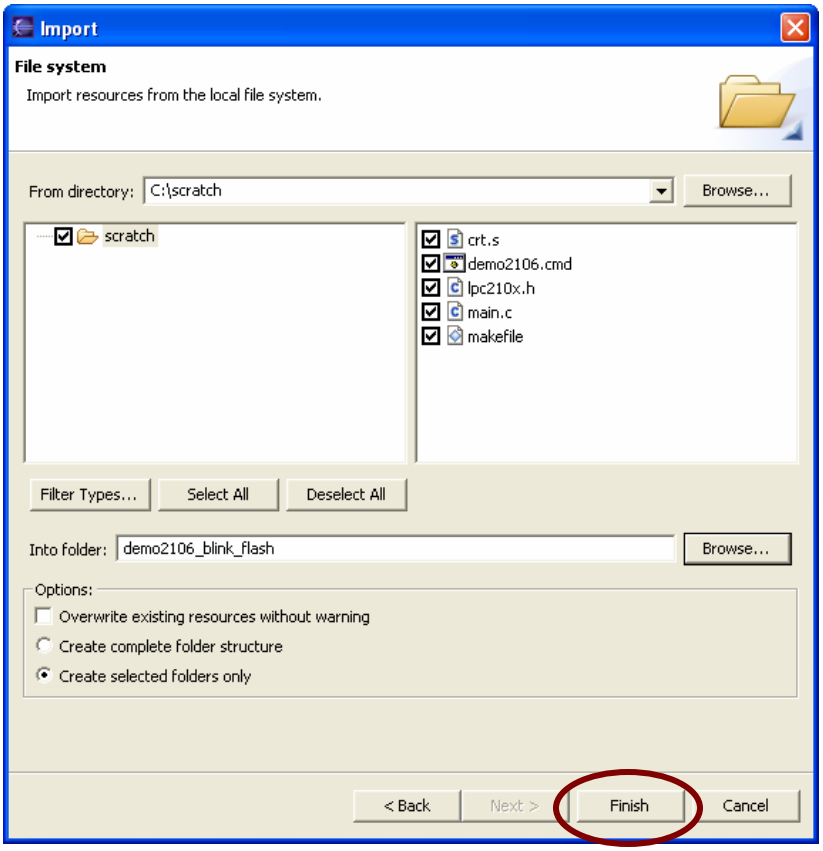
Now we have to indicate the destination for our source files. Click on “**Browse**” on the line to the right that says “**Into Folder**:”

The proper destination folder appears in the **Import Into Folder** window below.

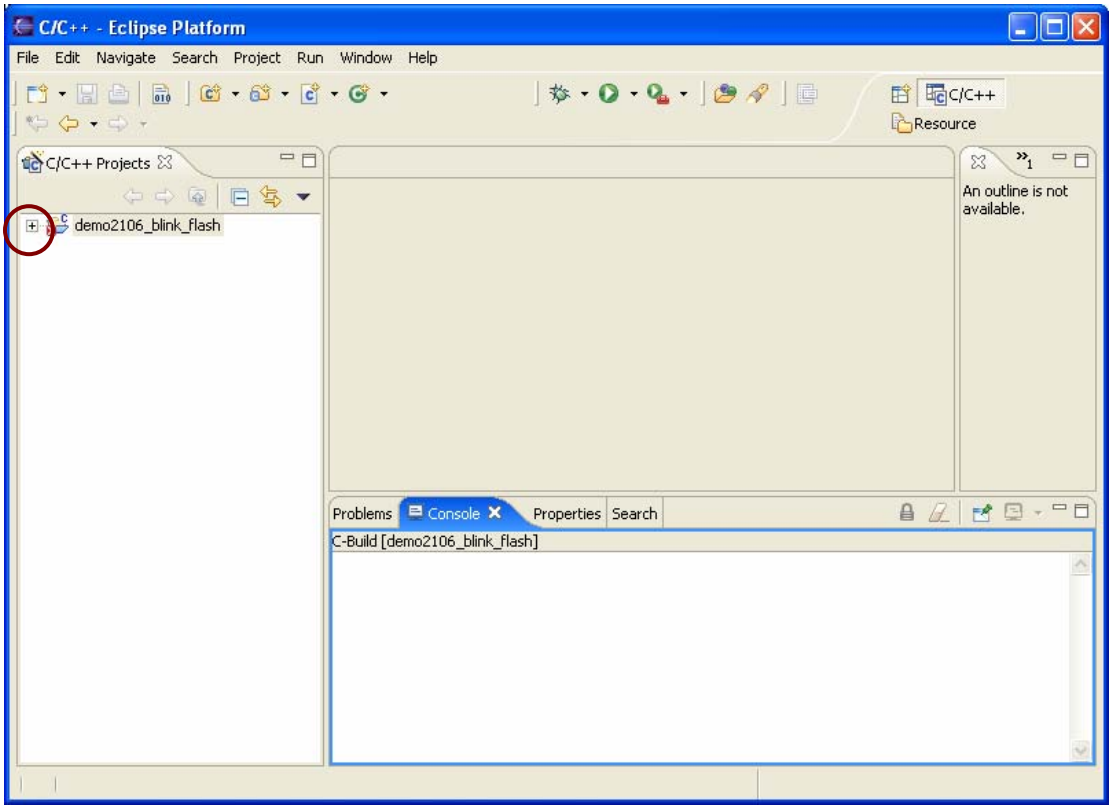
Click on the folder name "**demo2106_blink_flash**" and click "**OK**." The directory name "demo2106_blink_flash" should appear in the text box.



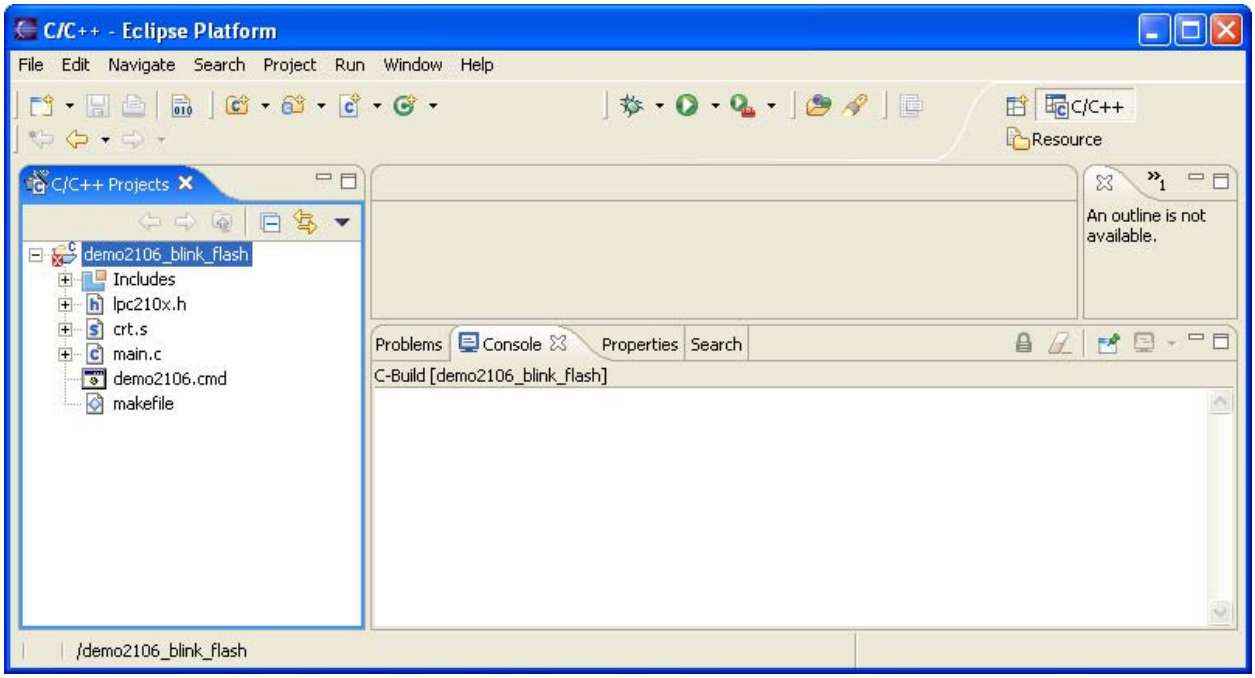
Now the Import dialog is completely filled out; we can click on "finish" to actually import the source files into our project.



Now the C/C++ perspective main screen will reappear. Click on the "**+**" expand symbol in the navigator pane to see if our files have been transferred.



Success is at hand, the expanded Projects view in the Navigator pane on the left shows our imported files.



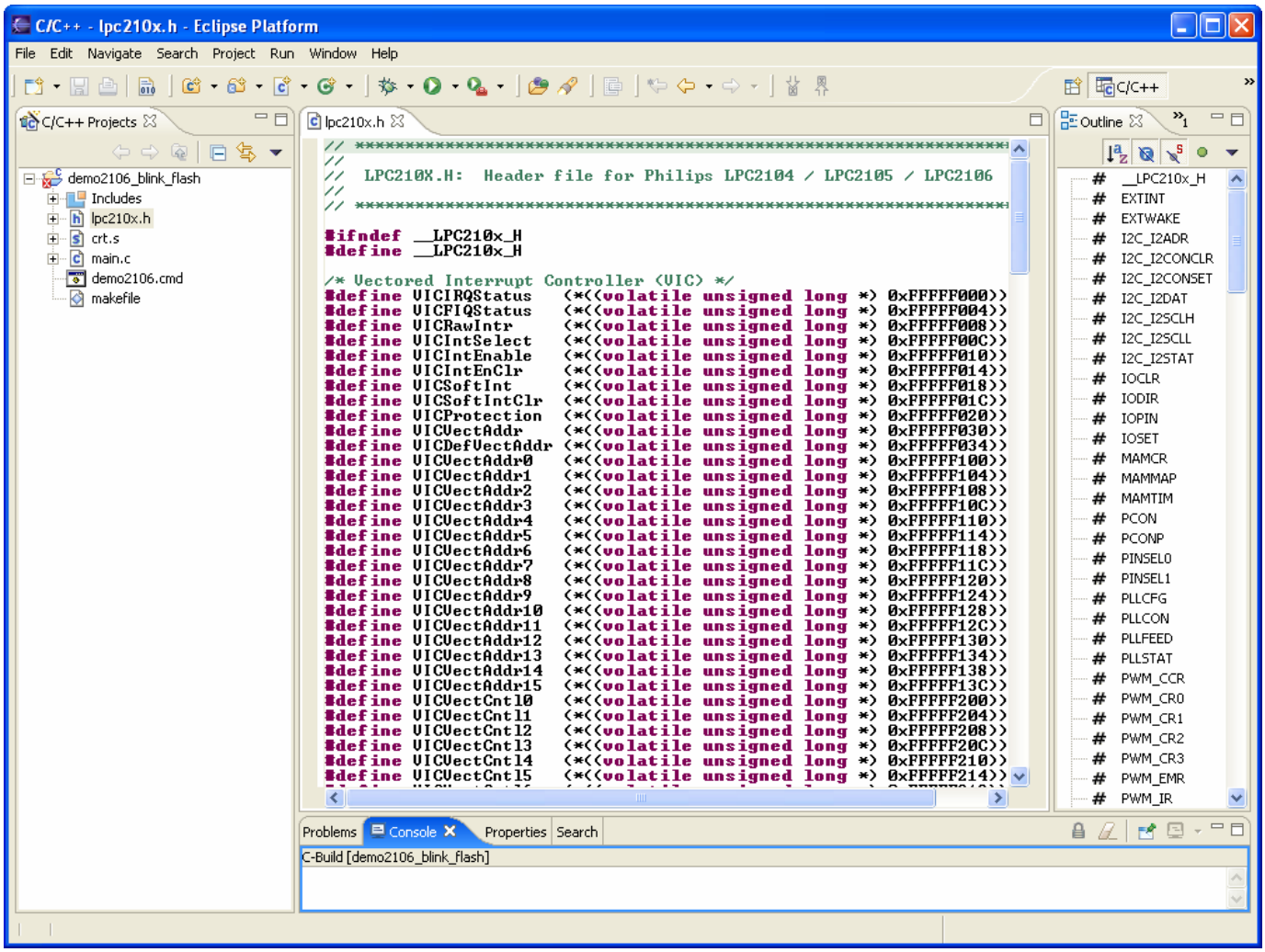
This is a good place to identify the imported source files.

Description of Project Files	
lpc210x.h	Standard LPC2106 header file
crt.s	Startup assembler file
main.c	Main C program
makefile	GNU makefile
demo2106_blink_flash.cmd	GNU Linker script file

13 Description of the LPC210X.H Include File

Let's look at the lpc210x.h header file. Double-click on it in the Project pane on the left'

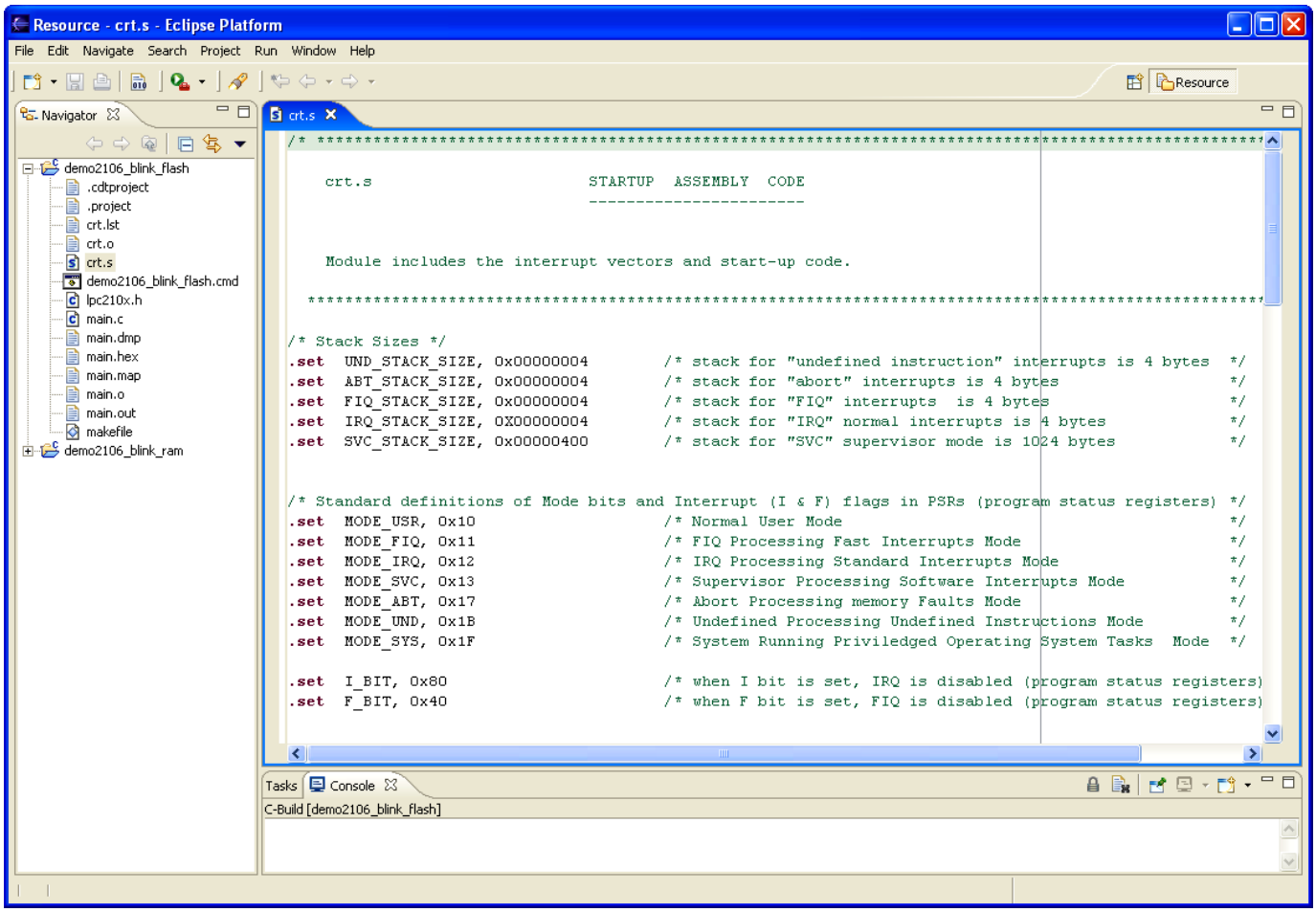
ARM peripherals are memory-mapped, so all I/O registers are defined in this file so you don't have to type in the absolute memory addresses.



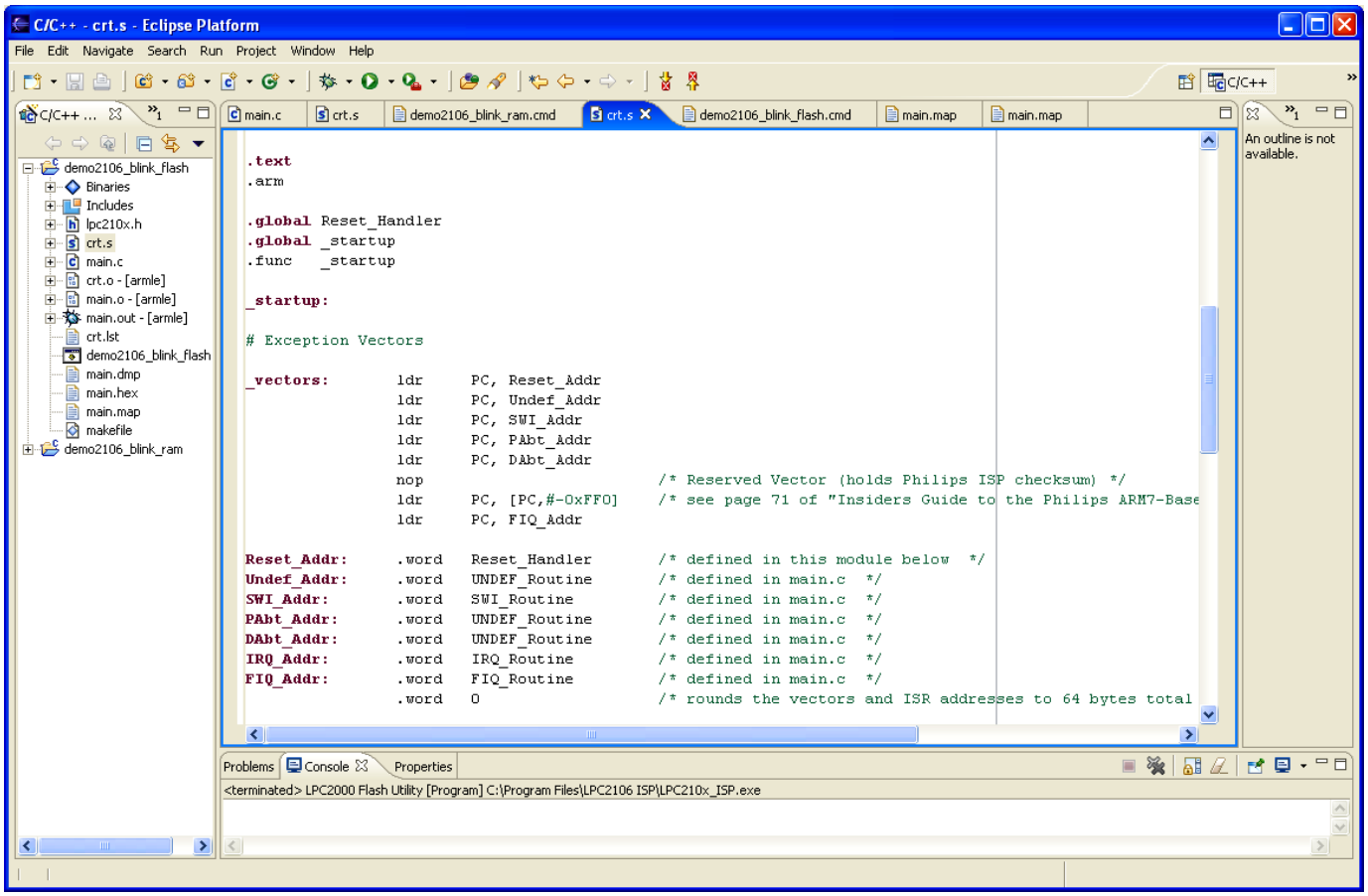
14 Description of the Startup File CRT.S

Now let's look on the startup assembler file, **crt.s**. Double-click on it.

This part of the **crt.s** file has some symbols set to the various stack sizes and mode bits.



This part of the **crt.s** file sets up the interrupt vectors.



Note that all of the code and data that follows goes into the **.text** section. It is also in ARM 32-bit code (not Thumb).

One label is made global, **_startup**. This will be available to other modules in the project and will also appear in the map.

The GNU assembler doesn't require you **.extern** anything. If a symbol is not defined in the assembler file, it is automatically assumed to be external.

The vector table is 32 bytes long and is **required** to be placed at address 0x000000.

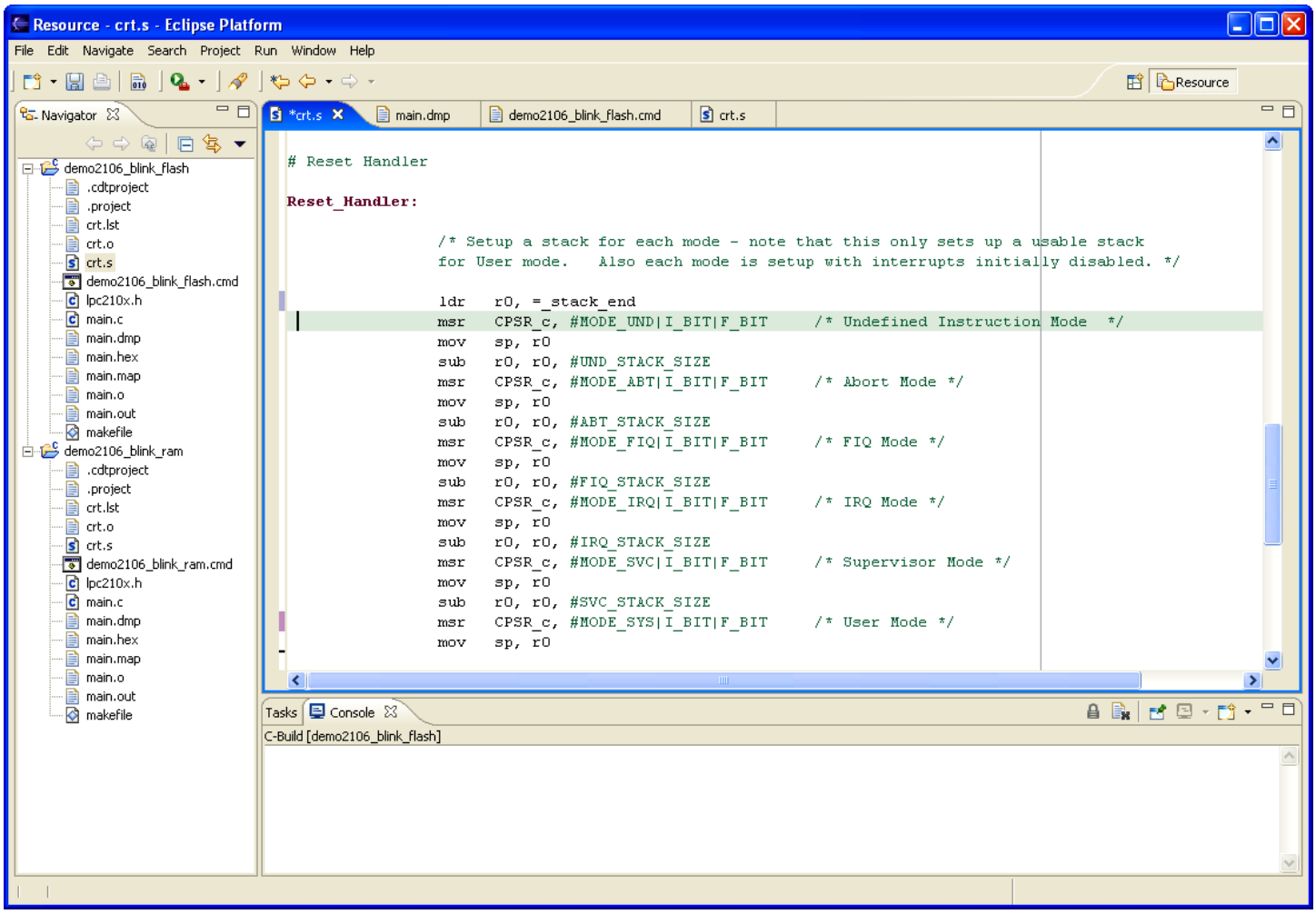
You will see later in this tutorial that the interrupt service routines referenced in the Vector Table are just endless-loop stubs in the main.c function and the interrupts are turned off.

The **NOP** instruction at address 14 is an empty spot to hold the checksum. Page 179 of the Philips LPC2106 manual states:

> **The reserved ARM interrupt vector location (0x0000 0014) should contain the 2's complement of the check-sum of the remaining interrupt vectors. This causes the checksum of all of the vectors together to be 0.**

Before you fall on your sword, you'll be happy to know that the Philips Flash Loader will calculate that checksum and insert it for you. That's why we show it as a NOP.

This part of the **crt.s** file sets up the various interrupt modes and stacks.



The label **Reset_Handler** is the beginning of the code. Recall that the first interrupt vector at address 0x000000 loads the PC with the contents of the address Reset_Addr, which contains the address of the startup code at the label Reset_Handler. This trick, used in the entire vector table, loads a 32-bit constant into the PC and thus can jump to any address in memory space.

```
_vectors:      ldr    PC, Reset_Addr
                 :
Reset_Addr:    .word  Reset_Handler
```

Whenever the LPC2106 is reset, the instruction at 0x000000 is executed first; it jumps to **Reset_Handler**. From that point, we are off and running!

The first part of the startup code above sets up the stacks and the mode bits.

The symbol **_stack_end** will be defined in the linker command script file demo2106.cmd. Here is how it will be defined. Knowing that the Philips ISP Flash Loader will use the very top 288 bytes of RAM for its internal stack and variables, we'll start our application stacks at **0x4000FEE0**.
(Note: 0x40010000 – 0x120 = 0x4000FEE0)

```
/* define a global symbol _stack_end, placed at the very end of RAM (minus 4 bytes)  */
stack_end = 0x4000FEE0 – 4;
```

Working that out with the Windows calculator, the _stack_end is placed at 4000FEDC.

The code snippet that sets up the stacks and modes is a bit complex, so let's explain it a bit.

First we load R0 with the address of the end of the stack, as described above.

```
ldr   r0, =_stack_end
```

Now we put the ARM into Undefined Instruction mode by setting the MODE_UND bit in the Current Program Status Register (CPSR). The four modes undefined, irq, abort and svc all have their own private copies of R13 (sp) and r14 (link return). The FIQ mode has private copies of registers R8 – R14. Thus, by writing R0 into the stack pointer sp (R13), it will use 0x4000FEDC as the initial stack pointer if we ever have processing of an undefined instruction. By subtracting the undefined stack size (4 bytes) from R0, we're limiting the stack for UND mode to just 4 bytes.

```
msr   CPSR_c, #MODE_UND|I_BIT|F_BIT     /*  This puts the CPU in undefined mode */
mov   sp, r0                            /* stack pointer for UND mode is 0x40000FEDC  */
sub   r0, r0, #UND_STACK_SIZE           /* Register R0 is now 0x4000FED8  */
```

Now we put the ARM into Abort mode by setting the MODE_ABT bit in the CPSR. As mentioned above, abort mode has its own private copies of R13 and R14. We now set the abort mode stack pointer to 0x4000FED8. Again by subtracting the abort stack size from R0, we're limiting the stack for ABT mode to just 4 bytes.

```
msr   CPSR_c, #MODE_ABT|I_BIT|F_BIT     /* this puts CPU in Abort mode  */
mov   sp, r0                            /* stack pointer for ABT mode is 0x4000FED8  */
sub   r0, r0, #ABT_STACK_SIZE           /* Register R0 is now 0x4000FED4  */
```

Now we put the ARM into FIQ (fast interrupt) mode by setting the MODE_FIQ bit in the CPSR. As mentioned above, FIQ mode has its own private copies of R14 through R8. We now set the abort mode stack pointer to 0x4000FED4. Again by subtracting the abort stack size from R0, we're limiting the stack for FIQ mode to just 4 bytes. We're not planning to support FIQ interrupts in this example.

```
msr   CPSR_c, #MODE_FIQ|I_BIT|F_BIT     /* this puts CPU in FIQ mode  */
mov   sp, r0                            /* stack pointer for FIQ mode is 0x4000FED4
sub    r0, r0, #FIQ_STACK_SIZE          /* Register R0 is now 0x4000FED0  */
```

Now we put the ARM into IRQ (normal interrupt) mode by setting the MODE_IRQ bit in the CPSR. As mentioned above, IRQ mode has its own private copies of R13 and R14. We now set the IRQ mode stack pointer to 0x4000FDE0. Again by subtracting the IRQ stack size from R0, we're limiting the stack for IRQ mode to just 4 bytes. We're not planning to support IRQ interrupts in this example.

```
msr   CPSR_c, #MODE_IRQ|I_BIT|F_BIT     /* this puts the CPU in IRQ mode  */
mov   sp, r0                            /* stack pointer for IRQ mode is 0x4000FED0  */
sub   r0, r0, #IRQ_STACK_SIZE           /* R0 is now 0x4000FECC  */
```

Now we put the ARM into SVC (Supervisor) mode by setting the MODE_SVC bit in the CPSR. As mentioned above, SVC mode has its own private copies of R13 and R14. We now set the supervisor mode stack pointer to 0x4000FDDC. Again by subtracting the SVC stack size(4 bytes) from R0, we're sizing the stack for SVC mode to 4 bytes.

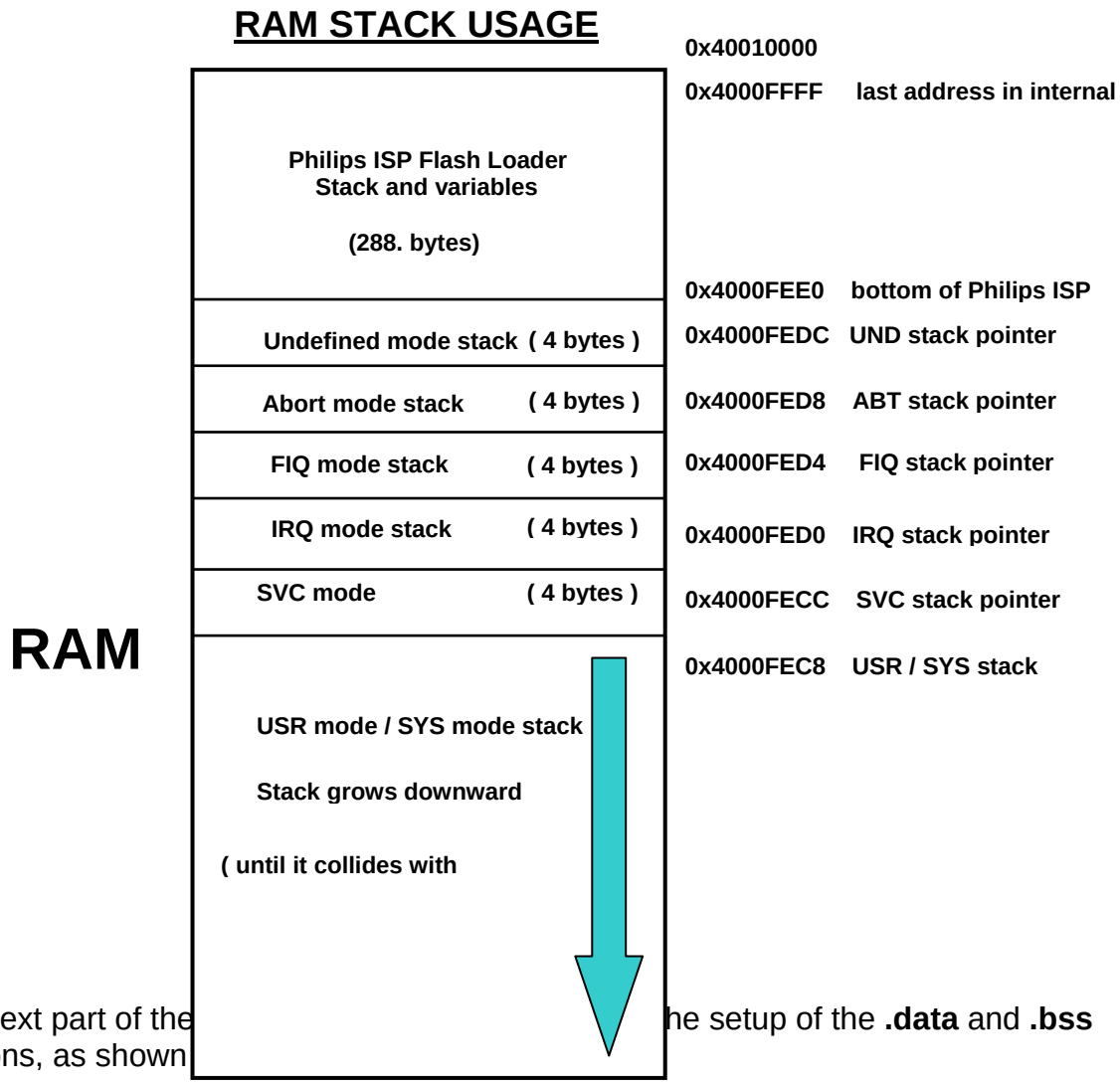
```
msr   CPSR_c, #MODE_SVC|I_BIT|F_BIT     /* This puts the CPU in SVC mode */
mov   sp, r0                            /* stack pointer for SVC mode is 0x4000FECC  */
sub   r0, r0, #SVC_STACK_SIZE           /* R0 is now 0x4000FEC8  */
```

The ARM "User" mode and the ARM "System" mode share the same registers and stack. For this very simple example, we'll run the application in "User" mode. Setting up the stack for User mode also sets up the stack for System mode.

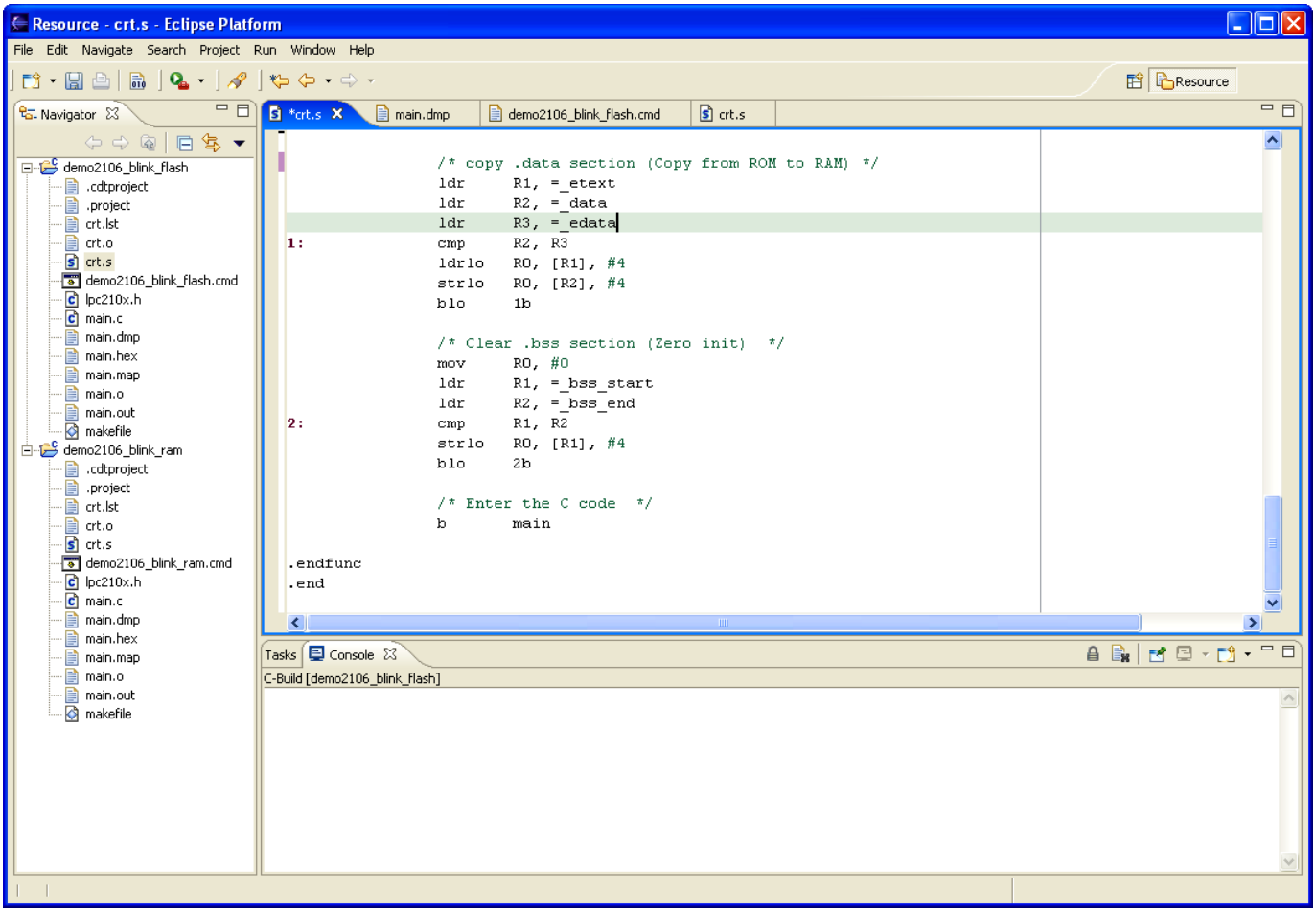
Now we put the ARM into USR (user) mode by setting the MODE_USR bit in the CPSR. We now set the USR mode stack pointer to 0x4000FEC8.

```
msr   CPSR_c, #MODE_USR|I_BIT|F_BIT             /* User Mode  */
mov   sp, r0
```

To summarize the above operations, let's draw a diagram of the stacks we just created.



The next part of the [image] he setup of the **.data** and **.bss** sections, as shown



The **.data** section contains all the initialized static and global variables. The GNU linker will create a exact copy of the variables in flash with the correct initial values loaded. The onus is on the programmer to copy this initialized flash copy of the data to RAM.

The location of the start of the **.data** section in flash is defined by symbol **_etext** (defined in the linker command script **demo2106.cmd**). Likewise, the location of the start and end of the **.data** section in destination RAM is given by the symbols **_data** and **_edata**. Both of these symbols are defined in the linker command script.

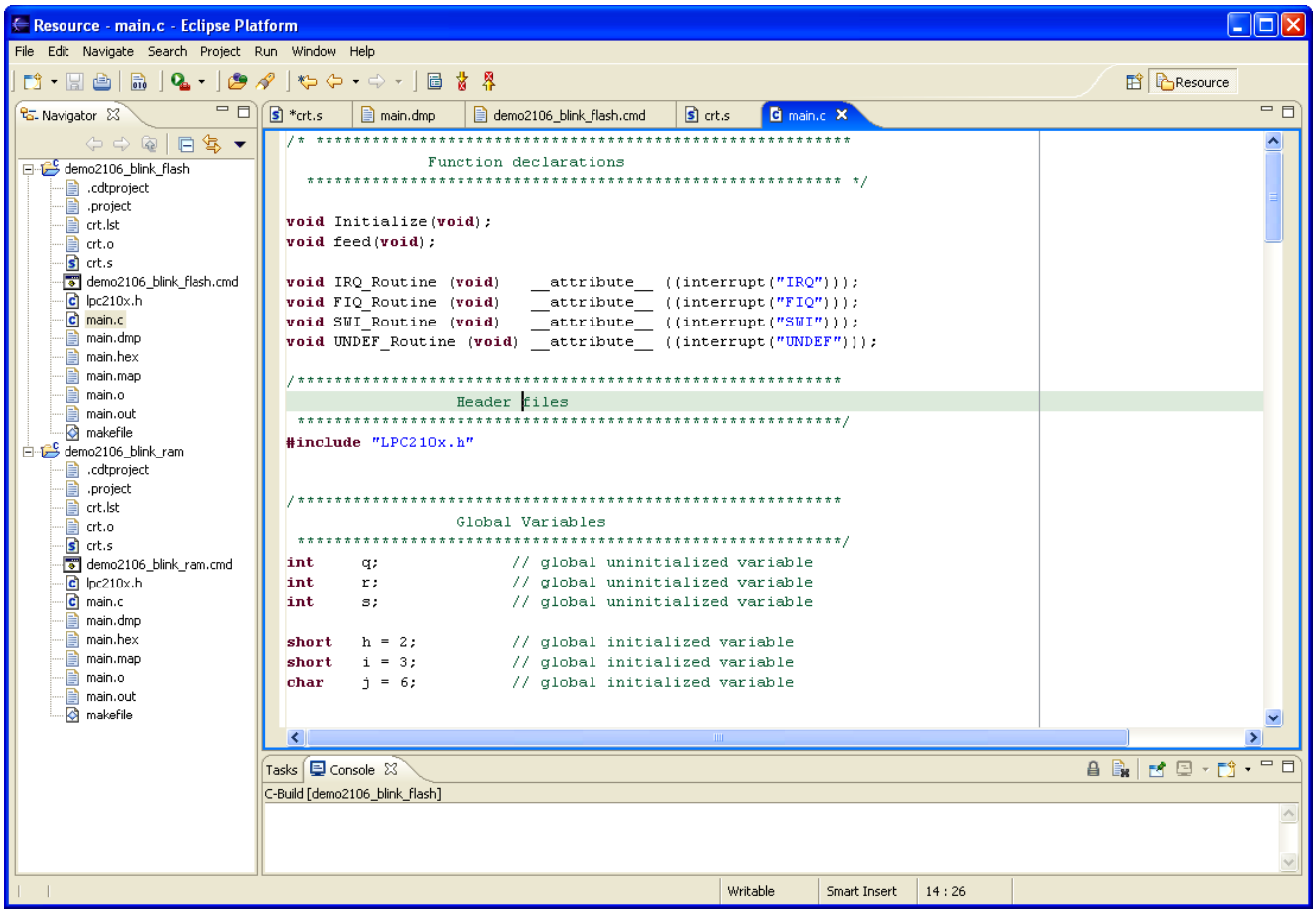
The **.bss** section contains all the uninitialized static and global variables. All we have to do here is clear this area. Likewise, the location of the start and end of the **.bss** section in destination RAM is given by the symbols **_bss_start** and **_bss_end**. Both of these symbols are defined in the linker command script.

Two simple assembly language loops load the **.data** section in RAM with the initializers in flash and clear out the **.bss** section in RAM.

The GNU linker specifies two addresses for sections, the Virtual Memory Address (**VMA**) and the Load memory Address (**LMA**). The **VMA** is the final destination for the section; for the **.data** section, this is the RAM address where it will reside. The **LMA** is where it will be loaded in Flash memory, the exact copy with the initial values. The GNU Linker will sort this out for us.

15 Description of the Main Program main.c

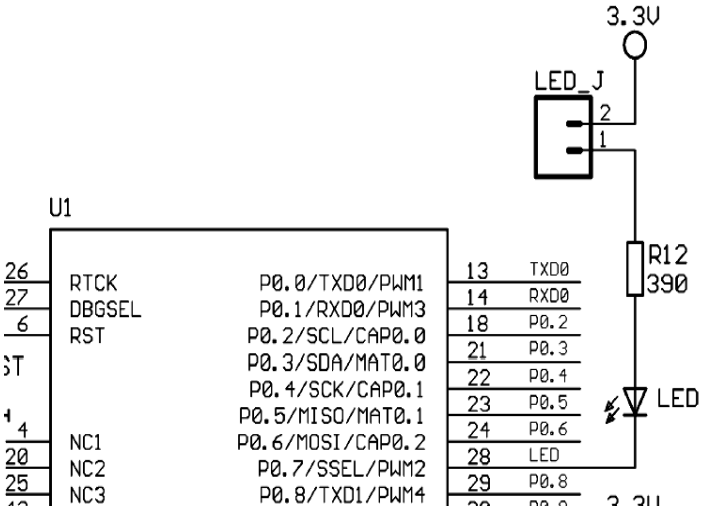
Now let's look at the main program.



The main program starts out with a few function prototypes. Note that the interrupt routines mentioned in the crt.s assembler program reside in the **main()** program. We've used the GNU C compiler syntax that identifies the interrupt routines and makes sure that the compiler will save and restore registers, etc. whenever the interrupt is asserted.

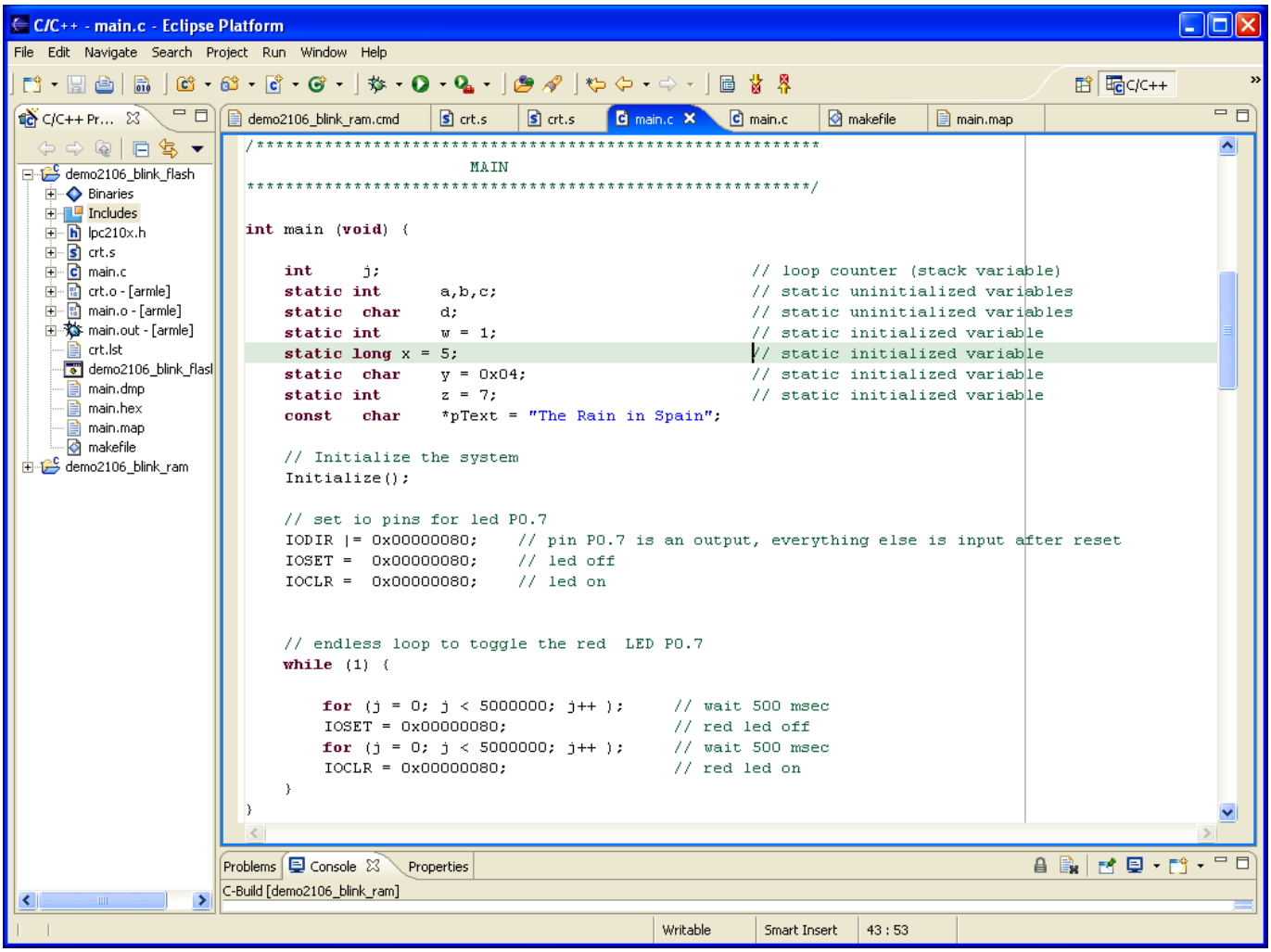
I've also included a few do-nothing variables, both initialized and uninitialized, to illustrate that the compiler will put the initialized variables into the **.data** section and the uninitialized ones into the .bss section.

We're going to try to toggle a single I/O bit, specifically P0.7 which is the Olimex red LED.



By the way, with this hardware arrangement:

P0.7 = 1 // turn off LED
P0.7 = 0 // turn on LED



The Philips LPC2106 has 32 I/O pins, labeled **P0.0** through **P0.31**. Most of these pins have two or three possible uses. For example, pin **P0.7** has three possible uses; digital I/O port, SPI Slave Select and PWM output 2. Normally, you select which function to use with the Pin Connect Block. The Pin Connect Block is composed of two 32-bit registers, PINSEL0 and PINSEL1. Each Pin Select register has two bits for each I/O pin, allowing at least three functions for each pin to be specified.

For example, pin **P0.7** is controlled by **PINSEL0**, bits 14 – 15. The following specification would select PWM2 output.

PINSEL0 = 0x00008000; // set PINSEL0 bits 14 – 15 to 01

Fortunately, the Pin Connect Block resets to zero, meaning that all port pins are General-Purpose I/O bits. So we don't have to set the Pin Select registers in this example.

We do have to set the I/O Direction for port **P0.7**, this can be done in this way.

```
IODIR |= 0x00000080;      // set IO Direction register, P0.7 as output
                          // 1 = output,   0 = input
```

The ARM I/O ports are manipulated by register **IOSET** and register **IOCLR**. You never directly write to the I/O Port! You set a bit in the **IOSET** register to set the port bit and you set a bit in the **IOCLR** register to clear the port bit. This little nuance will trip up novice and experienced programmers alike. Alert readers will ask; “What if both bits are set in IOSET and IOCLR?” The answer is “Last one wins.” The last IOSET or IOCLR instruction will prevail.

To turn the LED **P0.7** off, we can write:

```
IOSET = 0x00000080;       // turn P0.7 (red LED) off
```

Likewise, to turn the LED **P0.7** on, we can write:

```
IOCLR = 0x00000080;       // turn P0.7 (red LED) on
```

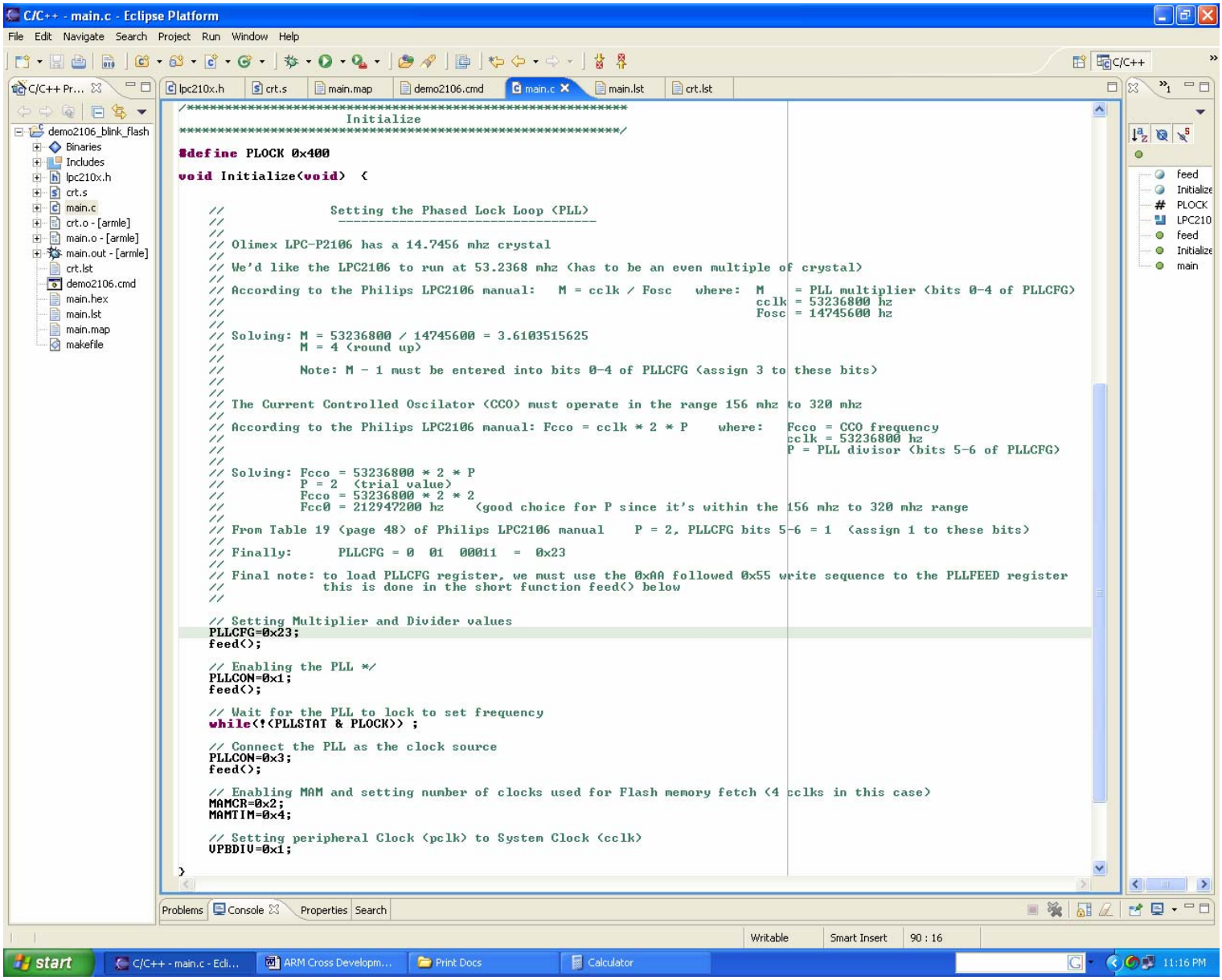
As you can see, it’s fairly simple to manipulate I/O bits on the ARM processor.

To blink the LED, a simple FOREVER loop will do the job. I selected the loop counter values to get a one half second blink on – off time.

```
// endless loop to toggle the red  LED P0.7
while (1) {

    for (j = 0; j < 5000000; j++ );          // wait 500 msec
    IOSET = 0x00000080;                      // red led off
    for (j = 0; j < 5000000; j++ );          // wait 500 msec
    IOCLR = 0x00000080;                      // red led on
}
```

This scheme is very inefficient in that it hog-ties the CPU while the wait loops are counting up.

The **Initialize();** function requires some explanation.



We have to set up the Phased Lock Loop (PLL) and that takes some math.

Olimex LPC-P2106 board has a 14.7456 Mhz crystal

We'd like the LPC2106 to run at 53.2368 Mhz (has to be an even multiple of crystal, in this case 3x)

According to the Philips LPC2106 manual: M = cclk / Fosc where: M = PLL multiplier (bits 0-4 of PLLCFG)

cclk = 53236800 hz
Fosc = 14745600 hz

Solving: M = 53236800 / 14745600 = 3.6103515625
M = 4 (round up)

Note: M - 1 must be entered into bits 0-4 of PLLCFG (assign 3 to these bits)

The Current Controlled Oscillator (CCO) must operate in the range 156 Mhz to 320 Mhz

According to the Philips LPC2106 manual: Fcco = cclk * 2 * P where: Fcco = CCO frequency

cclk = 53236800 hz

P = PLL divisor (bits 5-6 of PLLCFG)

Solving: Fcco = 53236800 * 2 * P
P = 2 (trial value)
Fcco = 53236800 * 2 * 2
Fcc0 = 212947200 hz (good choice for P since it's within the 156 mhz to 320 mhz range

From Table 19 (page 48) of Philips LPC2106 manual P = 2, PLLCFG bits 5-6 = 1 (assign 1 to these bits)

Finally: PLLCFG = 0 01 00011 = 0x23

Final note: to load PLLCFG register, we must use the 0xAA followed 0x55 write sequence to the PLLFEED register
this is done in the short function feed() below

With the math completed, we can set the Phase Locked Loop Configuration Register (**PLLCFG**)

```
// Setting Multiplier and Divider values
PLLCFG = 0x23;
feed();
```

To set values into the PLLCON and PLLCFG registers, you have to write a two-byte sequence to the PLLFEED register:

```
PLLFEED = 0xAA;
PLLFEED = 0x55;
```

This sequence is coded in a short function **feed();**
The net effect of the above setup is to run the ARM CPU at 53.2 Mhz.

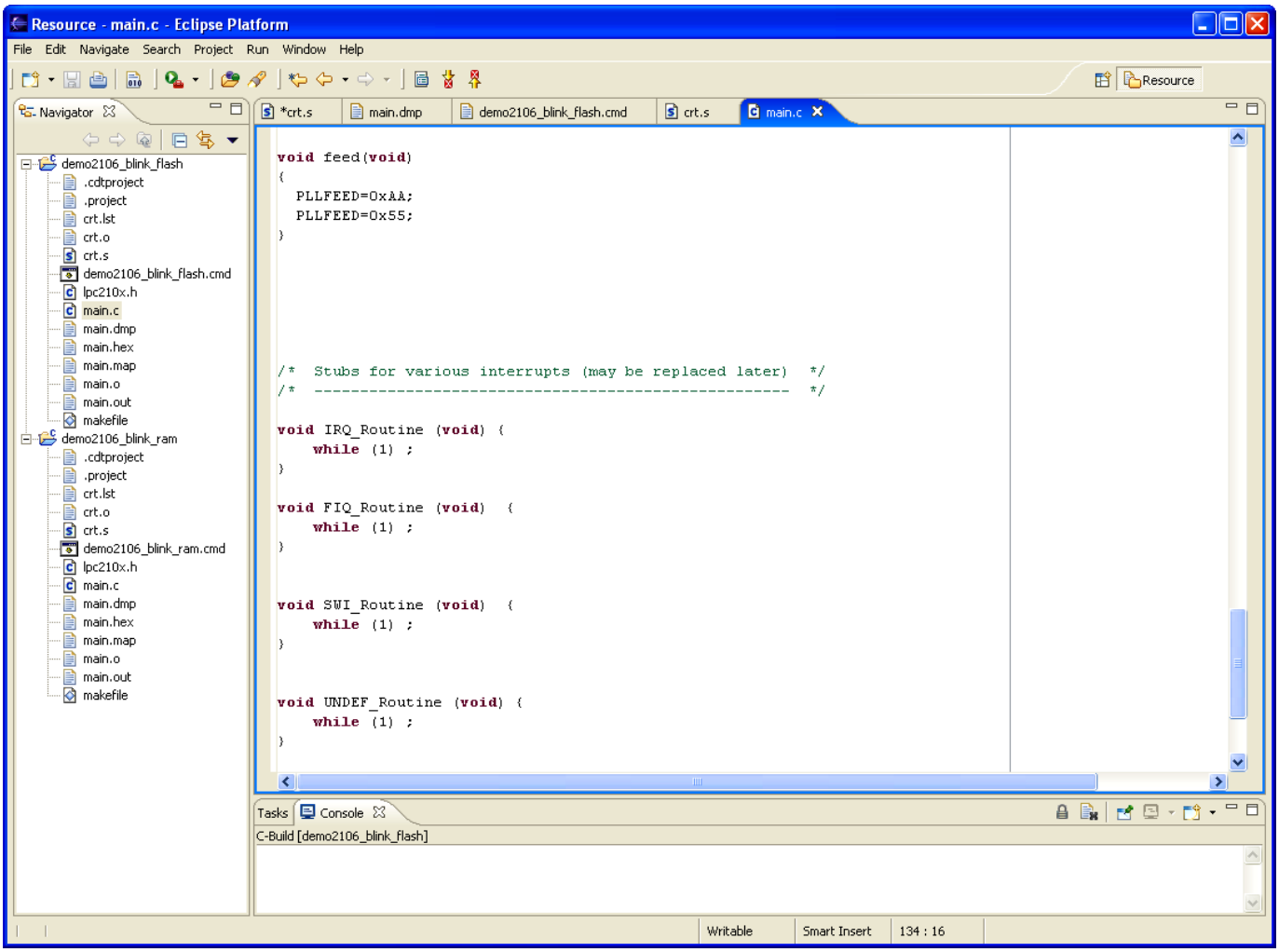
Next we fully enable the Memory Accelerator module and set the Flash memory to run at ¼ the clock speed. Now you see why some people prefer to execute out of RAM where it's much faster.

```
//  Enabling MAM and setting number of clocks used for Flash memory fetch
//  (4 cclks in this case)
MAMCR=0x2;
MAMTIM=0x4;
```

The clock speed of the peripherals is also run at 53.2 Mhz which is the full clock speed.

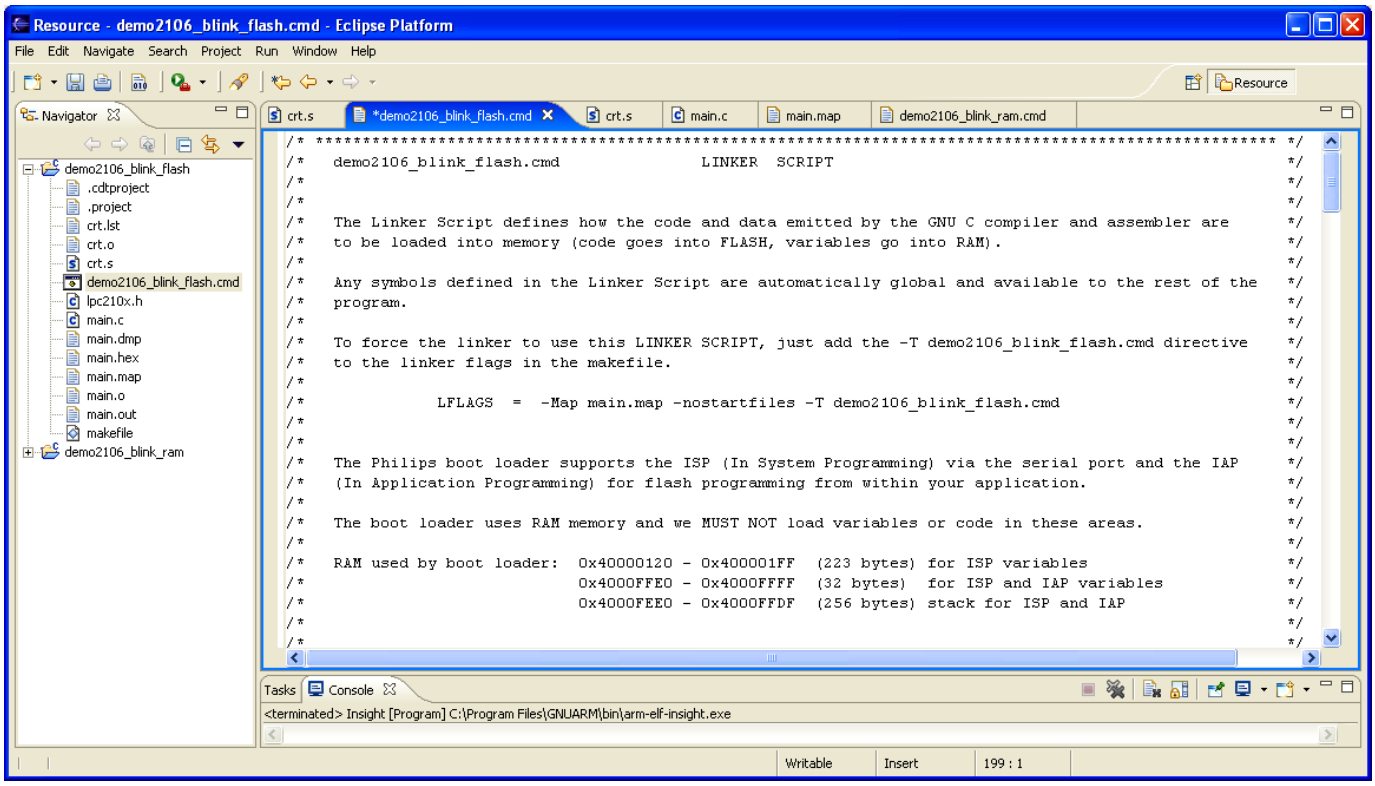
```
// Setting peripheral Clock (pclk) to System Clock (cclk)
VPBDIV=0x1;
```

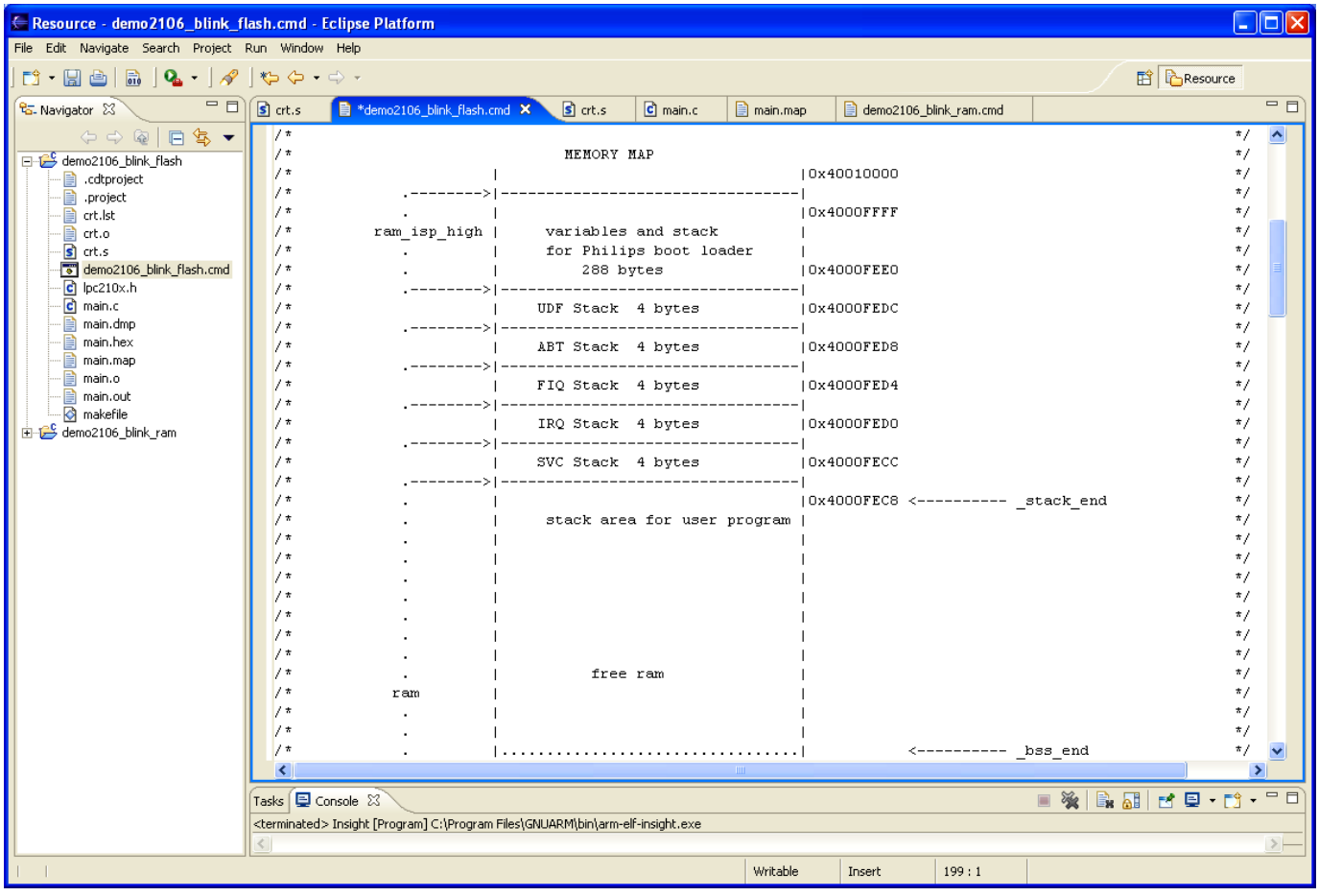
In the final snippet of the main() code, you can see the dummy interrupt service routines. They are just simple endless loops; we don't intent to allow interrupts in this simple example.

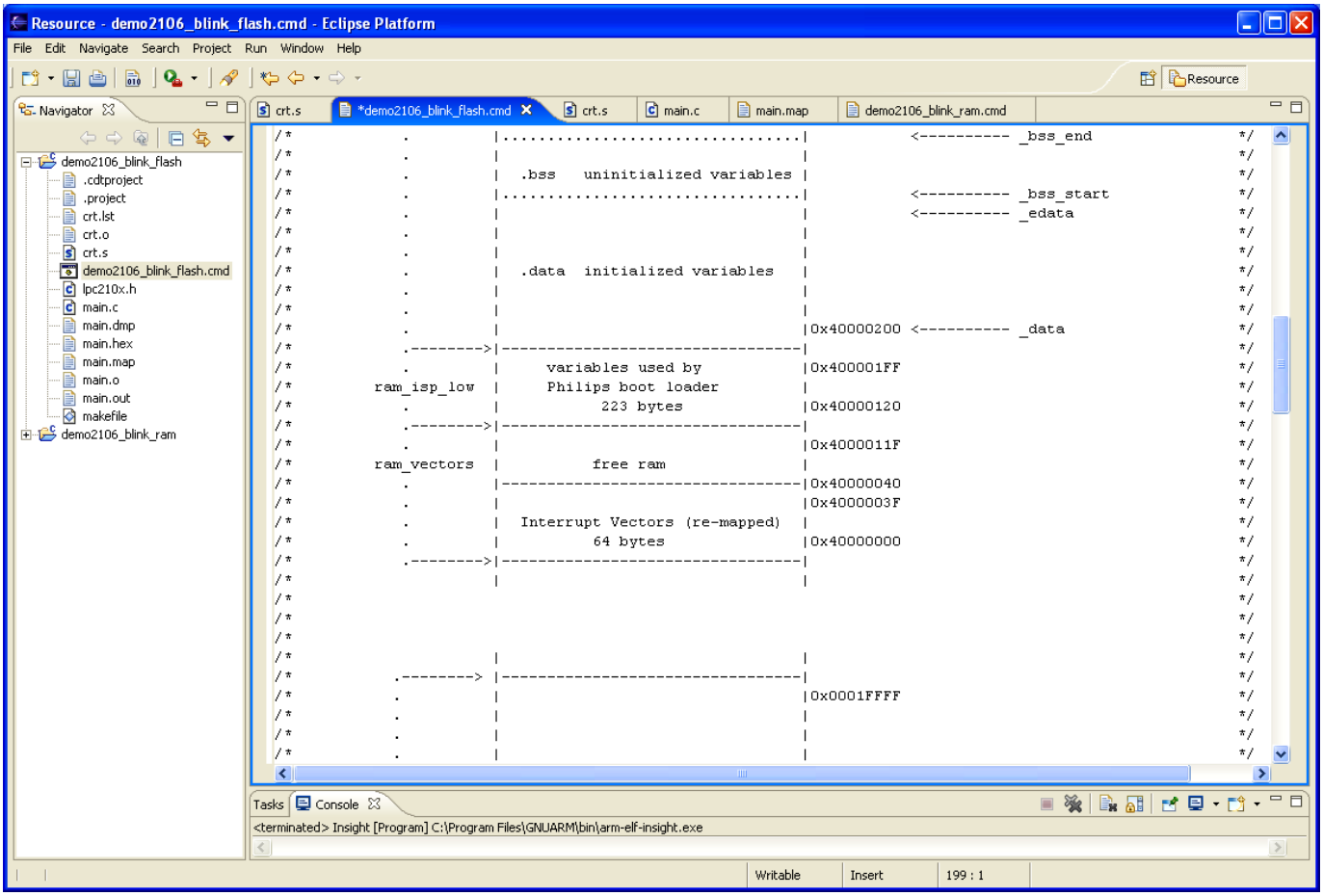


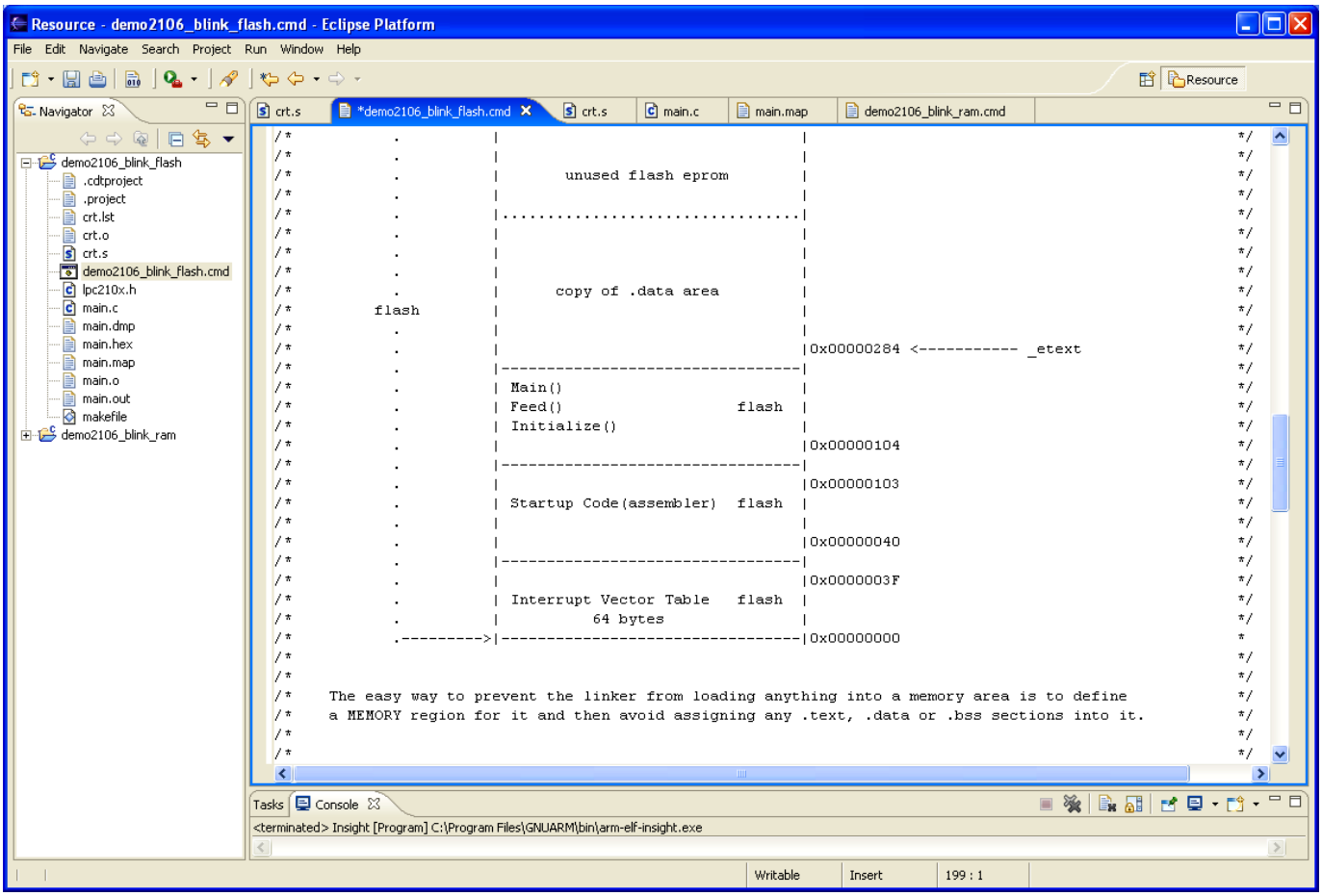
16 Description of the Linker Script demo2106_blink_flash.cmd

Let's look now at the linker command script, **demo2106_blink_flash.cmd**. I've included extensive annotation to make it very clear how the memory is organized.

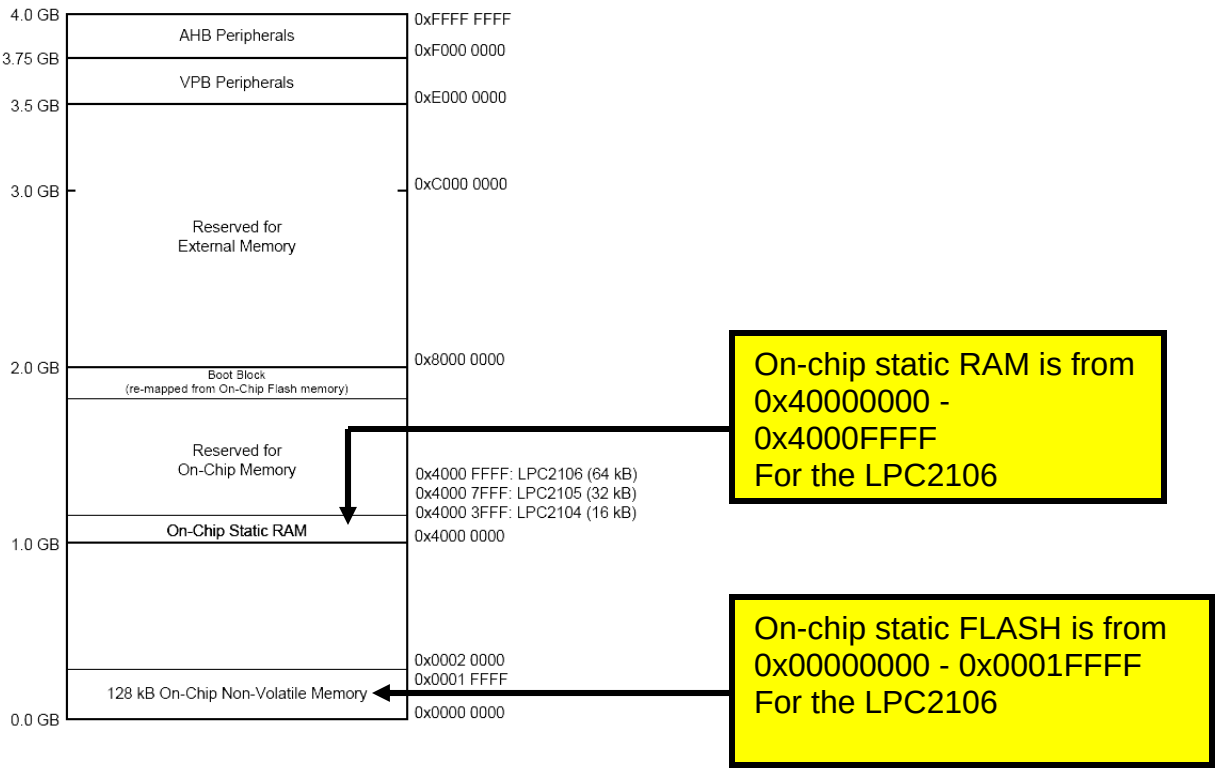


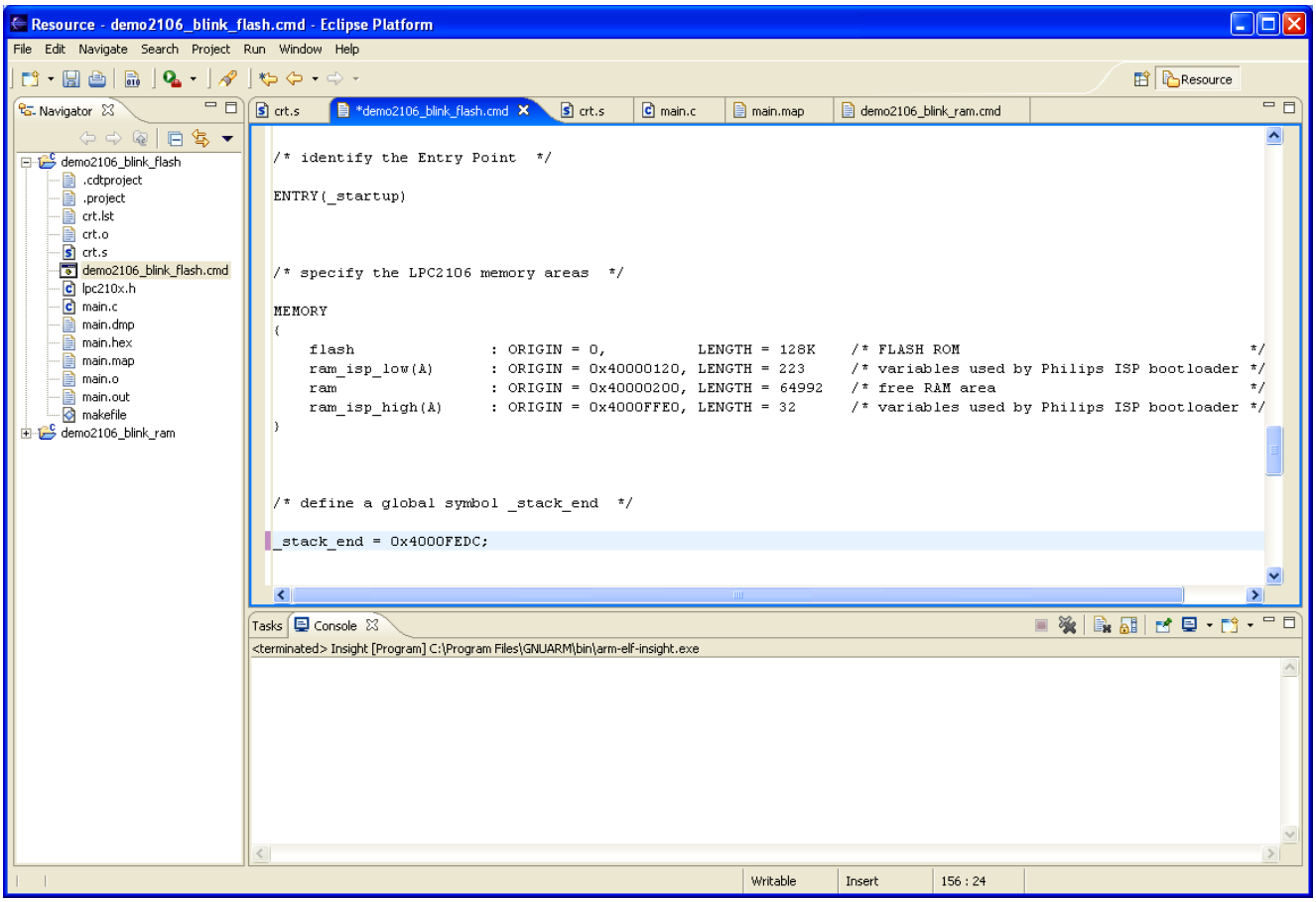



Resource - demo2106_blink_flash.cmd - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
demo2106_blink_flash
.cdtproject
.project
crt.lst
crt.o
crt.s
demo2106_blink_flash.cmd
lpc210x.h
main.c
main.dmp
main.hex
main.map
main.o
main.out
makefile
demo2106_blink_ram
crt.s
*demo2106_blink_flash.cmd
crt.s
main.c
main.map
demo2106_blink_ram.cmd
/* . |.....| <---------- _bss_end */
/* . | | */
/* . | .bss uninitialized variables | */
/* . |.....| <---------- _bss_start */
/* . | | <---------- _edata */
/* . | | */
/* . | | */
/* . | .data initialized variables | */
/* . | | */
/* . | | */
/* . | |0x40000200 <---------- _data */
/* . .--------->|---------------------------------| */
/* . | variables used by |0x400001FF */
/* ram_isp_low | Philips boot loader | */
/* . | 223 bytes |0x40000120 */
/* . .--------->|---------------------------------| */
/* . | |0x4000011F */
/* ram_vectors | free ram | */
/* . |---------------------------------|0x40000040 */
/* . | |0x4000003F */
/* . | Interrupt Vectors (re-mapped) | */
/* . | 64 bytes |0x40000000 */
/* . .--------->|---------------------------------| */
/* . | | */
/* */
/* */
/* */
/* . | | */
/* . .--------> |---------------------------------| */
/* . | |0x0001FFFF */
/* . | | */
/* . | | */
/* . | | */
Tasks Console
<terminated> Insight [Program] C:\Program Files\GNUARM\bin\arm-elf-insight.exe
Writable Insert 199 : 1


Resource - demo2106_blink_flash.cmd - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
demo2106_blink_flash
.cdtproject
.project
crt.lst
crt.o
crt.s
demo2106_blink_flash.cmd
lpc210x.h
main.c
main.dmp
main.hex
main.map
main.o
main.out
makefile
demo2106_blink_ram
crt.s
*demo2106_blink_flash.cmd
crt.s
main.c
main.map
demo2106_blink_ram.cmd
/* . | | */
/* . | | */
/* . | unused flash eprom | */
/* . | | */
/* . |.....| */
/* . | | */
/* . | | */
/* . | | */
/* . | copy of .data area | */
/* flash | | */
/* . | | */
/* . | |0x00000284 <---------- _etext */
/* . |---------------------------------| */
/* . | Main() | */
/* . | Feed() flash | */
/* . | Initialize() | */
/* . | |0x00000104 */
/* . |---------------------------------| */
/* . | |0x00000103 */
/* . | Startup Code(assembler) flash | */
/* . | | */
/* . | |0x00000040 */
/* . |---------------------------------| */
/* . | |0x0000003F */
/* . | Interrupt Vector Table flash | */
/* . | 64 bytes | */
/* . .---------->|---------------------------------|0x00000000 */
/* */
/* */
/* The easy way to prevent the linker from loading anything into a memory area is to define */
/* a MEMORY region for it and then avoid assigning any .text, .data or .bss sections into it. */
/* */
/* */
Tasks Console
<terminated> Insight [Program] C:\Program Files\GNUARM\bin\arm-elf-insight.exe
Writable Insert 199 : 1

The first order of business in the linker command script is to identify the memory available, this is easy in a Philips LPC2106 – the RAM and FLASH memory are on-chip and at fixed locations. Page 29 of the Philips LPC2106 User Manual shows the physical memory layout.





First we define an entry point; specifically _startup as defined in the assembler function **crt.s**.

```
ENTRY(_startup)
```

The Linker command script uses the following directives to lay out the physical memory.

```
MEMORY
{
        flash             : ORIGIN = 0, LENGTH = 128K                /* FLASH ROM  */
        ram_isp_low(A)    : ORIGIN = 0x40000120, LENGTH = 223        /* variables used by Philips
ISP */
        ram               : ORIGIN = 0x40000200, LENGTH = 64992      /* free RAM area        */
        ram_isp_high(A) : ORIGIN = 0x4000FFE0, LENGTH = 32           /* variables used by Philips
ISP  */
}
```

You might expect that we'd define only a flash and a ram memory area. In addition to those, we've added two dummy memory areas that will prevent the linker from loading code or variables into the RAM areas used by the Philips ISP Flash Utility (sometimes called a boot loader). See page 180 in the Philips LPC2106 User Manual for a description of the Boot Loader's RAM usage.

As you'll see in a minute, we'll be moving various sections **(.text** section, **.data** section, etc.) into flash and ram.

Note that we created a global symbol (all symbols created in the linker command script are global) called _**stack_end**. It's just located after the stack/variable area used by the Philips ISP Flash Utility (boot loader) as mentioned above.

```
_stack_end = 0x4000FEDC;
```

Now that the memory areas have been defined, we can start putting things into them. We do that by creating output sections and then putting bits and pieces of our code and data into them.

We define below four output sections:

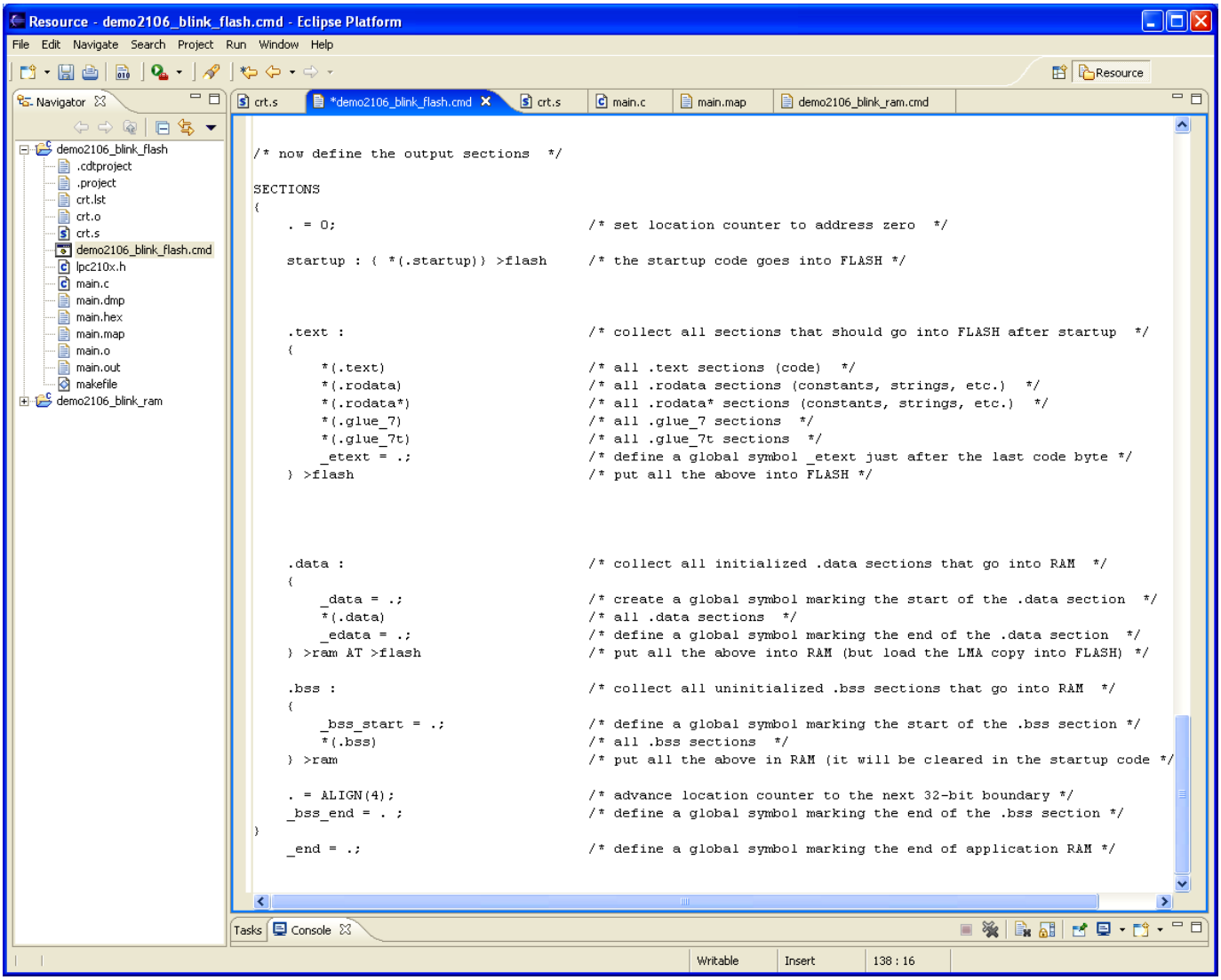
startup - this output section holds the code in the startup function, defined in **crt.s**

.text - this output section holds all other executable code generated by the compiler

.data - this output section contains all initialized data generated by the compiler

.bss - this output section contains all uninitialized data generated by the compiler

The next part of the Linker Command Script defines the sections and where they go in memory.



The first thing done within the SECTIONS command is to set the location counter.

The dot means “right here” and this sets the location counter at the beginning to 0x000000.

. = 0; /* set location counter to address zero */

Now we create our first output section, located at address 0x000000. This creates a output section named “**startup**” and it includes all sections emitted by the assembler and compiler named **.startup**. In this case, there is only one such section created in crt.s.

This startup output section is to go into FLASH at address 0x000000. Remember that the startup section has the interrupt vectors (must be placed at

0x000000) and the startup code also sets the stacks, modes and copies the **.data** and **.bss** sections.

```
startup : { *(.startup) }   >flash
```

Now we can follow the vector table and assembler startup code with all code generated by the assembler and C compiler; this code is normally emitted in **.text** sections. However, constants and strings go into sections such as **.rodata** and **.glue_7** so these are included for completeness. These code bits all go into FLASH memory.

```
        .text :         /* collect all sections that should go into FLASH after
startup  */
        {
                *(.text)                /* all .text sections (code)  */
                *(.rodata)              /* all .rodata sections (constants, strings,
etc.)  */
                *(.rodata*)             /* all .rodata* sections (constants, strings,
etc.)  */
                *(.glue_7)              /* all .glue_7 sections   */
                *(.glue_7t)             /* all .glue_7t sections  */
                _etext = .;             /* define a global symbol _etext after the last code
byte */
        } >flash                        /* put all the above into FLASH */
```

We follow the **.text:** output section (all the code and constants, etc) with a symbol definition, which is automatically global in the GNU toolset. This basically sets the next address after the last code byte to be the global symbol _**etext** (end-of-text).

There are two variable areas, **.data** and **.bss**. The initialized variables are contained in the **.data** section, which will be placed in RAM memory. The big secret here is that an exact copy of the **.data** section will be loaded into FLASH right after the code section just defined. The onus is on the programmer to copy this section to the correct address in FLASH; in this way the variables are "initialized" at startup just after a reset.

The **.bss** section has no initializers. Therefore, the onus is on the programmer to clear the entire **.bss** section in the startup routine.

Initialized variables are usually emitted by the assembler and C compiler as **.data** sections.

```
        .data :
        {
                _data = .;      // global symbol locates the start of .data section in RAM

                *(.data)        // tells linker to collect all .data sections together

                _edata = .;     // global symbol locates the end of .data section in RAM
```

```
} >ram AT>flash     // load data section into RAM, load copy of .data section
                    // into FLASH for copying during startup.
```

Note first that we created two global symbols, _**data** and _**edata**, that locate the beginning and end of the **.data** section in RAM. This helps us create a copy loop in the **crt.s** assembler file to load the initial values into the **.data** section in RAM.

The command **>ram** specifies the Virtual Memory Address that the .data section is to be placed into RAM (think of it as the final destination in RAM and all code references to any variables will use the RAM address.

The command **AT >flash** specifies the load memory address; essentially an exact copy of the RAM memory area with every variable initialized placed in flash for copying at startup.

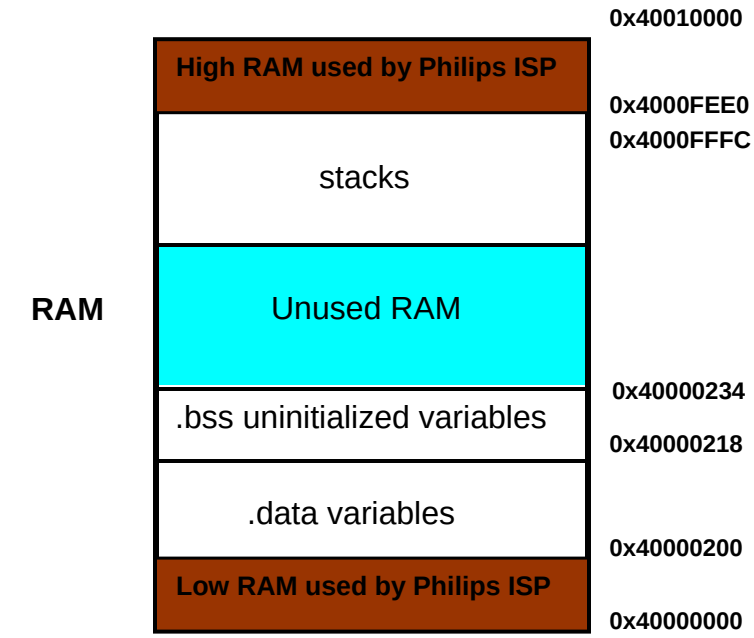
You might say “why not let the Philips boot loader load the initial values of the **.data** section in RAM directly from the hex file?” The answer is that would work once and only once. When you power off and reboot your embedded application, the RAM values are lost.

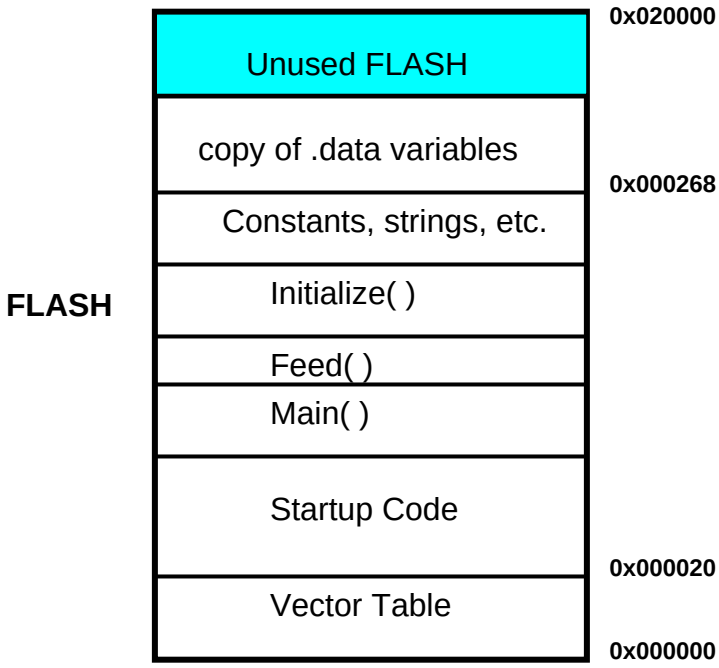
The copy of the **.data** area loaded into flash for copying during startup is placed by the GNU linker at the next available flash location. This is conveniently right after the last byte of the **.prog** section containing all our executable code.

The **.bss** section is all variables that are not initialized. It is loaded into RAM and we create two global symbols _**bss_start** and _**bss_end** to locate the beginning and end for clearing by a loop in the startup code.

```
        .bss :
        {
                _bss_start = .;
                *(.bss)
        } >ram

        . = ALIGN(4);
}
        _bss_end = . ;
        _end = .;
```

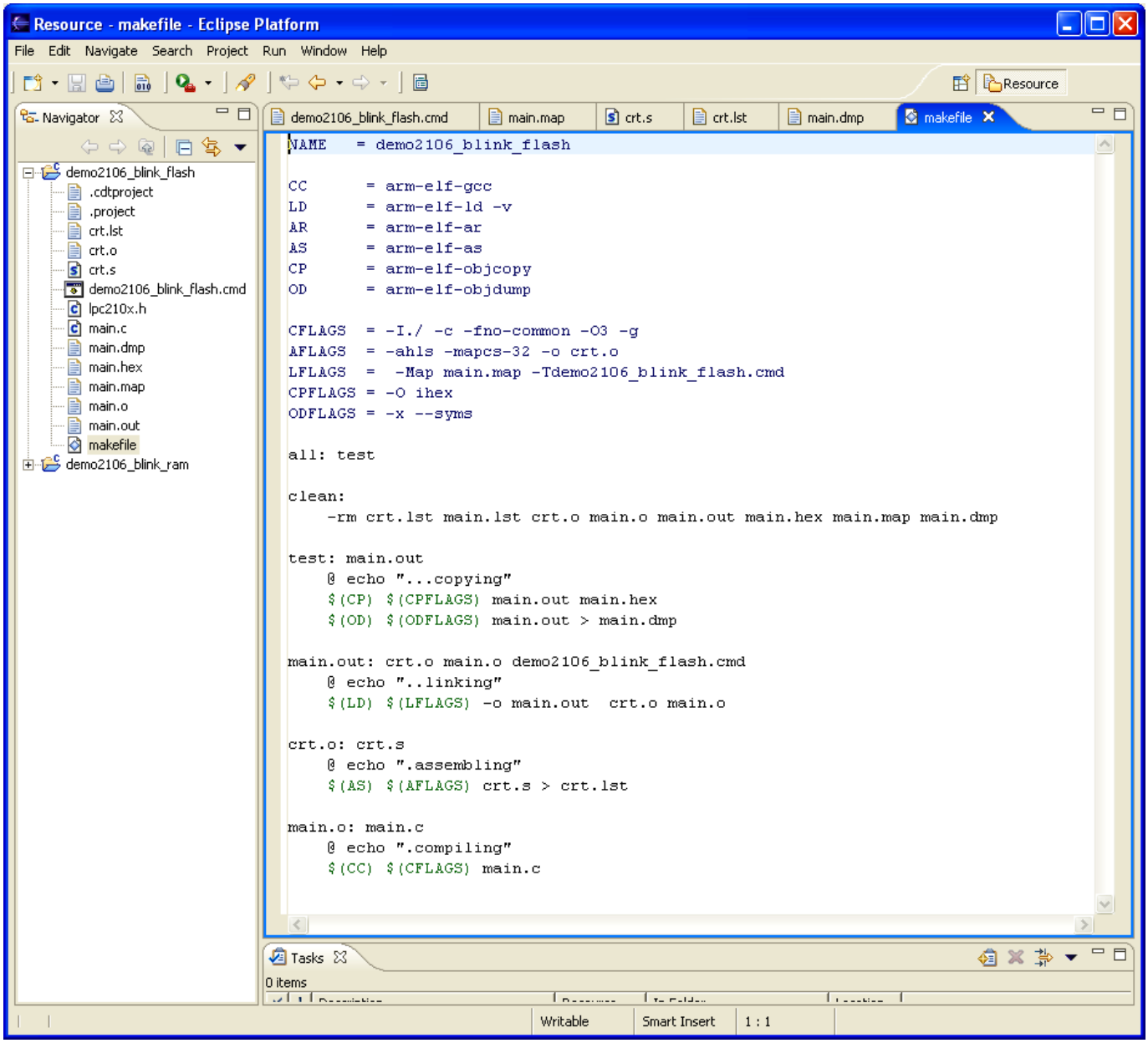
Now let's diagram just where everything is in RAM and FLASH memory.





17 Description of the Makefile

The makefile is the last source file we need to look at. I built the makefile to comply with the GNU make utility and be as simple as possible.



The general idea of the makefile is that a **target** (could be a file) is associated with one or more dependent files. If any of the dependent files are newer than the target, then the **commands** on the following lines are executed (to recompile, for instance). Command lines are indented with a **Tab** character!

main.o: main.c
arm-elf-gcc -I./ -c -O3 -g main.c

In the example above, if main.c is newer than the target main.o, the command or commands on the next line or lines will be executed. The command arm-elf-gcc will recompile the file main.c with several compilation options specified. If the target is up-to-date, nothing is done. Make works its way downward in the makefile, if you've deleted all object and output files, it will compile and link everything.

GNU make has a helpful "**variables**" feature that helps you reduce typing. If you define the following variable:

CFLAGS = -I./ -c -fno-common -O3 -g

You can use this multiple times in the makefile by writing the variable name as follows:

$(CFLAGS) will substitute the string **-I./ -c -O3 -g**

Therefore, the command-

arm-elf-gcc $(CFLAGS) main.c

is exactly the same as

arm-elf-gcc -I./ -c -O3 -g main.c

Likewise, we can replace the compiler name **arm-elf-gcc** with a variable too.

CC = arm-elf-gcc

Now the command line becomes

$(CC) $(CFLAGS) main.c

Now our "rule" for handling the main.o and main.c files becomes:

Commands MUST be indented with a TAB character!

```
main.o: main.c
	@ echo ".compiling"
	$(CC) $(CFLAGS) main.c
```

It's worth emphasizing that forgetting to insert the **TAB** character before the commands is the most common rookie mistake in using the GNU Make system.

The compilation options being used are:

-I./ **=** specifies include directories to search first (project directory in this case)

-c = do not invoke the linker, we have a separate make rule for that

-fno-common = gets rid of a pesky warning

-O3 = sets the optimization level (Note: set to **–O0** for debugging!)

-g = generates debugging information

The assembler is used to assemble the file crt.s, as shown below:

```
crt.o: crt.s
        @ echo ".assembling"
        $(AS) $(AFLAGS) crt.s > crt.lst
```

In the example above, if the object file **crt.o** is older than the dependent assembler source file **crt.s**, then the commands on the following lines are executed.

If we expand the make variables used, the lines would be:

```
crt.o: crt.s
        @ echo ".assembling"
        arm-elf-as    -ahls -mapcs-32 -o crt.o    crt.s   > crt.lst
```

The **> crt.lst** directive creates a assembler list file.

The assembler options being used are:

-ahls = listing control, turns on high-level source, assembly and symbols

-mapcs-32 = selects 32-bit ARM function calling method

-o crt.o = create an object output file named crt.o

The GNU linker is used to prepare the output from the assembler and C compiler for loading into Flash and RAM, as shown below:

```
main.out: crt.o main.o demo2106_blink_flash.cmd
        @ echo "..linking"
        $(LD) $(LFLAGS) -o main.out  crt.o main.o
```

If the target output file **main.out** is older than the two object files or the linker command file, then the commands on the following lines are executed.

The Linker options being used are:

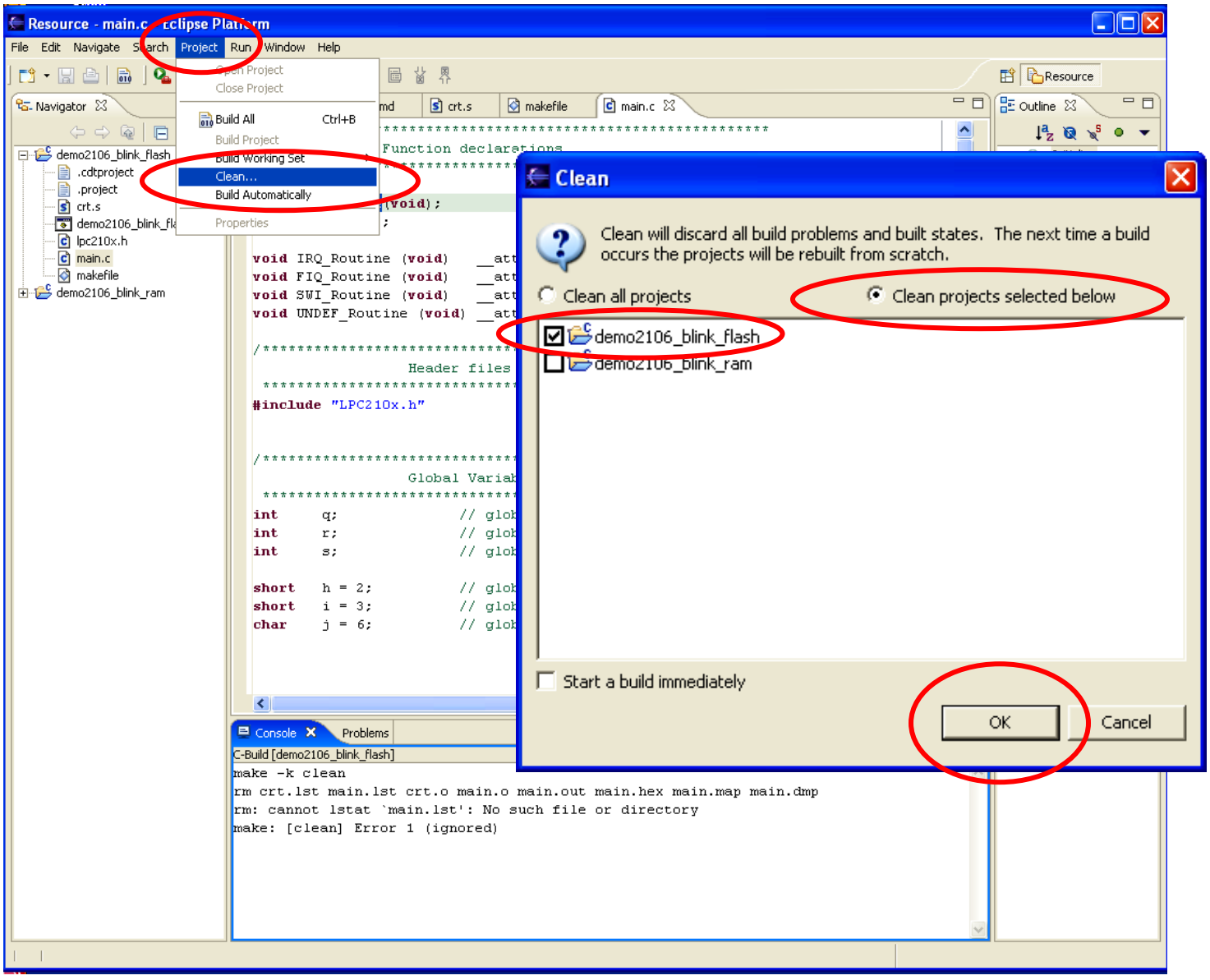
-Map main.map = creates a map file

**-T demo2106_blink_flash.cmd = ** identifies the name of the linker script file

Note that I've kept this GNU makefile as simple as possible. You can clearly see the assembler, C compiler and linker steps. They are followed by the **objcopy** utility that makes the hex file for the Philips ISP boot loader and an **objdump** operation to give a nice file of all symbols, etc.

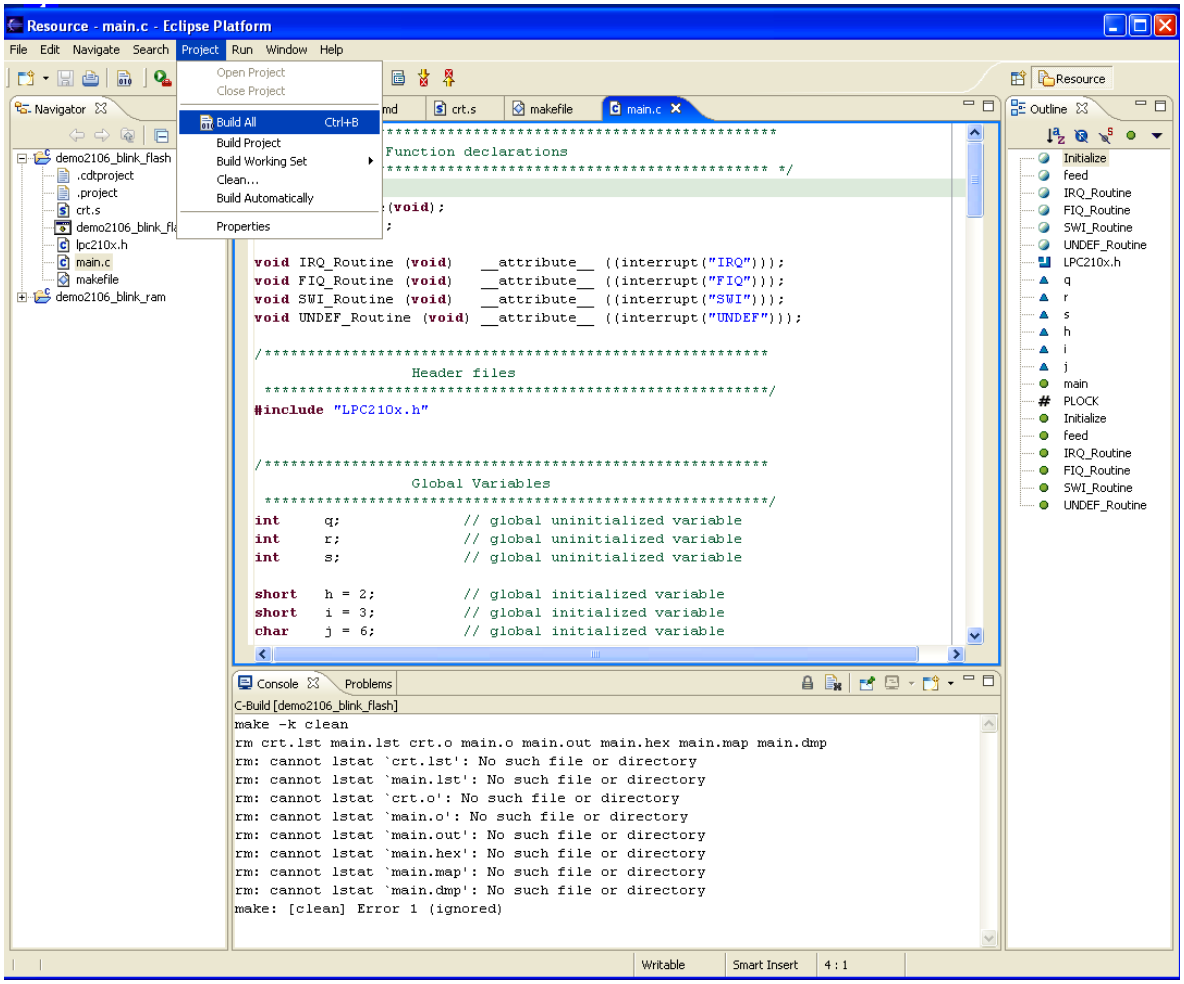
18 Compiling and Linking the Sample Application

OK, now it's time to actually do something. First, let's "**Clean**" the project; this gets rid of all object and list files, etc. Click on "**Project – Clean …**" and fill out the Clean dialog window.

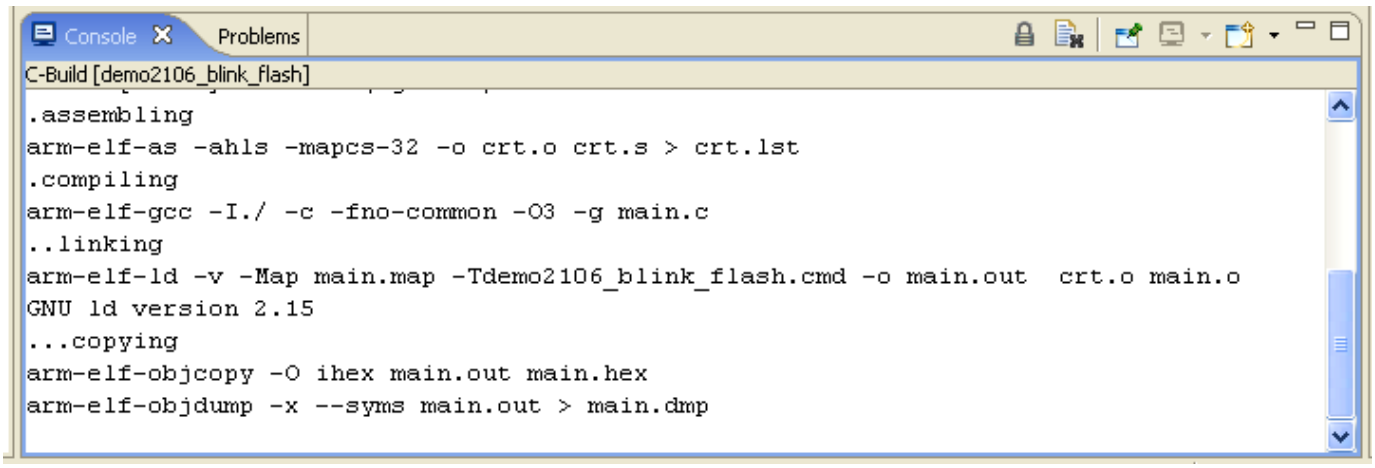


You can see the results of the “Clean” operation in the Console window at the bottom. Expect to see some warnings if there isn’t anything to delete.

To build the project, click on “Project – Build All”. Since we deleted all the object files and the main.out file via the clean operation, this “Build-all” will assemble the crt.s startup file, C compile the main.c function, run the linker and then run the **objcopy** utility to make a hex file suitable for downloading with the Philips ISP Flash Utility.

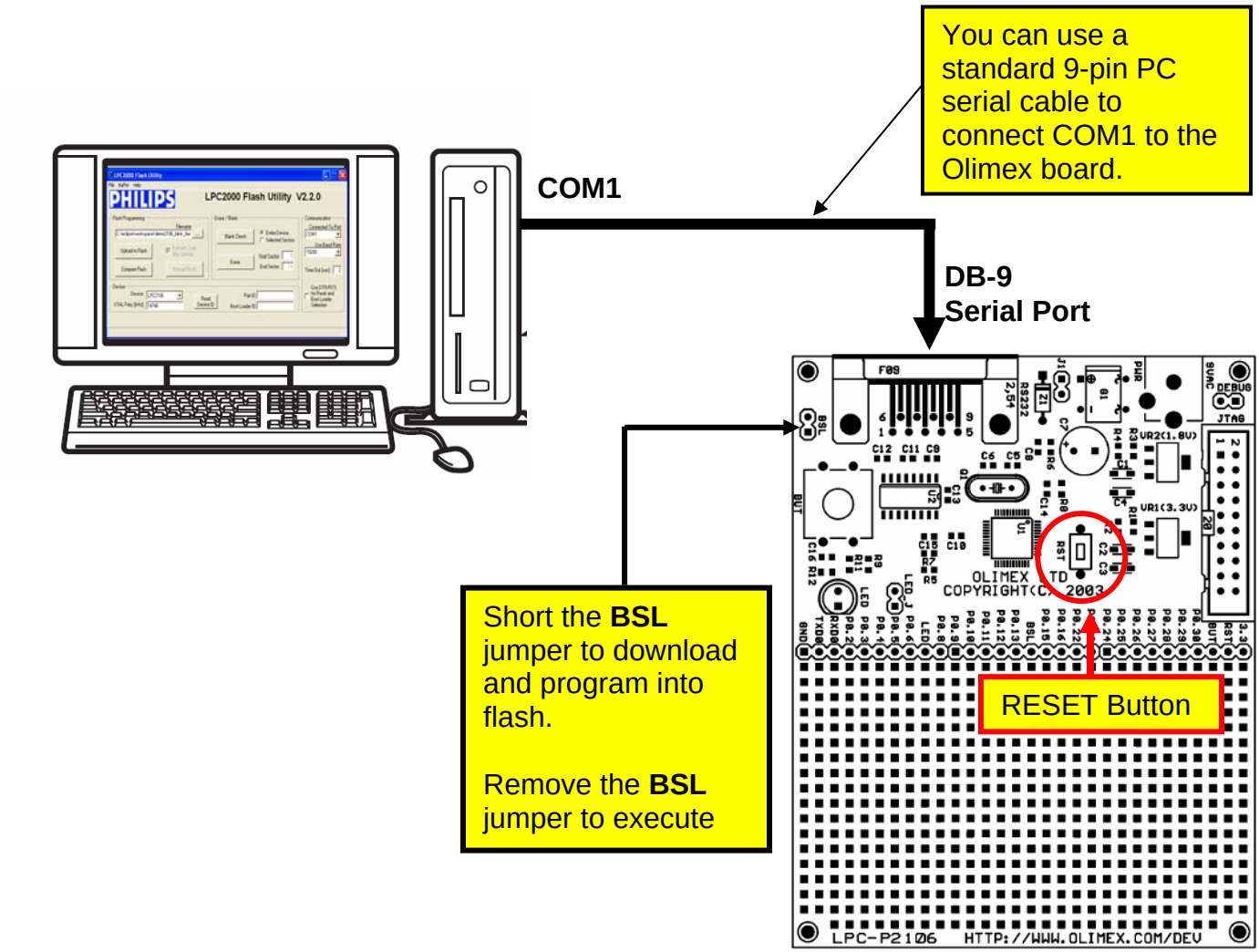


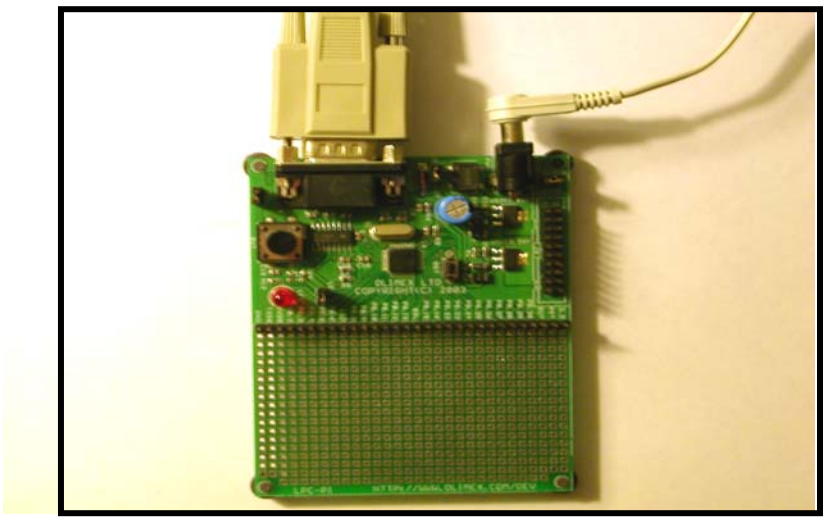
We can see the results in the Console Window at the bottom.



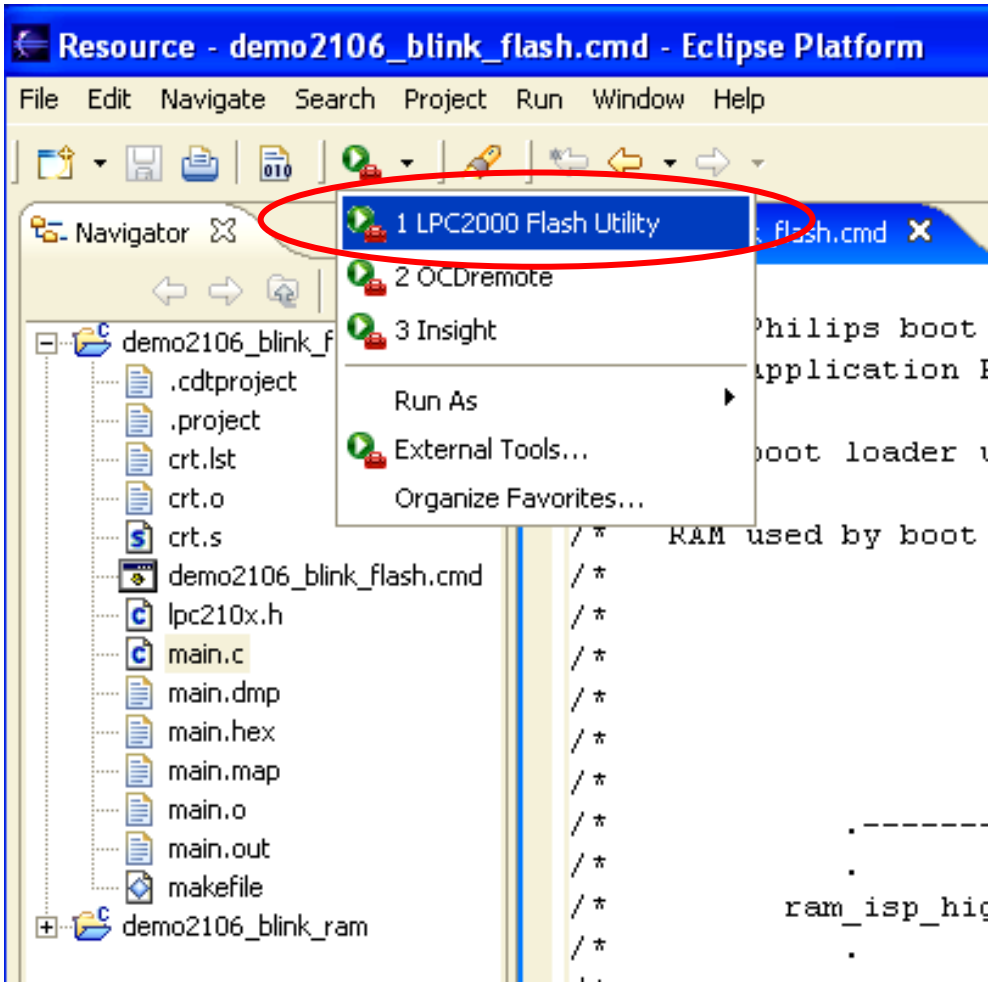
19 Setting Up the Hardware

For this tutorial, we'll be using the Olimex **LPC-P2106 Prototype Board**. Connect a straight-through 9-pin serial cable from your computer's COM1 port to the DB-9 connector on the Olimex board. Attach the 9-volt power supply to the PWR connector. Install the BSL jumper and the JTAG jumper.

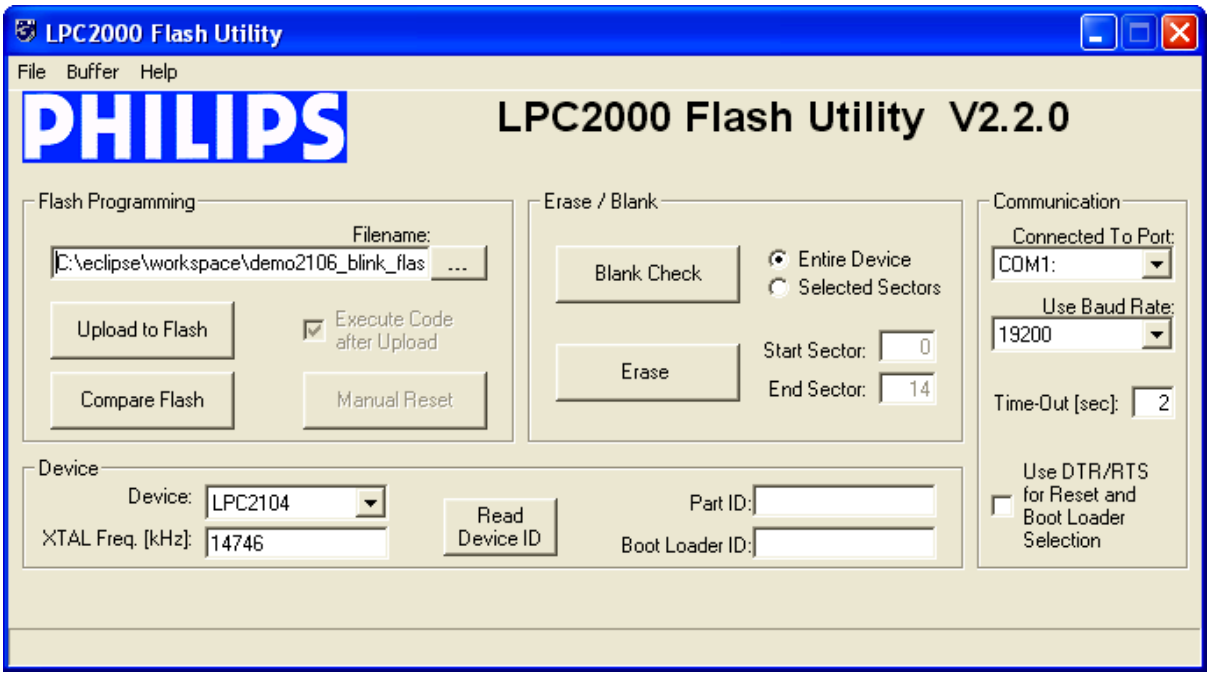




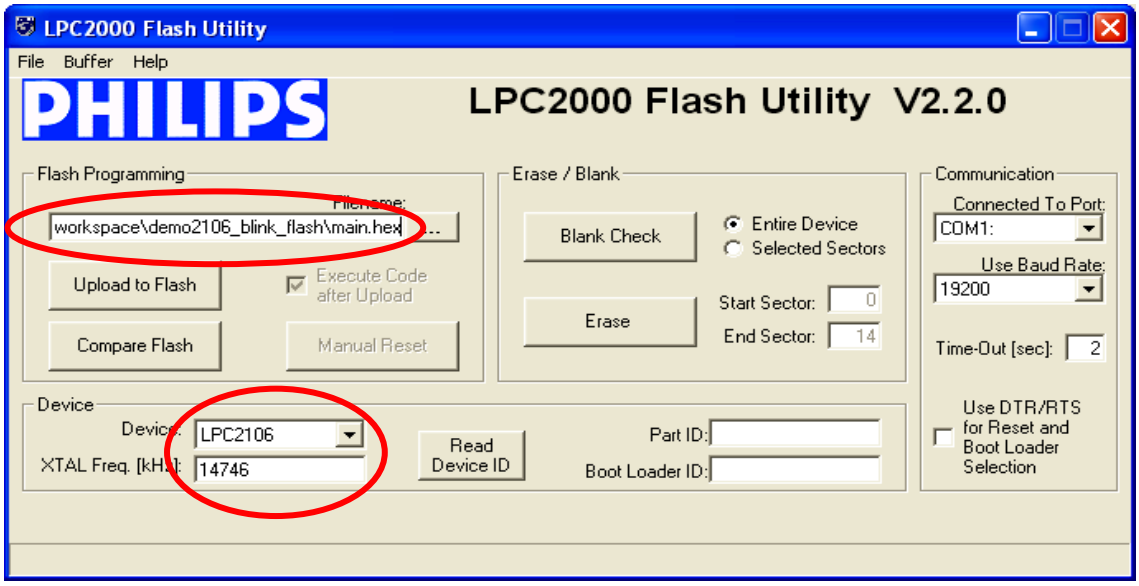
To run the Philips LPC2000 Flash Utility, it's easiest to just click on the "**External Tools**" button and its down arrow to pull-down the available tools. Click on "**LPC2000 Flash Utility**" to start the Philips Boot Loader.



The Philips LPC2000 ISP Flash Programming will start up.



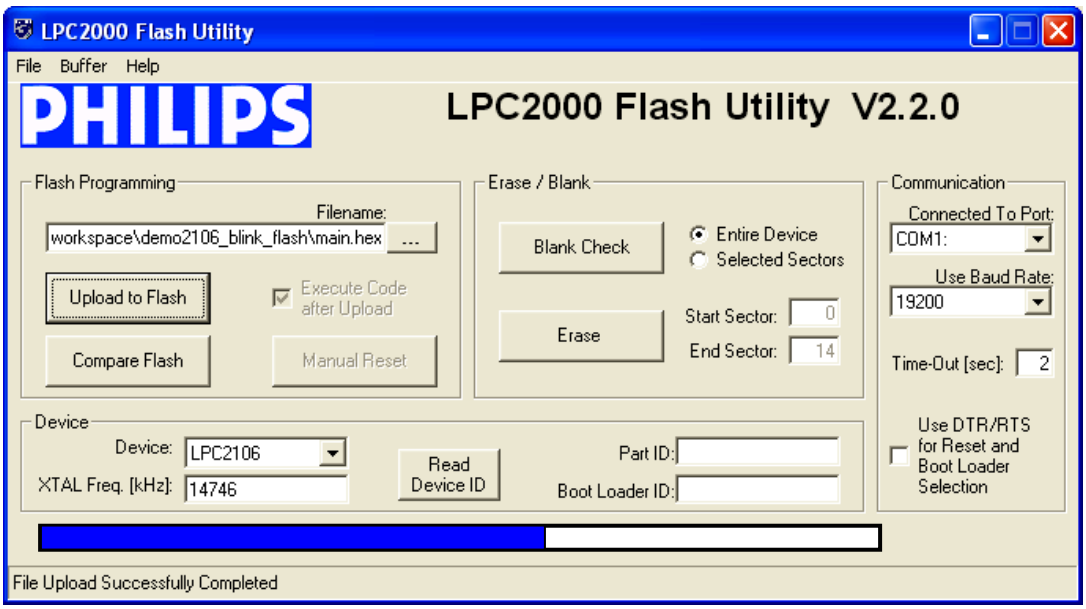
Now fill out the LPC2000 Flash Utility screen. Browse the workspace for the **main.hex** file. Set the Device to **LPC2106**. Set the crystal frequency to **14746**, as per the Olimex schematic. The default baud rate, COM port and Time-out are OK as is.



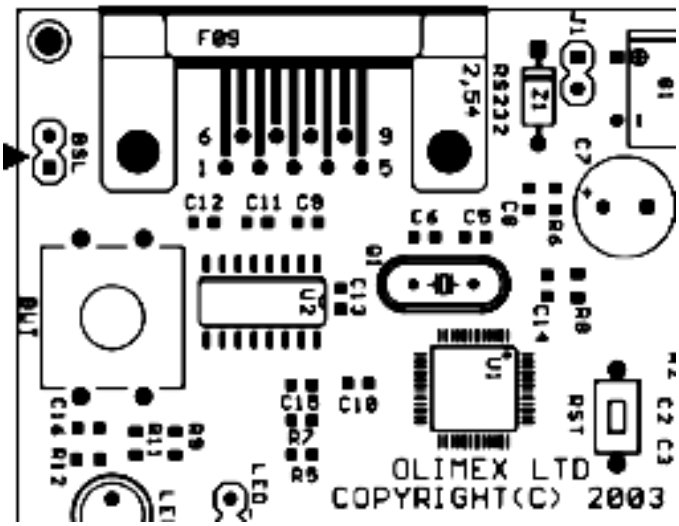
Now click on "**Upload to Flash**" to start the download.

The Philips ISP Flash Utility will now ask you to reset the target system. This is the tiny **RST** button near the CPU chip.

The download will now proceed; you'll see a blue progress bar at the bottom and then the status line will say "File Upload Successfully Completed".

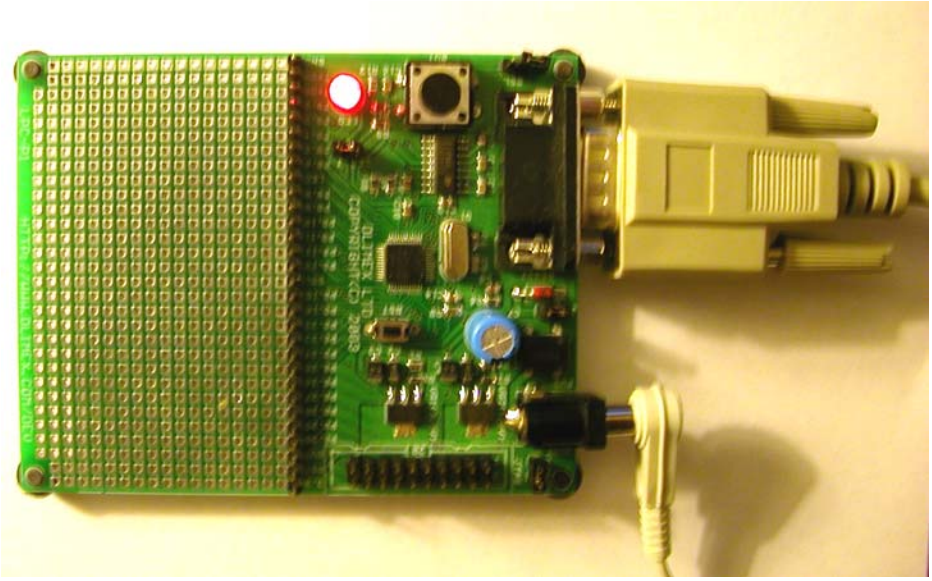


Remove the **BSL** (boot strap loader) jumper and hit the **RST** button.



Your application should start up and the LED will start blinking.

To prove that I am as honest as the sky is blue, here it is blinking away!

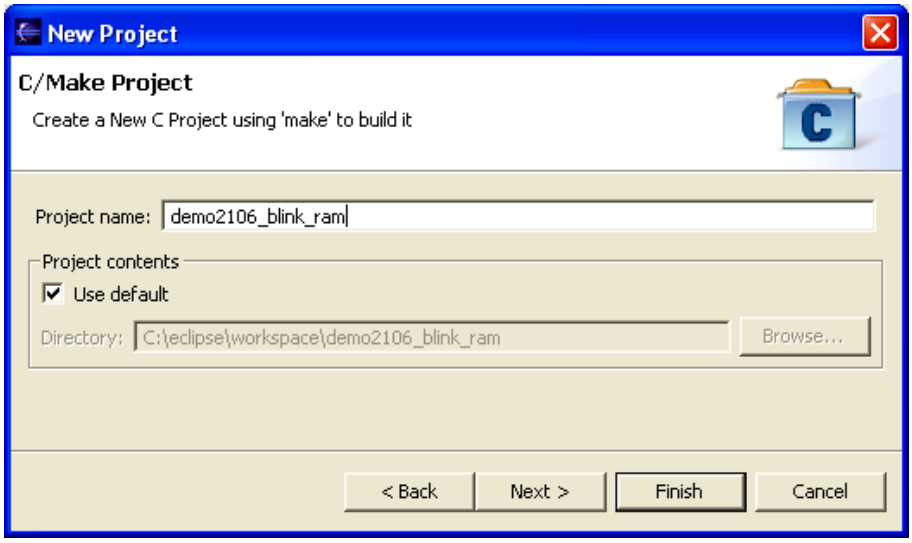


OK, I admit it; this photo has the reliability of a Bigfoot video!

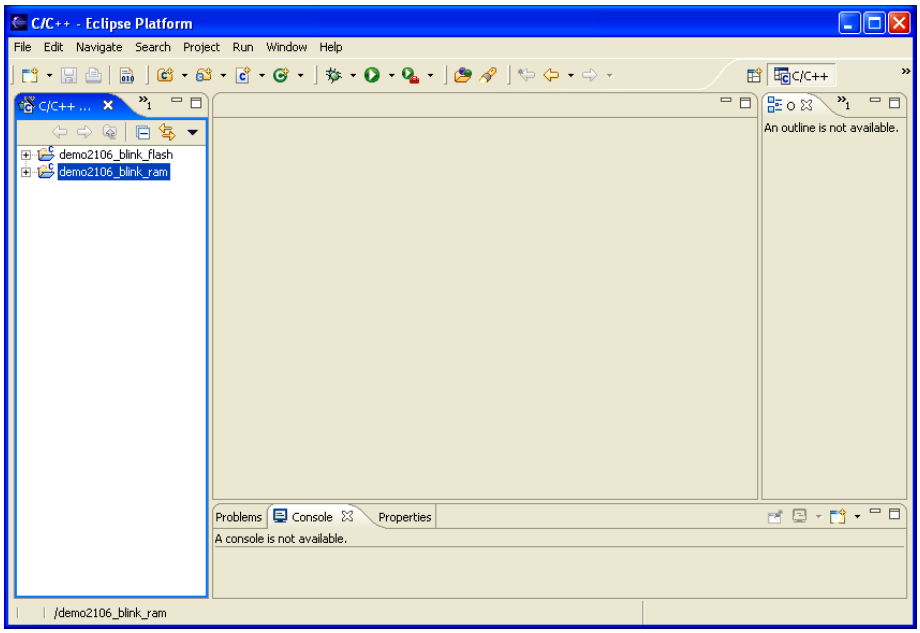
20 Create a New Project to Run the Code in RAM

Now we will create a new project that will run the blinker code in RAM. Only minor modifications to three files are required. We will show how to run the application using the Philips ISP flash utility. Later, we'll show how to use this very same RAM-based application with the Insight graphical debugger and a Wiggler JTAG interface.

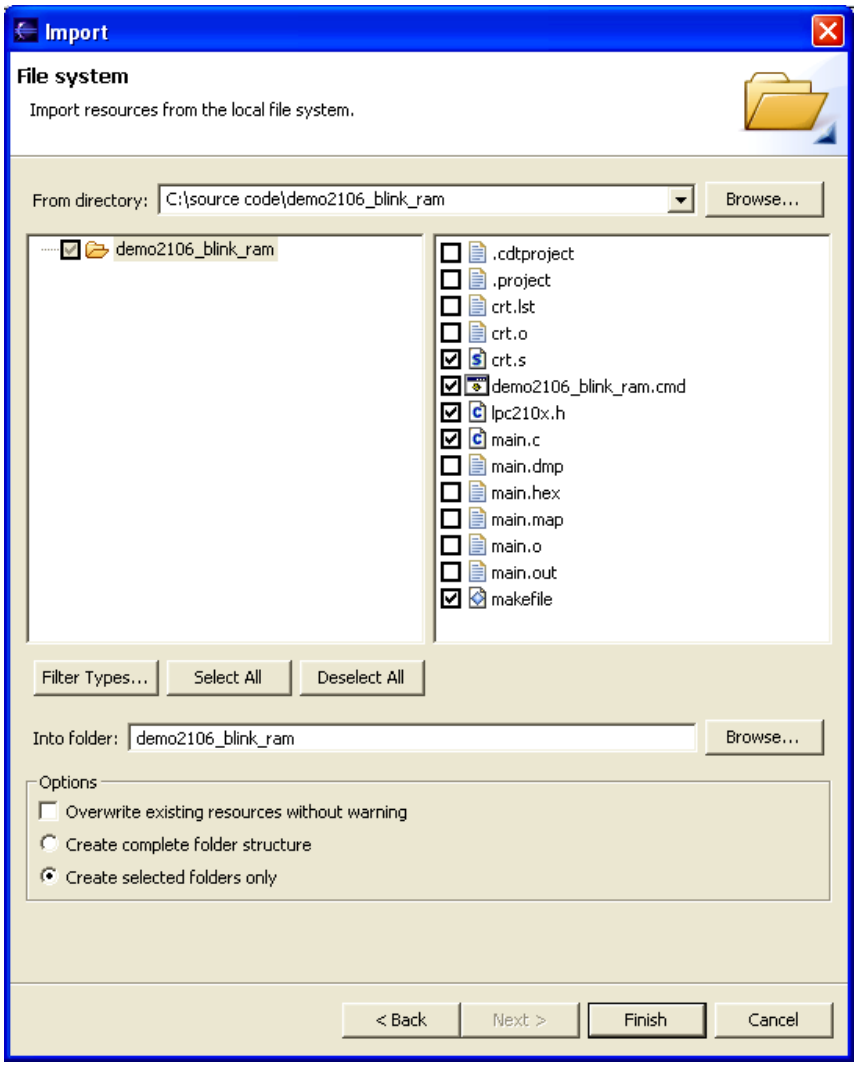
Using the techniques previously discussed, create a new project named **demo2106_blink_ram.**



Switch to the C/C++ Perspective and you will see that there are now two projects, although the new one contains no files.



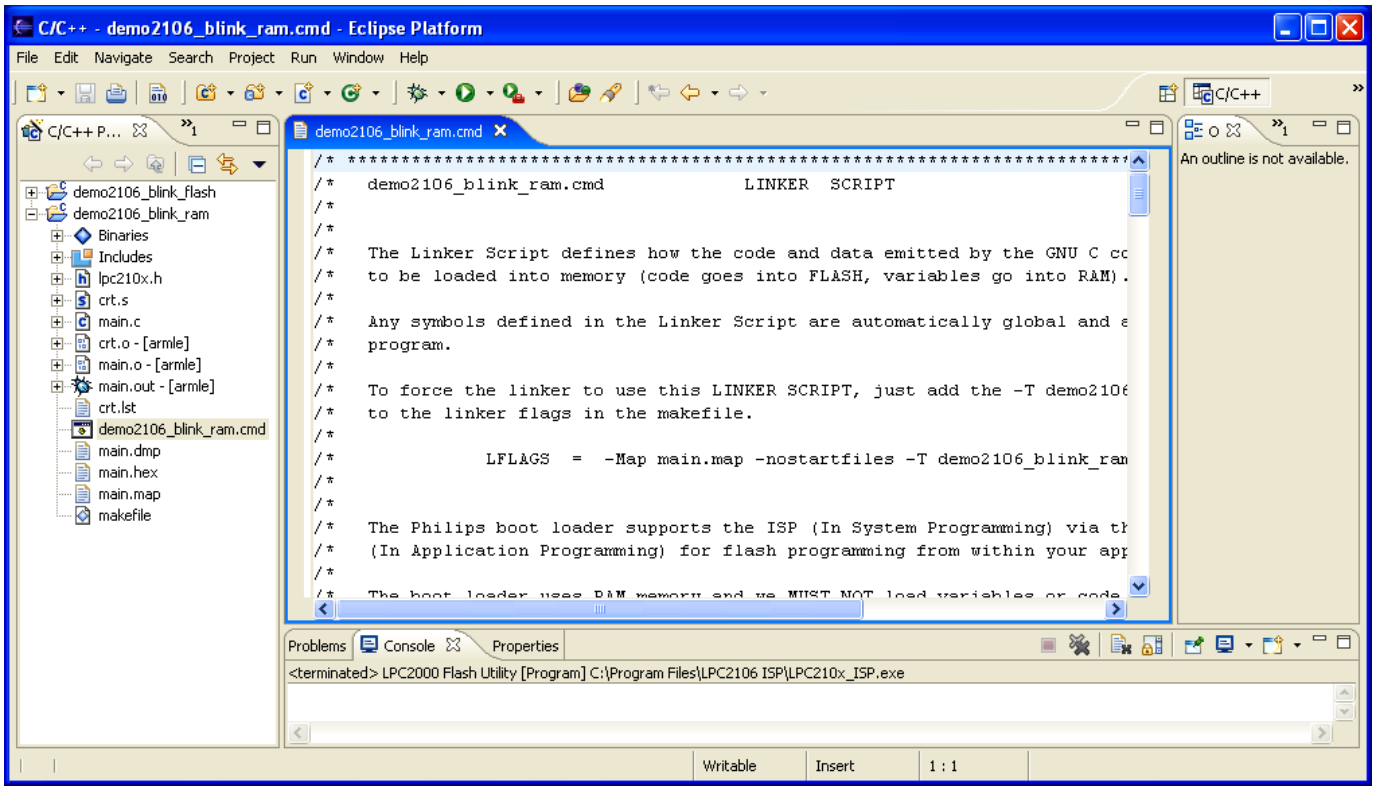
Now using the “**File Import**” procedure described earlier, fetch the source files for the project **demo2106_flash_ram** included in the zip distribution for this tutorial.



The files we import are:

crt.s
demo2106_blink_ram.cmd
lpc210x.h
main.c
makefile.mak

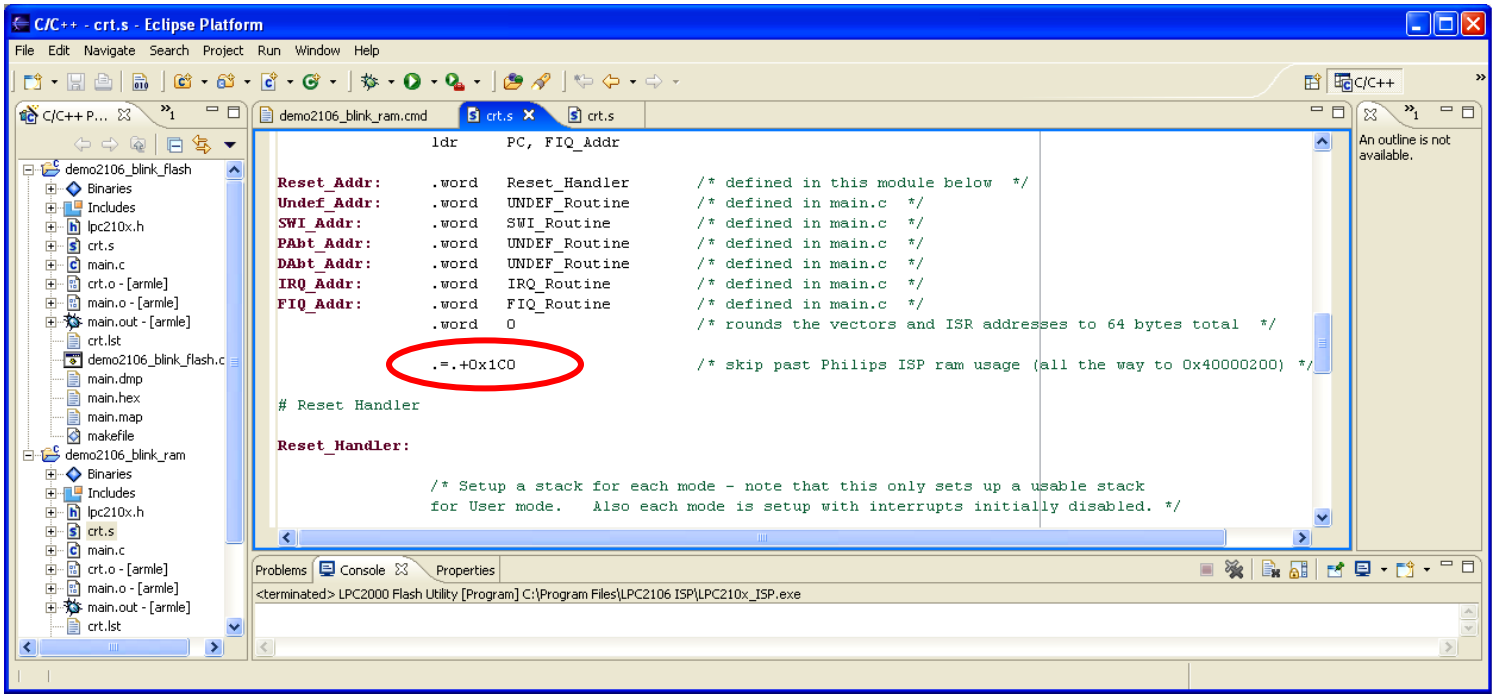
Now if you “Clean and Build” you should see a completed project with all the resultant files, as shown below.



21 Differences in the RAM Version

File CRT.S

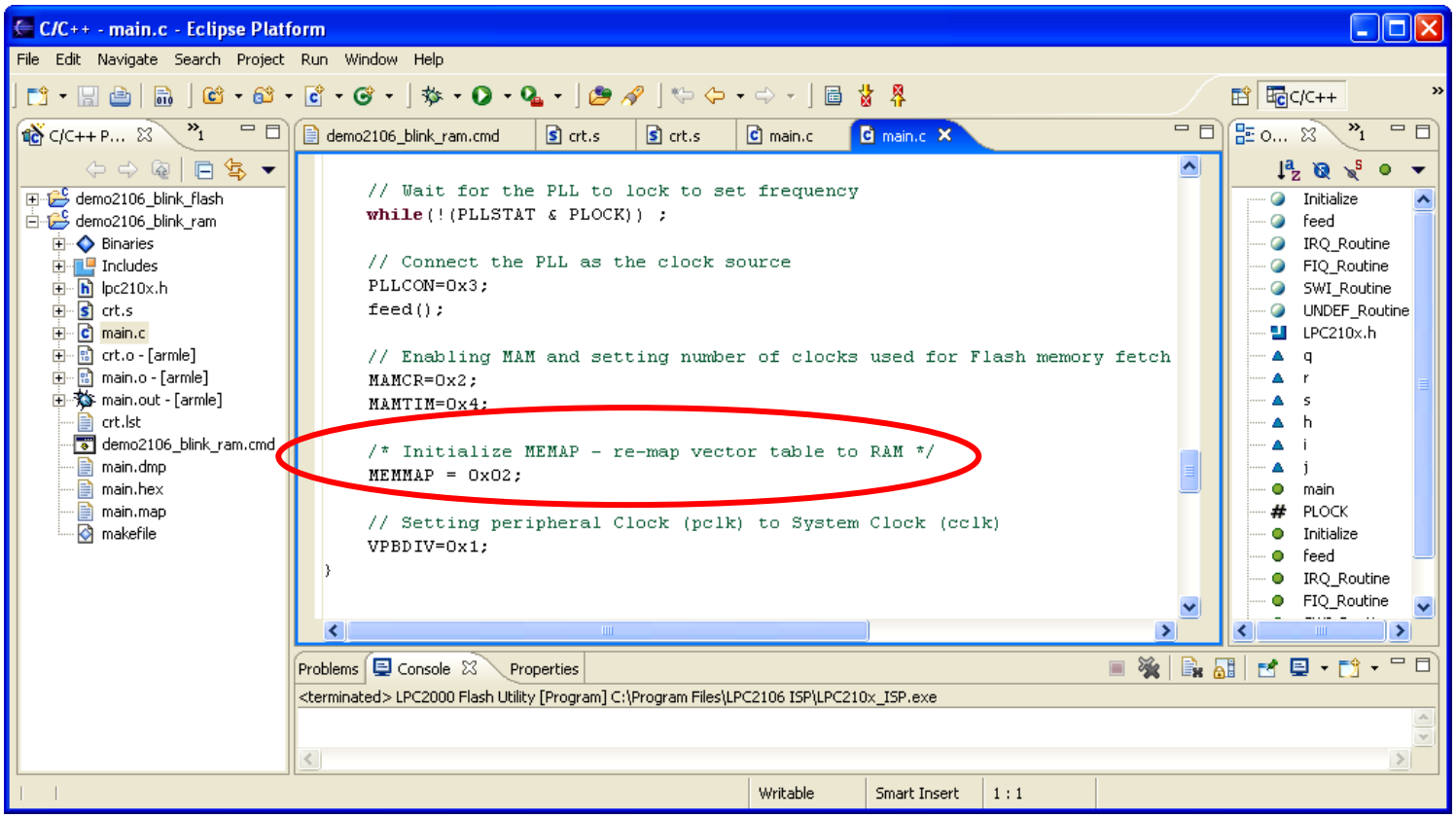
In the startup assembler file, I used a simple trick to move the startup code away from the vectors to ensure that it doesn't encroach on the Philips ISP Flash Loader low RAM area.



Remember that the entire project, code and variables, will be loaded into RAM starting at address 0x40000000. The location counter is advanced by the directive **.=.+0x1C0** to push the Reset_Handler to address 0x40000200. This leaves a hole where the Philips ISP Flash Utility will use the low RAM. There are other ways to do this.

File_MAIN.C

There is just one extra line of C code in the main program. It directs the LPC2106 to re-map the interrupt vectors to RAM at 0x40000000.



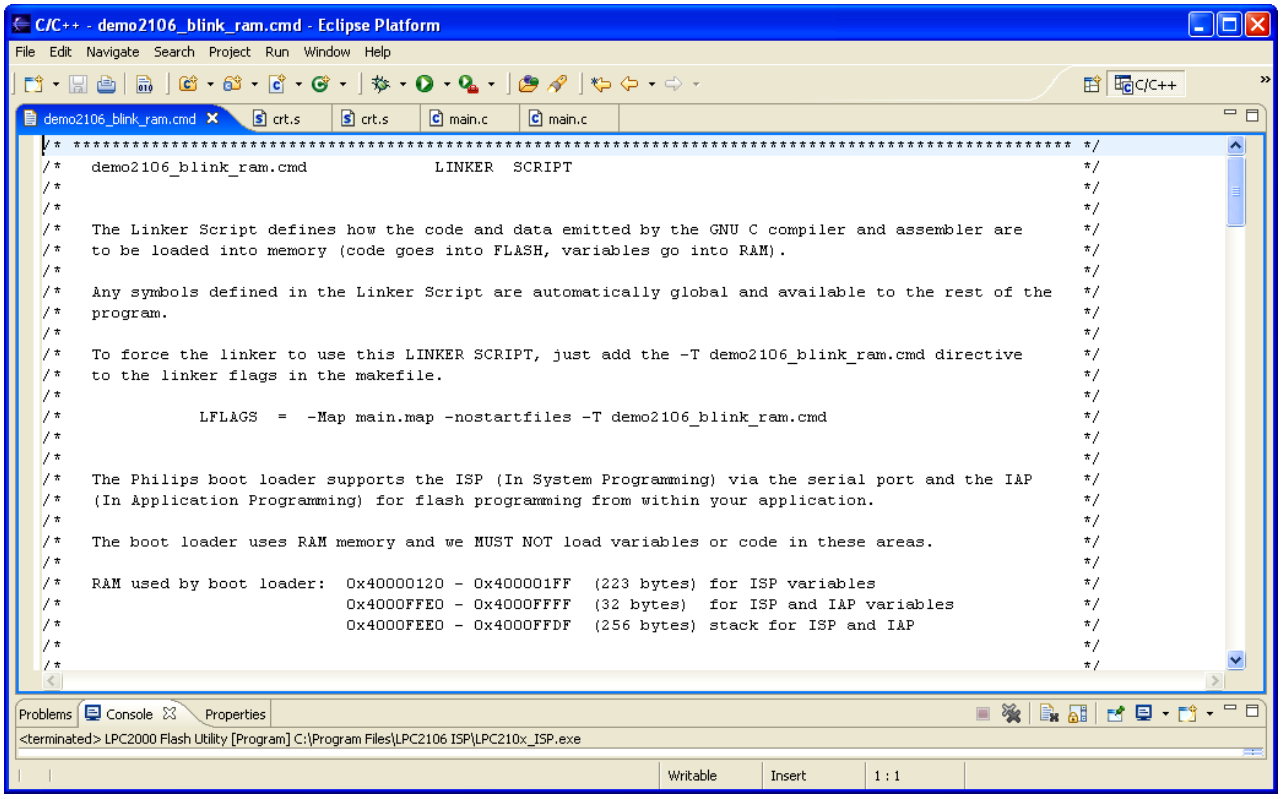
Since we are not using any interrupts in this example, this addition does not really matter. I've just added it for completeness; you should always do this when devising a project to run in RAM.

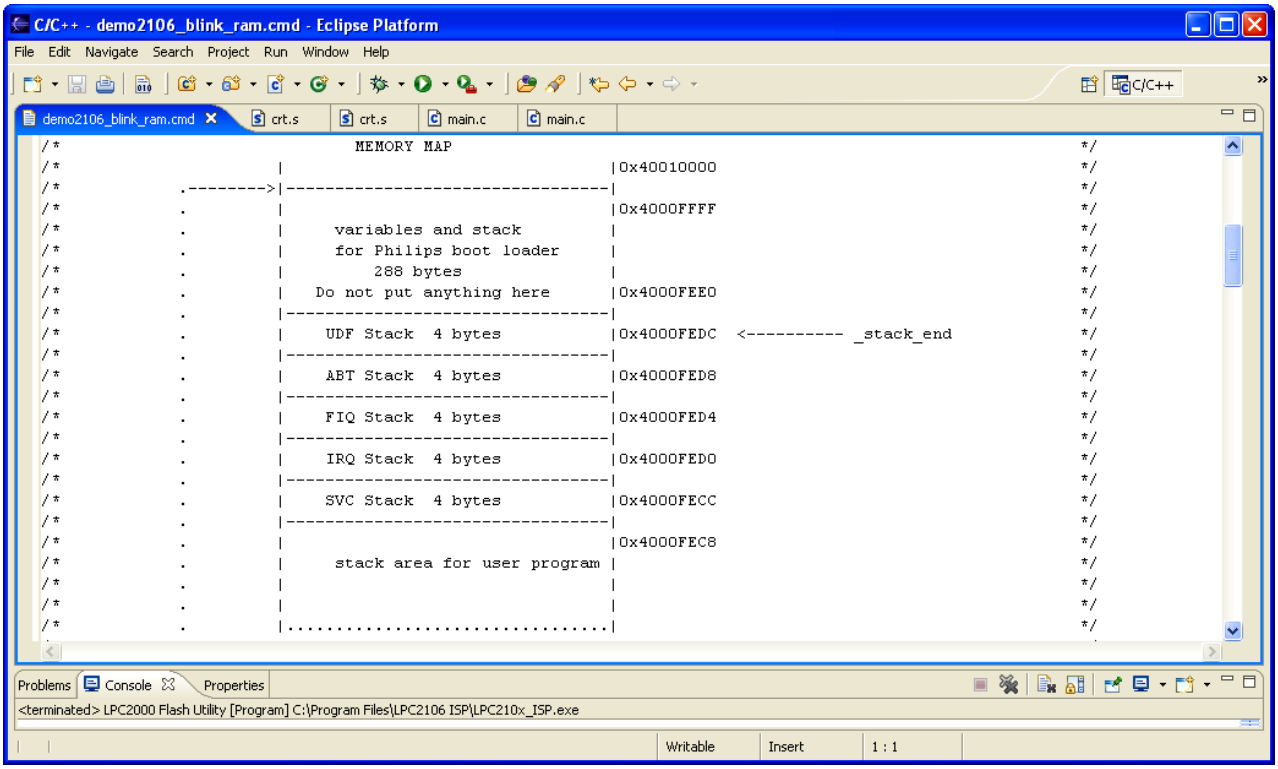
After you follow the next steps and get the application to execute out of RAM, you can run a little experiment and comment out the MEMMAP = 0x02; line. It will still run OK.

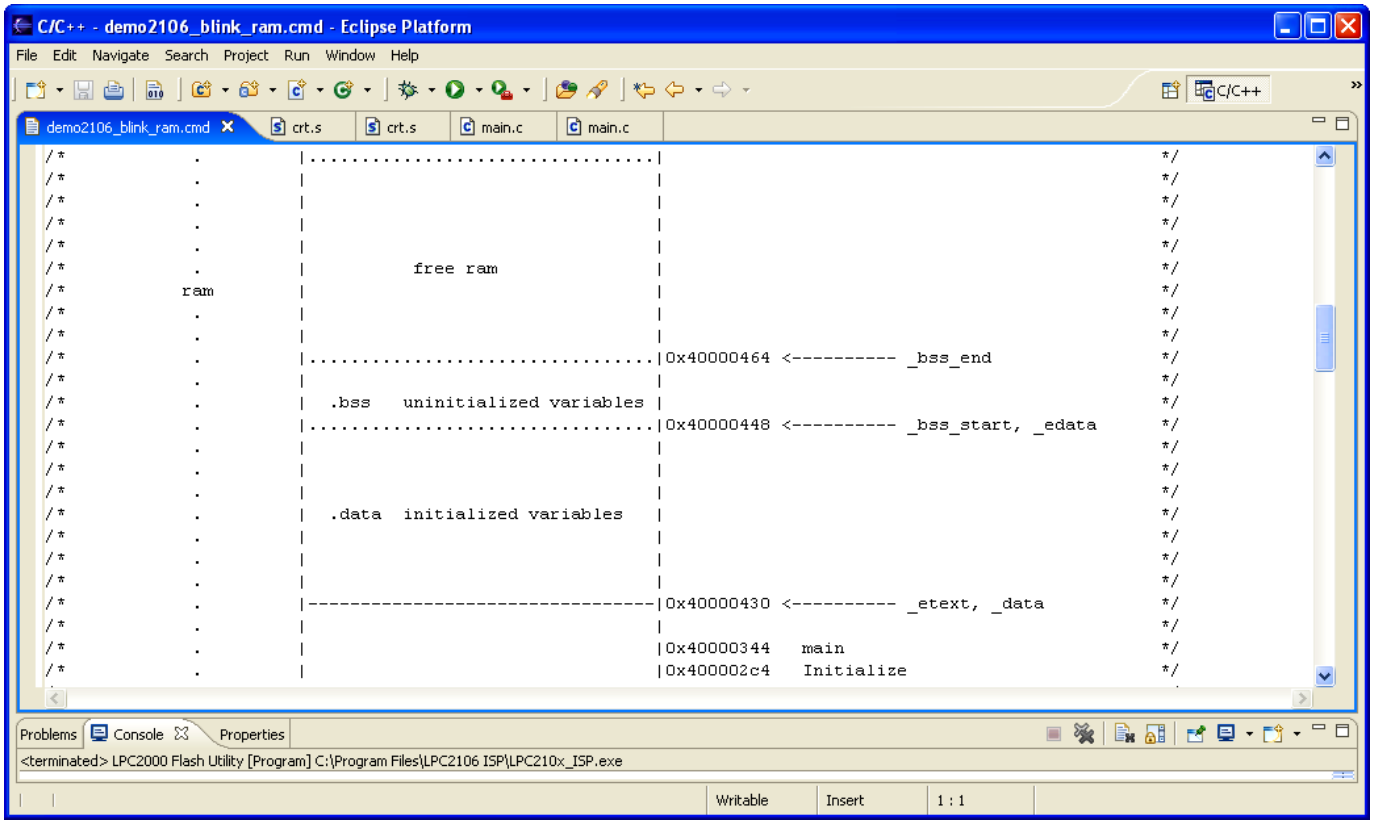
The reason for that is two-fold. First, we don't use interrupts in this example. Second, we use the Philips ISP Flash Loader to force the CPU to start at the address of Reset_Handler; which is at 0x40000200. This bypasses using the RESET vector at 0x4000000 to start the application.

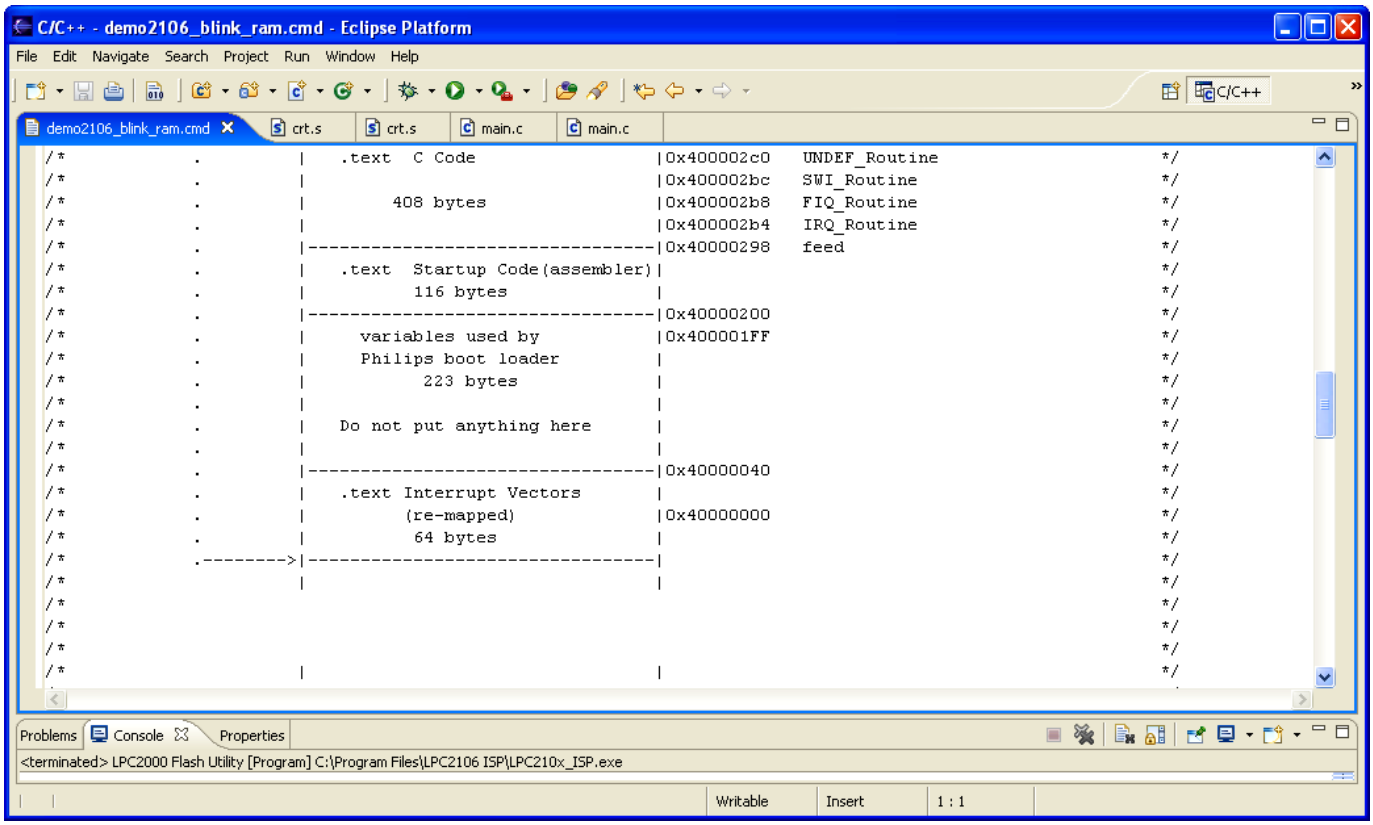
File DEMO2106_BLINK_RAM.CMD

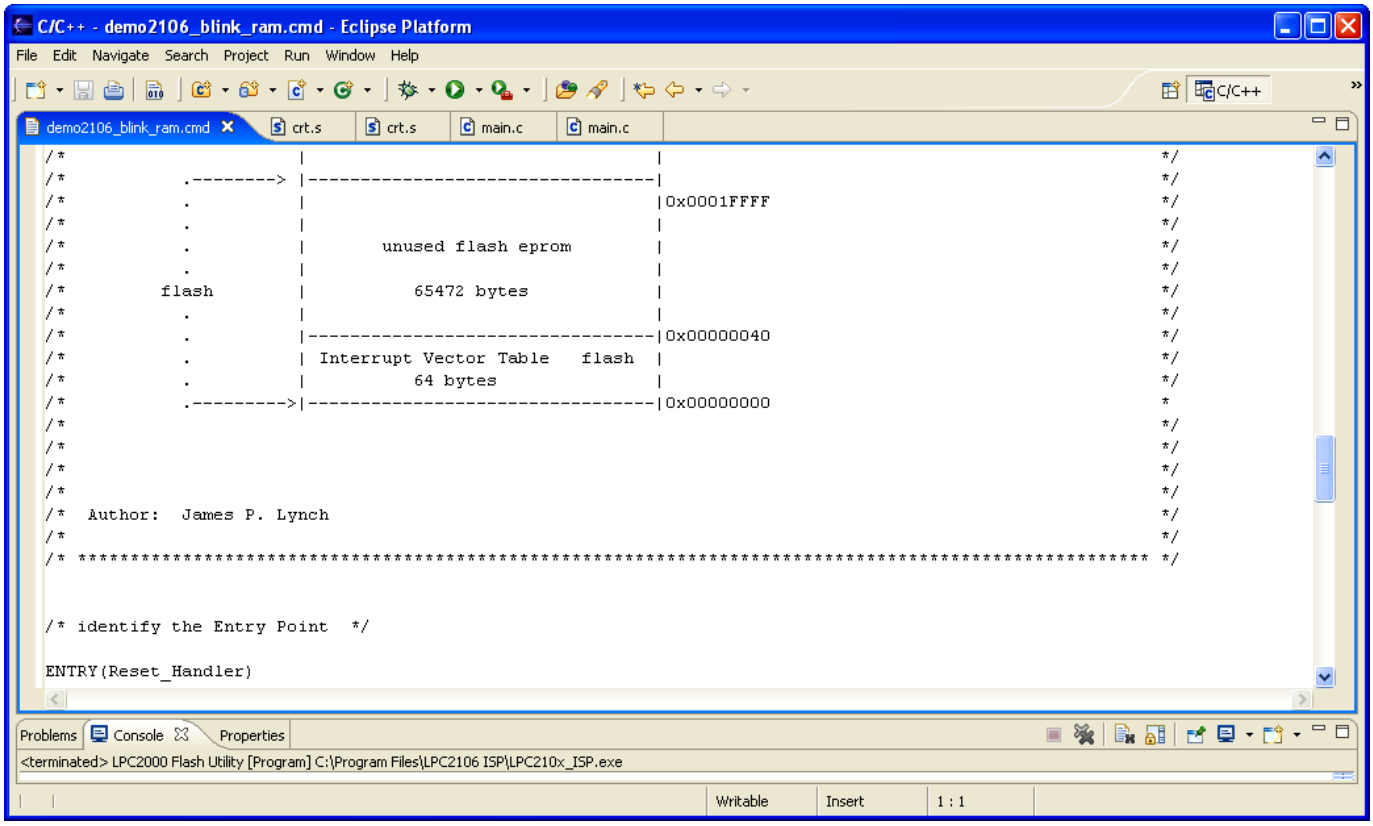
The entire project, both code and variables, is going to be loaded into RAM. Therefore, there are a few changes in the Linker Command Script file **demo2106_blink_ram.cmd**.



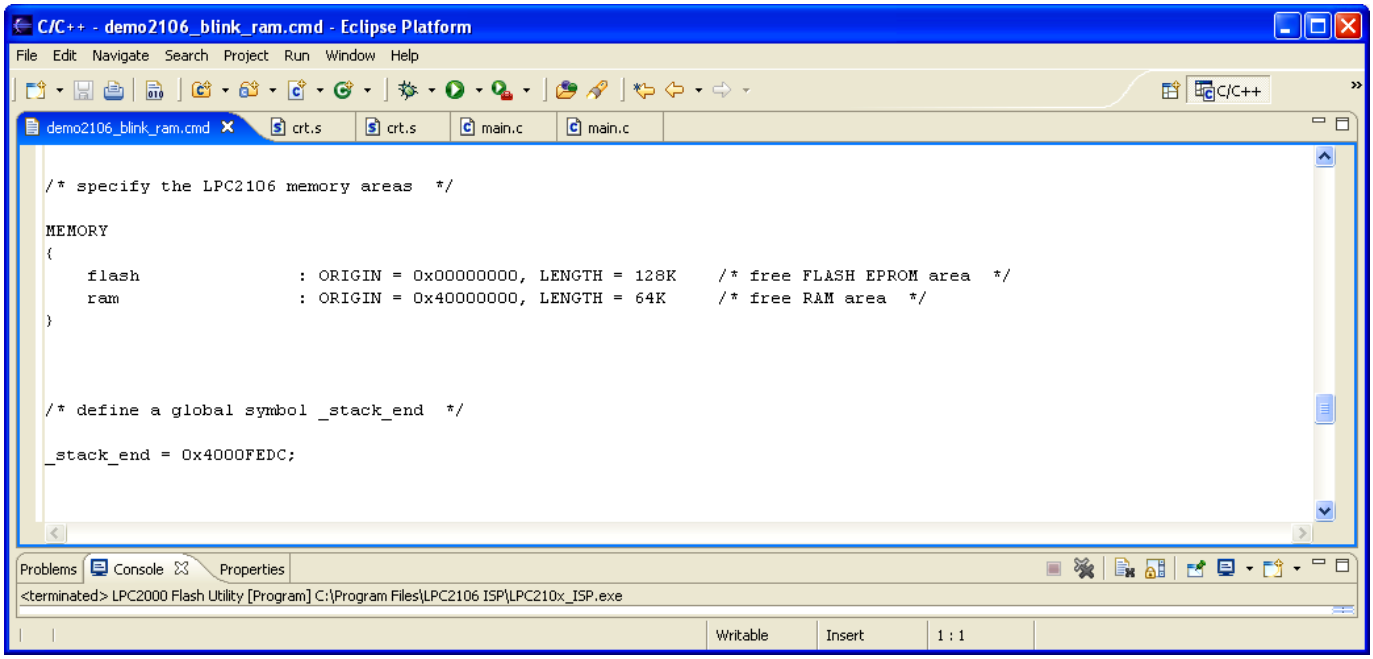



C/C++ - demo2106_blink_ram.cmd - Eclipse Platform
File Edit Navigate Search Project Run Window Help
demo2106_blink_ram.cmd
crt.s
crt.s
main.c
main.c
/* . |.....| */
/* . | | */
/* . | free ram | */
/* ram | | */
/* . | | */
/* . |.....|0x40000464 <---------- _bss_end */
/* . | | */
/* . | .bss uninitialized variables |0x40000448 <---------- _bss_start, _edata */
/* . | | */
/* . | | */
/* . | .data initialized variables | */
/* . | | */
/* . | | */
/* . |---------|0x40000430 <---------- _etext, _data */
/* . | |0x40000344 main */
/* . | |0x400002c4 Initialize */
Problems Console Properties
<terminated> LPC2000 Flash Utility [Program] C:\Program Files\LPC2106 ISP\LPC210x_ISP.exe
Writable Insert 1 : 1


C/C++ - demo2106_blink_ram.cmd - Eclipse Platform
File Edit Navigate Search Project Run Window Help
demo2106_blink_ram.cmd
crt.s
crt.s
main.c
main.c
/* . | .text C Code |0x400002c0 UNDEF_Routine */
/* . | |0x400002bc SWI_Routine */
/* . | 408 bytes |0x400002b8 FIQ_Routine */
/* . | |0x400002b4 IRQ_Routine */
/* . |---------|0x40000298 feed */
/* . | .text Startup Code(assembler)| */
/* . | 116 bytes | */
/* . |---------|0x40000200 */
/* . | variables used by |0x400001FF */
/* . | Philips boot loader | */
/* . | 223 bytes | */
/* . | | */
/* . | Do not put anything here | */
/* . | | */
/* . |---------|0x40000040 */
/* . | .text Interrupt Vectors | */
/* . | (re-mapped) |0x40000000 */
/* . | 64 bytes | */
/* .--------->|---------| */
Problems Console Properties
<terminated> LPC2000 Flash Utility [Program] C:\Program Files\LPC2106 ISP\LPC210x_ISP.exe
Writable Insert 1 : 1

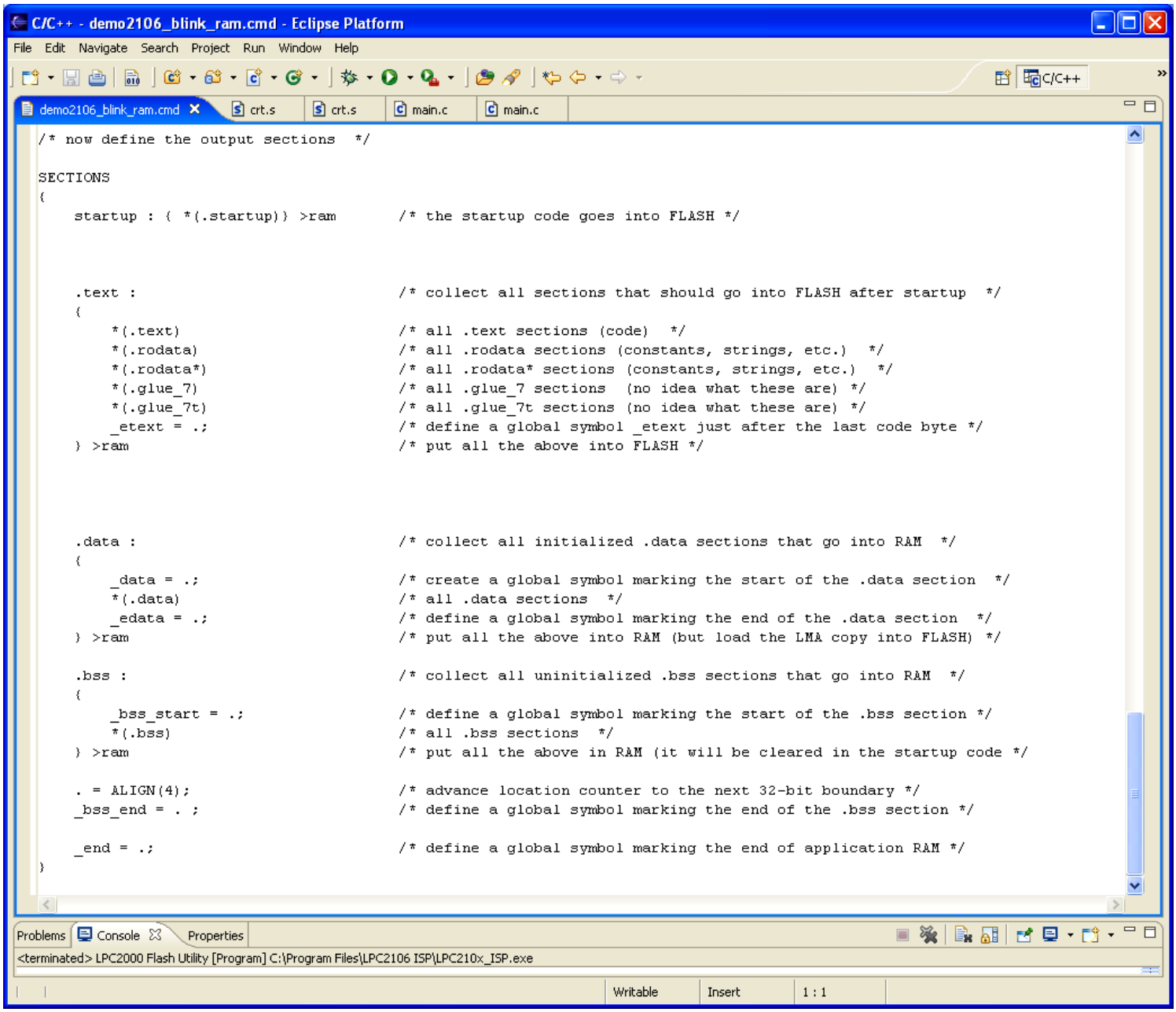


I added quite a bit of annotation above to make it very clear how the memory (flash and ram) is organized.



Above I defined two memory areas for flash and RAM, consistent with the LPC2106 memory map. Of course, we're going to load everything (code and variables) into RAM!

Note that I also created a global symbol, _stack_end, that is used in the startup routine to build the various stacks. The address is positioned just after the stacks and variables used by the Philips ISP Flash Utility.



Above is the final part of the Linker Command Script. Notice that everything is loaded into **RAM**.

You might ask, “Do we still copy the **.data** section initializers?” I left the copy operation intact in file CRT.S but it now essentially copies over itself (wasteful). I wanted to keep things very similar. You could delete the **.data** initializer copy code in **crt.s** to save space.

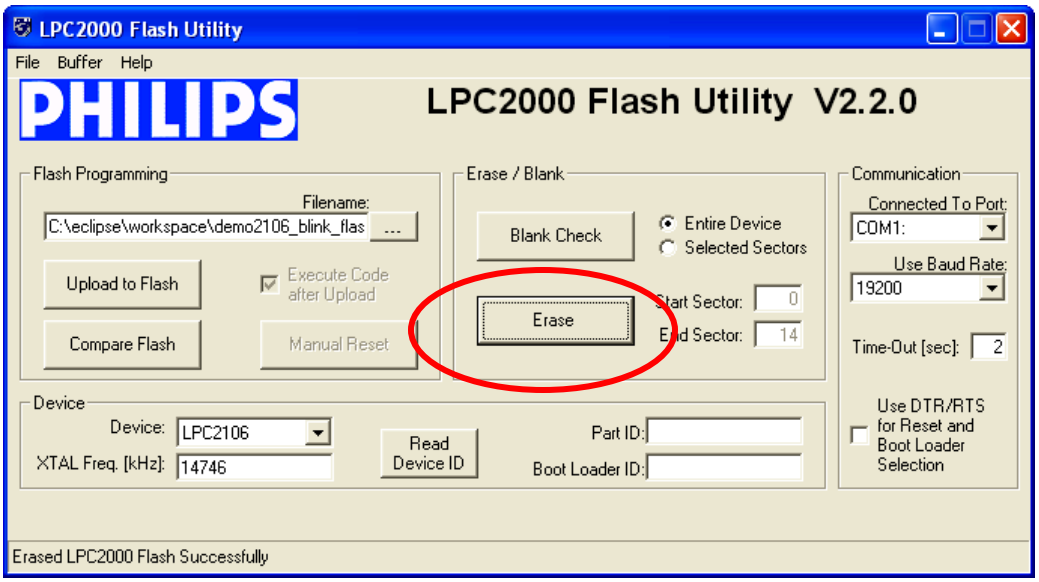
You might also ask, “Do we still clear the **.bss** section?” The answer is absolutely yes, RAM memory powers on into an unknown state. We want all uninitialized variables to

be zero ar start-up. Of course, stupid programmers rely on uninitialized variables to be zero at boot-up, this is how they get into trouble with uninitialized variables (not all compilers do this automatically).

At this point, if you haven't cleaned and built the project, do it now.

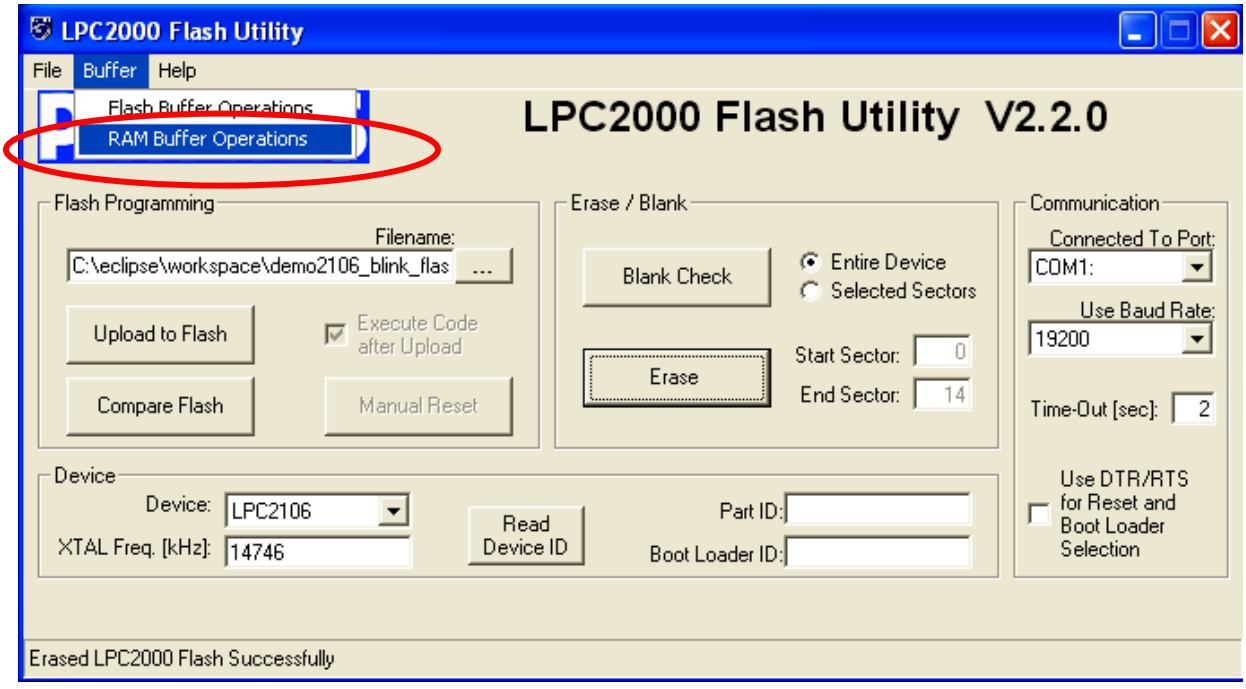
Make sure the BSL jumper is installed.

Now use the "External Tools" toolbar button to find the Philips ISP Flash Utility and start it. To make sure that we are not fooling ourselves, click on "Erase" to clear the flash memory.

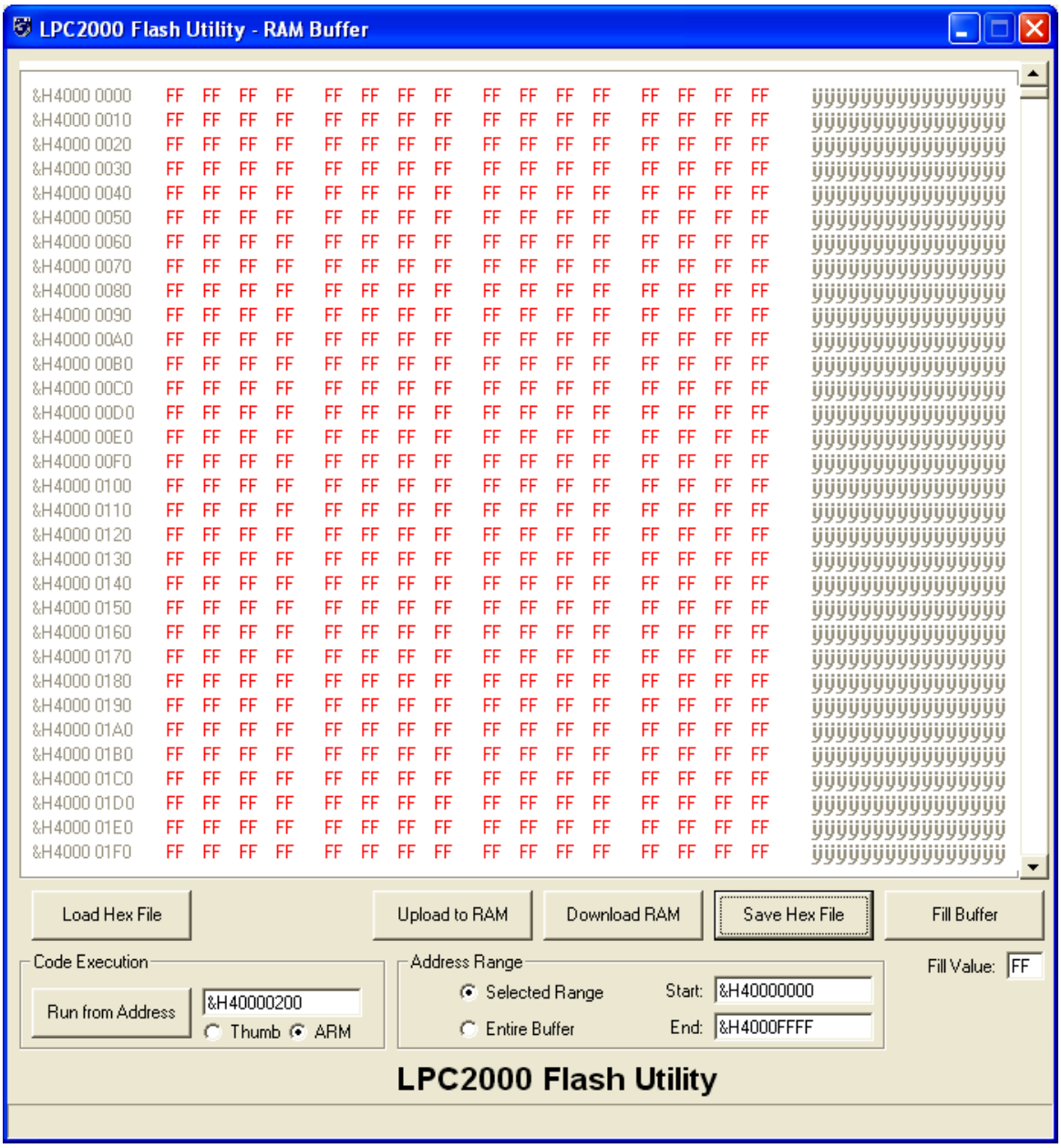


Now we can be sure that the blinking LED is not the Flash application running.

Click on "**Buffer – RAM Buffer Operations**."

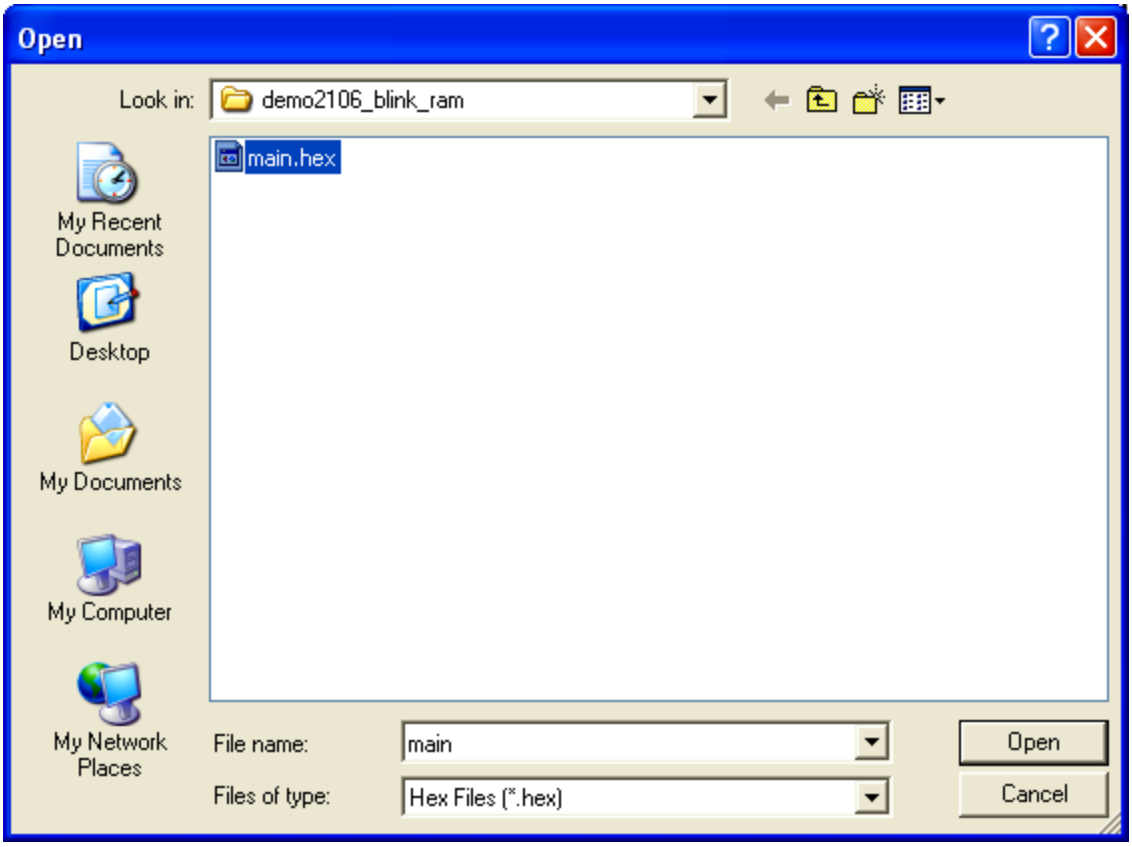


The RAM Buffer screen now appears. Click on “**Load Hex File**.” This is just an operation that fetches the hex file and puts it into the **Philips ISP Flash Utility**.



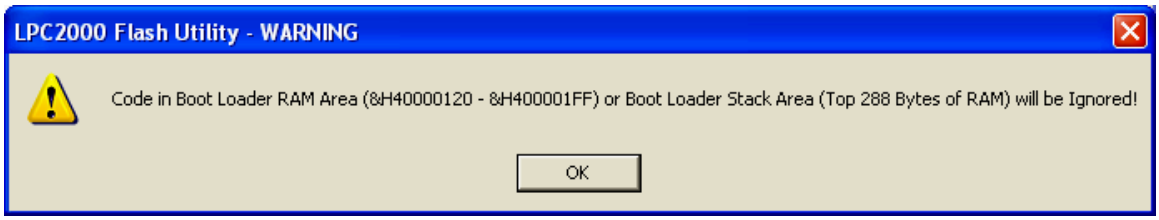
Notice that the button titled “Run from Address” has the value &H40000200 in it. This is thanks to the **ENTRY(Reset_Handler)** directive in the linker command script file. The Philips boot loader will simply load 0x40000200 into the PC register and let her rip!

When you click on the “Load Hex File” button, the following dialog will be presented.



Browse for the **main.hex** file in the project directory and click "**Open**".

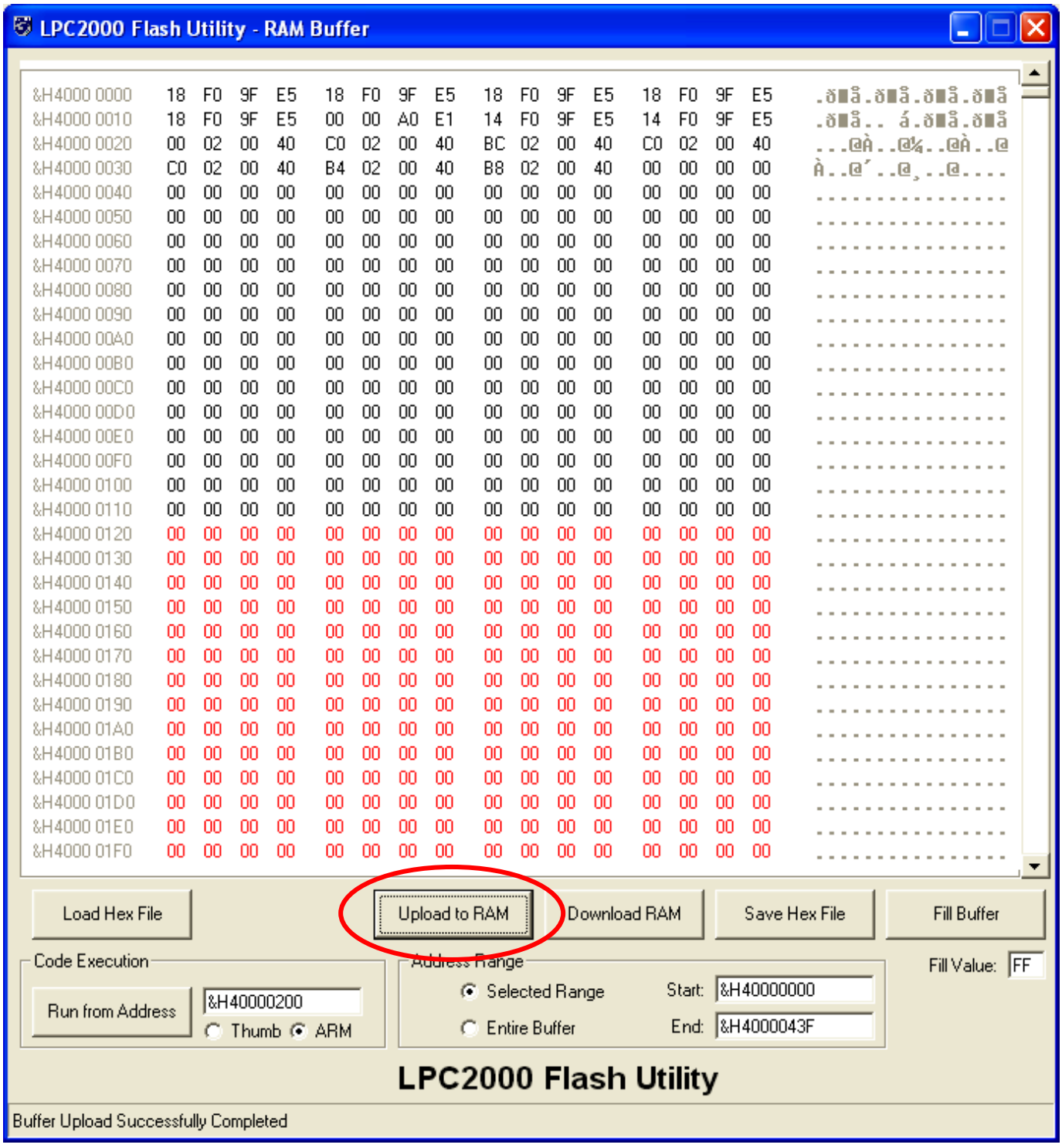
The following warning is presented. Since I advanced the location counter past the low RAM area used by Philips, it still thinks that there's code in there. If I had elected to make the interrupt vectors a separate section, I could have avoided this warning.



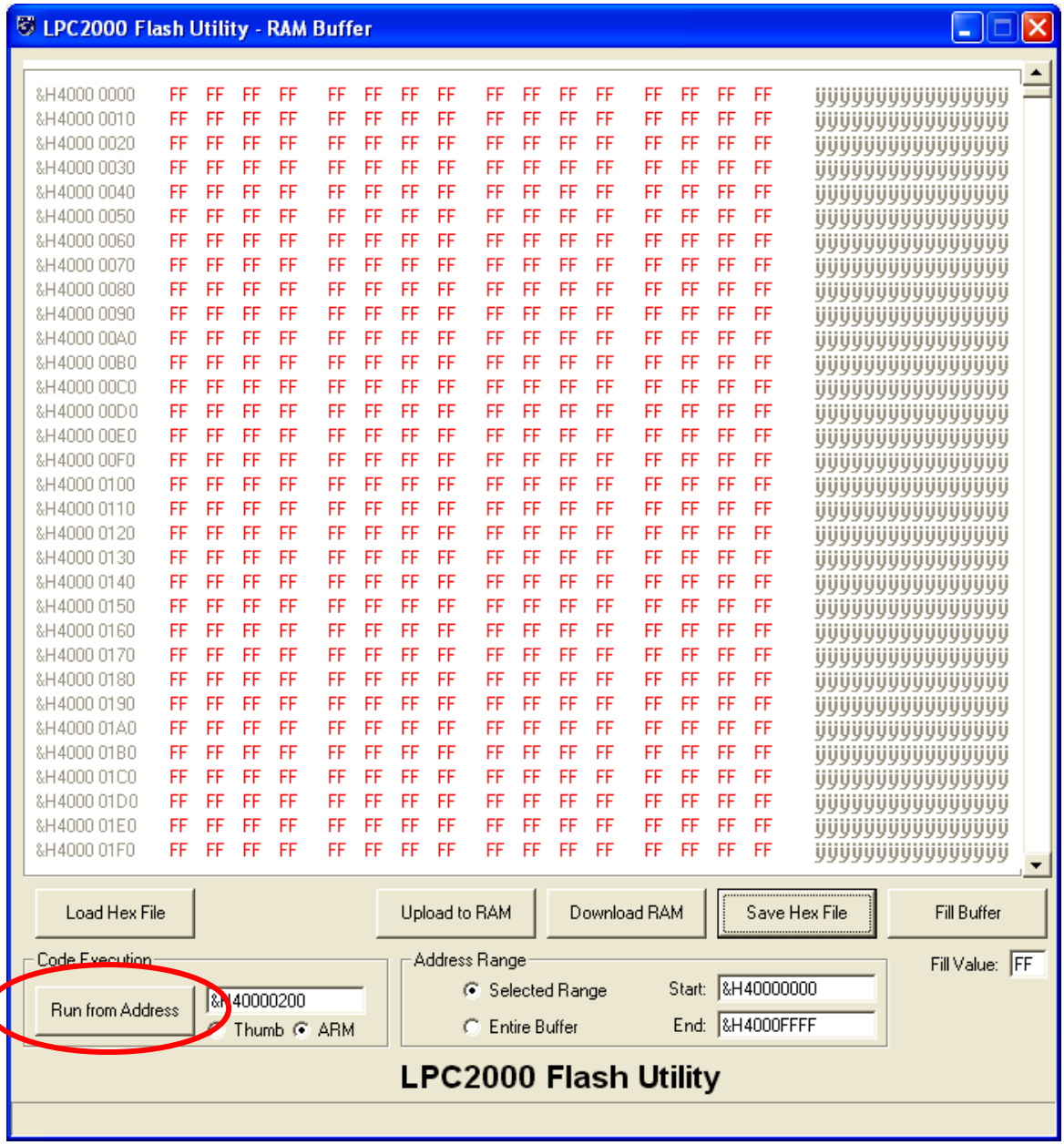
It will still execute OK, of course, since the hex file has no bytes defined for the area where we advanced the program counter past the Philips ISP low RAM usage.

Now click on the “**Upload to RAM**” button to load the hex file into the LPC2106 RAM memory.

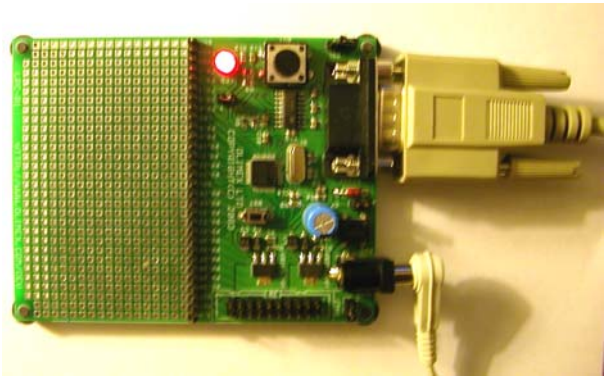
You will see a “progress bar” at the bottom of the screen and it will indicate that the operation has completed.



You do **NOT** have to remove the BSL jumper. Click on the “Run from Address” button to execute the program.



Your application should blink, just like the Flash EPROM version did. Time for the Bigfoot picture!

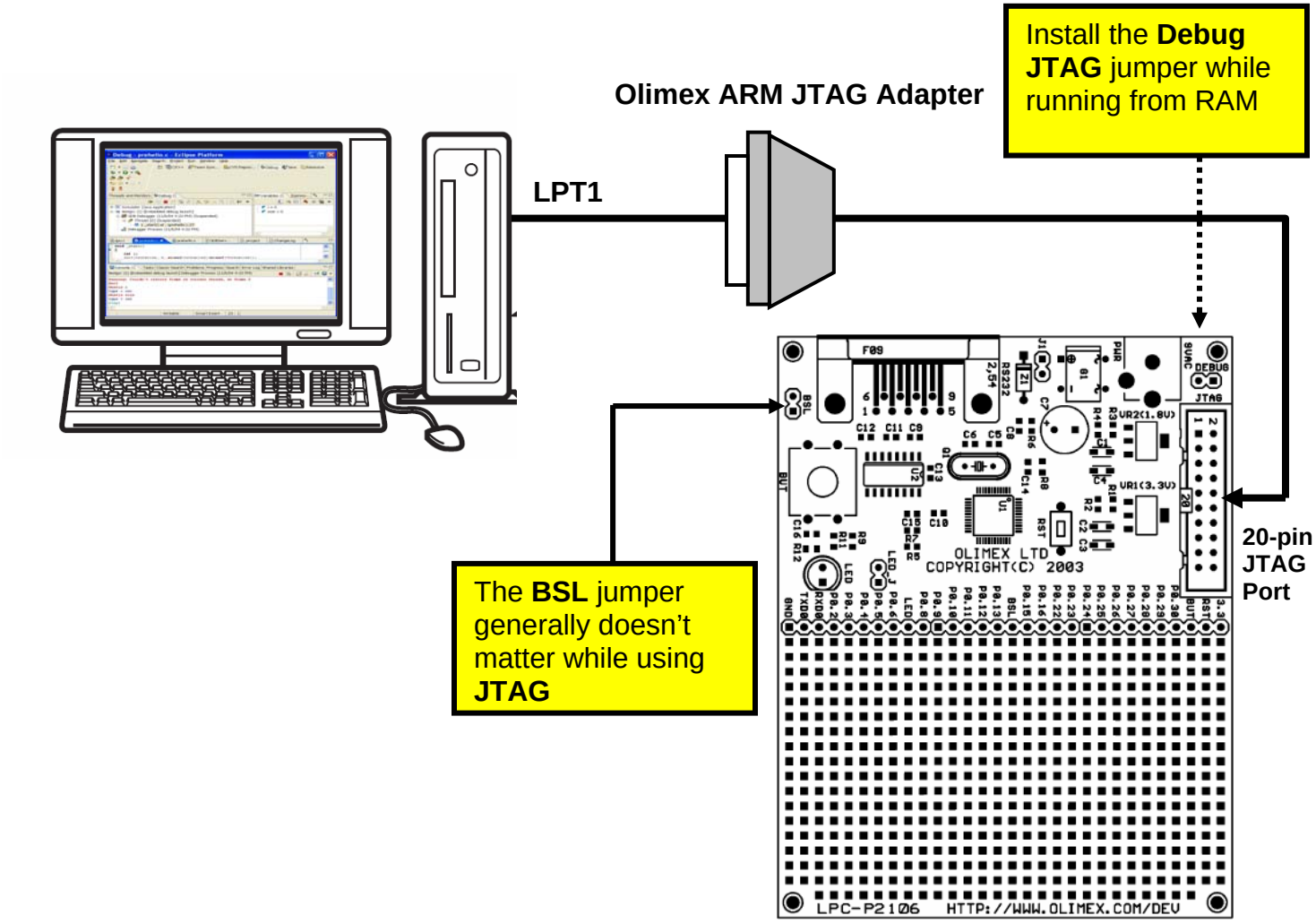


IT BLINKS!

22 Execute the RAM Project with the Insight Debugger

The previous exercise, running the RAM project from the Flash Utility, was of academic interest but essentially of no practical value. Well, it is kind of cool that you can do that with a flash utility.

Packaged within the GNUARM toolkit is the **Insight Graphical Debugger**. You can start it as an "external tool" very conveniently and it is specifically designed to debug GNU applications.
We will need the following hardware setup:



The **Olimex ARM JTAG Adapter** is a clone of the Macraigor **Wiggler** JTAG interface. It costs about $19.95 and all fits into a DB-25 shell. I bought a straight-through printer cable from my local computer retailer and fitted it from the LPT1 printer port to the ARM JTAG **Wiggler**. The **Wiggler** was then fitted to the 20-pin JTAG header on the Olimex **LPC-P2106** board.

The red stripe on the ribbon cable is pin 1 and should be nearest the power plug.

The **Debug JTAG** jumper should be fitted. It doesn't matter if the **BSL** jumper is installed or not. Make all these connections with the power off.

A. Blunt Talk About the Wiggler

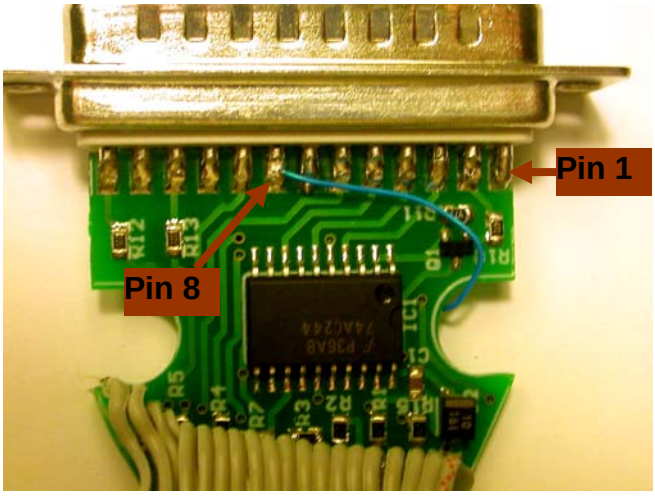
Let's talk bluntly about the **Wiggler**. The **Wiggler** is one of many products from the Canadian company Macraigor. It connects the parallel port of your PC to the 20-pin JTAG header on the Olimex **LPC-P2106** board. It is just a simple level shifter and a transistor. Macraigor charges $150 for it; the Olimex clone is about $19.

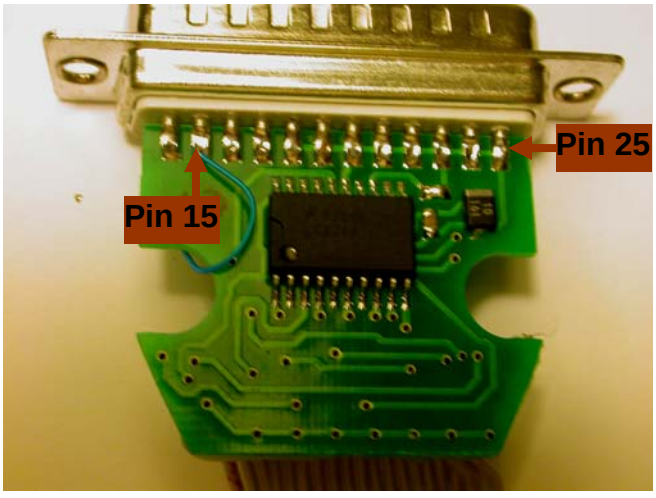
There are several schematic diagrams on the web for the **Wiggler**; notably Leon Heller has one on the LPC2000 message board on Yahoo. You could build your own but I doubt you'd save that much money after paying the shipping from Digikey and the gas to drive to Radio Shack. The Olimex version is a fair deal.

Obviously the Macraigor Company is not happy about all these clones running about, so recently they slipped an impediment into the works. The latest version of **OCDremote**; their free JTAG server for the **Wiggler** and other products, is expecting a connection (short circuit) between **pins 8 and 15** of the LPT1 printer port. This has made a lot of people fail.

Olimex has said that they would revise their design and modify their stock of **Wigglers** to make this connection, but there are large numbers of the device out there that don't have this modification (like my Olimex **Wiggler**).

Use an ohmmeter on the 25-pin printer connector on the **Wiggler** to see if these two pins are connected. If not, you can easily disassemble the Olimex **Wiggler** and tack-solder a jumper to do the job. Again, **<u>you must connect pin 8 to pin 15</u>**.





I used that 30 gauge Radio Shack blue Teflon coated hookup wire and a microscope to do the soldering above. If you have a good magnifier; the DB-25 pins on the wiggler have the pin numbers embossed in the white plastic above and below the rows of pins.

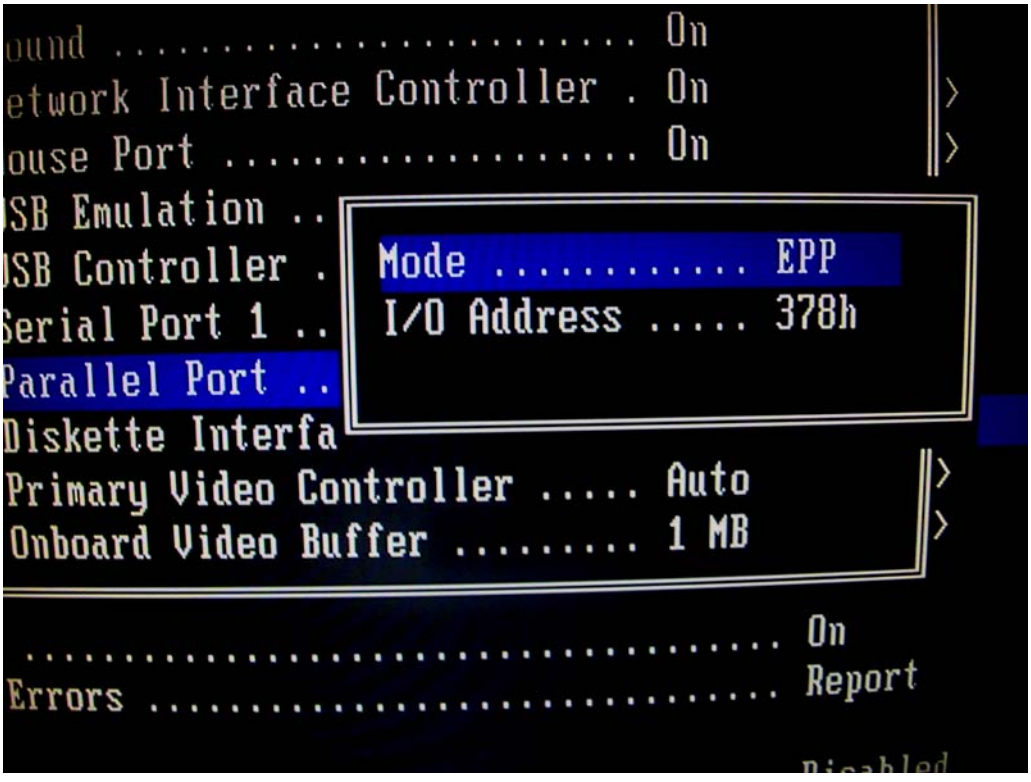
We're not quite finished with our **Wiggler** suffering. There's the final issue of the PC Printer port mode. Most modern PCs, like my new Dell, have the Printer Port defaulted to "**ECP**" mode.

The **Wiggler** will not work with the printer port configured for **ECP** mode.

The Macraigor web site has a FAQ with the following citation:

> **What mode must my parallel port be in?**
>
> **As far as the parallel port is concerned, a Wiggler is a simple uni-directional device. It will work with the parallel port in any mode EXCEPT "ECP". It will NOT work in ECP mode at all.**
>
> **The Raven works best with a parallel port in EPP mode. It may work in ECP mode. If the parallel port is in an older mode, such as uni-directional, AT, or compatible, the Raven will work but slower.**
>
> **Remember, the mode is set in the CMOS bios of your computer.**

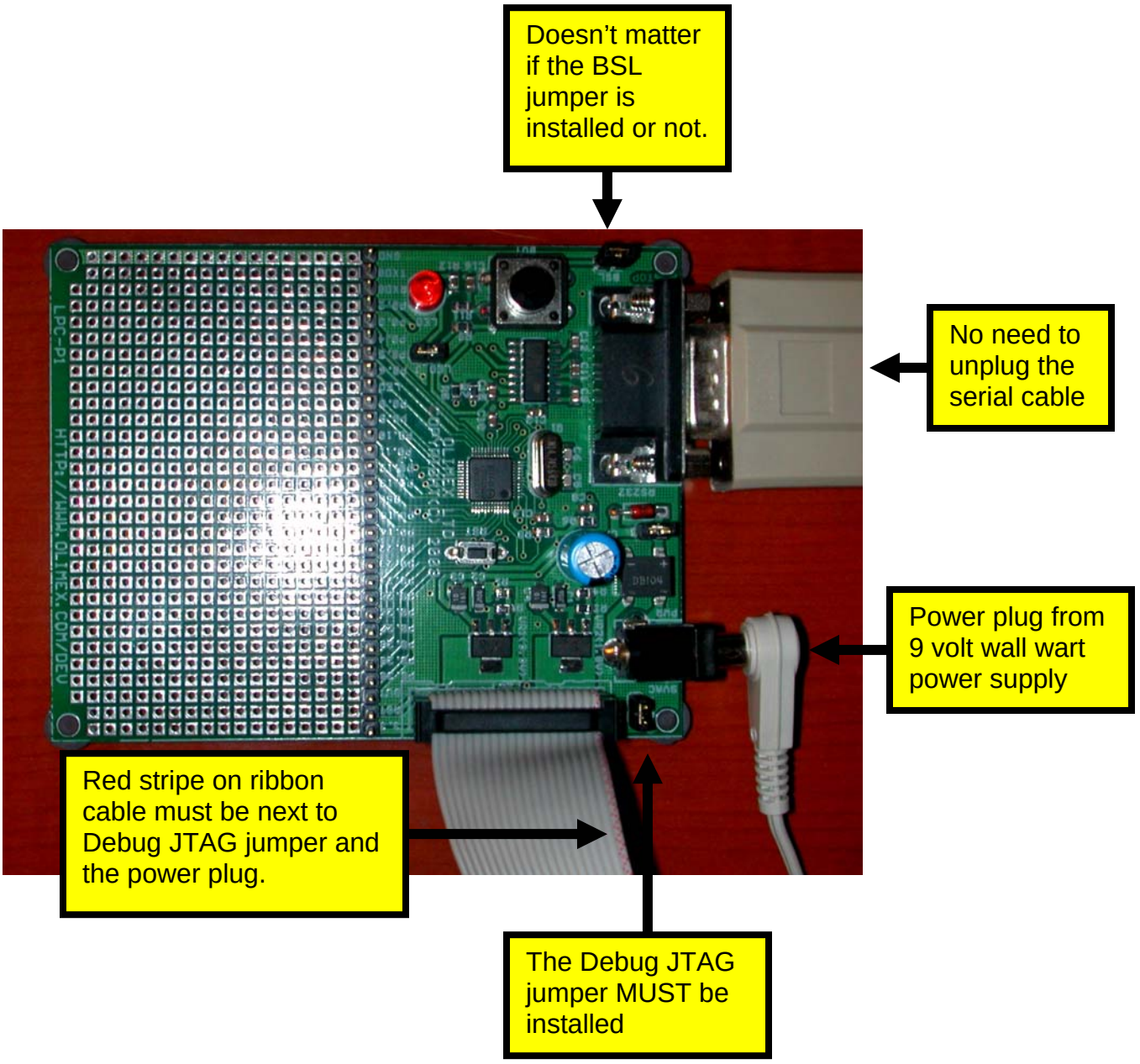
On my Dell Dimension Desktop PC, the CMOS setup can be entered if you hit the **F2** key as the machine boots up. By maneuvering around the CMOS setup, you can find the Parallel Port setup and see what mode it is set up as. If it's **ECP** mode, change it to **EPP** mode, as I did in the screen photograph below. Save the CMOS setup and exit.



My printer is a USB device, so this action didn't effect my printer operation.

It's disappointing to report that the **Wiggler** cannot set a breakpoint in FLASH. The **OCDremote** application cannot deal with GDB –z commands which refer to hardware breakpoints (the LPC2106 has two hardware breakpoints. This is the reason that debugging with this simple device is limited to applications configured to run from RAM exclusively. Obviously, the 64K limitation of the Philips LPC2106 limits the size of an application you can fit into RAM. However, it's better than nothing.

Let's review the hardware setup one more time.



Power up the Olimex LPC-P2106 board and press the **RST** button for good luck!

B. Final Preparations Before Starting Insight Debugger

The following procedure to start the **Insight** Debugger and download and debug our RAM application is based on very painful experimentation on my part. There may be other ways to do this, please have fun trying other approaches.

Before we start the **Insight** Graphical Debugger, I should mention that debuggers absolutely hate compiler optimization. This one is no different. We have been compiling with **–O3** and you will find some strange things happening when you single-step at that optimization level.

Just to be sure, let's turn off optimization. Go to the makefile and change the setting to **–O0** and rebuild!

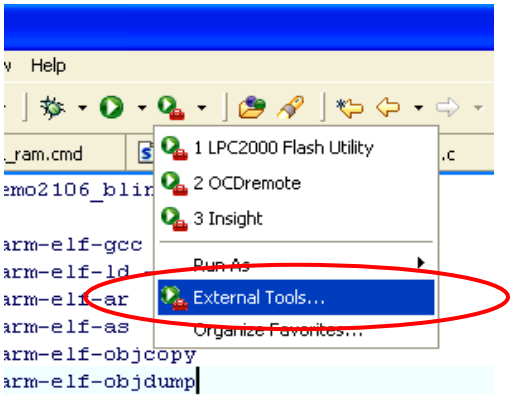
File: **makefile.mak**

```
NAME    = demo2106_blink_ram

CC       = arm-elf-gcc
LD       = arm-elf-ld -v
AR       = arm-elf-ar
AS       = arm-elf-as
CP       = arm-elf-objcopy
OD       = arm-elf-objdump

CFLAGS   = -I./ -c -fno-common -O0 -g
AFLAGS   = -ahls -mapcs-32 -o crt.o
LFLAGS   =  -Map main.map -Tdemo2106_blink_ram.cmd
CPFLAGS  = -O ihex
ODFLAGS  = -x --syms

all: test
```

Turn off compiler optimization by setting compiler flag to:

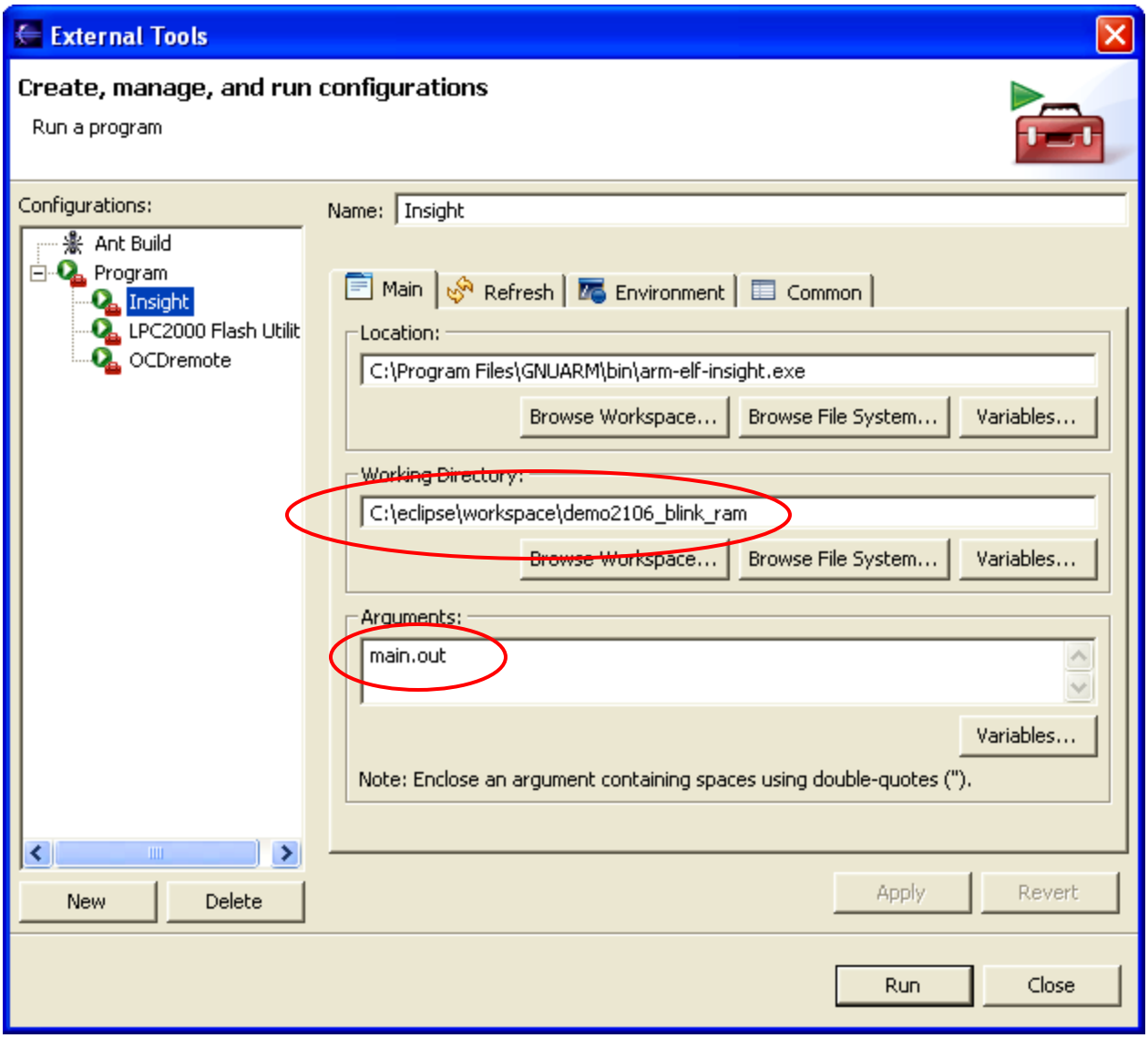
-O0 - no optimization

We also need to configure **Insight** to the specific project we are debugging. Remember the external tools configuration window set up earlier. Click on the **External Tools** toolbar button to bring up this window.



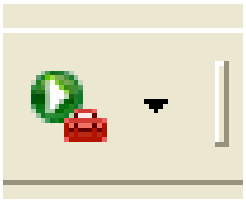
In the "**Working Directory**" text window, use the "**Browse File Space**" button to locate and select the folder that the project resides. In this example, it's **c:\eclipse\workspace\demo2106_blink_ram**.

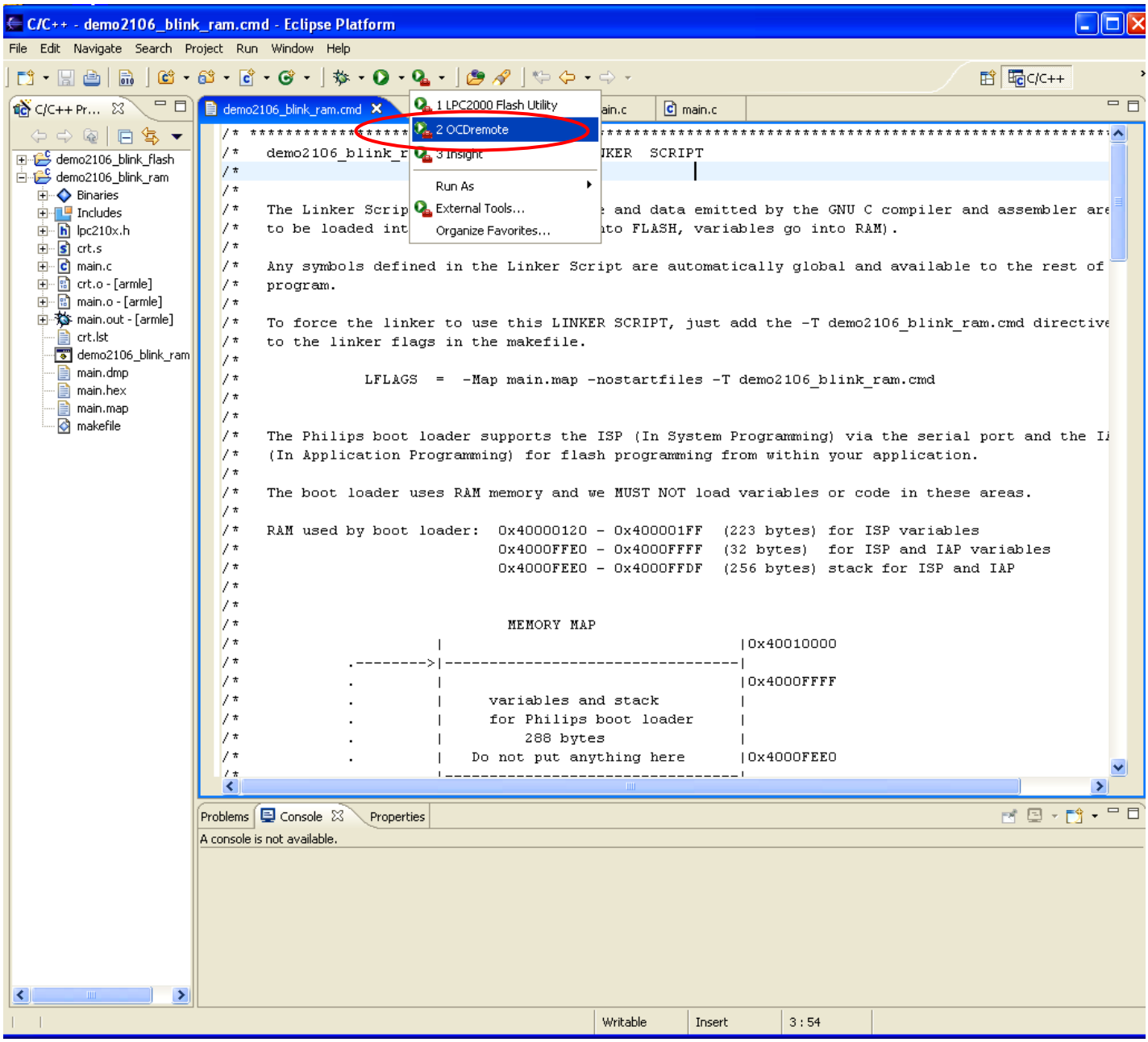
Under the "**Arguments**" text window, specify the GNU output file **main.out**. These changes will guarantee that Insight starts out with the correct source files.



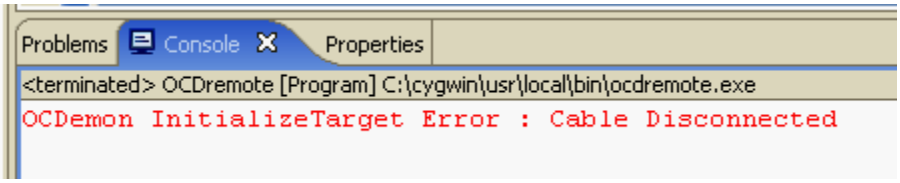
C. Start the Macraigor OCDremote application

Click on the “**External Tools**” toolbar button and start the **OCDremote**.

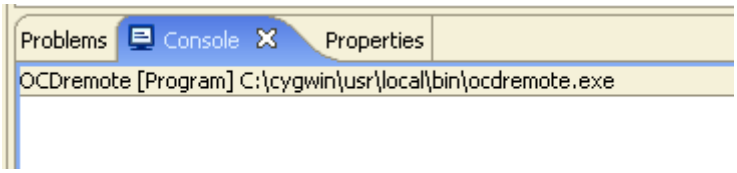




Now the **OCDremote** doesn't always start. You can tell when the console window at the bottom has a red error message that says"



Just keep starting it over and over until you get this response in the console window:

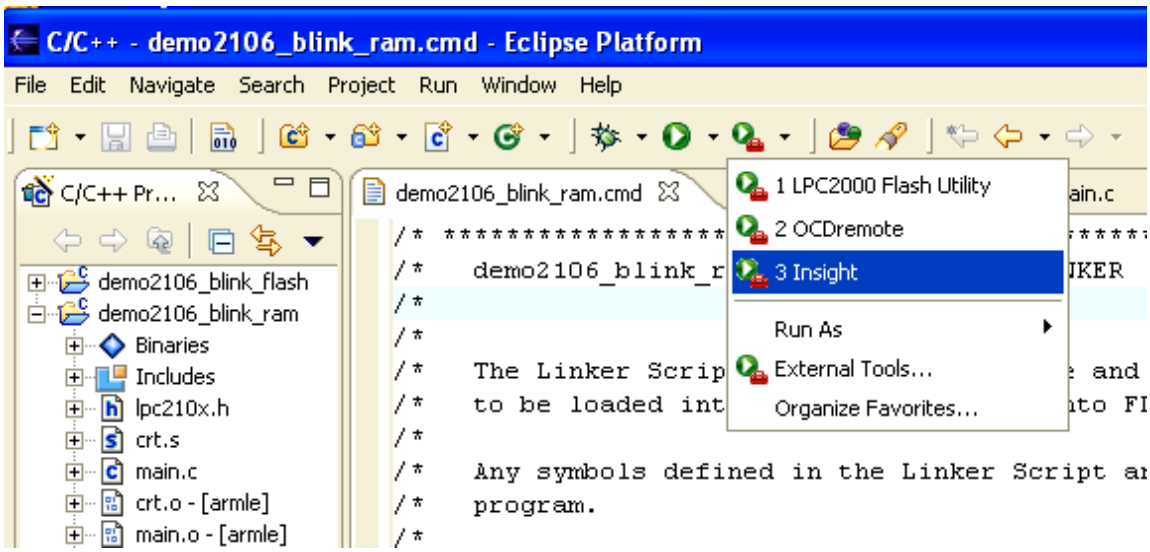


If you have trouble getting **OCDremote** to start; try these remedies:

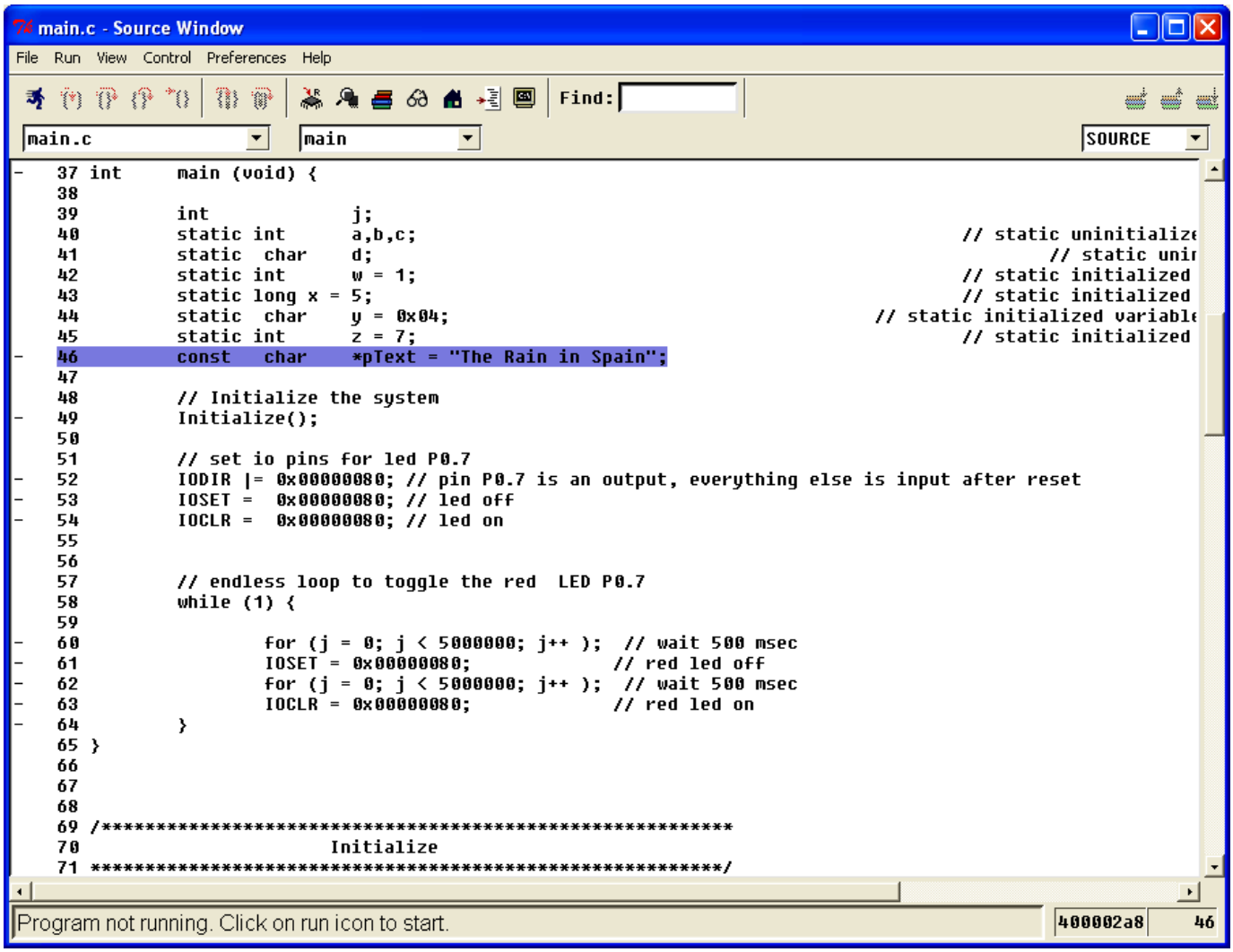
- You may have accidentally started multiple copies of **OCDremote**. Bring up the Windows Task Manager (**ctrl-alt-del**) and search the list of running tasks. If there are multiples, terminate all of them and start over.
- Keep trying; I've done it ten times before it started (this is simply Voodoo).
- Go to bed; let it win tonight.

D. Start the Insight Graphical Debugger

Assuming that it's now running; start the **Insight** Graphical Debugger by clicking on the "**External Tools**" toolbar button and select **Insight**.

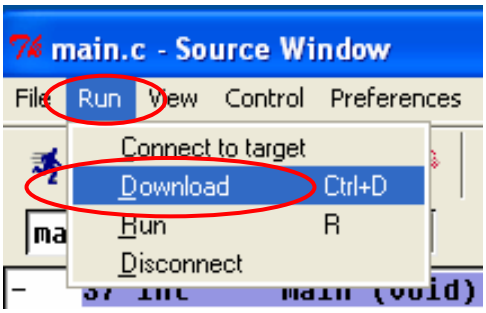


After a few seconds, **Insight** will start up in its own window.

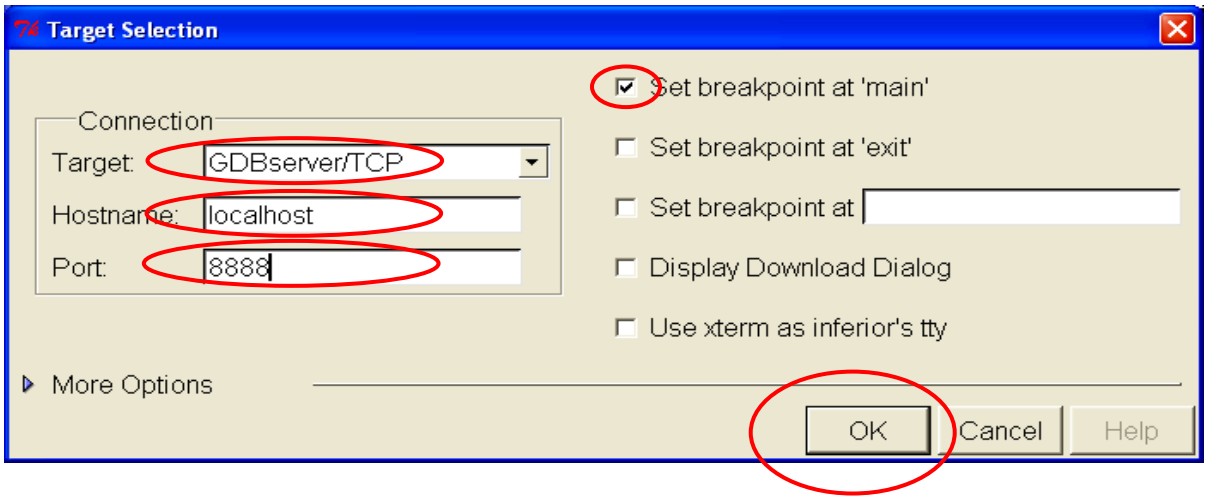


E. Download the Application into RAM

The first step is to "**download**" the application (main.out). Click on "**Run – Download**."



The **Insight** debugger recognizes straight away that it is not connected to the target board. To accomplish this, it will present a "**Target Properties**" dialog window.



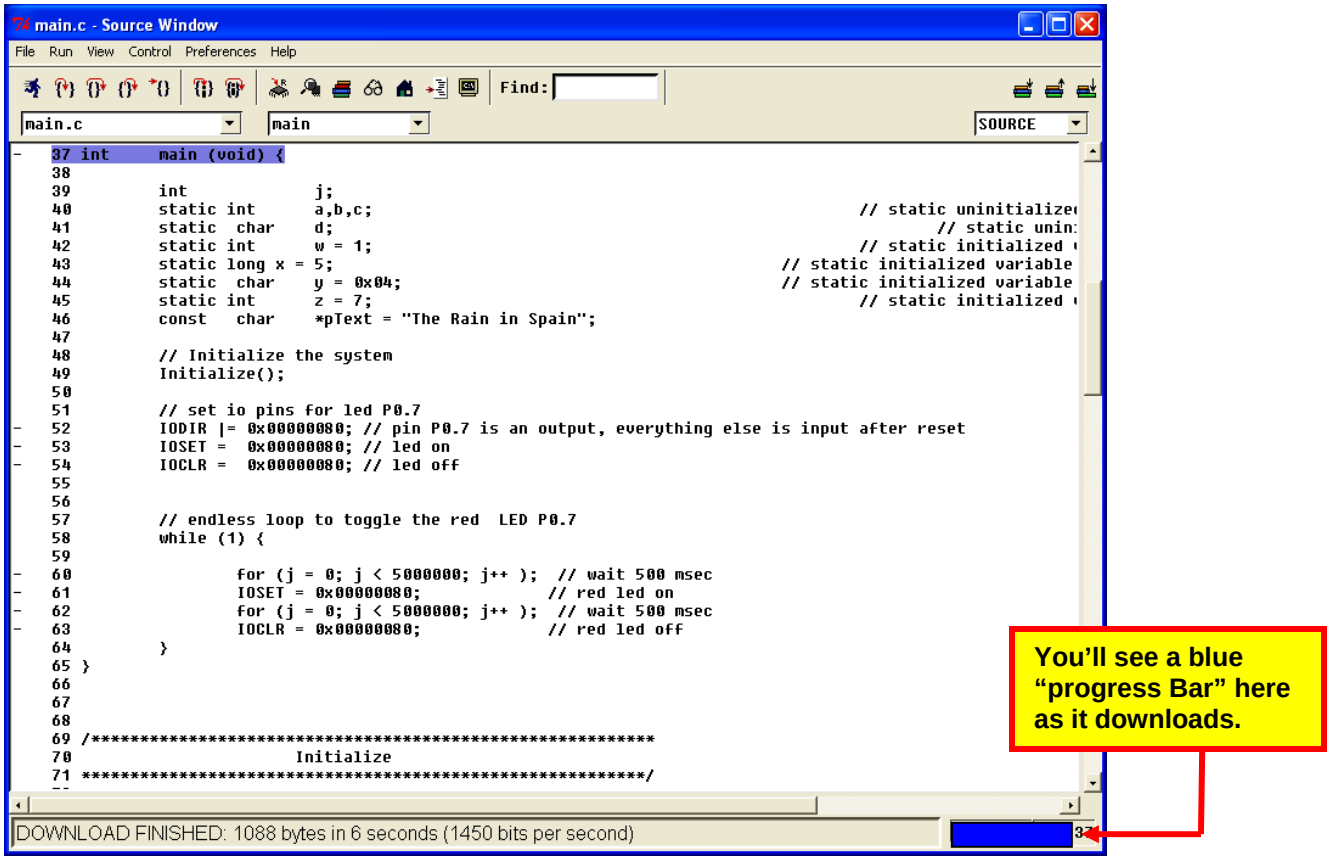
Use the pull-down list to select **GDBserver/TCP** for the target.

Enter the Macraigor default Hostname as **localhost**.

Enter the Macraigor default Port as **8888**.

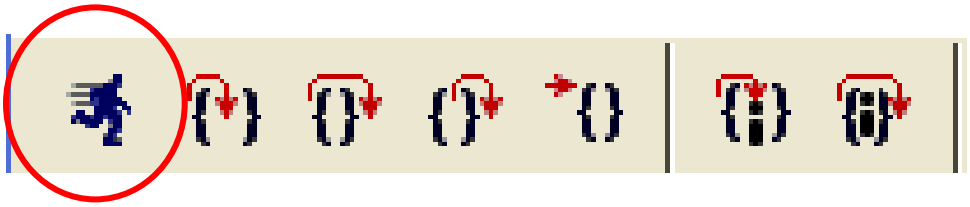
Check the box for **Set Breakpoint at 'main'**

Click **OK** to proceed. **Insight** will connect to the target through the **Wiggler**. It will then load the executable code into the LPC2106 RAM memory. There's a blue progress bar shown at the bottom right. Be patient, the **Wiggler** is painfully slow. This small file takes 6 seconds. Pray that it says "DOWNLOAD FINISHED" in the status bar at the bottom left.

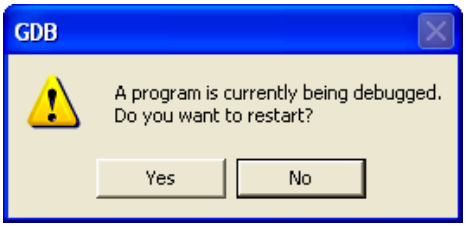


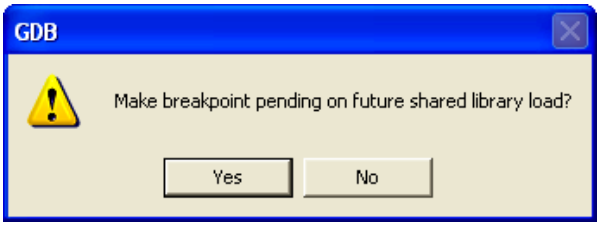
F. RUN the Application to MAIN()

Click on the **RUN** button. The application will start and breakpoint at the main() routine.

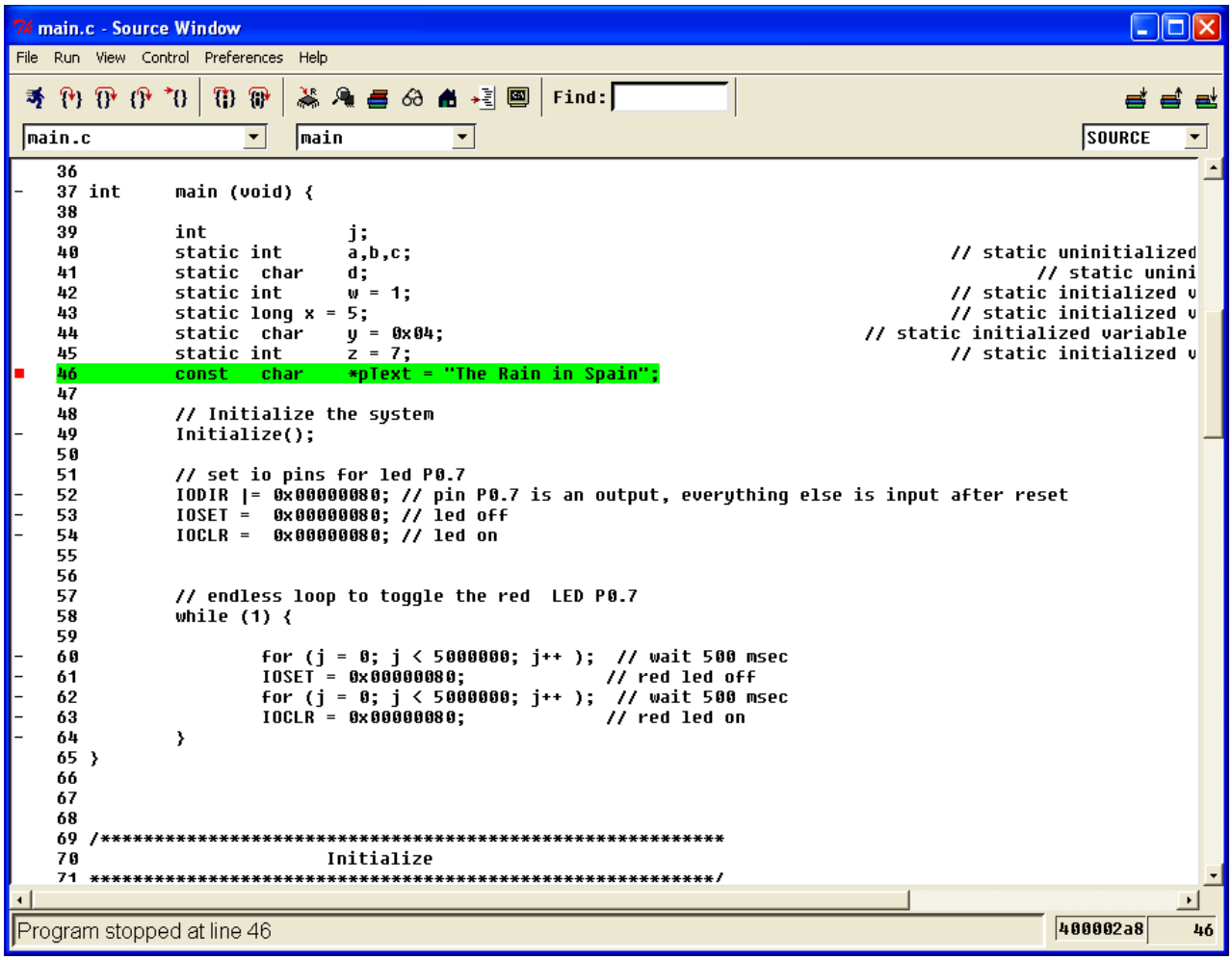


You may get either of these GDB windows, just answer **YES** .

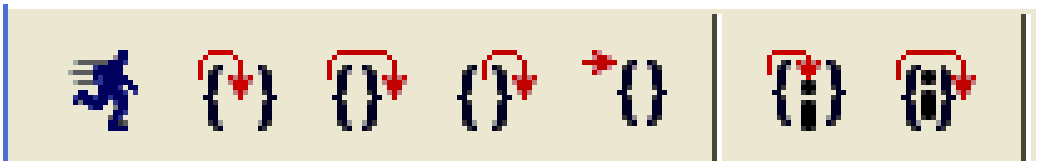




A red box on line 46 indicates that the debugger stopped at main(). If you study the assembly language generated for main(), you'll see that the breakpoint is just after the stack frame setup after entry to main(). A bit quirky, isn't it?



The Insight Debugger has the following toolbar buttons associated with executing the application.

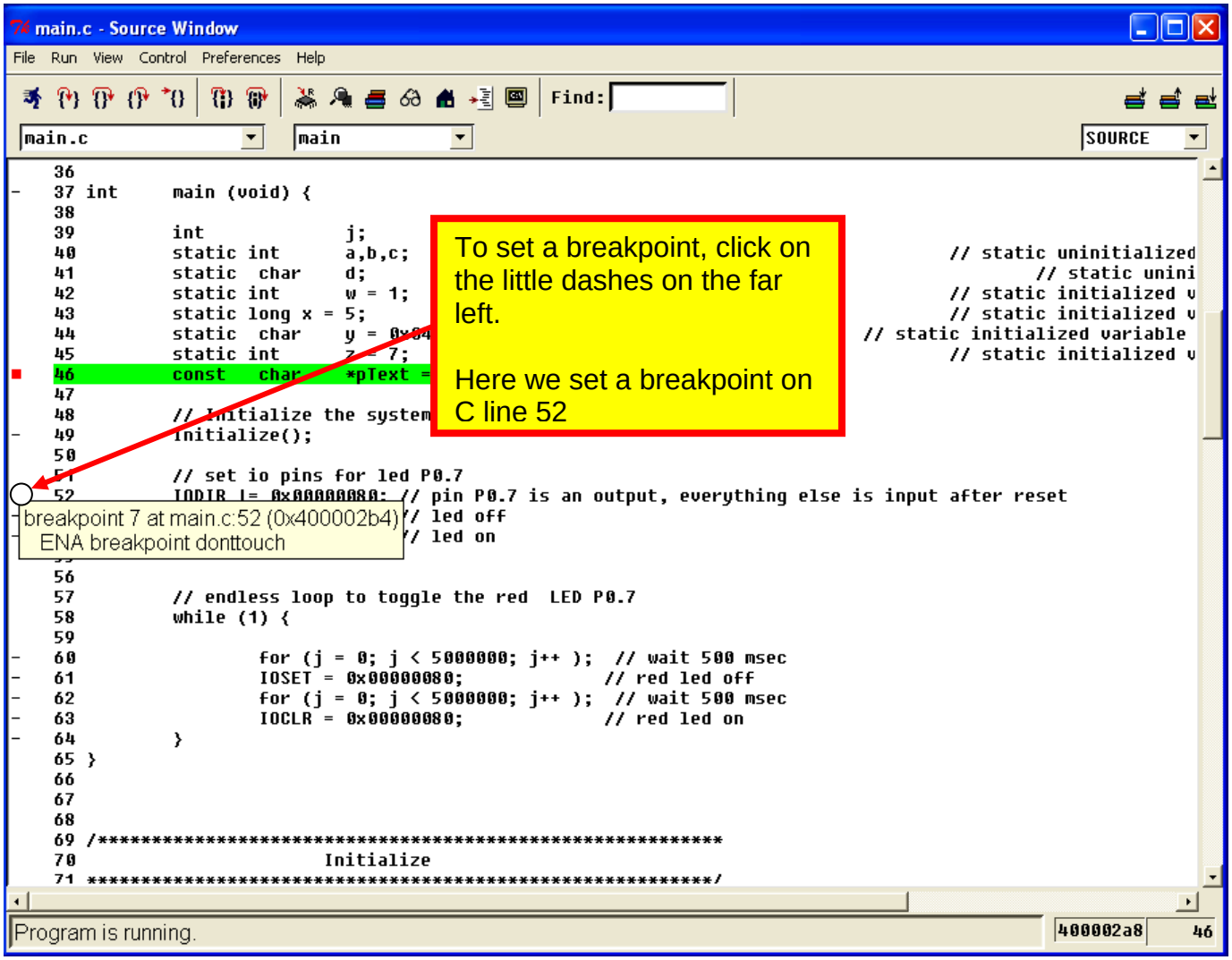


RUN **Step Into** **Step Over** **Step Out of Func** **Continue** **Step Into ASM** **Step Over ASM**

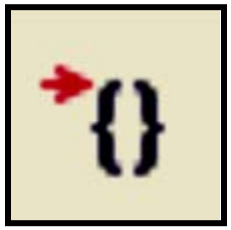
RUN **(STOP)**		Starts debugging – breaks at main() **Note: use this only after downloading.** Stops a running application
STEP INTO		Steps one C executable line If at a function call, it will step INTO the function.
STEP OVER		Steps one C executable line If at a function call, it will step OVER the function
STEP OUT OF FUNCTION		Steps out of a C Function
CONTINUE		Resumes execution to the next breakpoint, watchpoint or exception If no breakpoints are encountered, it will run continuously
STEP INTO ASM		Steps one Assembler instruction If at a subroutine branch, it will enter the subroutine
STEP OVER ASM		Steps one Assembler instruction If at a subroutine branch, it will step over the branch to the next instruction.

G. Set a Breakpoint

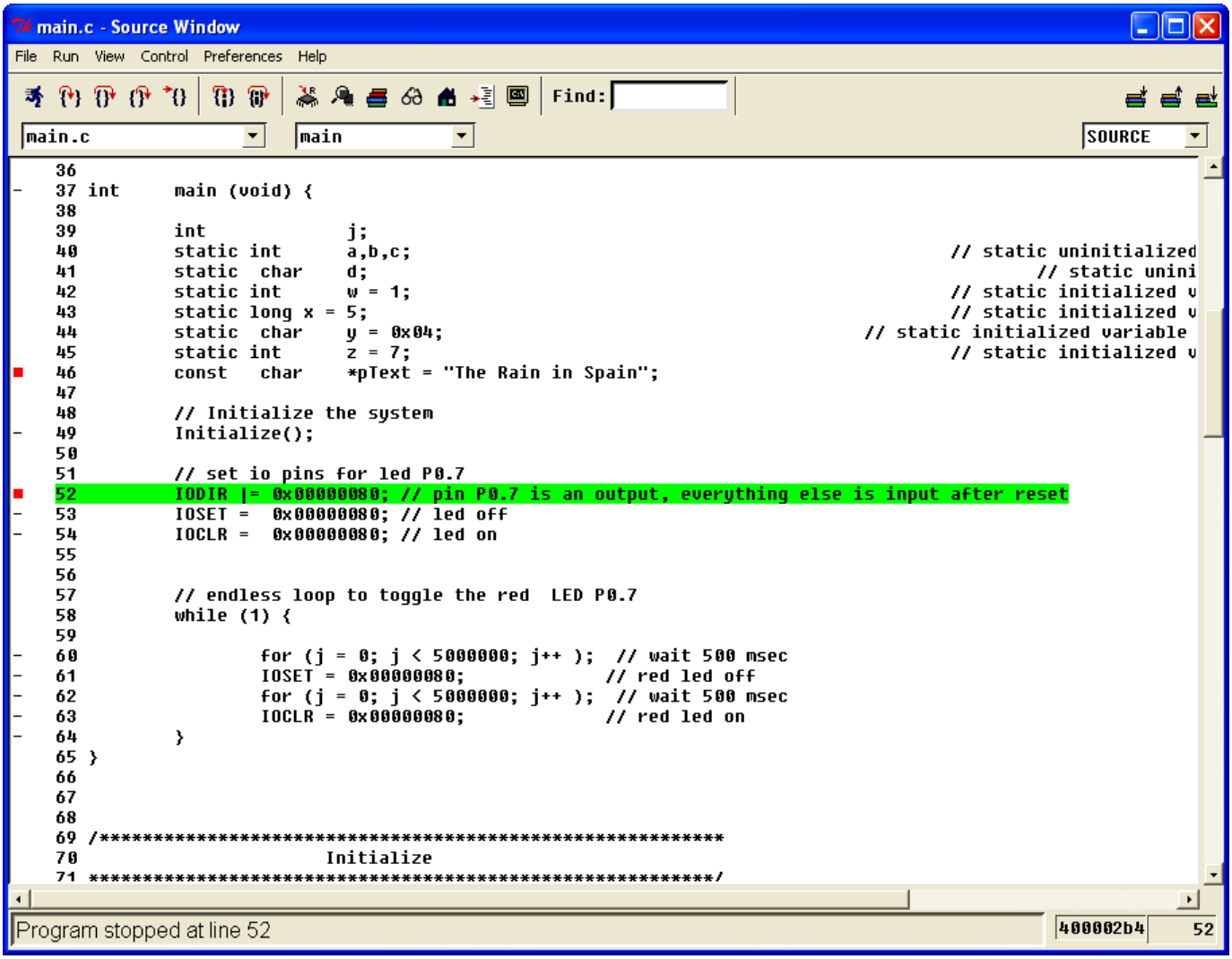
On the far left of the source screen, you'll see a series of dashes. These indicate C executable lines where you can set a breakpoint. Just cursor over to one of them and the cursor will transform into a little circle. Click on the dash and a breakpoint will be set (a little red box indicates this). Clicking again will remove the breakpoint.



Now click on the Continue button to execute from main() to the breakpoint.

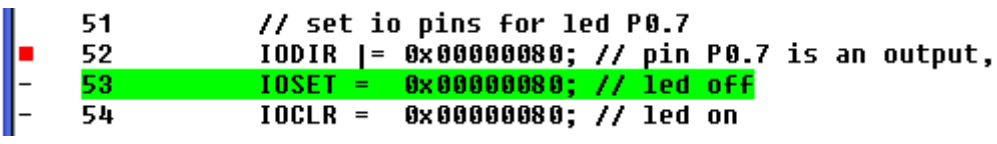


The green highlight at line xx indicates that we've hit a breakpoint. The status line at the bottom left also indicates this: **Program stopped at line 52**.

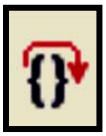


H. Step Through a Few Lines of Code

Click the "STEP OVER" button, the program executes one line, stopping on the highlighted line.
The green highlight indicates the line to be executed next.



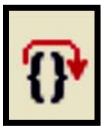
Click the “**STEP OVER**” button, the program executes one line. Note that the LED goes off.



```
51          // set io pins for led P0.7
52          IODIR |= 0x00000080; // pin P0.7 is an output,
53          IOSET =  0x00000080; // led off
54          IOCLR =  0x00000080; // led on
55
```

Click the “**STEP OVER**” button, the program executes one line. Note that the LED goes on.

Also note that the program executes to the next line with a “dash” on the far left.



```
51          // set io pins for led P0.7
52          IODIR |= 0x00000080; // pin P0.7 is an output, everything el
53          IOSET =  0x00000080; // led off
54          IOCLR =  0x00000080; // led on
55
56
57          // endless loop to toggle the red  LED P0.7
58          while (1) {
59
60                  for (j = 0; j < 5000000; j++ );  // wait 500 msec
61                  IOSET = 0x00000080;              // red led off
62                  for (j = 0; j < 5000000; j++ );  // wait 500 msec
63                  IOCLR = 0x00000080;              // red led on
64          }
65 }
```

I. Instant Inspection of any Variable or Data Structure

Anytime the Insight Debugger is stopped, you can maneuver the cursor over any variable name and it will display its current value. If the variable is a C Structure, then a “+” sign will appear that will expand the structure display if clicked.

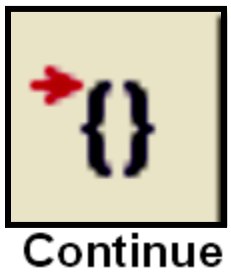
```
36
37 int      main (void) {
38
39          int             j;
40          static int      a,b,c;
41          static  char    d;
42          static int      w =
43          static long x = 5;
44          static  char    y = 0x04;
45          static int      z = 7;
46          const   char    *pText = "The Rain in Spain";
47
48          // Initialize the system
49          Initialize();
50
51          // set io pins for led P0.7
52          IODIR |= 0x00000080; // pin P0.7 is an output,
```

b=0

If you hold the tip of the cursor over the variable b, a pop-up window will appear with its current value.

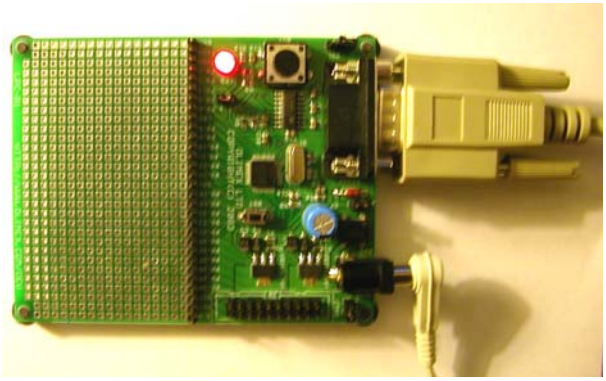
J. Resuming Execution

To let the program take off and start running continuously or run to the next breakpoint, you use the "**Continue**" button.

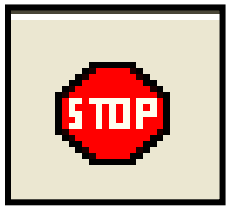


Now you should see the LED Blinking. The "**Run**" button has changed shape into a "**Stop**" button.

Let's show that Bigfoot photograph again of the Olimex LPC-P2106 board executing the blinker application!



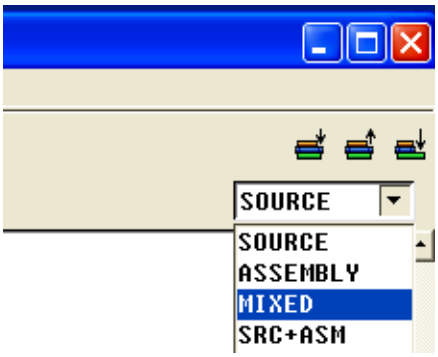
Click on "**Stop**" to terminate execution.



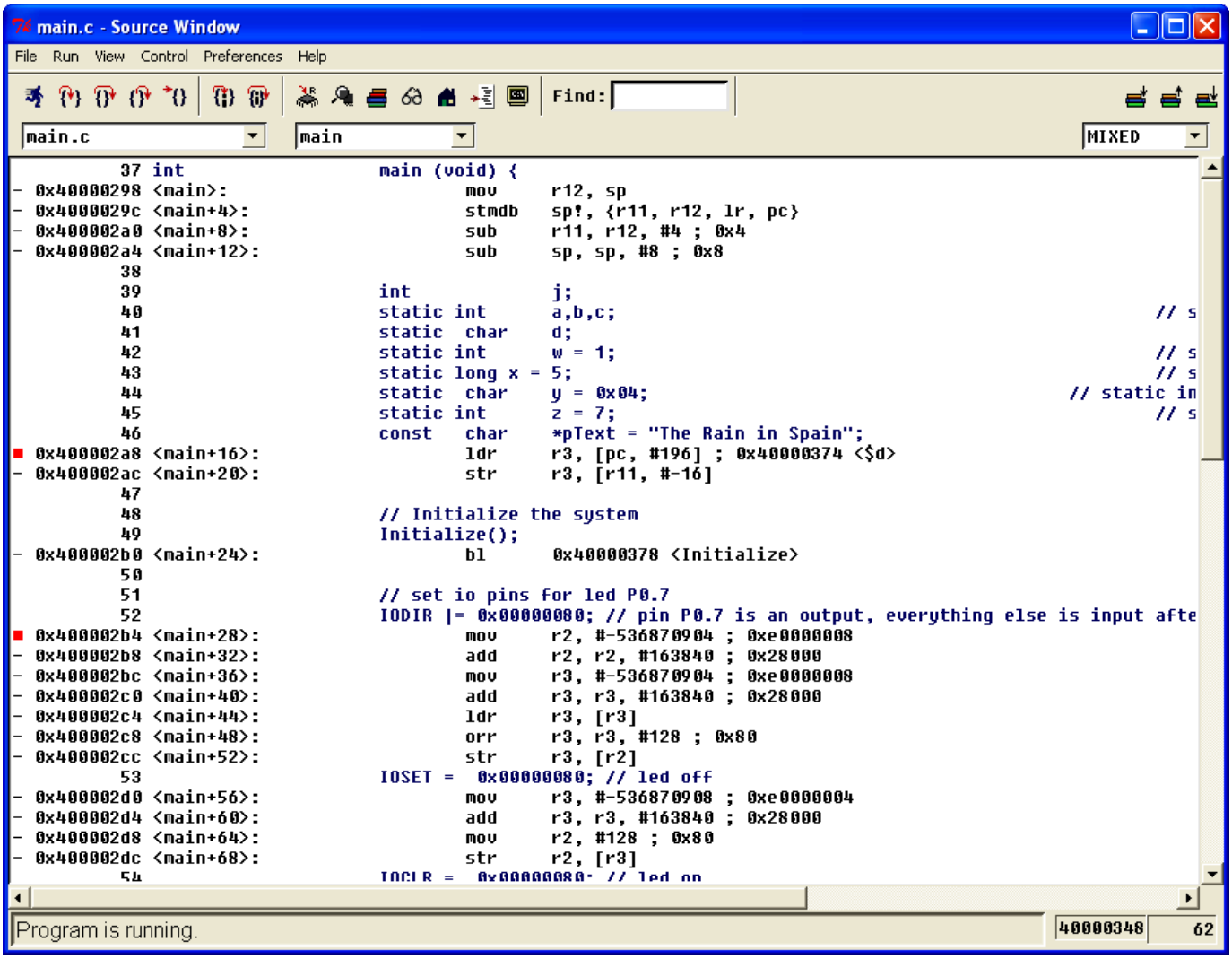
K. Looking at Assembler Code

You can view the code as simple C Source, assembler source or a mixed version.
On the upper right of the Insight main screen is a pull-down list of these display formats.

Click on "**Mixed**" to see the combination C Source intermixed with Assembler source display.



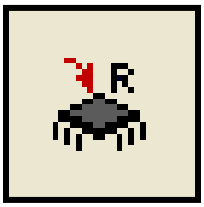
The resultant display looks like this.

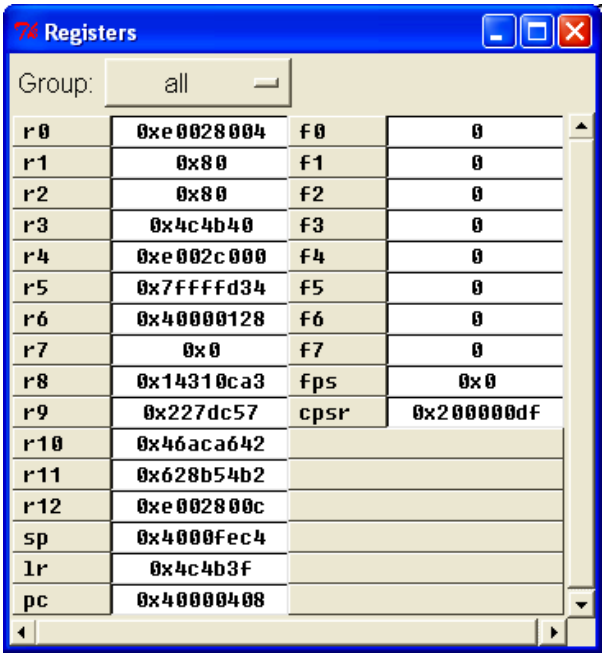


You can use the two assembler step buttons to iterate through the assembler code.

L. Inspecting the ARM Registers

You can see the ARM registers by clicking on the "**Registers**" toolbar button.



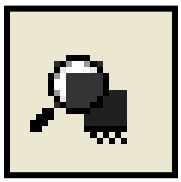


You can edit any register by clicking on it and typing a new value (be careful about this one!).

There's a right-click pop-up menu that lets you change to decimal display, add a register to the watch window, etc.

M. Displaying the Contents of Memory.

A nice screen dump of the memory is available with the "**Memory**" toolbar button:



Here I've entered the address 0x400004B4, the address of the string "The Rain in Spain".

Memory

Addresses

Address 0x400004b4 Target is LITTLE endian

	0	4	8	C	ASCII
0x400004b4	0x20656854	0x6e696152	0x206e6920	0x69617053	The Rain in Spai
0x400004c4	0x0000006e	0x00030002	0x00000006	0x00000007	n.....
0x400004d4	0x00000004	0x00000005	0x00000001	0x00000000	.....
0x400004e4	0x00000000	0x00000000	0x00000000	0x00000000	.....
0x400004f4	0x00000000	0x00000000	0xddb69d29	0x573ad2d8	.....)....:W
0x40000504	0x7dcb1f85	0xda1a014c	0x9ac45a5b	0x8ca084ba	...}L...[Z.....
0x40000514	0x7b73d4ae	0x0a001416	0xb2b0e27a	0x7a13b0dc	..s{....z.....z
0x40000524	0xb2b255b7	0xa821cb42	0xc6995c65	0x7b2148f2	.U..B.!.e\...H!{
0x40000534	0x4eb09f07	0x47620862	0xae2dfbf6	0xa54f1289	...Nb.bG..-...O.
0x40000544	0xdab99bff	0x4e4958d6	0x8fbf721c	0x8848f170	.....XIN.r..p.H.
0x40000554	0x76b1f9b9	0xe611c111	0x5fe9a898	0xda1f719c	...v....._.q..
0x40000564	0x283f6e55	0x80338fe0	0x24ff4cf3	0x3011ecaa	Un?(..3..L.$...0

Click on the "**Addresses**" pull-down menu and then select "**Preferences**." This will allow you to change to a byte-display. If you're confused, remember that the byte display doesn't show the effect of "**little endian**" memory organization.

Memory

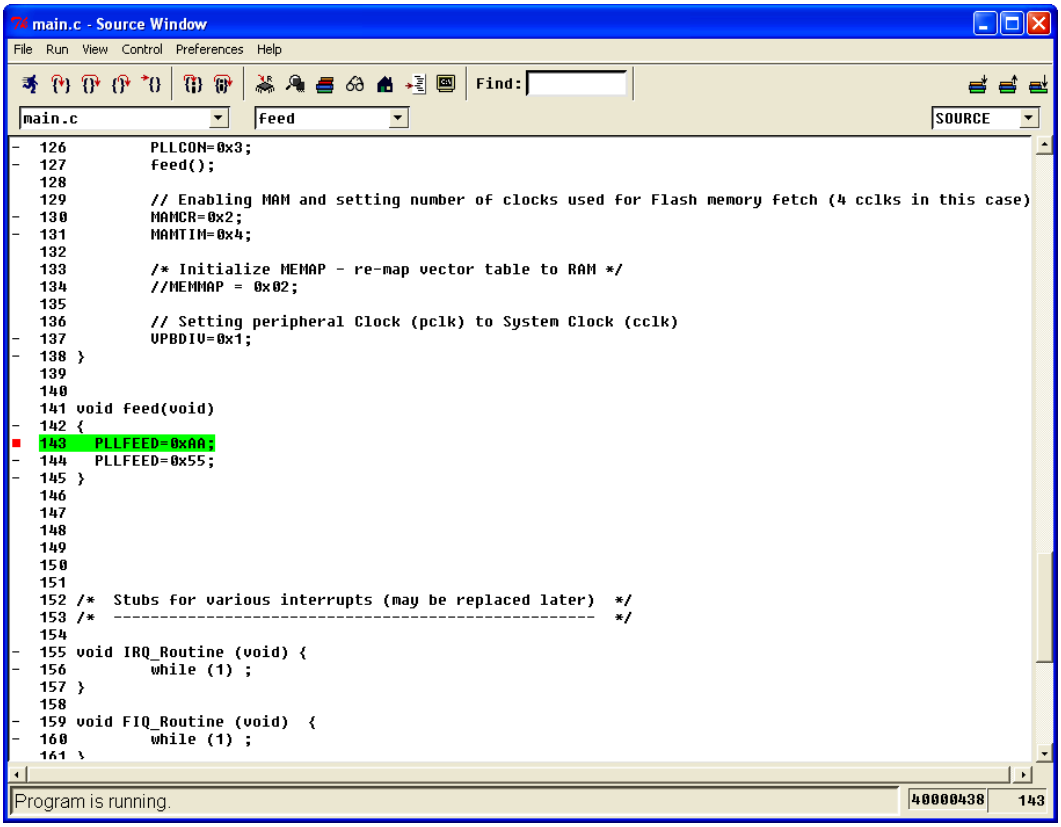
Addresses

Address 0x400004b4 Target is LITTLE endian

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
0x400004b4	0x54	0x68	0x65	0x20	0x52	0x61	0x69	0x6e	0x20	0x69	0x6e	0x20	0x53	0x70	0x61	0x69	The Rain in Spai
0x400004c4	0x6e	0x00	0x00	0x00	0x02	0x00	0x03	0x00	0x06	0x00	0x00	0x00	0x07	0x00	0x00	0x00	n.....
0x400004d4	0x04	0x00	0x00	0x00	0x05	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00	.....
0x400004e4	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	.....
0x400004f4	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x29	0x9d	0xb6	0xdd	0xd8	0xd2	0x3a	0x57	.....)....:W
0x40000504	0x85	0x1f	0xcb	0x7d	0x4c	0x01	0x1a	0xda	0x5b	0x5a	0xc4	0x9a	0xba	0x84	0xa0	0x8c	...}L...[Z.....
0x40000514	0xae	0xd4	0x73	0x7b	0x16	0x14	0x00	0x0a	0x7a	0xe2	0xb0	0xb2	0xdc	0xb0	0x13	0x7a	..s{....z.....z
0x40000524	0xb7	0x55	0xb2	0xb2	0x42	0xcb	0x21	0xa8	0x65	0x5c	0x99	0xc6	0xf2	0x48	0x21	0x7b	.U..B.!.e\...H!{
0x40000534	0x07	0x9f	0xb0	0x4e	0x62	0x08	0x62	0x47	0xf6	0xfb	0x2d	0xae	0x89	0x12	0x4f	0xa5	...Nb.bG..-...O.
0x40000544	0xff	0x9b	0xb9	0xda	0xd6	0x58	0x49	0x4e	0x1c	0x72	0xbf	0x8f	0x70	0xf1	0x48	0x88	.....XIN.r..p.H.
0x40000554	0xb9	0xf9	0xb1	0x76	0x11	0xc1	0x11	0xe6	0x98	0xa8	0xe9	0x5f	0x9c	0x71	0x1f	0xda	...v....._.q..
0x40000564	0x55	0x6e	0x3f	0x28	0xe0	0x8f	0x33	0x80	0xf3	0x4c	0xff	0x24	0xaa	0xec	0x11	0x30	Un?(..3..L.$...0

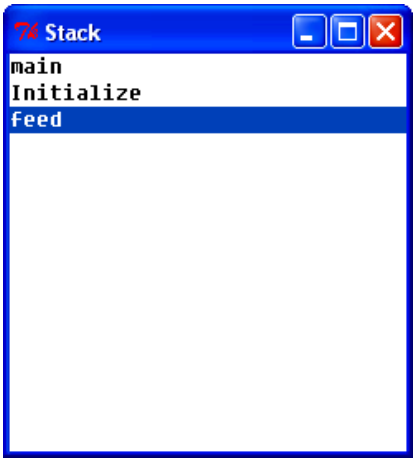
N. Inspecting the Stack Frame

I ran the application with a breakpoint set inside the feed() routine.



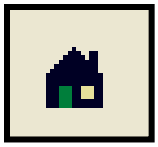
Click on the "**Stack Window**" toolbar button to inspect the call stack.

The **Stack Window** shows that **main()** called **Initialize()**. **Initialize()** then called **feed()**.

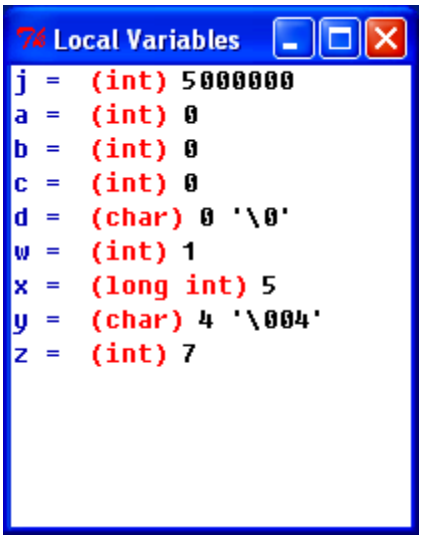


O. Inspecting Local Variables

In addition to just maneuvering the cursor over a variable name to see its value, you can also bring up the "**Local Variables**" display box by clicking on the "**Local Variables**" toolbar button:



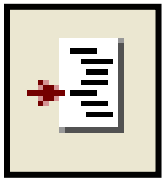
Remember that Local Variables are your "stack-based" variables defined within a function. There is no window for Global Variables; you'll have to create Watch Windows for them.



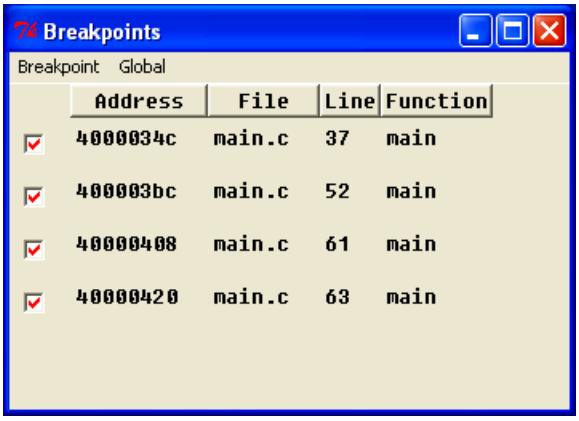
P. Inspecting Breakpoints

You probably know that breakpoints are set by clicking on the little dashes to the far left of the Insight Source Window (click again and the breakpoints are removed).

Insight also has a breakpoint summary window which can be accessed by clicking the "Breakpoints" button on the Insight toolbar.



The “Breakpoints” window shows every breakpoint you’ve created.

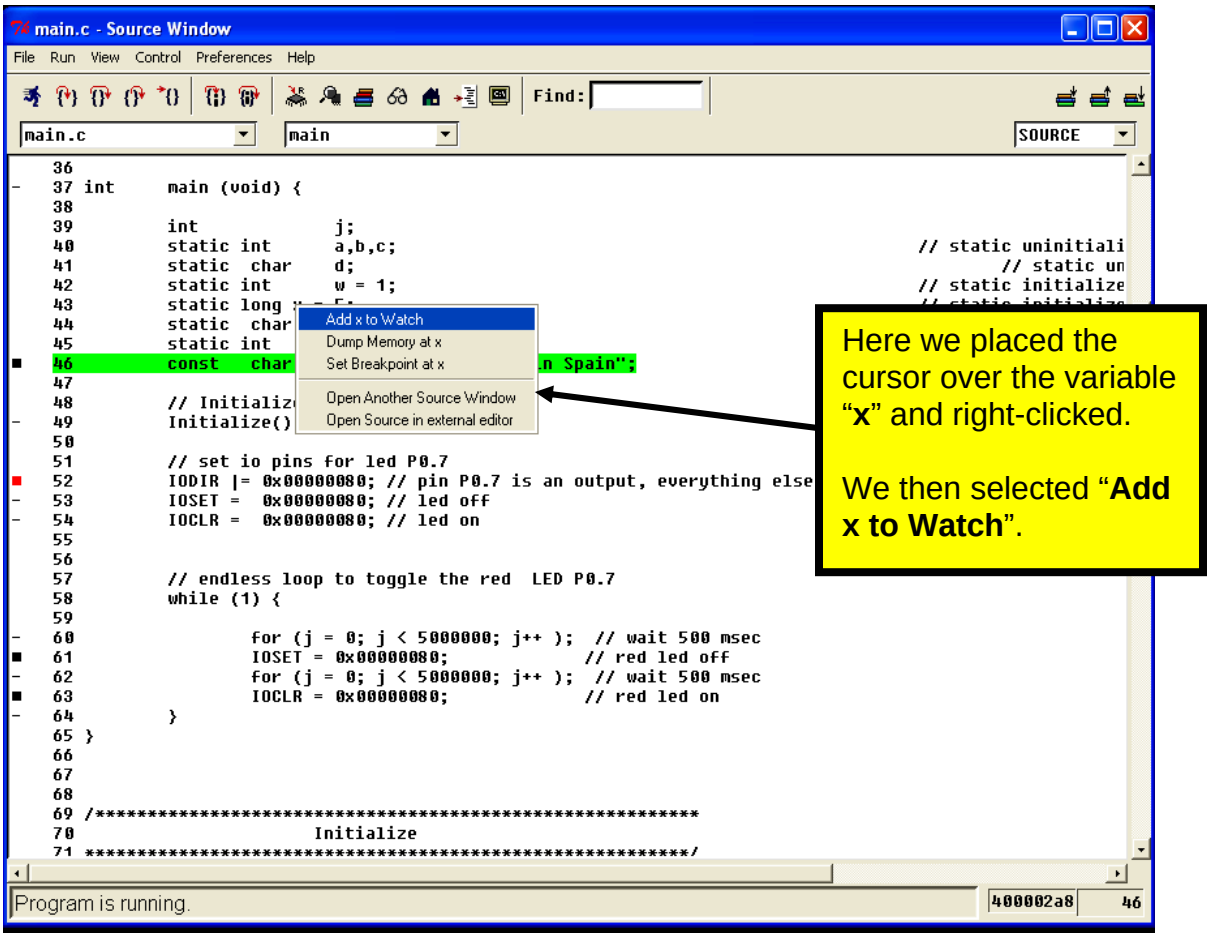


Using the pull-down menus and the right-click menu, you can easily disable any or all breakpoints, remove them completely, etc.

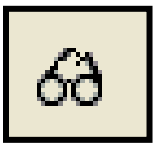
Q. Watch Window

The **Watch Window** allows you to create a display of all your favorite variables that will be displayed whenever the application has stopped.

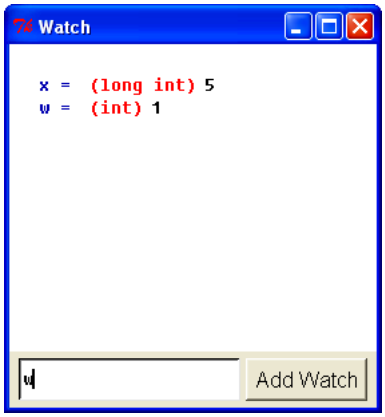
There are several ways to add a variable to the **Watch Window**. The most convenient method is to hover over a variable with the cursor, right-click and select “**Add to Watch**.”



The **Watch Window** itself can be displayed by clicking on the "**Watch Expressions**" toolbar button.



Another easy way to enter variables into the **Watch Window** is to just type the variable name into the text box at the bottom. When you hit the "**enter**" key, it will appear in the list of watched variables. Here I typed in the variable name "**w**".



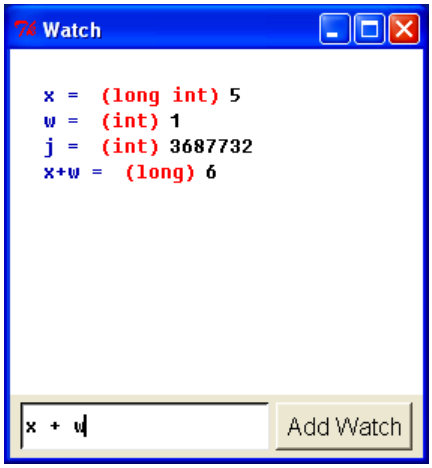
WARNING

Do not close the watch window using the

You will loose your setup.

Just minimize it using

You can also type expressions into the **Watch Window**. For example, you could type in **x + w** and this will be displayed.



Structured variables will have **+** symbols that can be clicked on to expand the structure so you can see all the inner bits.

Remember that the **Watch Window** only updates when the **Insight Debugger** hits a breakpoint or stops.

R. Entering GDB Commands

For those who know the original text-only GDB debugger well, you can open a "**GDB Console**" and start typing.

For example, to ask GDB to display the variable **j** in the function main(); type the following command into the **Console**" window.

S. Some Insight Observations

To **restart** the application from the beginning, I recommend **downloading** and hitting the "**Run**" button again. It will not ask you for the connection details.

Click the "close" button at the top right when you're finished debugging. This will terminate and remove **Insight** and also terminate **OCDremote**.

If you crash, you'll probably have to use the Windows Task Manager (**ctrl-alt-del**) to stop **OCDremote**.

23 The Author Sounds Off

This tutorial was designed for students and hobbyists; those with limited funds. It described in great detail how to download and install all the component parts of a complete ARM software development system and gave two simple code examples to try out. Of course, the beauty of this is that it's completely free.

If you are a professional engineer attempting to build an ARM development system with these techniques, you have a fool for a chief engineer. The professional compilers such as IAR, Rowley, and Keil etc. are more efficient, generally bug free and interface seamlessly with debuggers. They allow debugging with either ram or flash executables and flash programming is usually accomplished with a single click. You also have telephone support with these systems. These professional packages save your company time and money in the long run.

This tutorial was written for students and grown up "kids at heart"; its purpose is to foster their interest in computer science and electrical engineering. It's a shame that the big players like Microsoft, Kiel, Borland and others don't develop a "student/hobbyist" version of their software development packages, priced at a give-away point that a third world high school student could afford. Bill Gates has criticized my country's school system for not developing enough computer scientists and engineers; why not provide a "non-commercial" version of his Visual Studio for students (and provide code targeting for every popular microprocessor being sold today)?

I am not happy with the debugger I described in the tutorial. The Wiggler/Insight combination works only for RAM-based applications and thus limits software to less than 64K. It's extremely slow and a bit unreliable. Professional USB or Ethernet-based debuggers are very expensive and out of the price range of hobbyists.

A better solution might be an Olimex LPC-P2106 board outfitted with one of those Spark Fun CP2106 USB-to-Serial converters to accept GDB debugger serial protocol from the PC and convert it into ARM JTAG signals. The JTAG signals are documented at the ARM web site and the GDB serial protocol is fully specified at the GNU web site. The LPC2106 could be programmed to know the device ID of the LPC2000 series microcontroller it is fitted to and convert any download files into flash programming commands if needed. Even software breakpoints can be handled by reading an 8k block of code, changing one word and flash programming it back into the target. Just using parts from Olimex and Spark Fun, this could cost less than $100. The software programming job would be rather extensive. Still, it'd make a very nice open-source project. I'm thinking about it.

I'm not finished writing tutorials. My next tutorial will involve using ARM interrupts and how to design and implement I2C port expanders to interface to LCD displays and keypads. Later tutorials will go into motion control, free real-time operating systems and other hardware projects. Stay tuned, just like you, I'm just getting started!

24 About the Author

Jim Lynch lives in Grand Island, New York and is a Project Manager for Control Techniques, a subsidiary of Emerson Electric. He develops embedded software for the company's industrial drives (high power motor controllers) which are sold all over the world.

Mr. Lynch has previously worked for Mennen Medical, Calspan Corporation and the Boeing Company. He has a BSEE from Ohio University and a MSEE from State University of New York at Buffalo. Jim is a single Father and has two children who now live in Florida and Nevada. He has two brothers, one is a Viet Nam veteran in Hollywood, Florida and the other is the Bishop of St. Petersburg, also in Florida. Jim plays the guitar and is collecting woodworking machines for future projects that will integrate woodworking and embedded computers. Lynch can be reached via e-mail at: lynchzilla@aol.com

24 Some Books That May Be Helpful

The following is a short compendium of books that I've found helpful on the subject of ARM microprocessors and the GNU tool chain. I've reproduced the Amazon.com data on them.

GCC: The Complete Reference

by Arthur Griffith "The GNU Compiler Collection (GCC) is the most important piece of open source software in the world..." (more)
SIPs: instruction scheduling parameters, builtin apply, execute the configure script, release egcs, call insn (more)

List Price: ~~$59.99~~
Price: **$39.59** and this item ships for **FREE with Super Saver Shipping**. See details
You Save: $20.40 (34%)
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.
Only 5 left in stock--order soon (more on the way).

57 used & new available from $8.70
Edition: Paperback

An Introduction to GCC

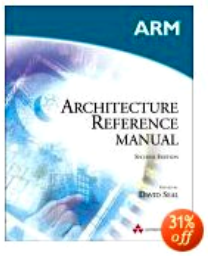
by Brian J. Gough, Richard M. Stallman (Foreword) "The purpose of this book is to explain the use of the GNU C and C++ compilers, gcc and g++..." (more)
SIPs: void hello, math library libm, default gcc, object file containing, options gcc (more)

List Price: ~~$19.95~~
Price: **$13.57** and eligible for **FREE Super Saver Shipping** on orders over $25. See details
You Save: $6.38 (32%)
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.

14 used & new available from $13.16
Edition: Paperback

ARM Architecture Reference Manual (2nd Edition)

by David Seal



List Price: ~~$57.99~~
Price: **$40.24** and this item ships for **FREE with Super Saver Shipping**. See details
You Save: $17.75 (31%)
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.

Want it delivered Tuesday, June 21? Order it in the next 44 hours and 57 minutes, and choose **One-Day Shipping** at checkout. See details

39 used & new available from $28.00
Edition: Paperback

ARM System-on-Chip Architecture (2nd Edition)

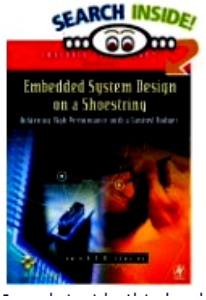
by Steve Furber

List Price: ~~$44.99~~
Price: **$29.39** and this item ships for **FREE with Super Saver Shipping**. See details
You Save: $15.60 (35%)
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.

Want it delivered Tuesday, June 21? Order it in the next 41 hours and 55 minutes, and choose **One-Day Shipping** at checkout. See details

64 used & new available from $20.00
Edition: Paperback

Embedded System Design on a Shoestring (Embedded Technology Series)
by Lewin Edwards "There exist a large body of literature focused on teaching both general embedded systems principles and design techniques, and tips and tricks for specific microcontrollers..." (more)
SIPs: current output section, bss end, qdb stubs, sourcecode files, clear bss (more)



List Price: $49.95
Price: $49.95 and this item ships for **FREE with Super Saver Shipping**. See details
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.

11 used & new available from $47.97
Edition: Paperback

The ARM documentation can be downloaded free from the ARM web site.

http://www.arm.com/documentation/

The Philips Corporation has extensive documentation on the LPC2000 series here:

http://www.semiconductors.philips.com/pip/LPC2106.html

All the GNU documentation, in PDF format, is maintained by, among others, the University of South Wales in Sidney, Australia. I found the GNU assembler and linker manuals very readable; the GNU C compiler manuals are very difficult.

http://dsl.ee.unsw.edu.au/dsl-cdrom/gnutools/doc/

Of course, the bookstore is full of Eclipse books but they are all about the JAVA toolkit. So far, no one has published anything on the CDT plugin.

Finally, avail yourself of the many discussion groups on the web:

www.yahoo.com	GNUARM group LPC2000 group
www.sparkfun.com	tech support forum
www.newmicros.com	tech support forum
www.eclipse.org	**C/C++ Development Tools User Forum**

HAVE FUN, EVERYBODY!