

ARM Cross Development with Eclipse Version 2

By: James P. Lynch
September 10, 2005

Preface to Version 2

When I developed this tutorial in June 2005, Eclipse was at release 3.0.2 and the CDT (C Development Toolkit) was at version 2.1.1. Preview versions of the next releases of these components were available at that time, but I prefer to stick with official releases in my tutorials for beginners.

Also, the Eclipse debugger worked well when the application was targeted for a PC platform, but it was nearly useless in a “remote debugging” environment where the target is, for example, an ARM microprocessor connected through a JTAG interface. A Norwegian company, Zylín, developed a “custom” version of CDT that modified the debugger for proper operation in a “remote debugging” environment. I found the Zylín version that I downloaded in June 2005 to be a bit unreliable and fairly easy to lock up. For that reason, I wrote the original tutorial with the classic Redhat **Insight** debugger as the principal debugging agent; it’s simple and reliable.

In just two months, Eclipse has released Version 3.1 and CDT released Version 3.0 as a companion edition to the new Eclipse release. Zylín also updated their “custom” version of CDT 3.0 (an effort done with the cooperation of the Eclipse CDT team).

I tested the new Zylín CDT 3.0 release and found that the remote debugging feature now works fairly well. Therefore, it seemed to be a good idea to republish the “**ARM Cross Development with Eclipse**” tutorial with the new Eclipse and Zylín/CDT components.

The original tutorial is still usable; Eclipse and CDT will always have their legacy versions archived. You will have to browse their web sites to find them.

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 than \$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 SPITM.
- 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 mm²) 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**”. The Norwegian company Zylind has created a custom version of CDT that enables the debugger to work better with cross-development applications; 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.

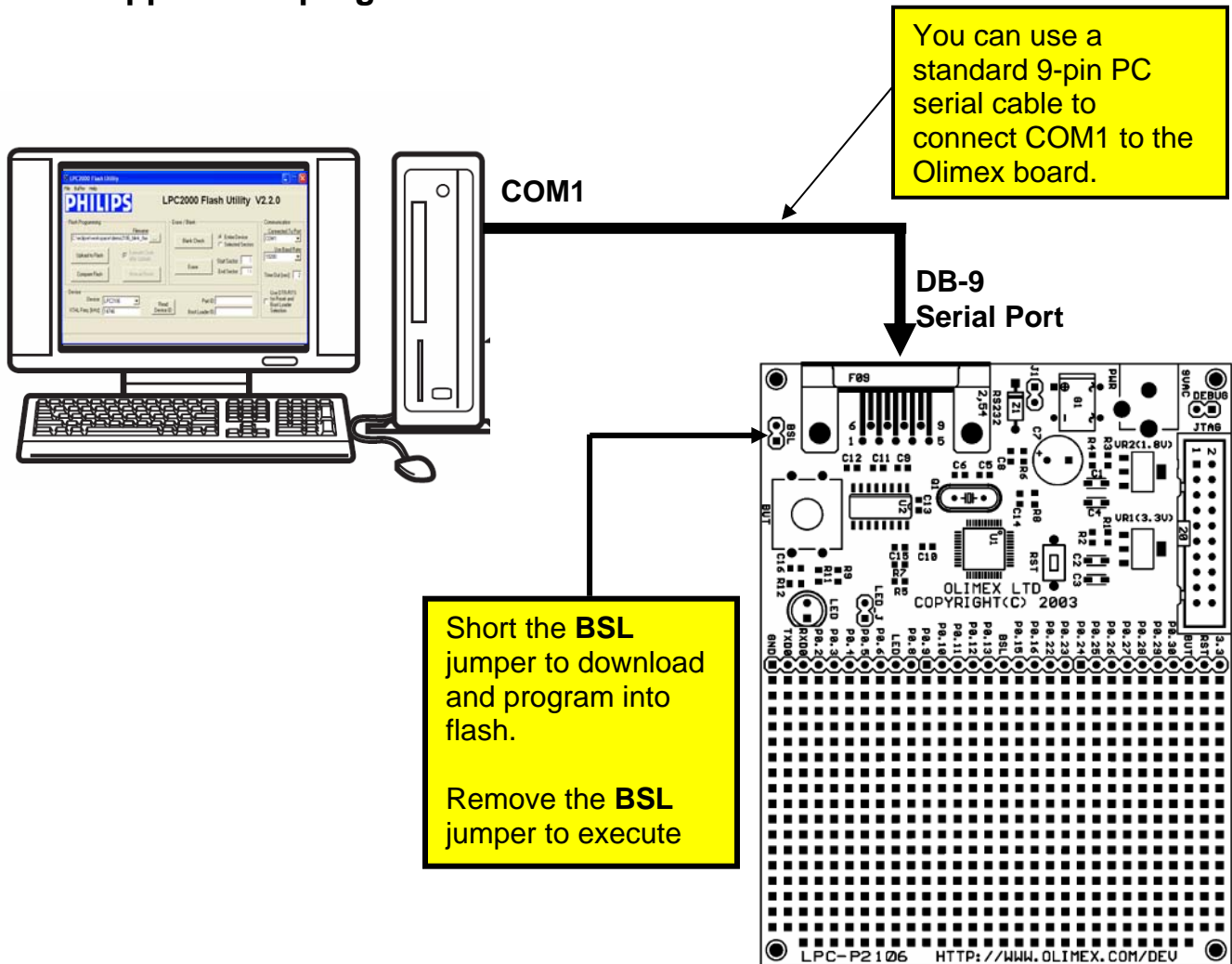
- The GNU tools do not currently generate as efficient code as the professional compilers.
- The Eclipse CDT Debugger cannot set a software breakpoint in FLASH since it can’t erase and reprogram the FLASH.

- 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**.

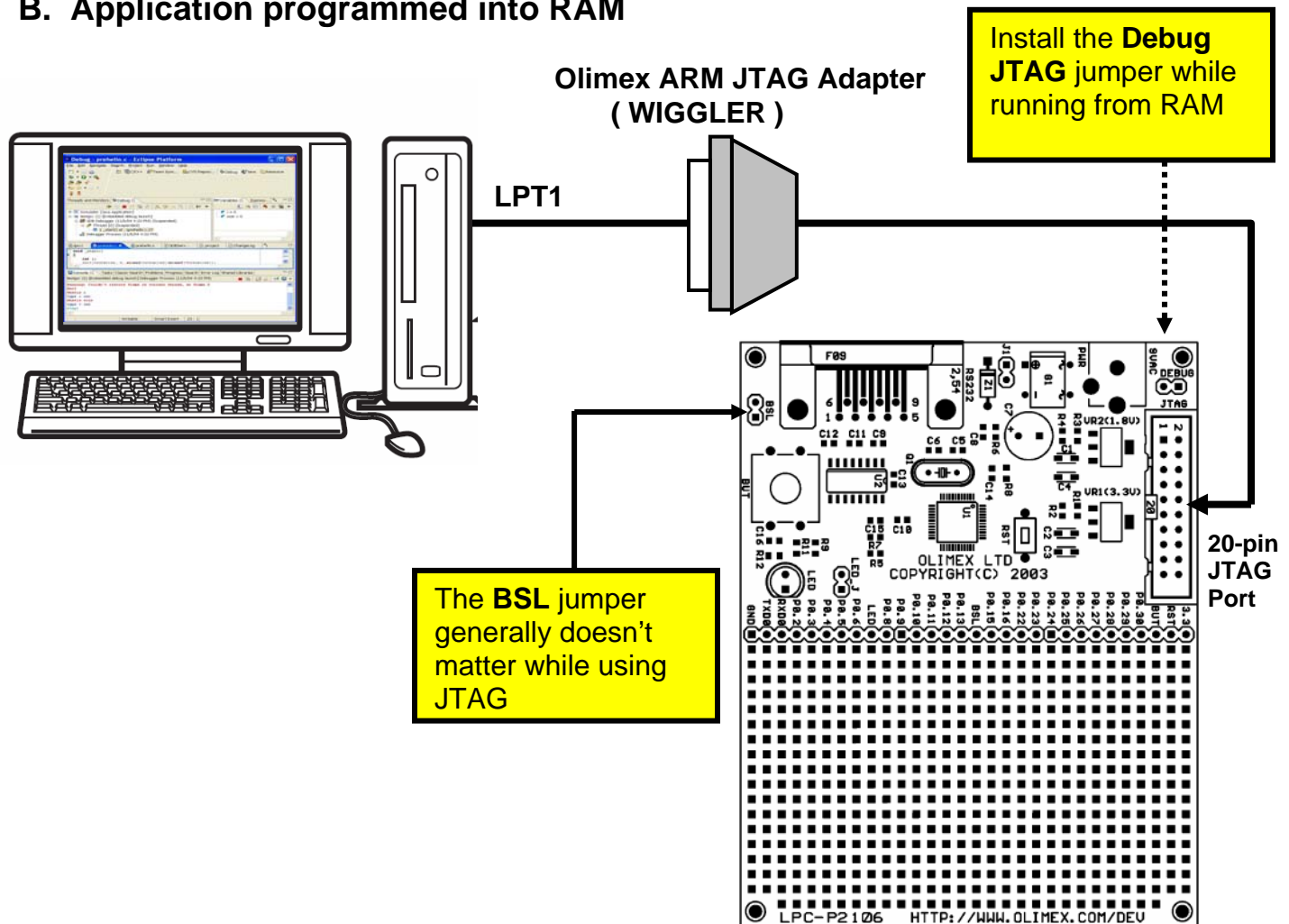
The Philips In-System Programming (**ISP**) utility is started within Eclipse and will download your hex file and program the flash memory through the standard COM1

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 **CDT** debugger (started from within Eclipse) communicates with the Macraigor **OCDRemote** program that operates the JTAG port using the **Wiggler**. With the **CDT** 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.

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 (Zylin custom version)**
4. **CYGWIN GNU C++/C Compiler and Toolset for Windows**
5. **GNUARM GNU C++/C Compiler for ARM Targets**
6. **Philips Flash Programmer for LPC2100 Family CPUs**
7. **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.



The screenshot shows the Sun Microsystems website. The main header features the Sun Microsystems logo and the StorageTek logo. Below the logos, there is a large banner with the text "THE LEADER IN DATA MANAGEMENT" and a sub-headline "Sun Microsystems Completes Acquisition of StorageTek to deliver excellent service and value to our shared customers – and a broad portfolio of open products to meet your data management needs." To the right of the banner, there are three promotional boxes: "Blogging the Enterprise", "Summit for Change", and "Roam Free". The navigation bar includes links for "Products", "Downloads", "Services & Solutions", "Support", "Training", and "Research". The "Downloads" menu is open, and "Java 2 Standard Edition" is highlighted with a red circle. Other items in the menu include "Solaris 10", "Developer Tools", "Top Downloads", "New Downloads", "Patches & Updates", and "See All". The footer contains contact information, copyright notice, and a "powered by Sun Microsystems" logo.

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

Sun Developer Network
Products and Technologies Technical Topics

Developers Home > Products & Technologies > Java Technology > J2SE >

J2SE Downloads

Downloads
- Early Access

Reference
- API Specifications
- Documentation
- FAQs
- Code Samples & Apps
- BluePrints
- Technical Articles & Tips
- White Papers
- Third-Party
- Compatibility

- **J2SE 5.0**
- J2SE 1.4.2
- J2SE 1.3.1

Download Archived Releases

Sun maintains a download site for previously released versions of the J2SE platform and related products and are no longer covered by standard support contracts. These downloads are made available as a courtesy resolution.

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

Download Java 2 Platform, Standard Edition 5.0 - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address <http://java.sun.com/j2se/1.5.0/download.jsp>

Sun Developer Network
Products and Technologies Technical Topics

Developers Home > Products & Technologies > Java Technology > J2SE > Core Java > J2SE 5.0 >

J2SE 5.0

Download Java 2 Platform Standard Edition 5.0

Downloads Confused or having trouble downloading or installing? See the download help page.

Reference Supported System Configurations

Community
- Bug Database
- Forums

Learning
- Tutorials & Code Camps
- Online Sessions & Courses
- Instructor-Led Courses
- Course Certification

NetBeans IDE + JDK 5.0 Update 3

netBeans

This distribution of the J2SE Development Kit (JDK) includes NetBeans IDE, which is a powerful integrated development environment platform. More info...
[Download JDK 5.0 Update 3 with NetBeans 4.1 Bundle](#)

JDK 5.0 Update 3 includes the JVM technology

The J2SE Development Kit (JDK) supports creating J2SE applications. More info...
[Download JDK 5.0 Update 3](#)
[Installation Instructions](#) [ReadMe](#) [ReleaseNotes](#)
[Sun License](#) [Third Party Licenses](#)

JRE 5.0 Update 3 includes the JVM technology

The J2SE Runtime Environment (JRE) allows end-users to run Java applications. More info...
[Download JRE 5.0 Update 3](#)
[Installation Instructions](#) [ReadMe](#) [ReleaseNotes](#)
[Sun License](#) [Third Party Licenses](#)

J2SE 5.0 Documentation

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.

developers.sun.com

> search tips | Search: in Developers' Site

The Source for Java Developers

Terms of Use

Please indicate whether you accept or do not accept the following software license agreement(s) by choosing either "Accept" or "Decline" and clicking the "Continue" button.

NOTE: If you do not accept the license agreement for a product you have chosen, you will not be able to purchase or download that product.

LICENSE AGREEMENT

J2SE(TM) Runtime Environment 5.0 Update 2, Download

In order to obtain J2SE(TM) Runtime Environment 5.0 Update 2 you must agree to the software license below.

[Printer Friendly Page](#)

Sun Microsystems, Inc. Binary Code License Agreement

for the JAVA 2 PLATFORM STANDARD EDITION RUNTIME ENVIRONMENT 5.0

SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT"). PLEASE READ THE AGREEMENT CAREFULLY. BY DOWNLOADING OR INSTALLING THIS SOFTWARE, YOU ACCEPT THE TERMS OF THE AGREEMENT. INDICATE ACCEPTANCE BY SELECTING THE "ACCEPT" BUTTON AT THE BOTTOM OF THE

Accept Decline

[Continue](#)

Select the "accept" radio button and click "continue" to proceed.

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

developers.sun.com

> search tips | Search: in Developers' Site

The Source for Java Developers

Download

J2SE(TM) Runtime Environment 5.0 Update 2

NOTE: The list offers files for different platforms - please be sure to select the proper file(s) for your platform. Carefully review the files listed below to select the ones you want, then click the link(s) to download. If you don't complete your download, you may return to the Download Center anytime, sign in, then click the "Download/Order History" link on the left to continue.

How long will it take? [?](#)

Download problems or Questions? See the [Sun Download Center FAQ](#)

J2SE(TM) Runtime Environment 5.0 Update 2	Click below to download
Windows Platform	Windows Offline Installation, Multi-language (jre-1_5_0_02-windows-i586-p.exe, 15.25 MB)
	Windows Online Installation, Multi-language (jre-1_5_0_02-windows-i586-p-iftw.exe, 221.27 KB)
Linux Platform	Linux RPM in self-extracting file (jre-1_5_0_02-linux-i586-rpm.bin, 15.27 MB)
	Linux self-extracting file (jre-1_5_0_02-linux-i586.bin, 15.78 MB)
Solaris SPARC Platform	Solaris SPARC 32-bit self-extracting file (jre-1_5_0_02-solaris-sparc.sh, 19.45 MB)
	Solaris SPARC 64-bit self-extracting file (jre-1_5_0_02-solaris-sparcv9.sh, 8.33 MB)
Solaris x86 Platform	Solaris x86 self-extracting file (jre-1_5_0_02-solaris-i586.sh, 14.44 MB)
Solaris AMD64 Platform	Solaris AMD64 self-extracting file (jre-1_5_0_02-solaris-amd64.sh, 4.72 MB)
Linux AMD64 Platform	Linux AMD64 RPM in self-extracting file (jre-1_5_0_02-linux-amd64-rpm.bin, 14.91 MB)
	Linux AMD64 self-extracting file (jre-1_5_0_02-linux-amd64.bin, 15.41 MB)

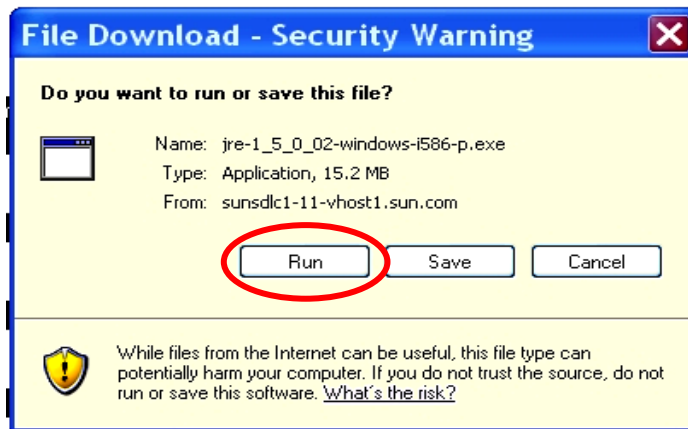
For Customer Service, e-mail SDLC-EXT@sun.com

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

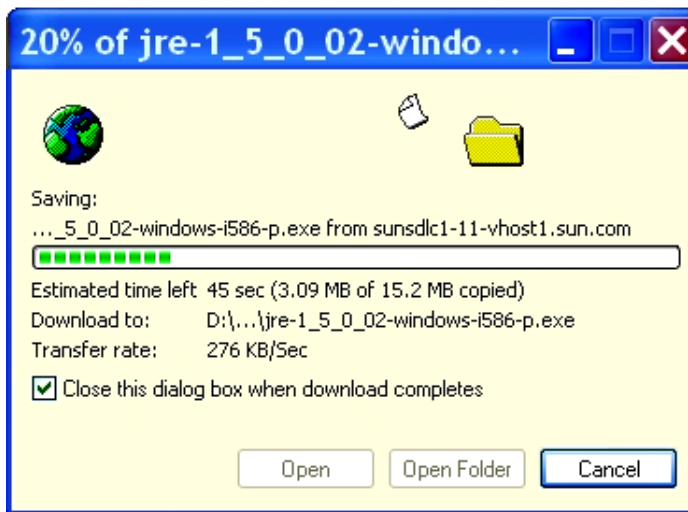
[Windows Offline Installation, Multi-language \(jre-1_5_0_02-windows-i586-p.exe, 15.25 MB\)](#)

[Windows Online Installation, Multi-language \(jre-1_5_0_02-windows-i586-p-iftw.exe, 221.27 KB\)](#)

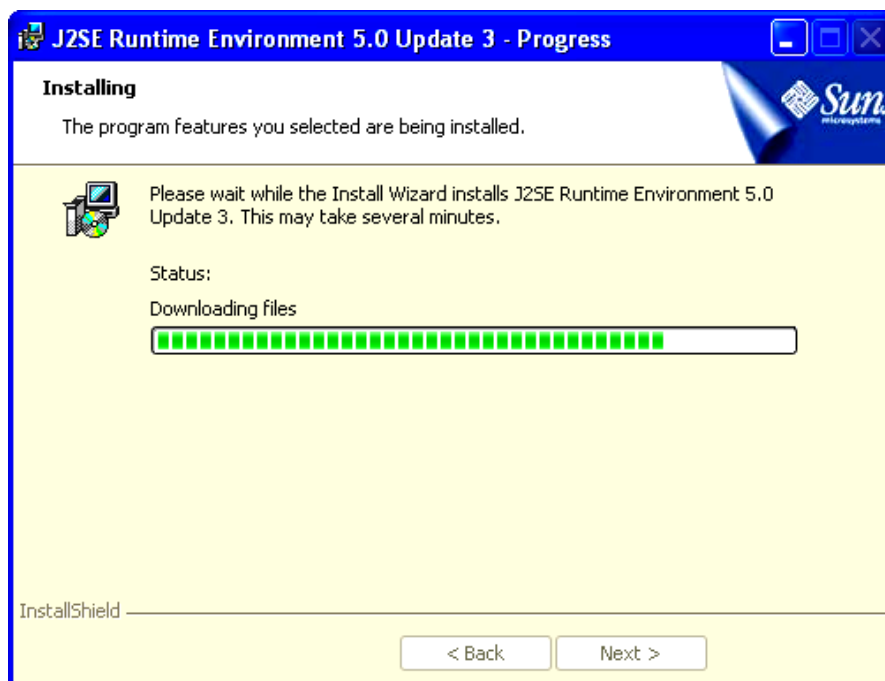
Finally the “file download” window appears. Click on “Run” to download and run the installation.



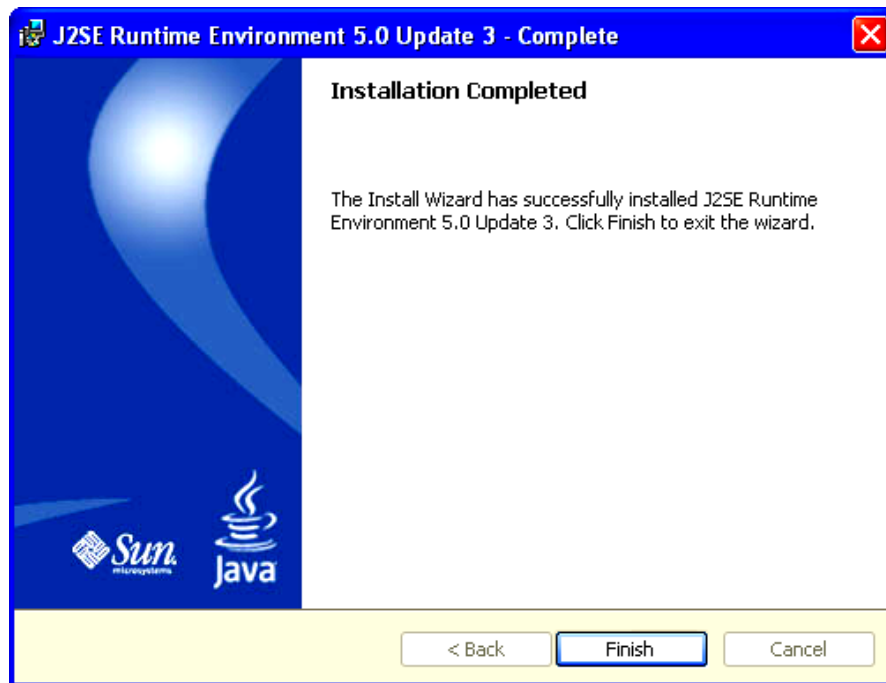
Now the downloading will start.



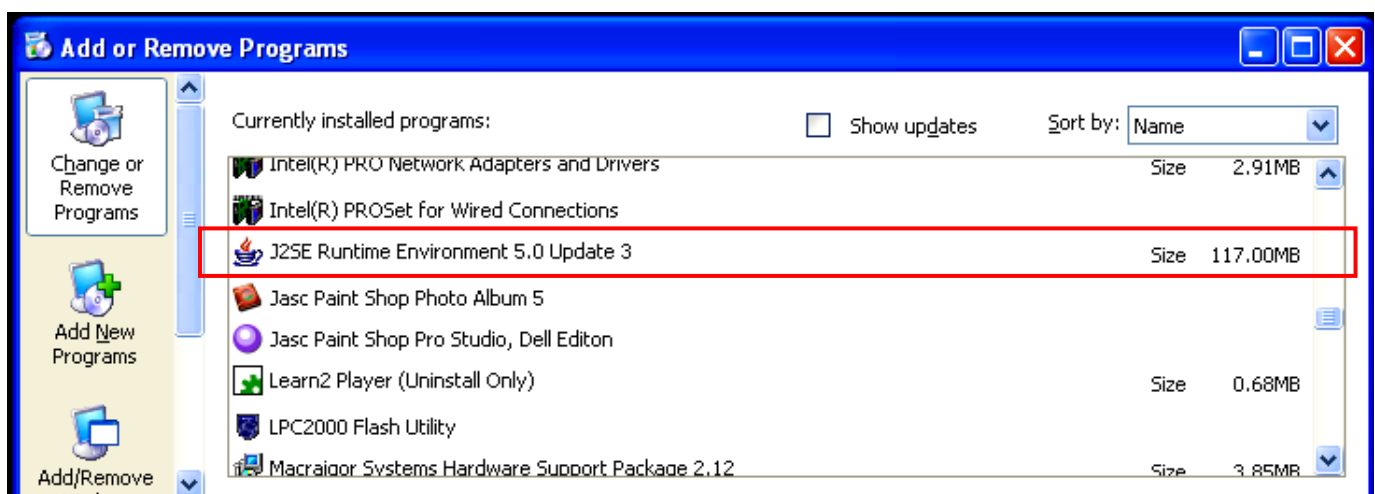
After downloading, the installation will proceed automatically.



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 the latest versions of Eclipse.



The Eclipse download window will appear. Eclipse is constantly being improved and new releases come several times a year. Usually the safest thing to download is the “official” latest release. When this tutorial was created, the latest release was **Eclipse SDK 3.1**

Click on “Downloads” to get things started.



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 3.1** 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).

eclipse

- home
- about us
- projects
- downloads
- articles
- newsgroups
- mailing lists
- community
- search
- bugs

eclipse downloads

Your preferred mirror appears to have this file: [eclipse-SDK-3.0.2-win32.zip](#)

United States

- [United States] [University of Buffalo CSE Department](#)

Please select a mirror for this file: [eclipse-SDK-3.0.2-win32.zip](#)

Africa

- [South Africa] [University of Stellenbosch](#)

Asia

- [Japan] [Japan Advanced Institute of Science and Technology](#)
- [Korea, Republic Of] [Areum](#)

Australia/Oceania

- [Australia] [Pacific Internet](#)

North America

- [Canada] [Groupe d'utilisateurs de Linux de l'UdeS](#)
- [Canada] [Reachable.ca](#)
- [United States] [Calvin College \(ftp\)](#)
- [United States] [TDS Internet Services](#)
- [United States] [University of Buffalo CSE Department](#)
- [United States] [University of Florida](#)
- [United States] [WMW WEB inc.](#)

South America

- [Brazil] [Eclipse@Rio, PUC-Rio](#)

Main Download Site

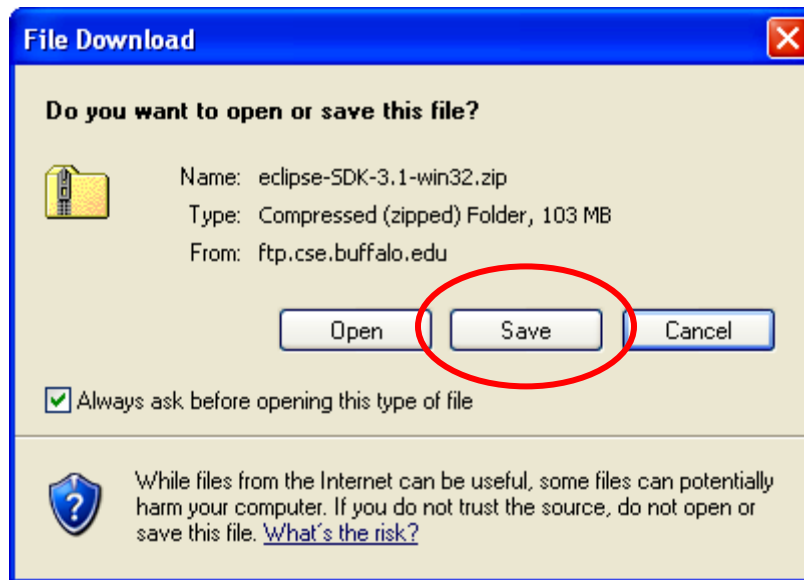
Canada

- [Main eclipse.org downloads area](#)

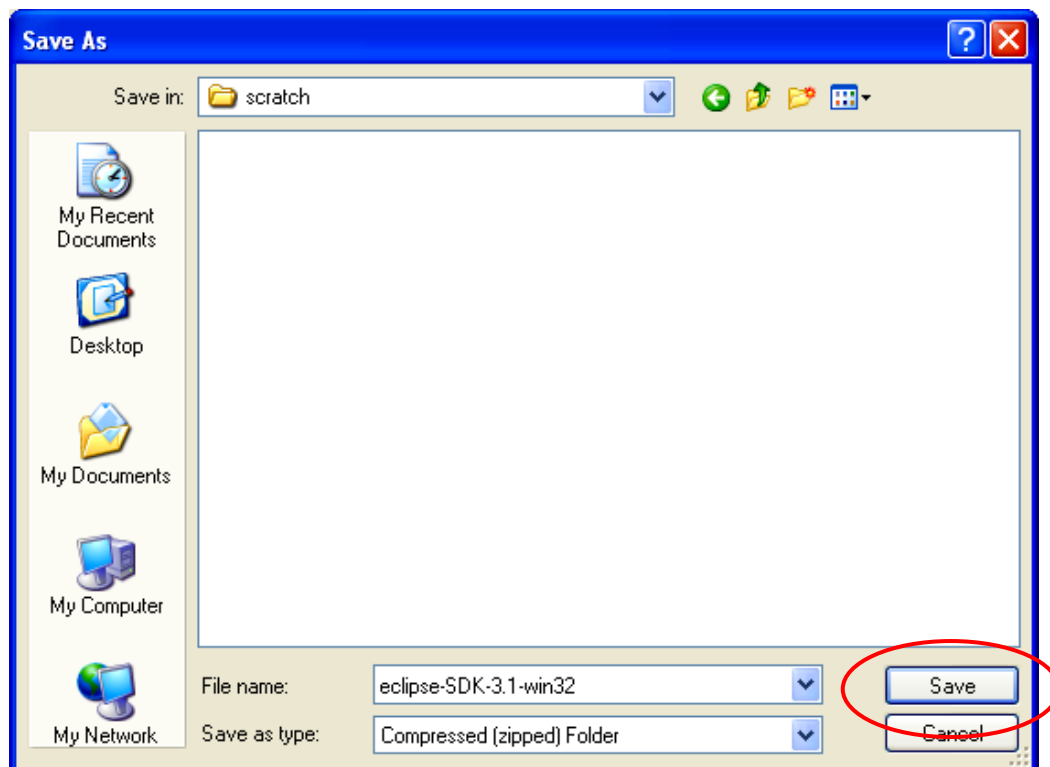
Great! This mirror site is in my home town.

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.

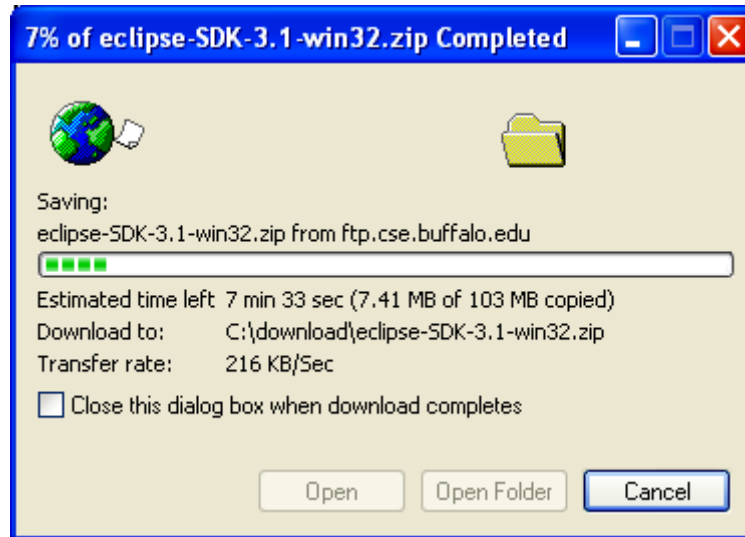


Now browse to the **c:/scratch** directory that you created previously.

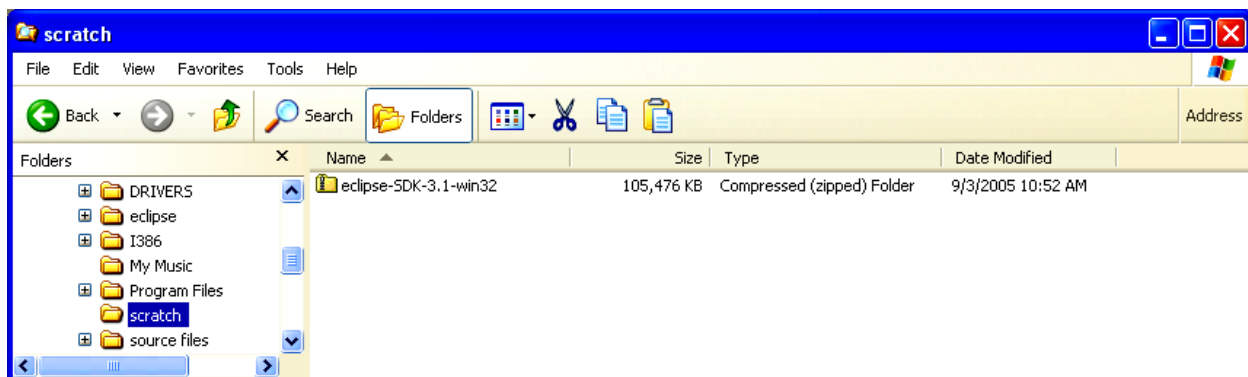


Click on **Save** to start the download.

Now the download will start. Eclipse is delivered as a ZIP file. It's 103 megabytes in length and takes 9 minutes 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.

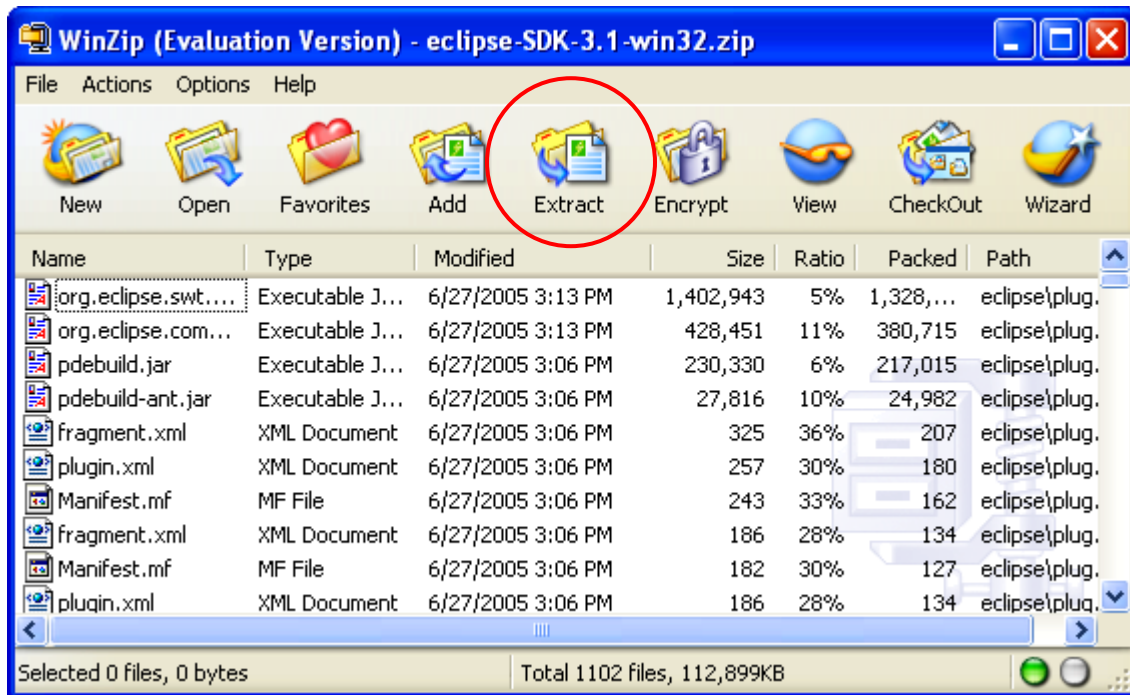


Eclipse is delivered as a ZIP file (**eclipse-SDK-3.1-win32.zip**). You can use WinZip to decompress this file and load its constituent parts on your hard drive. If you don't have WinZip, you can get a free evaluation version from this address:

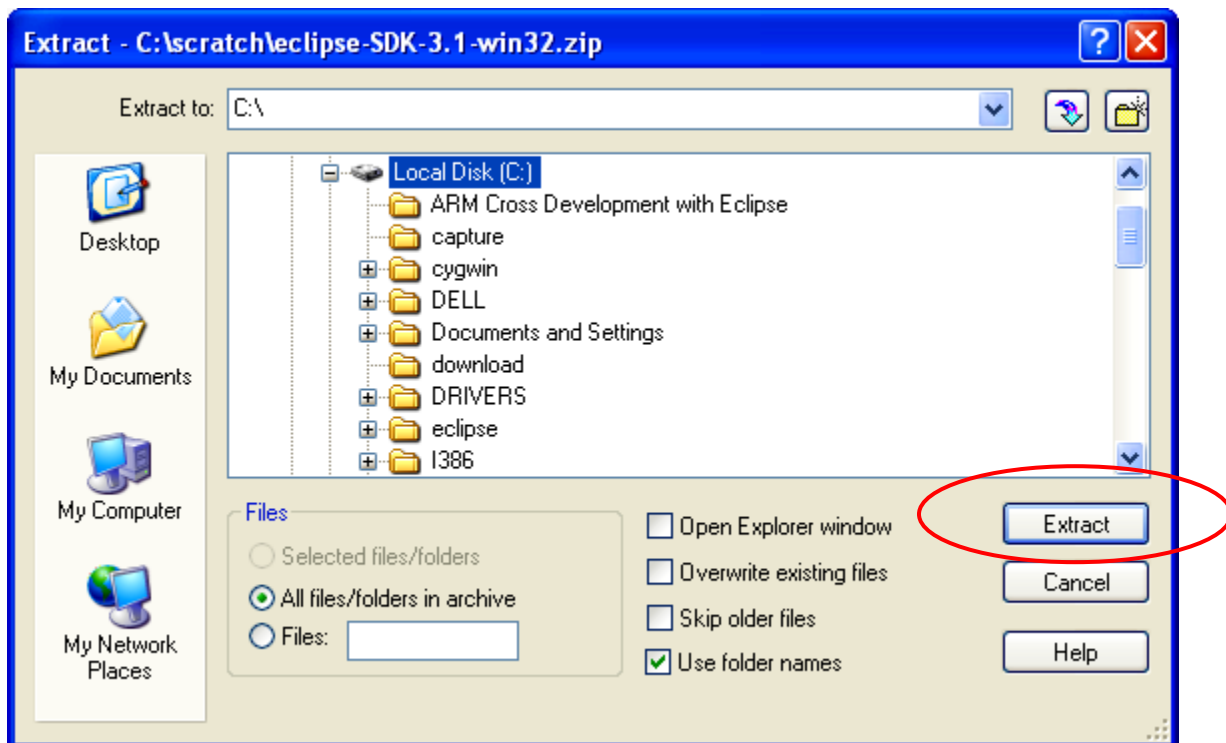
<http://www.winzip.com/>

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.

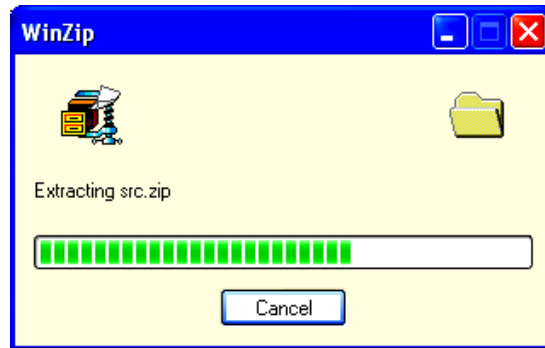
In my computer, with WinZip installed, double-clicking on the zip file name (**eclipse-SDK-3.1-win32.zip**) in the Windows Explorer display above will automatically start up WinZip. Click on “**Extract**” to start the Eclipse file decompression.



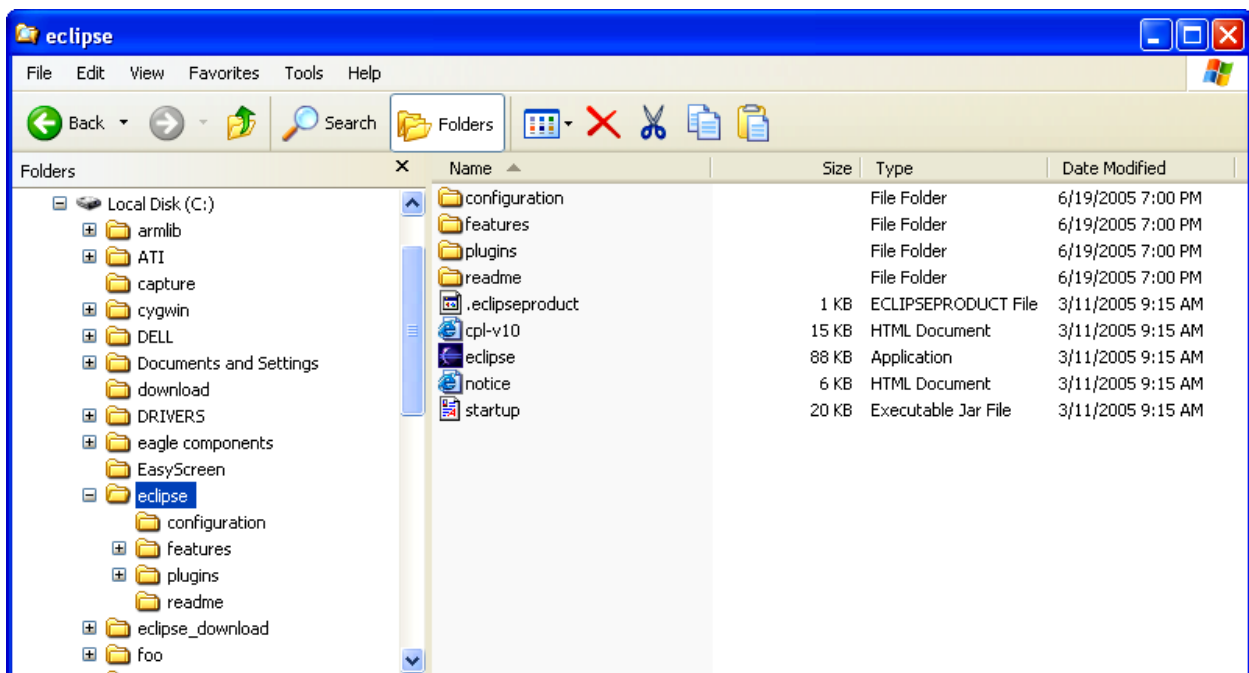
WinZip will ask you into what directory you wish to extract the contents of the zip file. In this case, you must specify the root drive **C:**



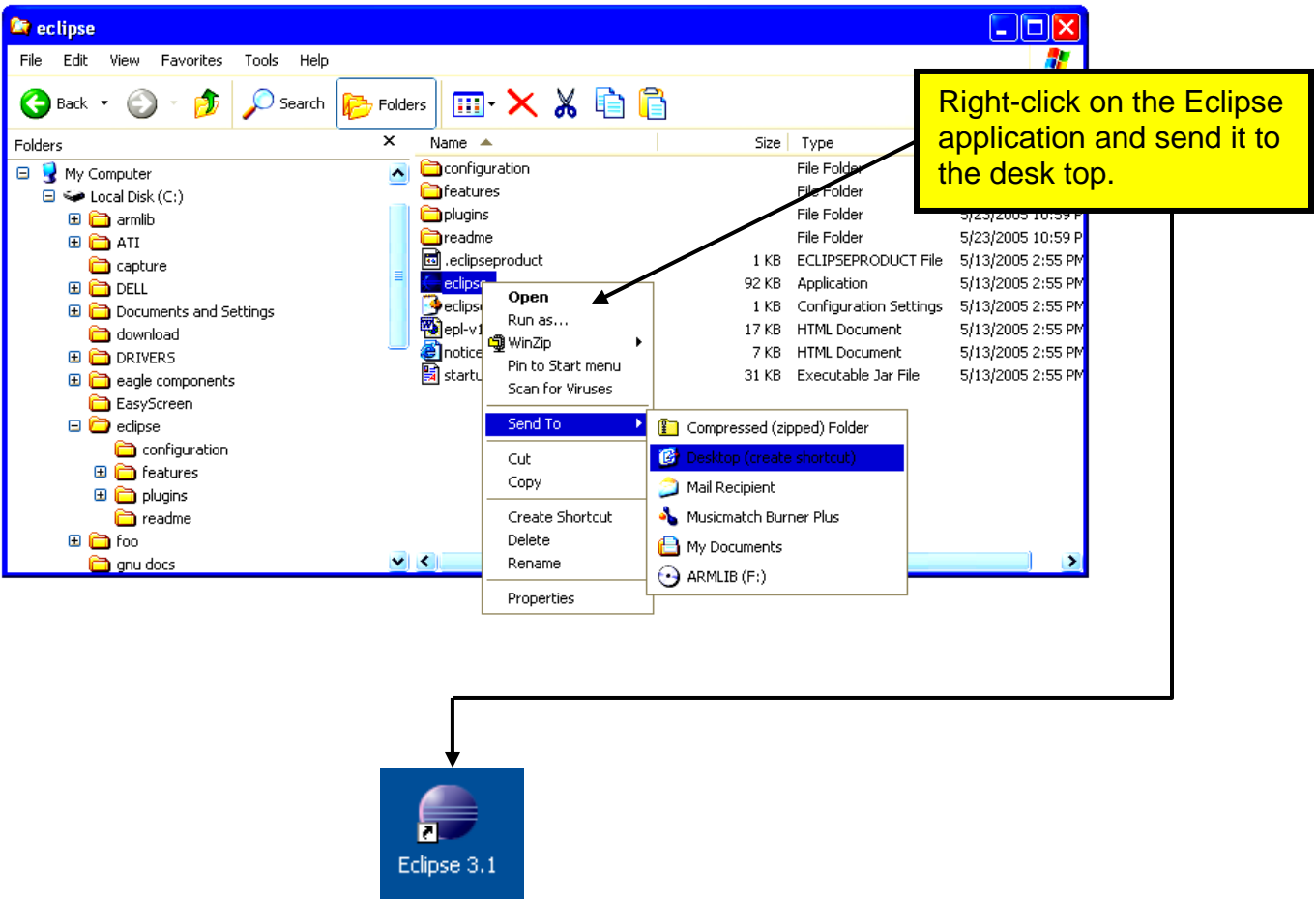
The WinZip Utility will start extracting all the Eclipse files and directories into a c:/eclipse directory on your root drive C:



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.

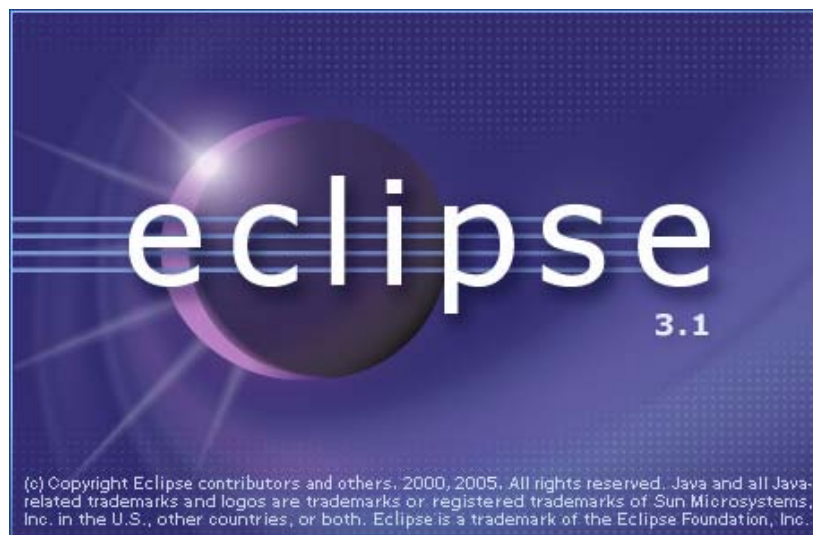


You can create a desktop icon for conveniently starting Eclipse by right-clicking on the Eclipse application above and sending it to the desk top.

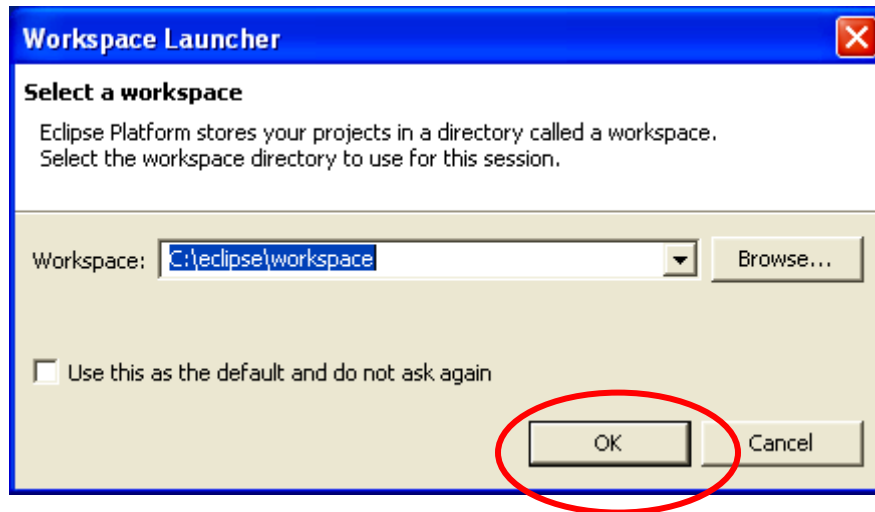


Now is a good time to test that Eclipse will actually run. Click on the desktop icon to start the Eclipse IDE.

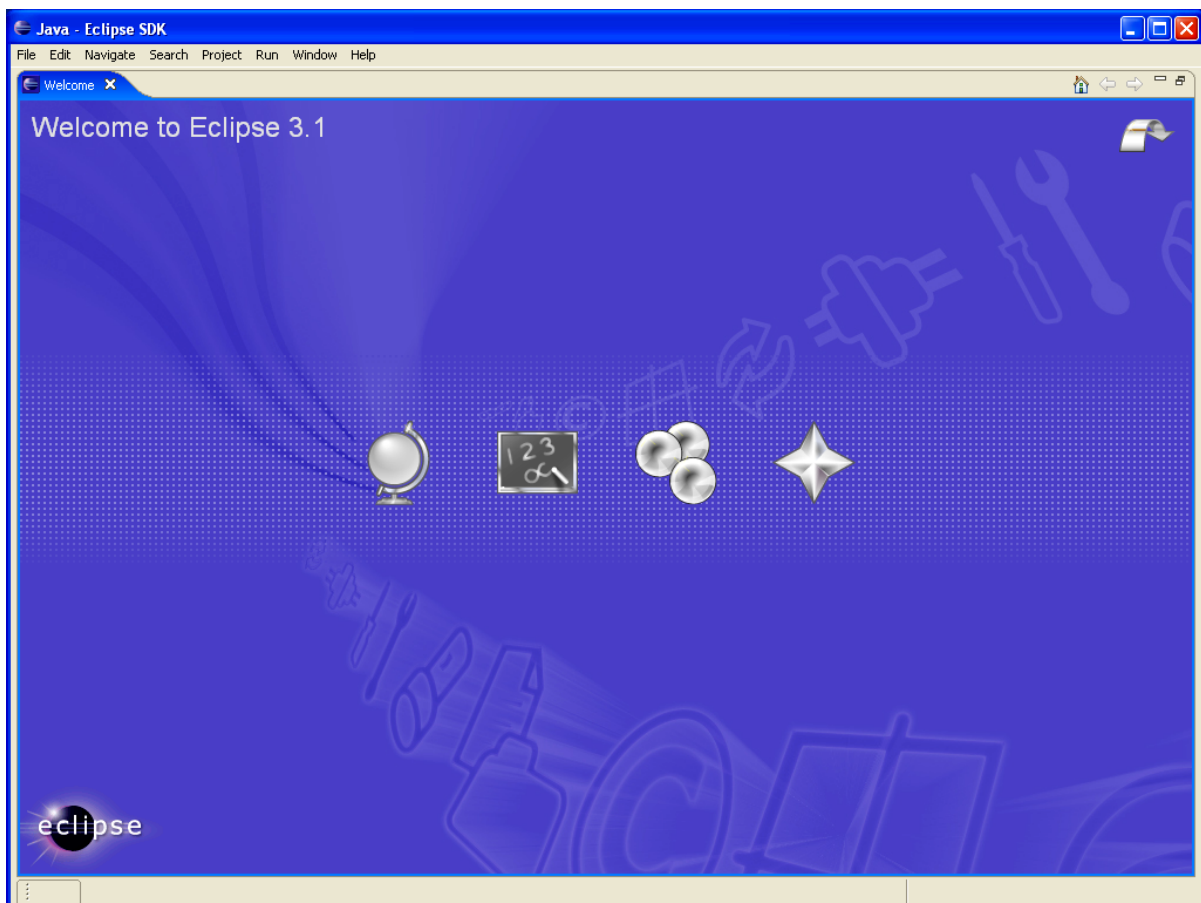
If the Eclipse Splash Screen appears, we have succeeded. If not, chances are that the Java Run Time Environment is not in place. Review and repeat the instructions on installing Java on your computer.



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 plug-ins 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 zip files that are unzipped into the Eclipse directory.

Unfortunately, the **CDT** plug-in from the Eclipse web site has some problems debugging applications in a cross-development environment (e.g. where the target is a circuit board with an ARM microprocessor and a JTAG interface). To the rescue is the Norwegian engineering company Zylin who have developed a special custom version of **CDT** that properly interfaces the GDB debugger to a remote target. The Zylin version of **CDT** was developed with the cooperation of the **CDT** Development Team and is essentially a copy of the latest version of **CDT** with the special debug modifications. The open source community owes a debt of thanks to Øyvind Harboe and his associates at Zylin.

To download the Zylin version of the CDT plug-in, click on the following link:

<http://www.zylin.com/embeddedcdt.html>

The Zylin website page devoted to the CDT plug-in will have a link to the latest “snapshot”. This snapshot is two zip files that you will extract to the c:\eclipse folder.

Zylin AS
Zylin soft CPU
Hardware
Embedded software
eCos
Open source
- Eclipse CDT
- eCos and libstdc++
- Mailinglist
Contact

Eclipse CDT plugin

Eclipse CDT has excellent GDB support. However, there are a few stumbling blocks when trying to debug embedded applications.

Zylin has made some modifications in Eclipse CDT for Windows + a plugin to improve support for GDB embedded debugging in CDT for eCos applications.

Zylin would like to extend a special thanks to Alain Magloire of the CDT team for making this possible.

Download

[Latest snapshot](#)

Mini FAQ

- Q: Is the plugin specific to eCos?
- A: No.
- Q: Is the plugin specific to ARM?
- A: No. In principle it can be used with any CPU that GDB supports.
- Q: Does the plugin work under Linux?
- A: Yes. You can even debug native applications that do not live inside CDT projects.
- Q: Does the plugin work under Windows?
- A: Yes.
- Q: Source?
- A: See inside the zylincdt-200xmdd.zip archive.

Install

- **These plugins require the latest Eclipse 3.1 release.**
- **Both plugins must be installed together.**
- Delete previous CDT and Zylin Embedded CDT directories from eclipse\features and eclipse\plugins directory.
- unzip embeddedcdt-200xmdd.zip and zylincdt-200xmdd.zip to the Eclipse directory.

Click on this link to get the latest Zylin CDT snapshot.

Download the following two files from the Zylin web site.

<http://www.zylin.com/embeddedcdt-20050810.zip>
<http://www.zylin.com/zylincdt-20050810.zip>

[Zylin-discuss] embeddedcdt binary snapshot - CDT 3.0 - RC3

*Oyvind Harboe [oyvind.harboe at zylin.com](mailto:oyvind.harboe@zylin.com)
Wed Aug 10 09:12:41 CEST 2005*

- Previous message: [\[Zylin-discuss\] display the remote memory content.](#)
- Next message: [\[Zylin-discuss\] embeddedcdt binary snapshot - CDT 3.1](#)
- Messages sorted by: [\[date \]](#) [\[thread \]](#) [\[subject \]](#) [\[author \]](#)

Changes:

- updated to CDT CVS HEAD.
- before installing this version, take special care to delete:
 - *all* previous versions of the Zylin CDT embedded plugin
 - *all* launch entries for the Zylin CDT embedded plugin

Installation:

<http://www.zylin.com/embeddedcdt.html>

Download binary:

<http://www.zylin.com/embeddedcdt-20050810.zip>

<http://www.zylin.com/embeddedcdt-linux-gtk-20050810.zip>

<http://www.zylin.com/zylincdt-20050810.zip>

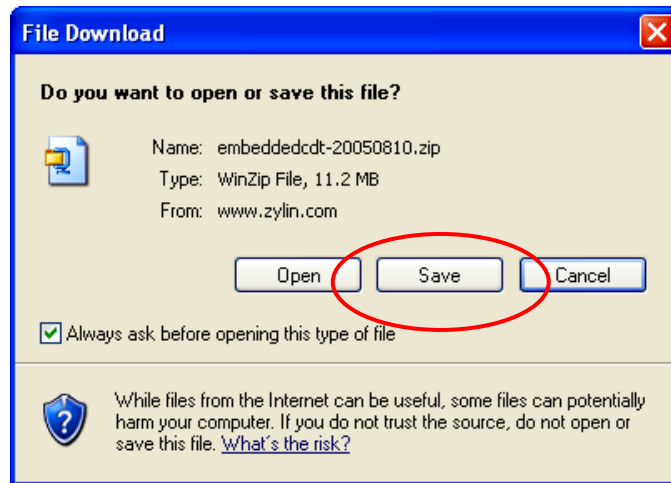
Download these two files to c:/scratch

Source code:

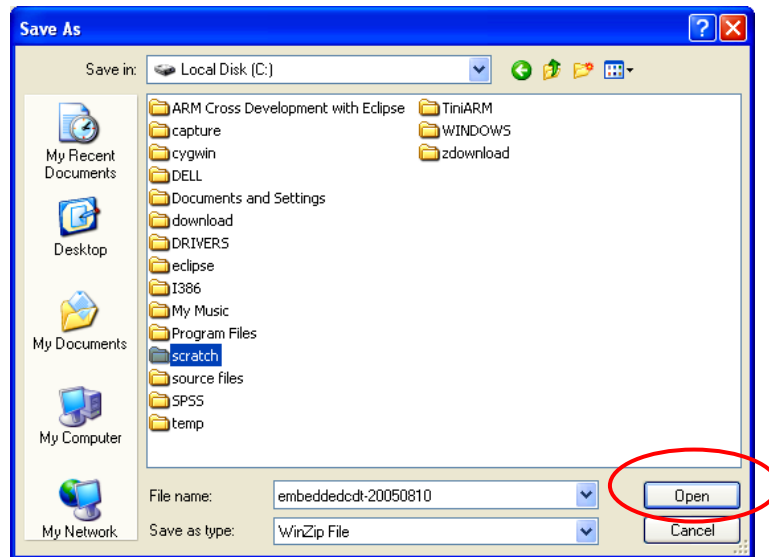
In zylincdt-yyyyymmdd.zip above. The patches to CDT are in the files named "org.eclipse.cdt.debug.mi.core.txt"

--
Oyvind Harboe
<http://www.zylin.com>

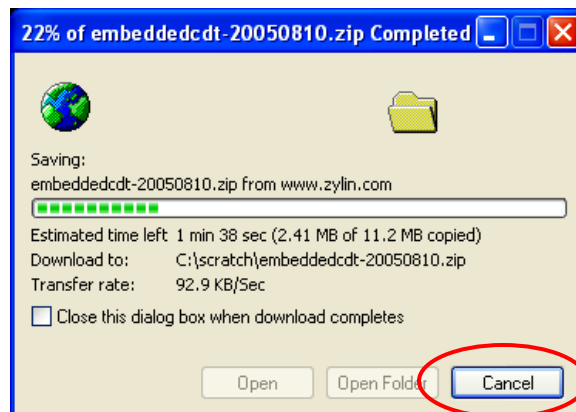
First, click on <http://www.zylin.com/embeddedcdt-20050810.zip> to download. Then click on “**Save**” in the File Download window.



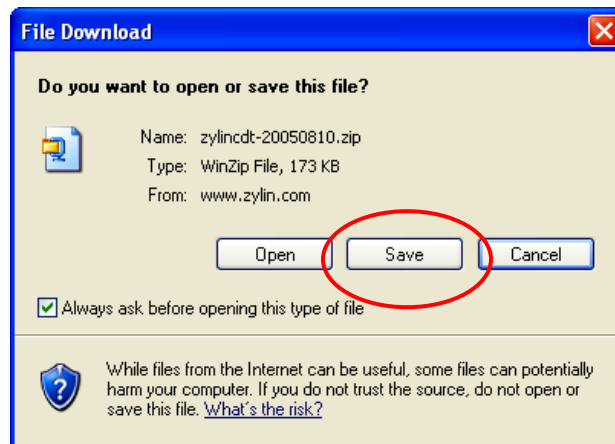
Select the temporary **c:\scratch** directory as the target of the download and click “**Open**.”



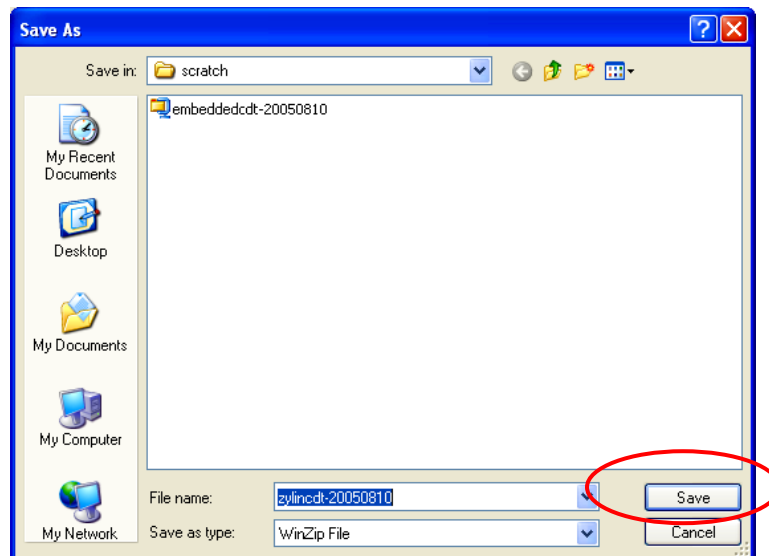
The first Zylin CDT zip file will download into the **c:\scratch** folder. This file is an 11 Mb download.



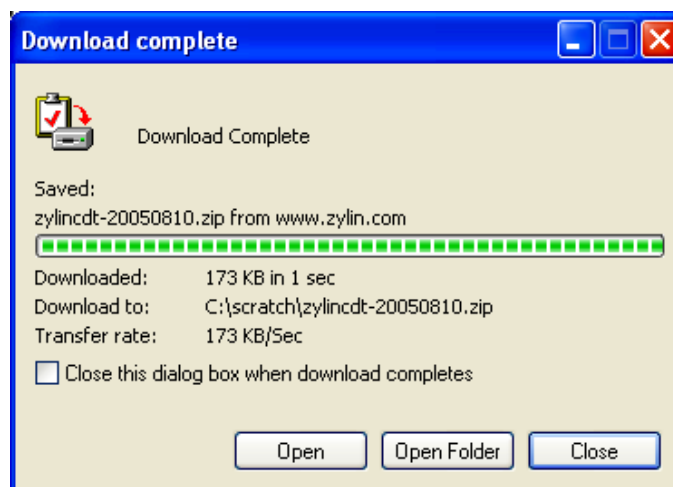
Next, click on <http://www.zylin.com/zylincdt-20050810.zip> to download. Then click on “Save” in the File Download window.



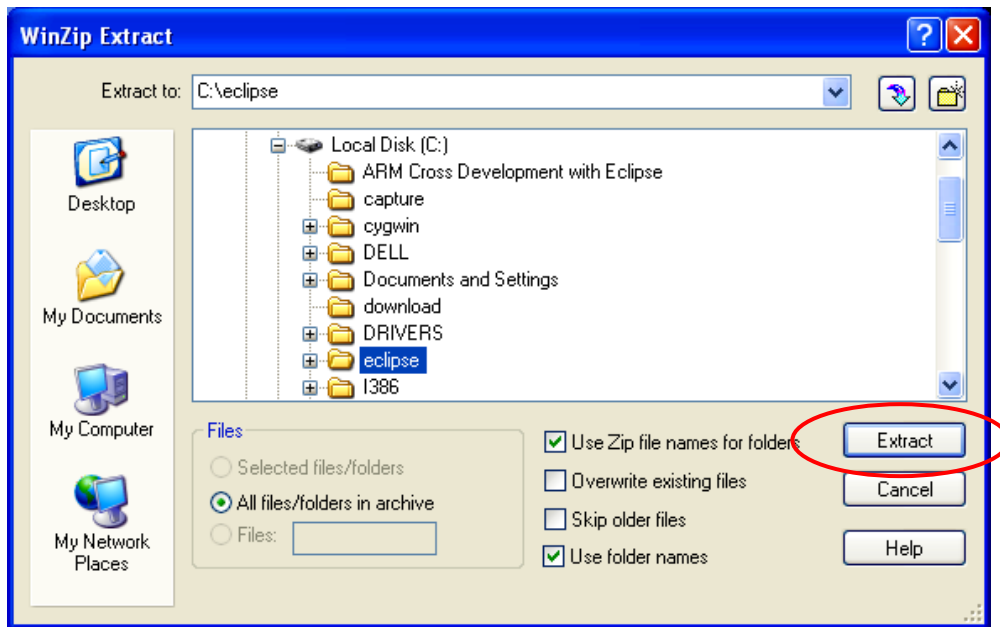
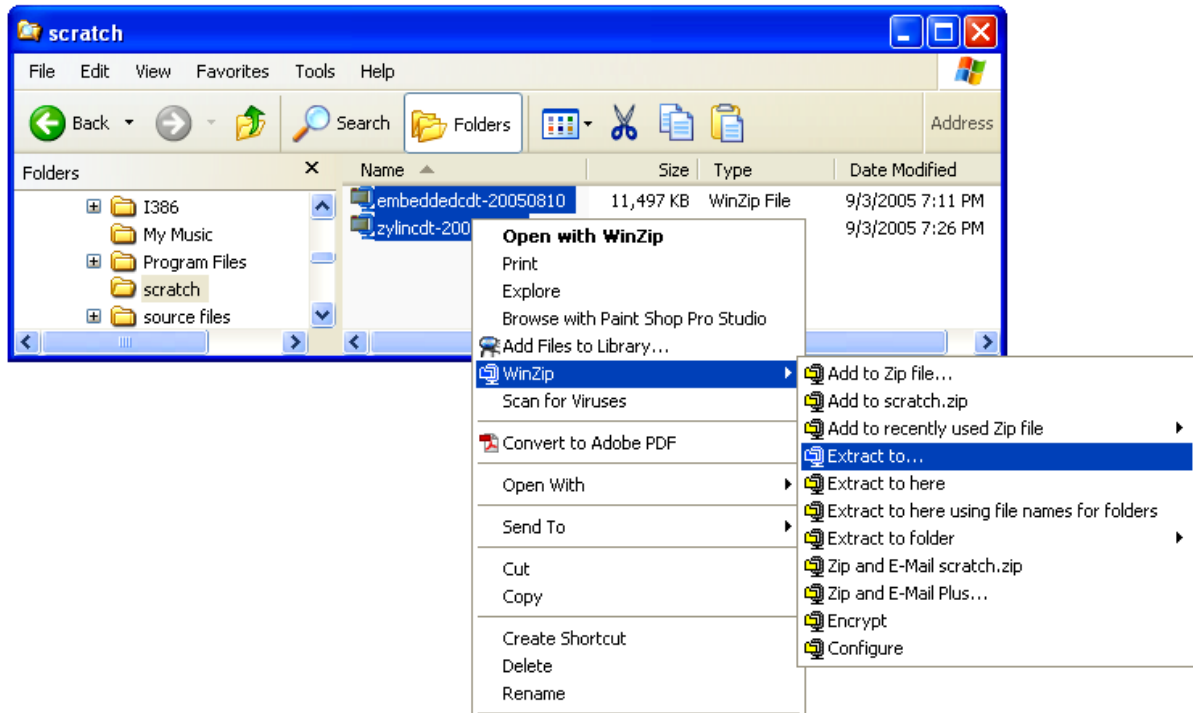
Select the temporary **c:\scratch** directory as the target of the download.



The second Zylin CDT zip file will download into the **c:\scratch** folder. This file is a shorter file, only 173 Kb.



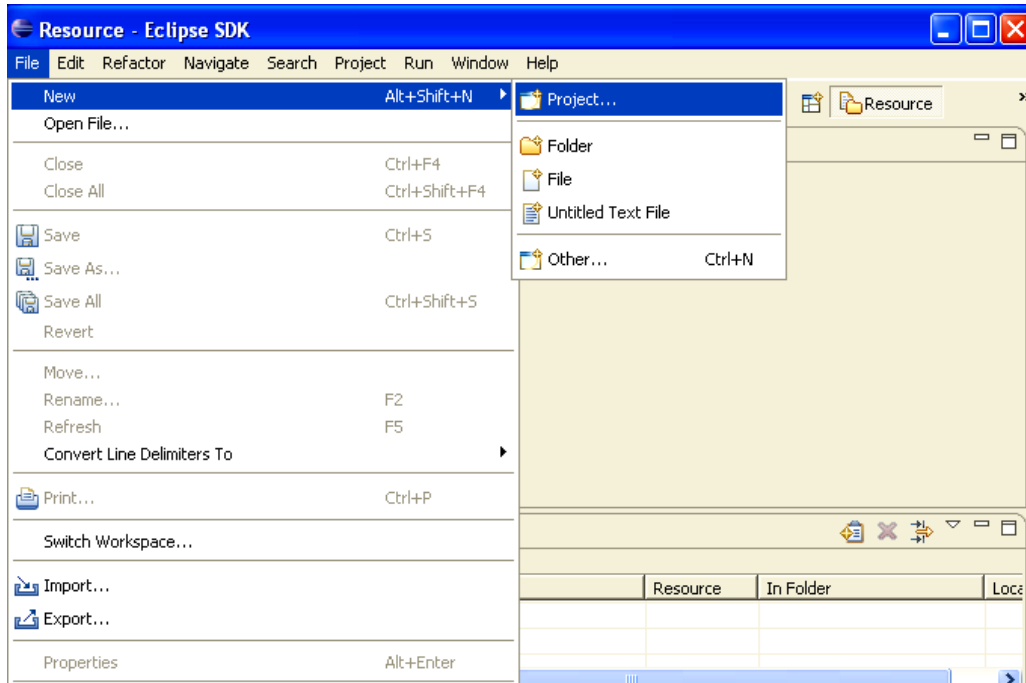
Select both Zylin **CDT** files in the **c:\scratch** folder using Windows Explorer and use WinZip to extract them to the **c:\eclipse** folder.



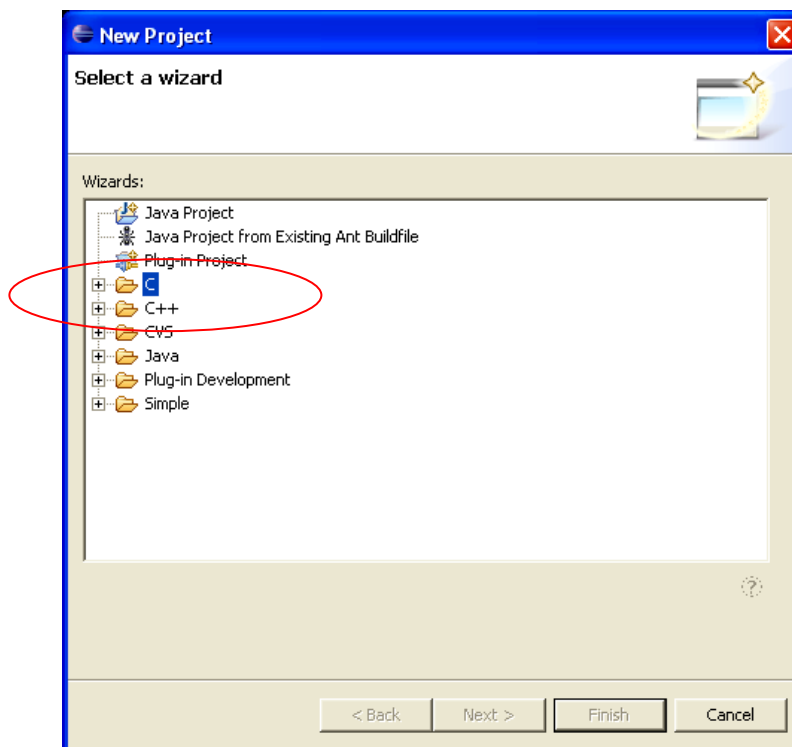
To verify that Eclipse had the **CDT** installed properly, start Eclipse by clicking on the desktop icon.



When Eclipse starts, click on **"File – New - Project..."**



When the New Project window appears, check if C and C++ appear as potential projects. If this is true, Eclipse CDT has been installed properly.



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.

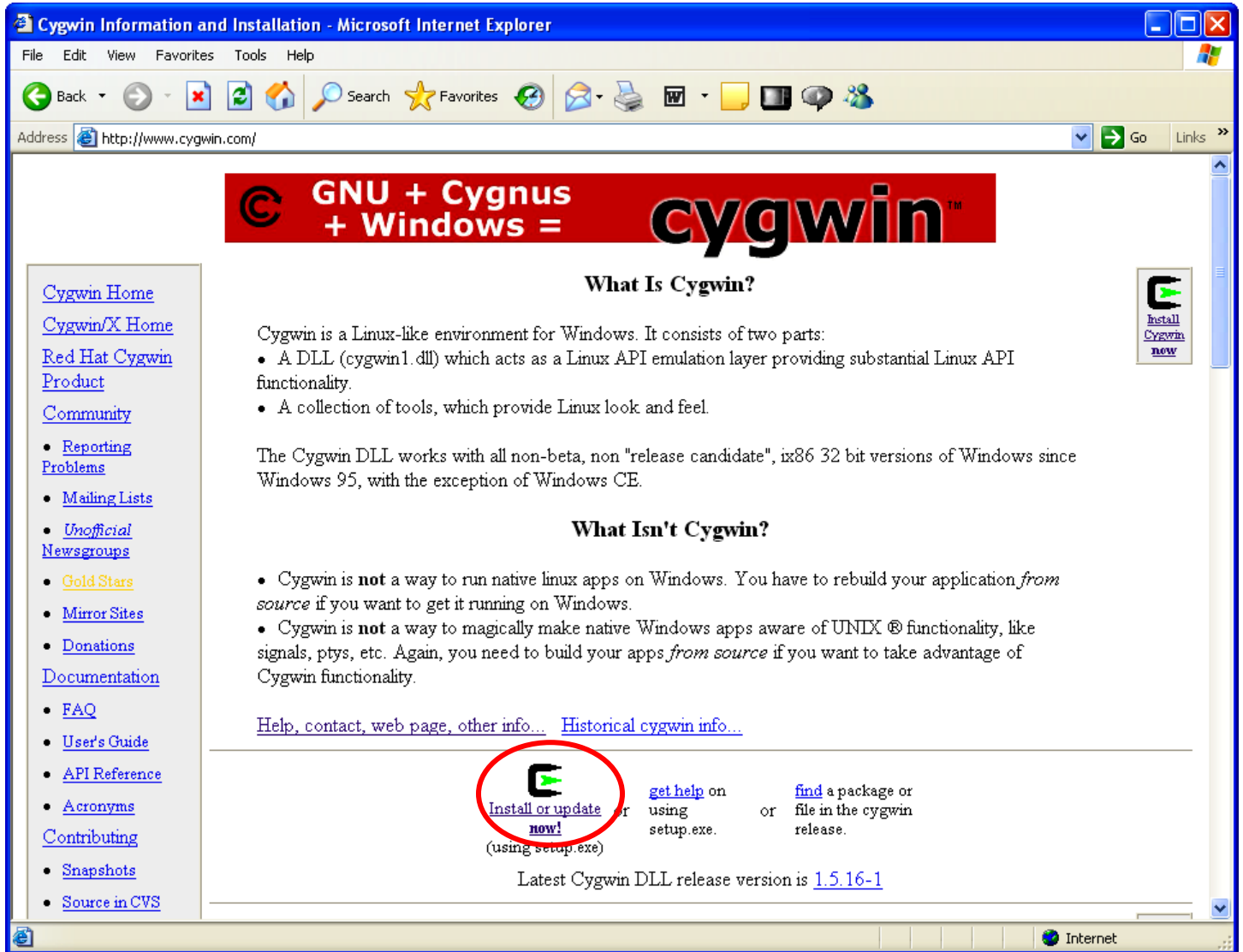
It is possible to build a compiler toolset for the ARM processors using the generic Cygwin GNU toolkit. In his book “**Embedded System Design on a Shoestring**”, Lewin A.R.W. Edwards gives detailed instructions on just how to do that. Fortunately, there are quite a few pre-built tool chains on the internet 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:



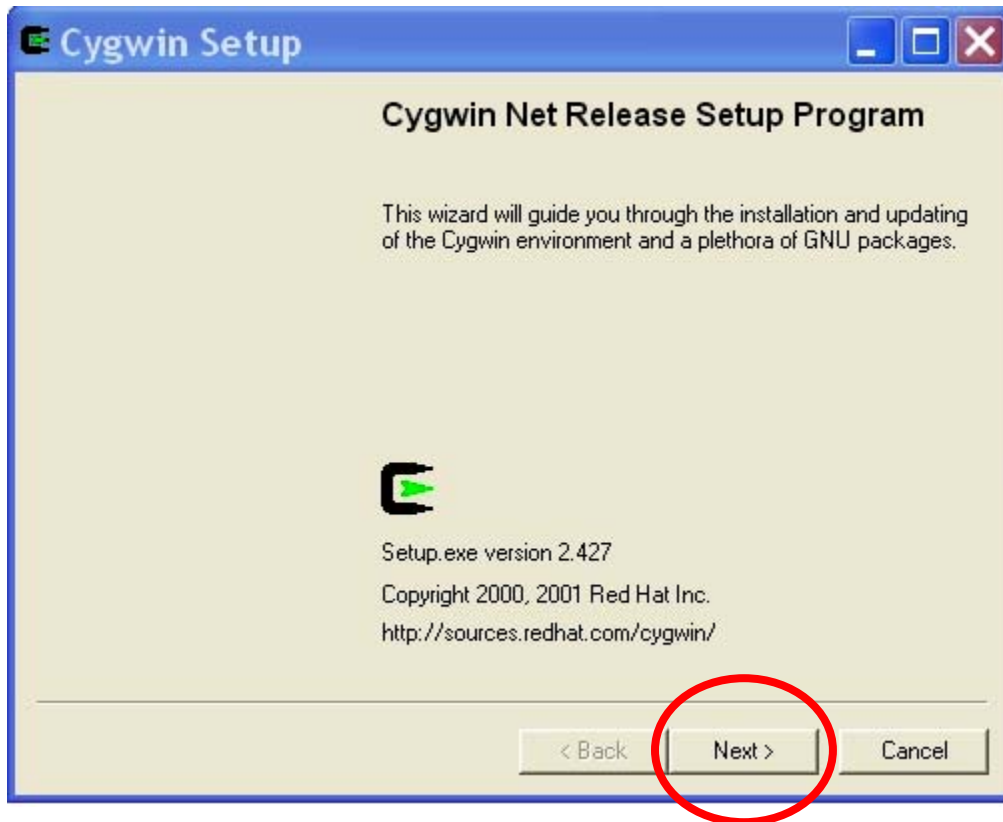
The first thing to do is to click on the install icon:

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

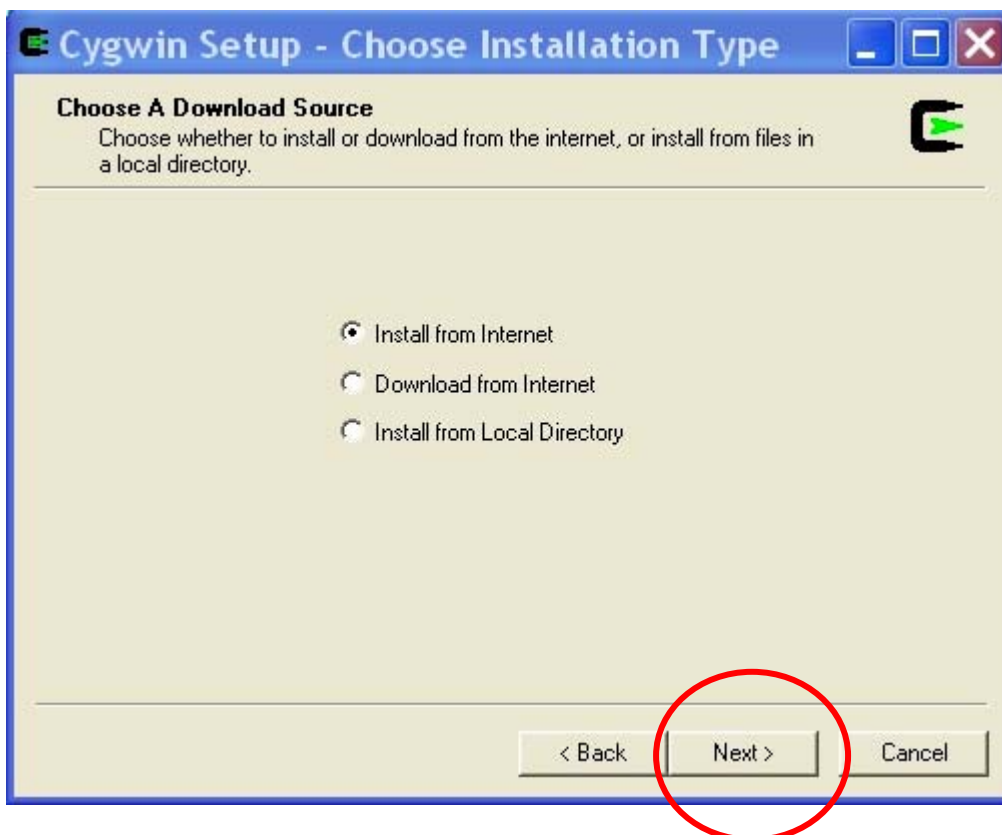


Click on "Run" to download and run the Cygwin setup program.

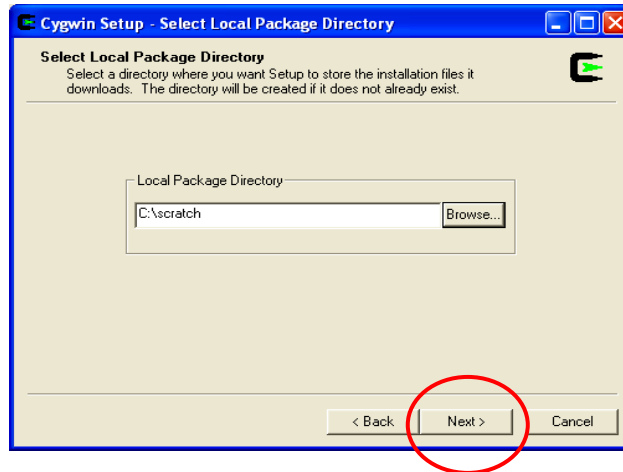
Now the Cygwin wizard will start up. Select **“Next”** to continue.



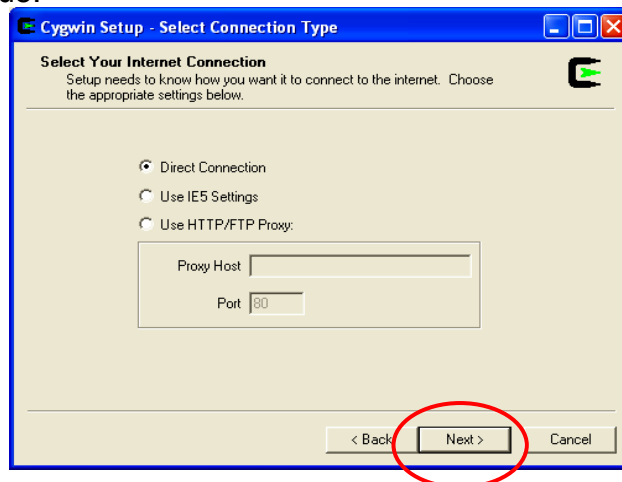
Choose **“Install from Internet”** and then click **“Next.”**



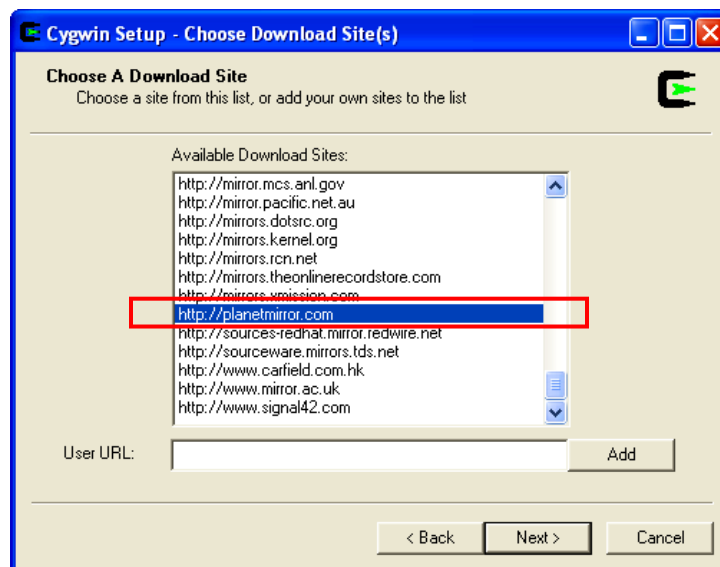
Now we specify a directory where all the downloaded components go, our `c:/scratch` folder will do just fine.



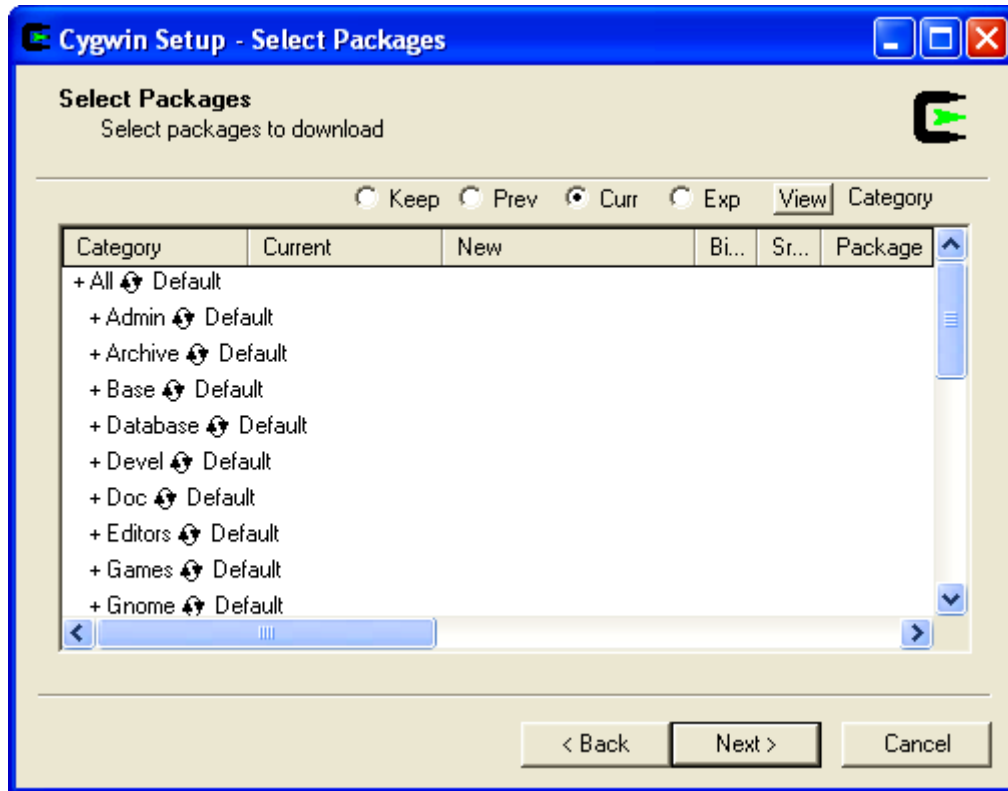
Since I have a high speed internet connection, I always select “**Direct Connection.**” Click “**Next**” to continue.



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 download.



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.

+ Devel  Default

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

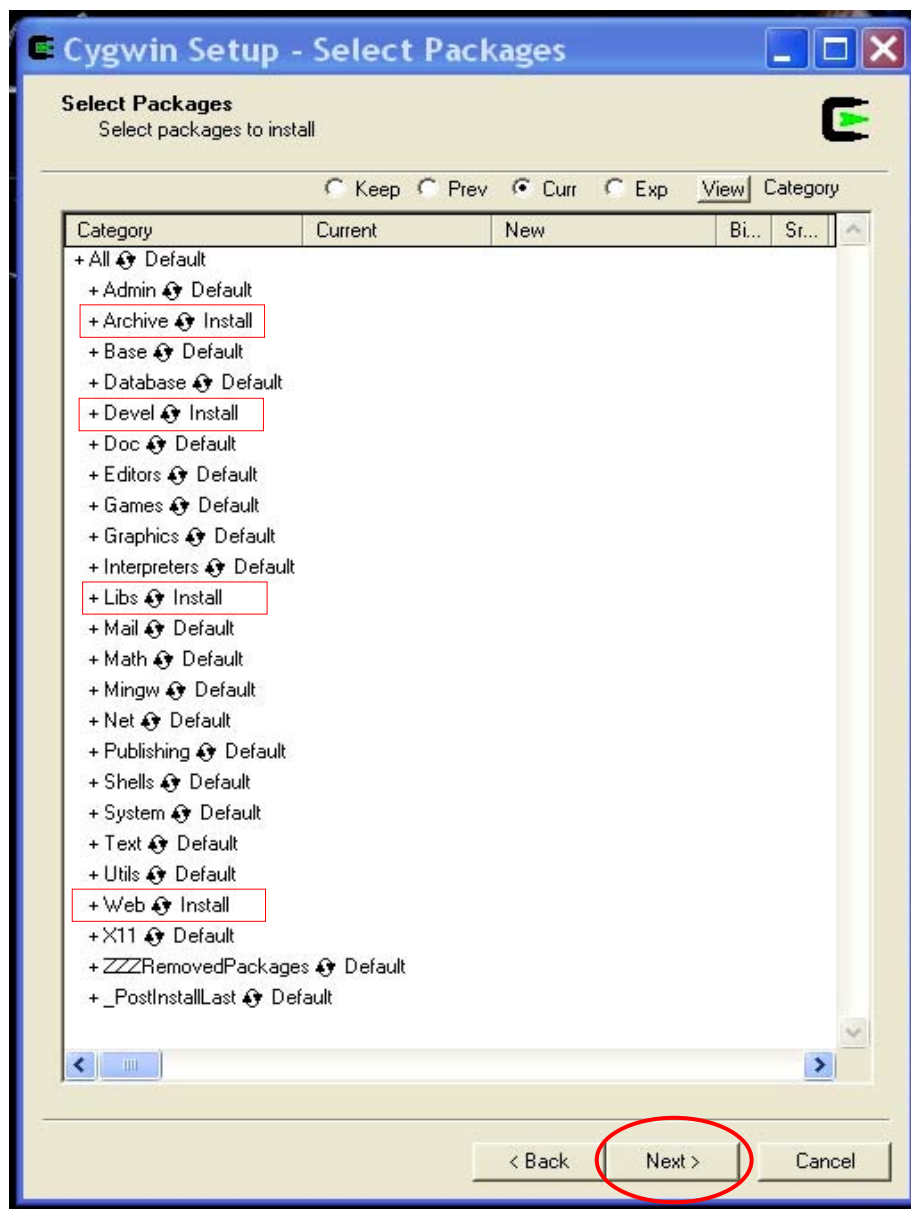
+ Devel  Install

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	<input type="radio"/>	Default	→	Archive	<input type="radio"/>	Install
Devel	<input type="radio"/>	Default		Devel	<input type="radio"/>	Install
Libs	<input type="radio"/>	Default		Libs	<input type="radio"/>	Install
Web	<input type="radio"/>	Default		Web	<input type="radio"/>	Install

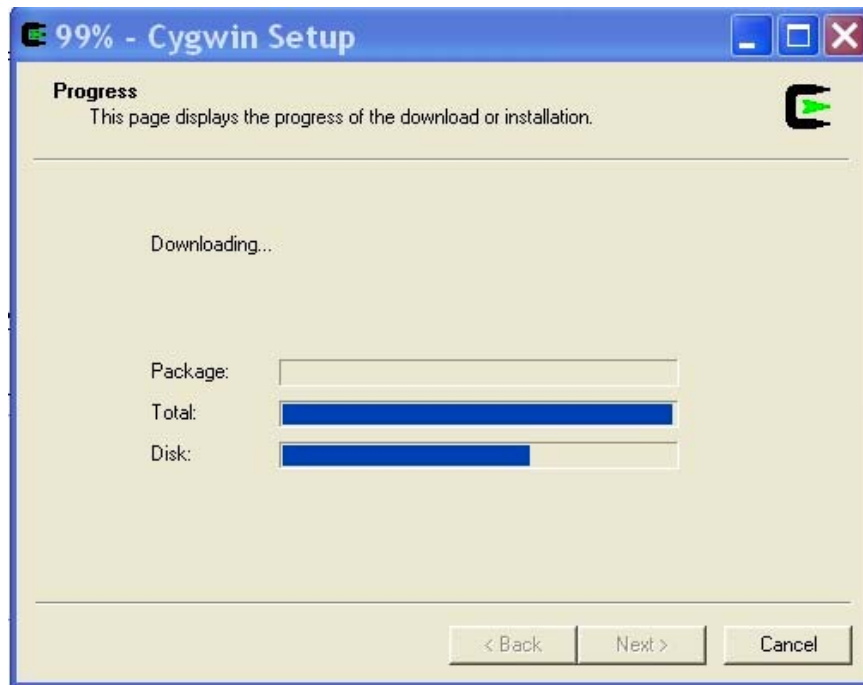
Click on the little circle with the arrowheads until you change the four packages listed above from "default" to "install."

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.



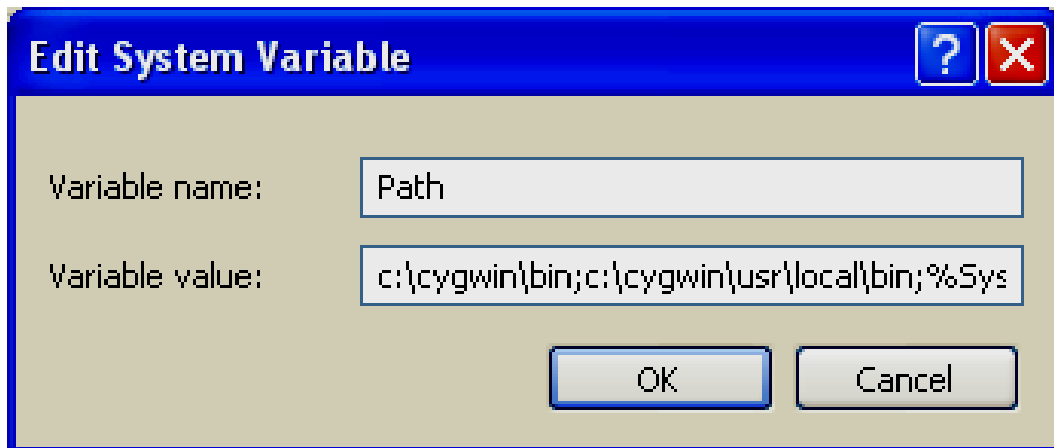
Now the Cygwin installation manager completes and shows the following result.



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.



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.



GNU ARM

HOME FILES SUPPORT LIST RESOURCES

“Steve is one of the brightest guys I’ve ever worked with - brilliant; but when we decided to do a microprocessor on our own, I made two great decisions - I gave them [Steve Furber and Sophie Wilson] two things which National, Intel and Motorola had never given their design teams: the first was no money; the second was no people. The only way they could do it was to keep it really simple.” -- Hermann Hauser

Last update: 2004-07-05 18:01

GNU ARM toolchain for Cygwin, Linux and MacOS

Welcome! In this page you will find a pre-compiled binary distribution for the (hopefully) latest GNU ARM/Newlib toolchain for [Cygwin](#), Linux and MacOS.

The toolchain consists of the GNU binutils, compiler set (GCC) and debugger (Insight for Windows and Linux, GDB only for MacOS). Newlib is used for the C library. The toolchain includes the C and C++ compilers. Details of the build process appear [here](#). The Windows installer executable files are generated with Inno Setup. The MacOS toolchain is bundled with Apple’s PackageMaker.

If you have any problems using these files please contact us using our [mailing list](#).

Please note: Some people have been asking us for permission to re-distribute the GNUARM installer and associated files along with their commercial products. This is totally encouraged provided that the software licenses are fulfilled and that there are no charges except for, possibly, a small fee for the media and handling. In this way you will be helping both the GNUARM project and your customers.

Also note that we have avoided purely-commercial pointers in our projects section at our [resources](#) page. To be fair with everyone, we will be only adding links to projects that provide useful, unbiased, technical information online. [Contact us](#) if you wish your site to be listed there.

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

Binaries

GCC-3.3 toolchain

Mac OS X

[binutils-2.14, gcc-3.3.2-c-c++, newlib-1.12.0, gdb-6.0, PKG TGZ \[35,2 MB\]](#)

GCC-3.4 toolchain

Cygwin

[binutils-2.15, gcc-3.4.3-c-c++-java, newlib-1.12.0, insight-6.1, setup.exe \[17,0MB\]](#)

GNU/Linux (x86)

[binutils-2.15, gcc-3.4.3-c-c++-java, newlib-1.12.0, insight-6.1, TAR BZ2 \[56,0MB\]](#)

GCC-4.0 toolchain

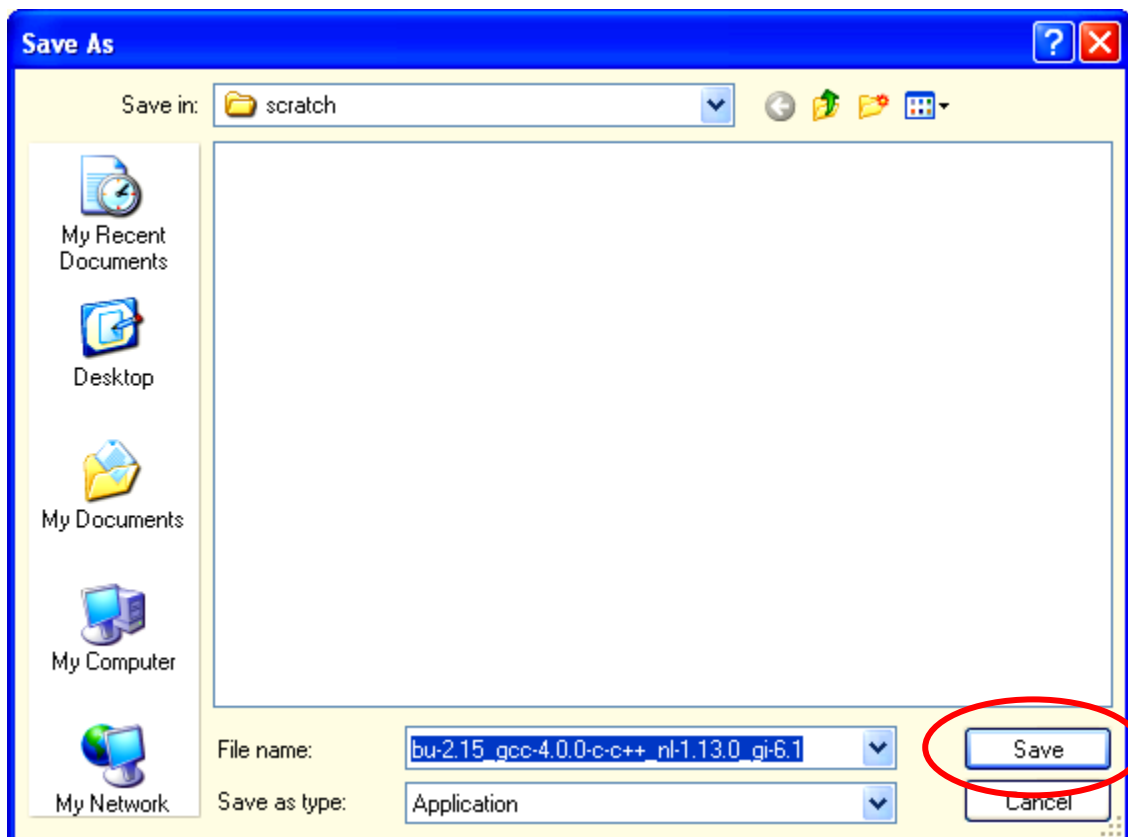
Cygwin

[binutils-2.15, gcc-4.0.0-c-c++, newlib-1.13.0, insight-6.1, setup.exe \[23,0MB\]](#)

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.

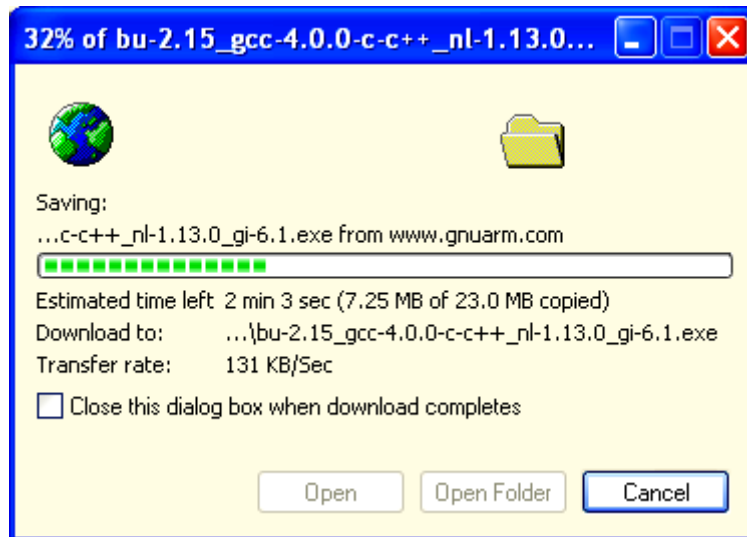


Once again, our **c:/scratch** directory will suffice.



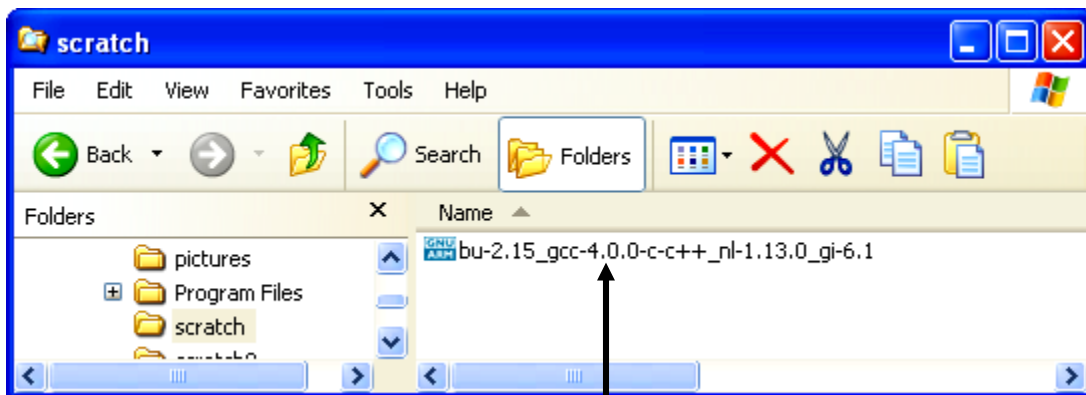
As you can see, this download has a very long name!

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



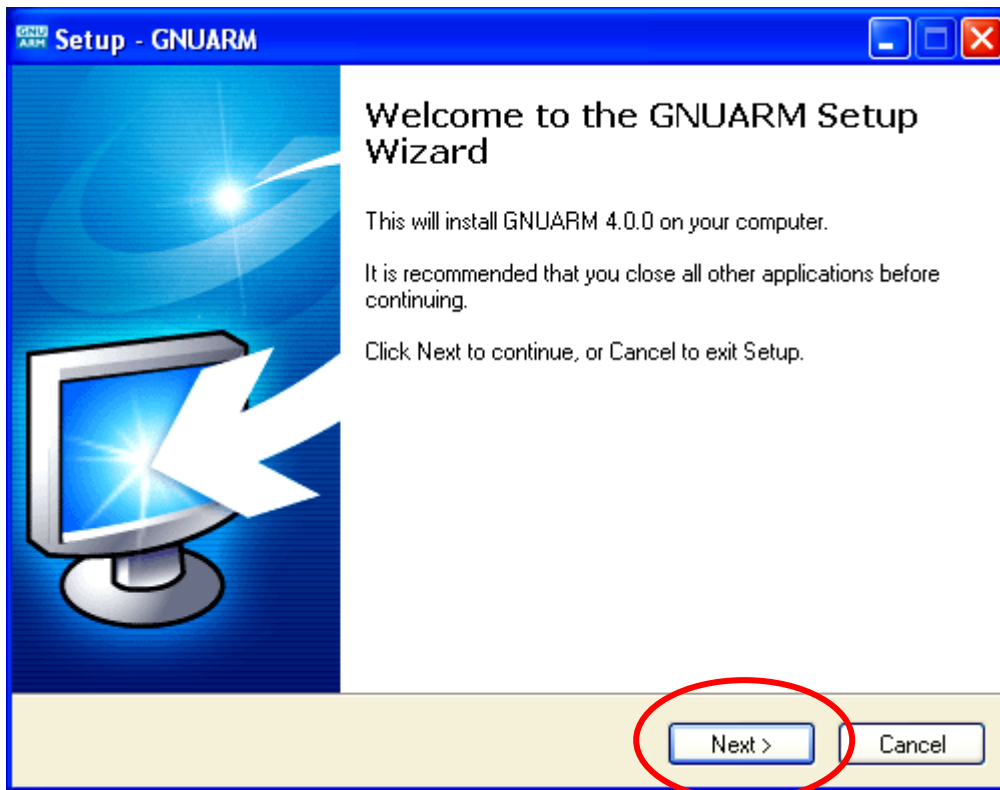
The download directory now has the following setup application with the following unintelligible filename: **bu-2.15_gcc-3.4.1-c-c++-java_nl-1.12.0_gi-6.0.exe**

Click on that filename to start the installer.

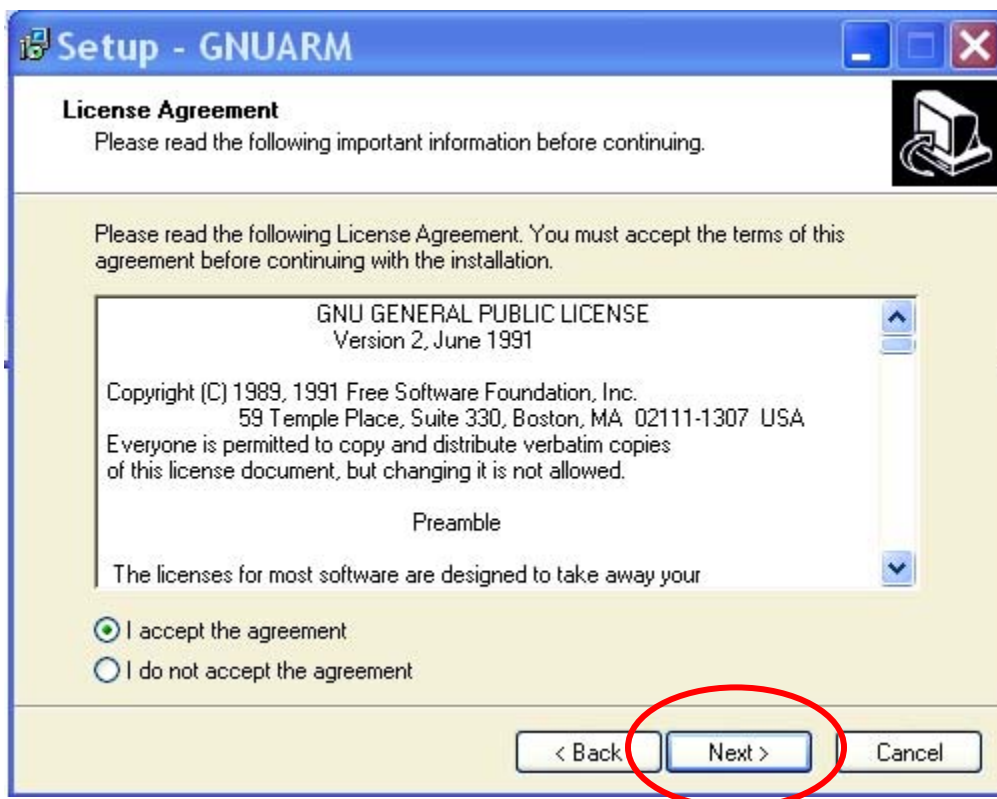


Click on this application to start the GNUARM installer

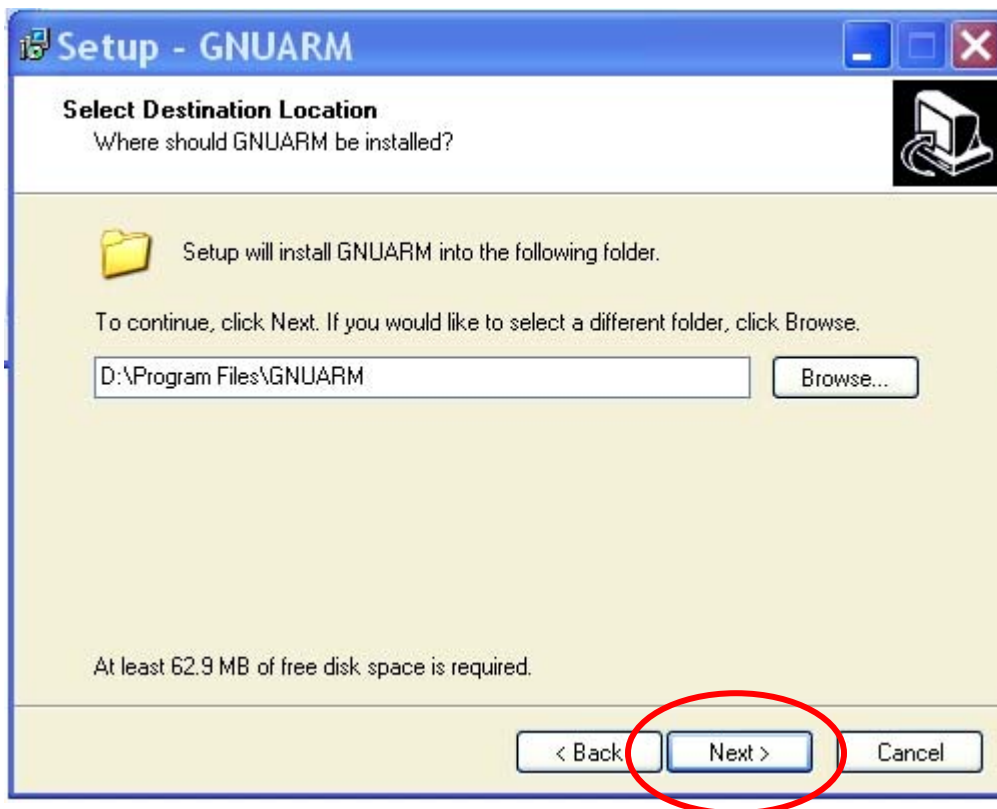
The GNUARM installer will now start. Click **“Next”** to continue.



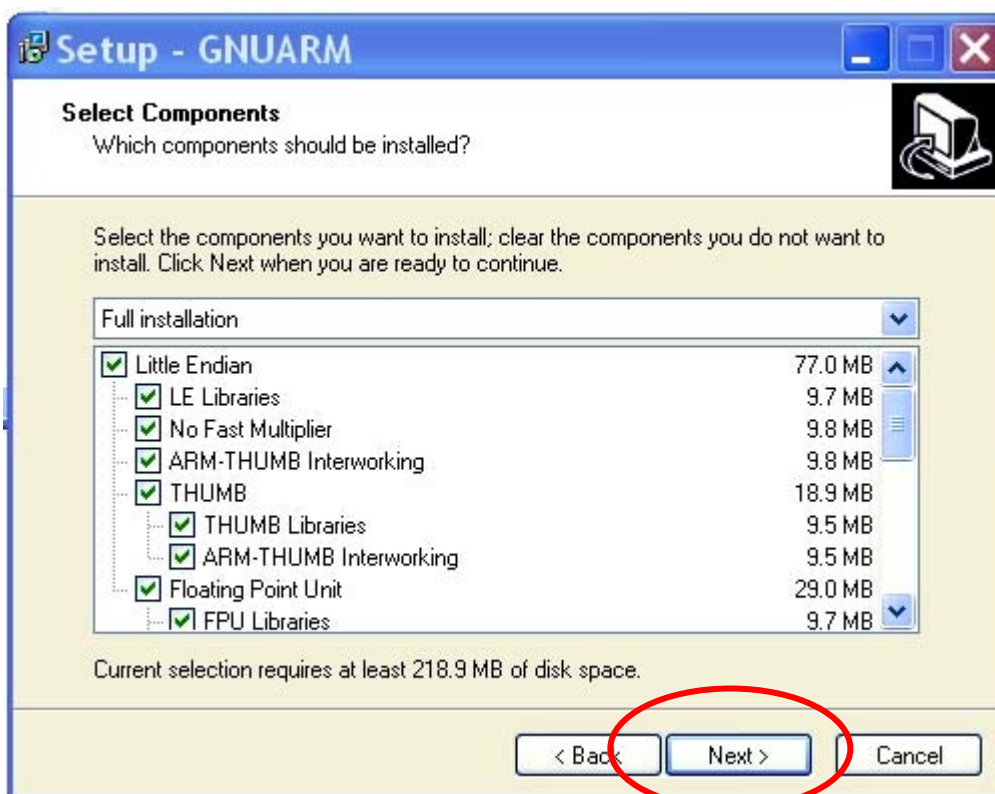
Accept the GNU license agreement – don't worry, it's still free. Click **“Next”** to continue.



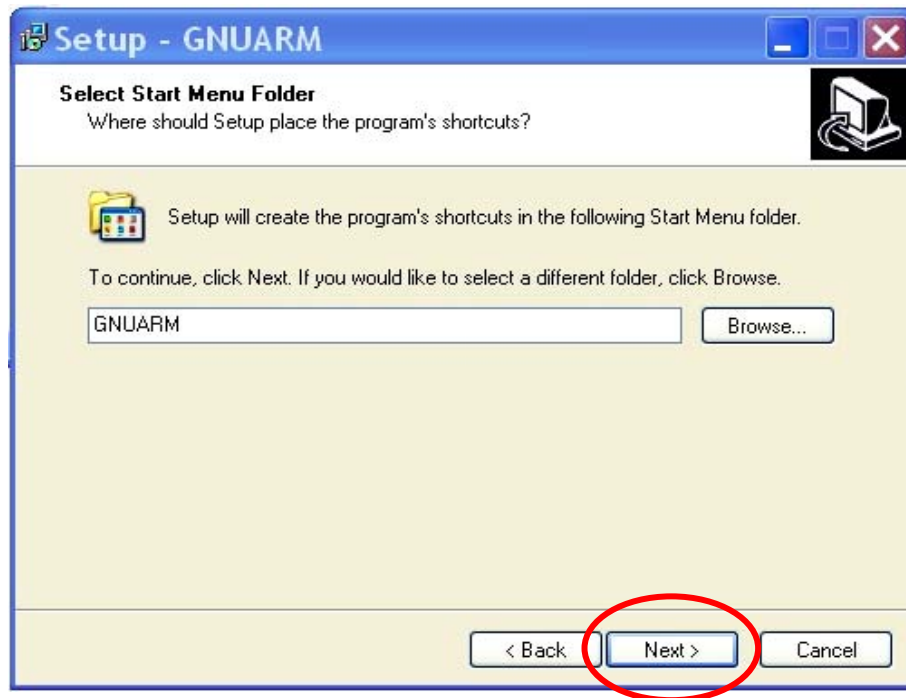
We'll take the default and let it install into the "Program Files" directory. Click "Next" to continue.



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

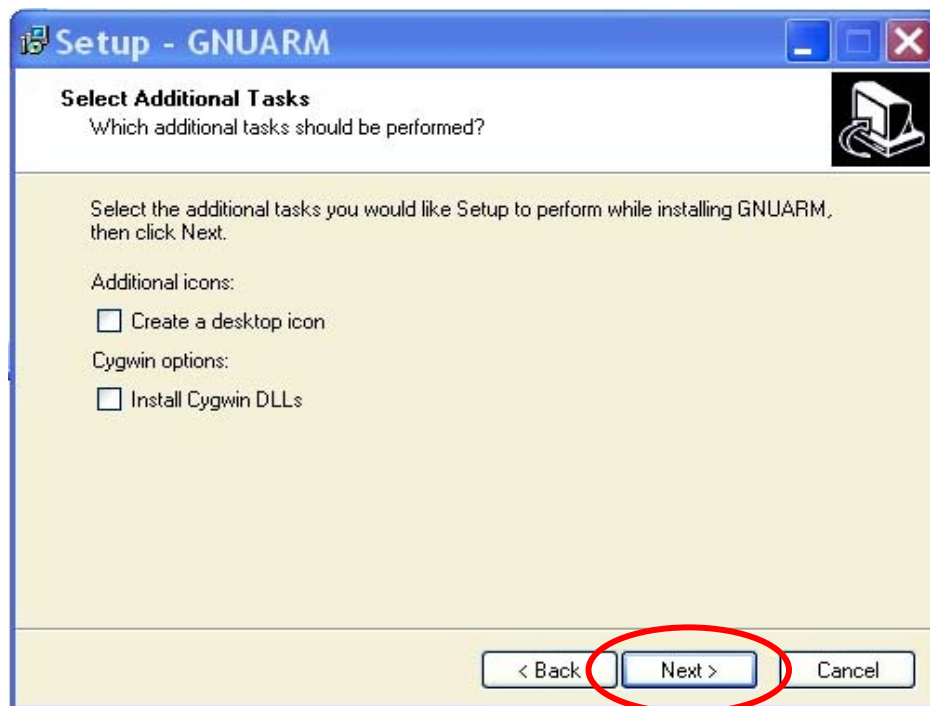


Take the default on this screen. Click **“Next”** to continue.

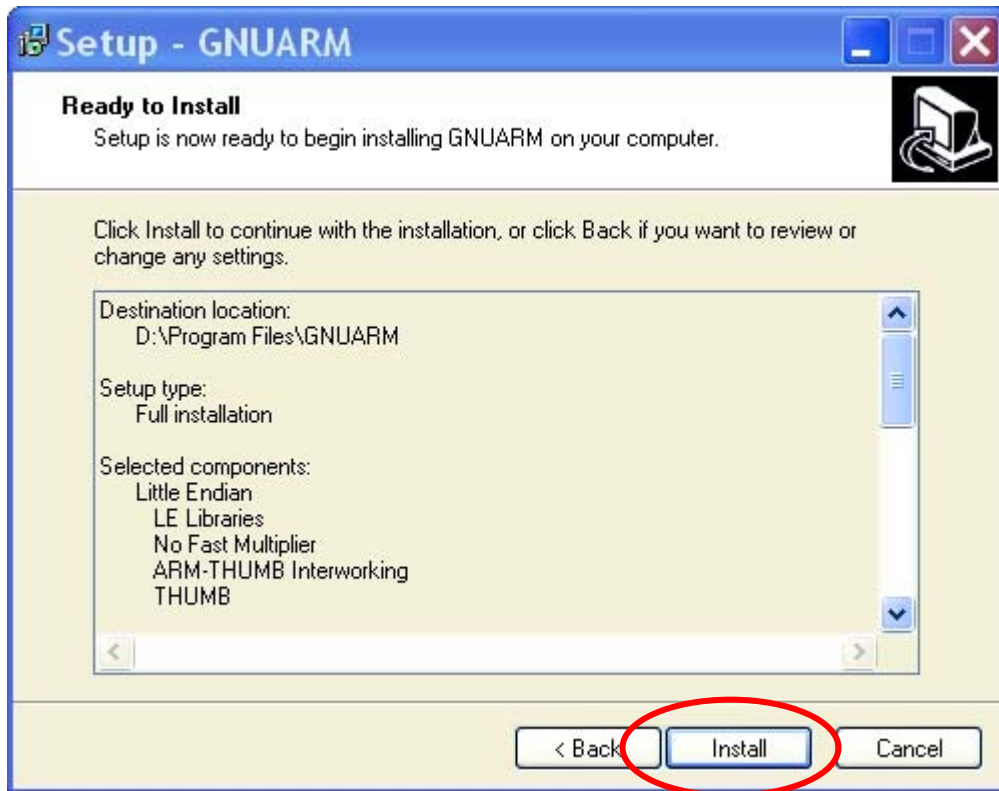


It's very important that you don't check **“Install Cygwin DLLs”** below. We already have the Cygwin DLLs installed from our Cygwin environment installation. In fact, the ARM message boards have had recent comments suggesting that the Cygwin DLL installation from within GNUARM has some problems.

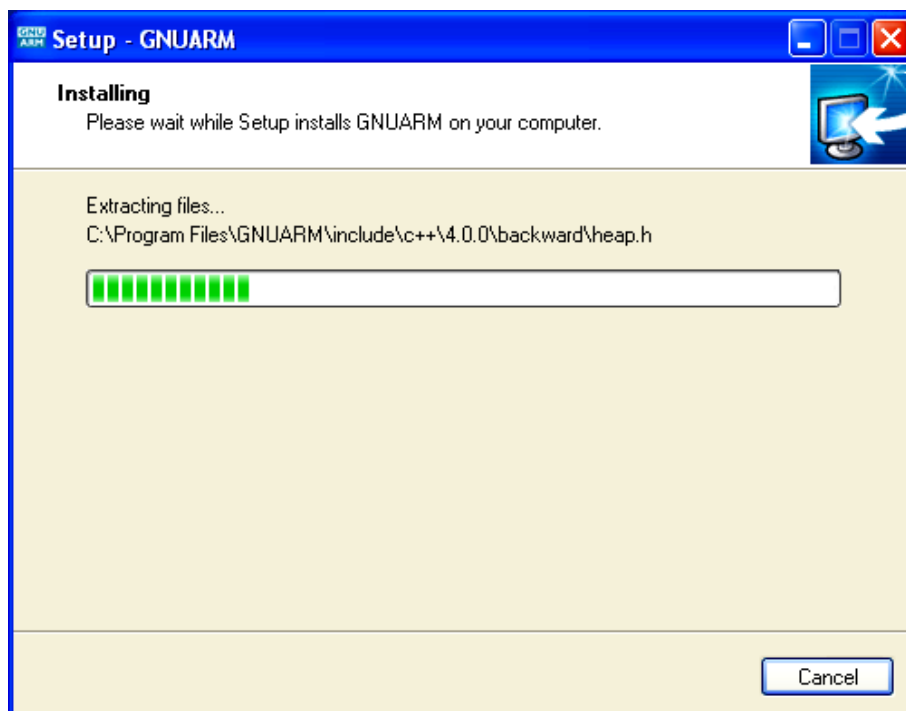
Since all operations are called from within Eclipse, we don't need a **“desktop icon”** either. Click **“Next”** to continue.



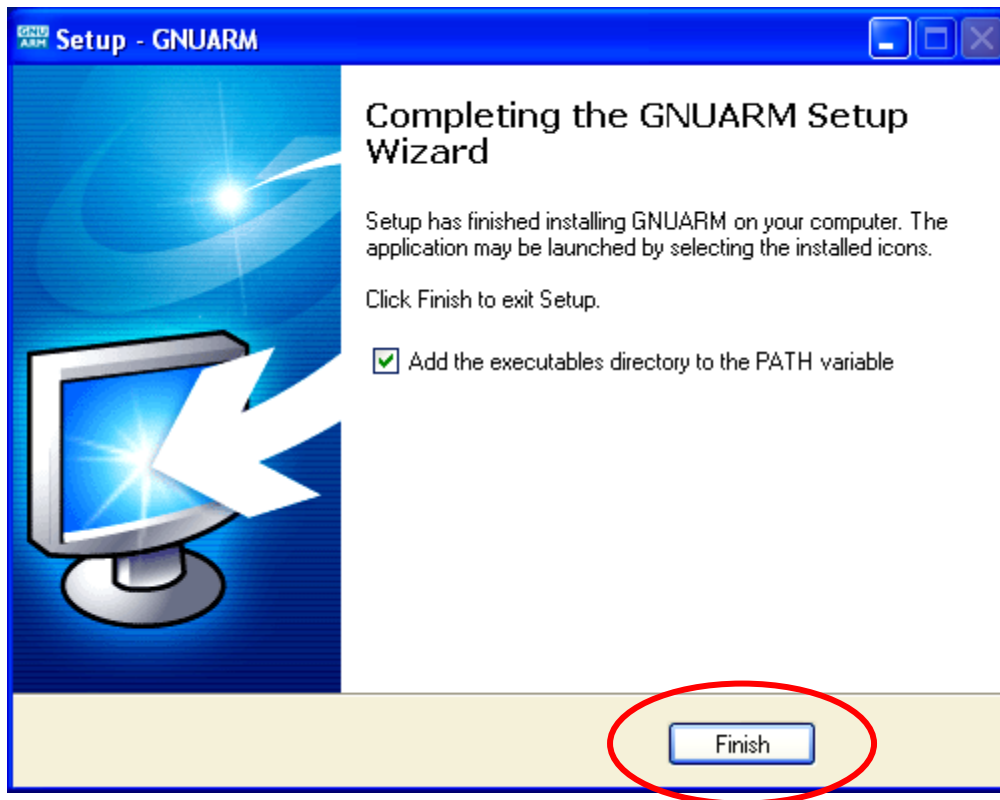
Click on **“Install”** to start the GNUARM installation.



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.

The screenshot shows the Philips website's product information page for the LPC2104/2105/2106 microcontrollers. The page features a blue header with the Philips logo and navigation menus for 'YOUR COUNTRY', 'CONSUMER PRODUCTS', and 'PROFESSIONAL PRODUCTS'. A search bar is also present. Below the header, there are navigation icons and a 'Product Categories' sidebar on the left. The main content area is titled 'Product Information' and includes a brief description of the microcontrollers, a table of links for further information, and sections for 'General description' and 'Features'.

PHILIPS

YOUR COUNTRY ▼ CONSUMER PRODUCTS ▼ PROFESSIONAL PRODUCTS ▼ SEARCH [input]

► PHILIPS SEMICONDUCTORS News Center | Markets | Key Technologies | Products | Jobs | Company Profile |

Product Information

LPC2104/2105/2106; Single-chip 32-bit microcontrollers; 128 kB ISP/IAP Flash with 64 kB/32 kB/16 kB RAM

Information as of 2004-07-10

Stay informed Download datasheet

<input checked="" type="checkbox"/> General description	<input checked="" type="checkbox"/> Features	<input checked="" type="checkbox"/> Applications	<input checked="" type="checkbox"/> Datasheet
<input checked="" type="checkbox"/> Block diagram	<input checked="" type="checkbox"/> Buy online	<input checked="" type="checkbox"/> Support & tools	<input checked="" type="checkbox"/> Email/translate
<input checked="" type="checkbox"/> Products & packages	<input checked="" type="checkbox"/> Parametrics	<input checked="" type="checkbox"/> Similar products	<input checked="" type="checkbox"/> Disclaimer

General description

The LPC2104, 2105 and 2106 are based on a 16/32 bit ARM7TDMI-S CPU with real-time emulation and embedded trace support, together with 128 kbytes (kB) of embedded high speed flash memory. A 128 bit wide memory interface and a unique accelerator architecture enable 32 bit code execution at maximum clock rate. For critical code size applications, the alternative 16-bit Thumb Mode reduces code by more than 30pct with minimal performance penalty.

Due to their tiny size and low power consumption, these microcontrollers are ideal for applications where miniaturization is a key requirement, such as access control and point-of-sale. With a wide range of serial communications interfaces and on-chip SRAM options up to 64 kilobytes, they are very well suited for communication gateways and protocol converters, soft modems, voice recognition and low end imaging, providing both large buffer size and high processing power. Various 32 bit timers, PWM channels and 32 GPIO lines make these microcontrollers particularly suitable for industrial control and medical systems.

Features

Key features

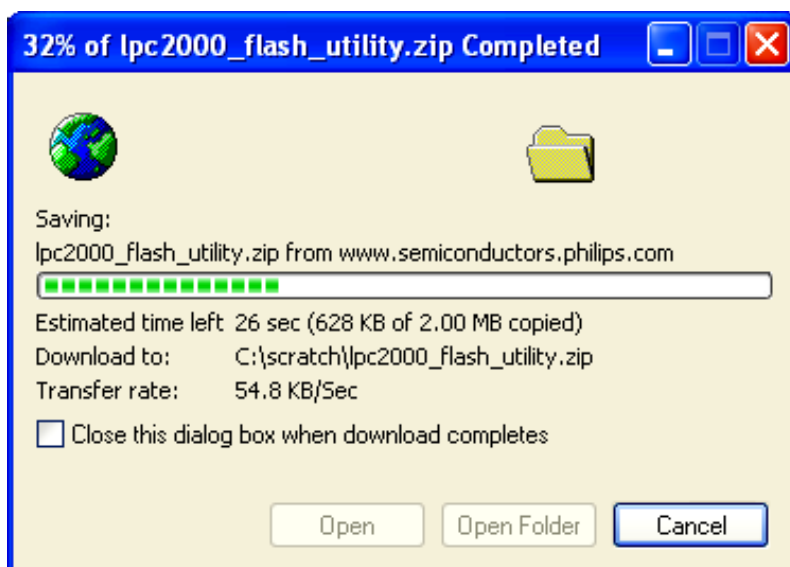
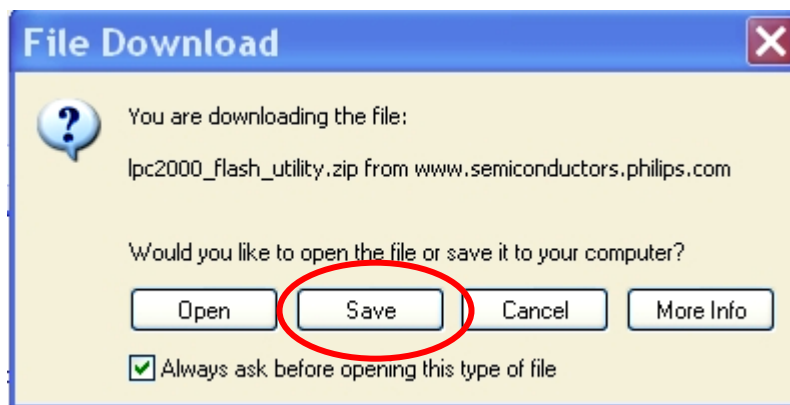
- 16/32 bit ARM7TDMI-S processor.
- 16/32/64 kB on-chip Static RAM.

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)

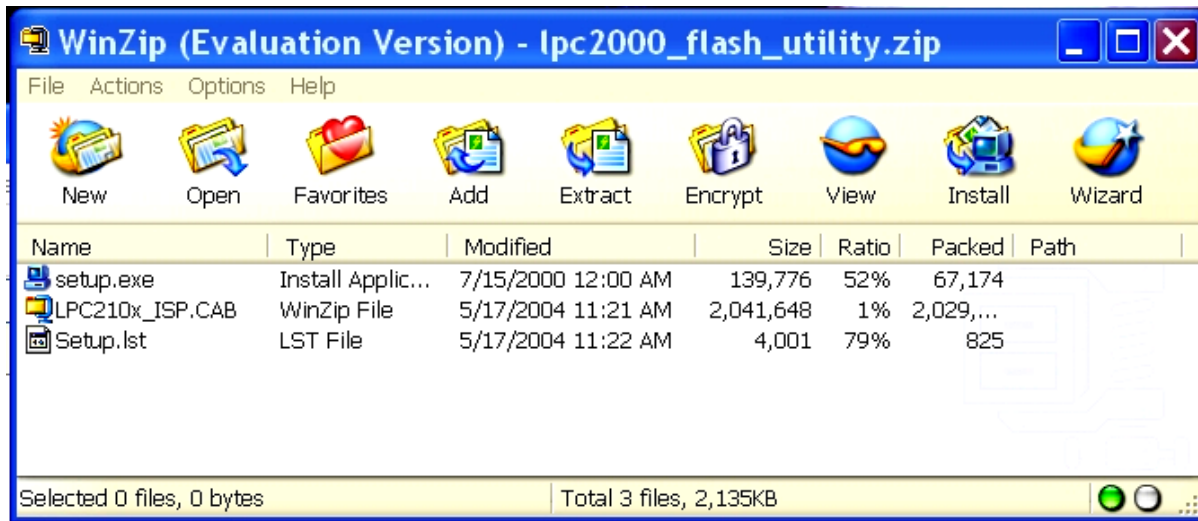
Support & tools

- PDF LPC2104 Single Chip 32-bit Microcontroller Erratasheet(date 2004-06-01)
- PDF LPC2105 Single Chip 32-bit Microcontroller Erratasheet(date 2004-06-01)
- PDF LPC2106 Single Chip 32-bit Microcontroller Erratasheet(date 2004-06-01)
- PDF LPC2104 Erratasheet(date 2003-12-10)
- PDF LPC2105 Erratasheet(date 2003-12-10)
- PDF LPC2106 Erratasheet(date 2003-12-10)
- PDF Philips Microcontroller Line Card(date 2004-03-05)
- PDF LPC2104/2105/2106 Leaflet(date 2004-02-24)
- PDF Philips -- The Innovation Leader in Mococontrollers(date 2004-06-30)
- PDF LPC2106/2105/2104 User Manual(date 2003-09-17)
- ZIP LPC2000 Flash Utility(date 2004-03-01)
- WEBSITE Development Tools for LPC2100 devices(date 2003-05-21)

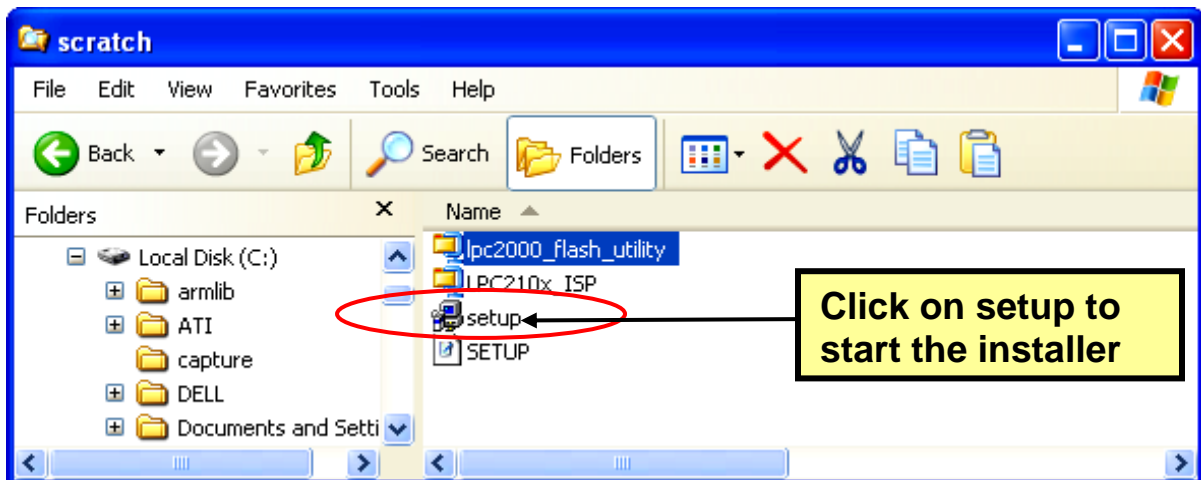
As before, we'll save the downloaded zip file in our empty **c:/scratch** directory. This is a fairly short download, only about 2 megabytes.



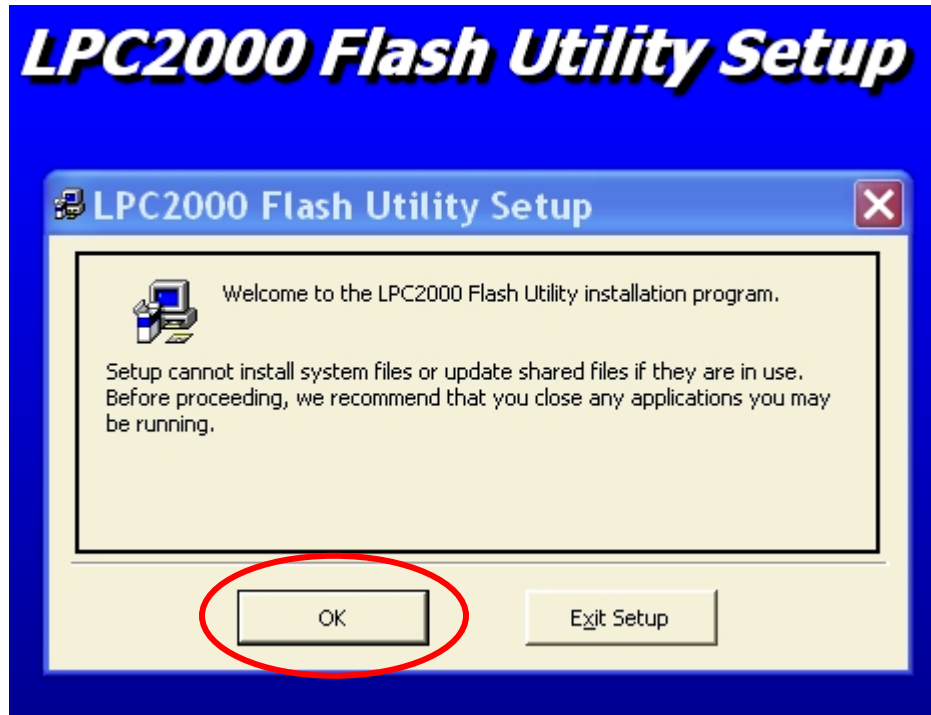
We'll use WinZip to unzip this into the **c:/scratch** directory.



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.



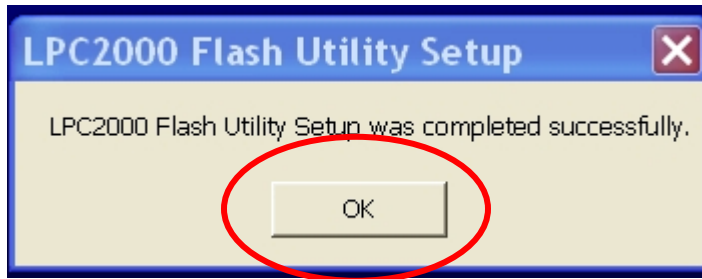
The LPC2000 Flash Utility setup now starts. Click on **OK** to proceed.



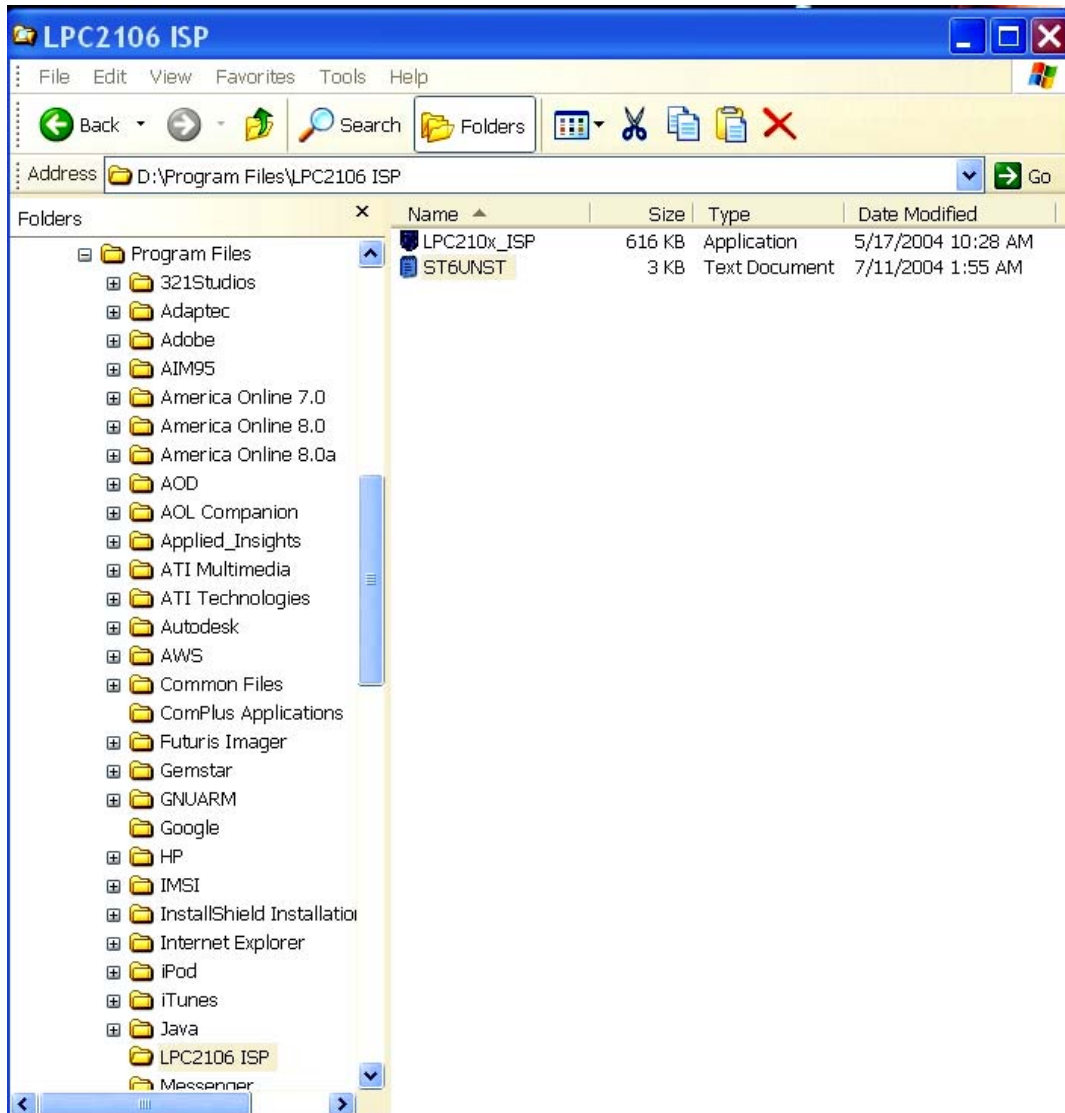
Take the default on this screen below and let it install the LPC2000 Flash Utility into the Program Files directory.



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



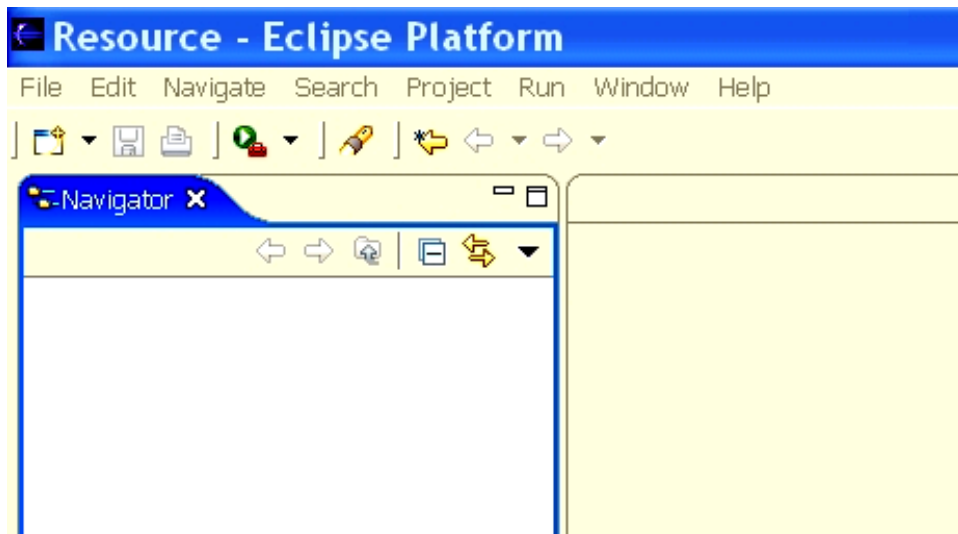
Here we see the utility residing in the Program Files directory, just as promised.



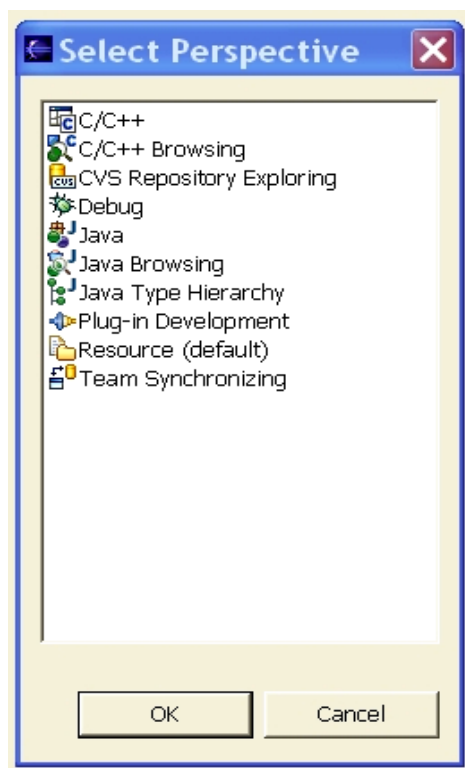
Now that the Philips LPC2000 Flash Utility is properly installed on our computer, we'd like to install it into Eclipse so that it can be invoked from the RUN pull-down menu under the "external tools" option. Start Eclipse by clicking on the desktop icon.



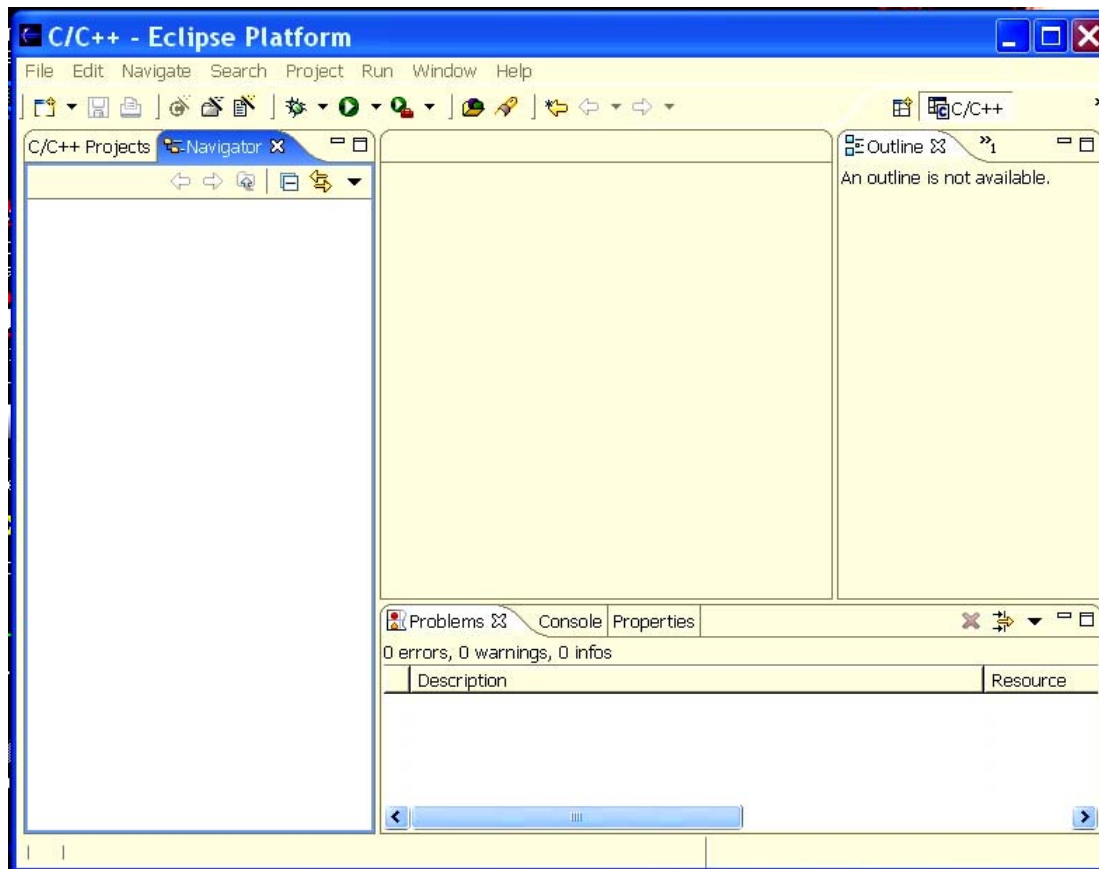
The layout of the Eclipse screen is called a "perspective." The default perspective is the "resource" perspective, as shown below.



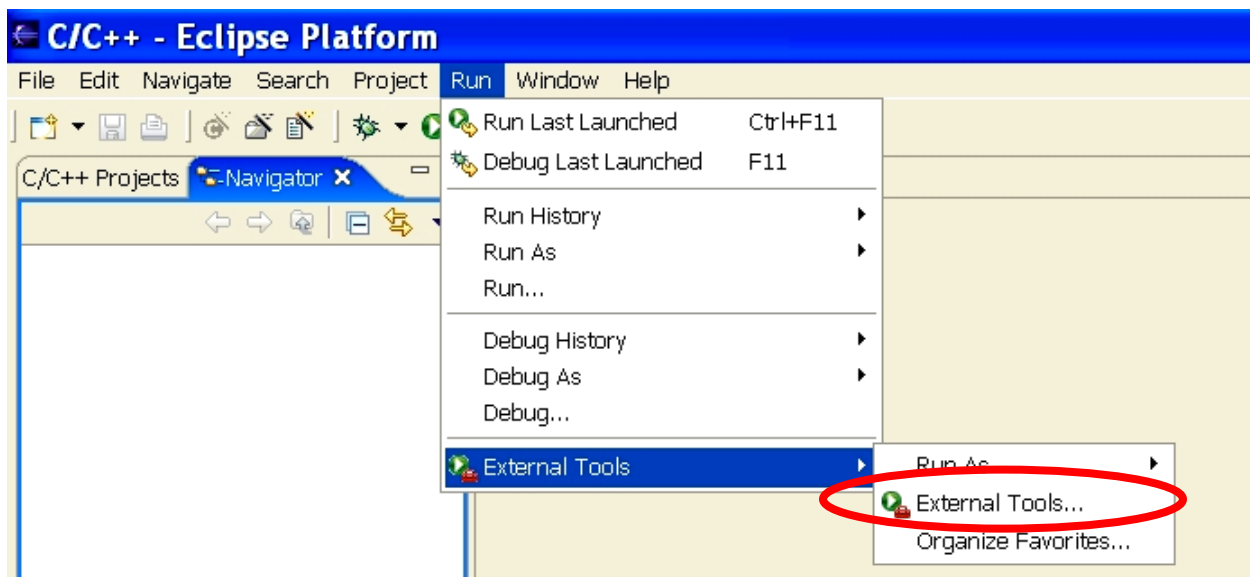
We need to change it into the C/C++ perspective. In the **Window** pull-down menu, select **Window – Open Perspective – Other – C/C++** and then click **OK**.



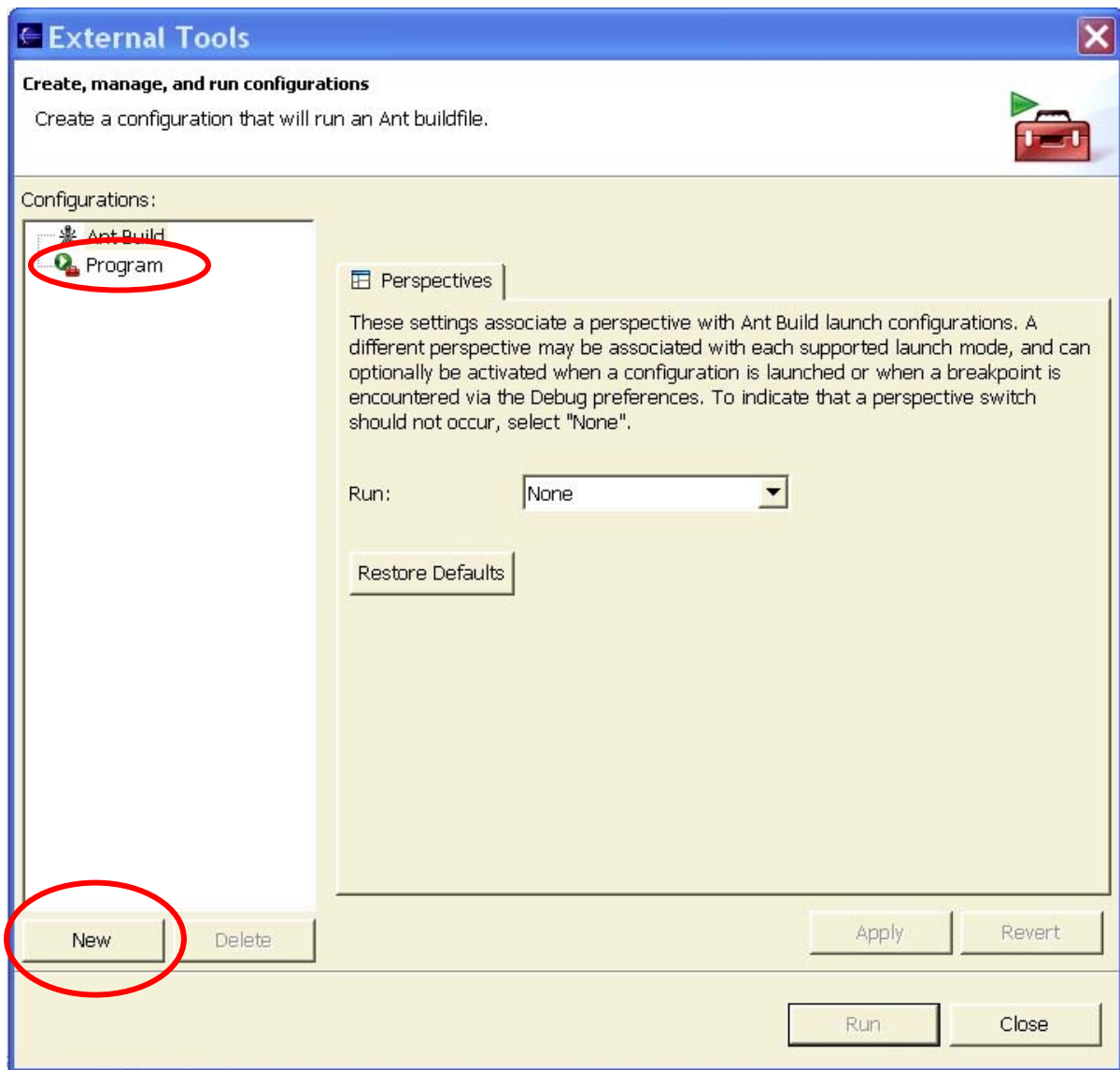
Eclipse will now switch to the **C/C++** perspective shown below and will remember it when you exit.



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**.

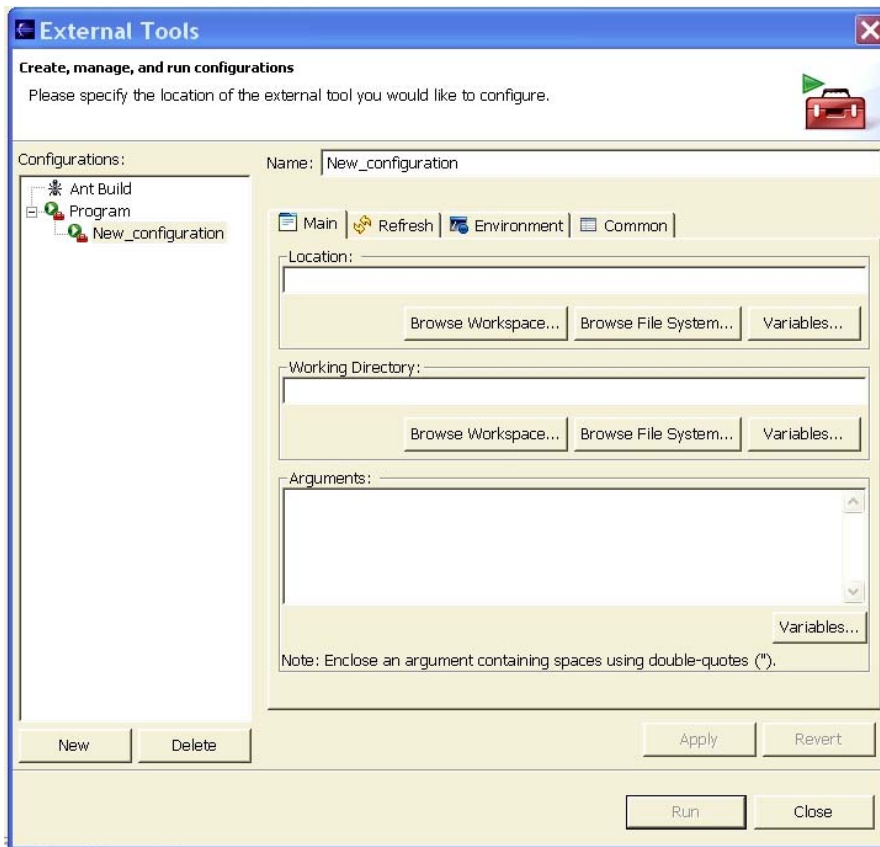


Note below that there's a new program under the "program" tree with the name **New_configuration** and there's no specifications as to what it is.

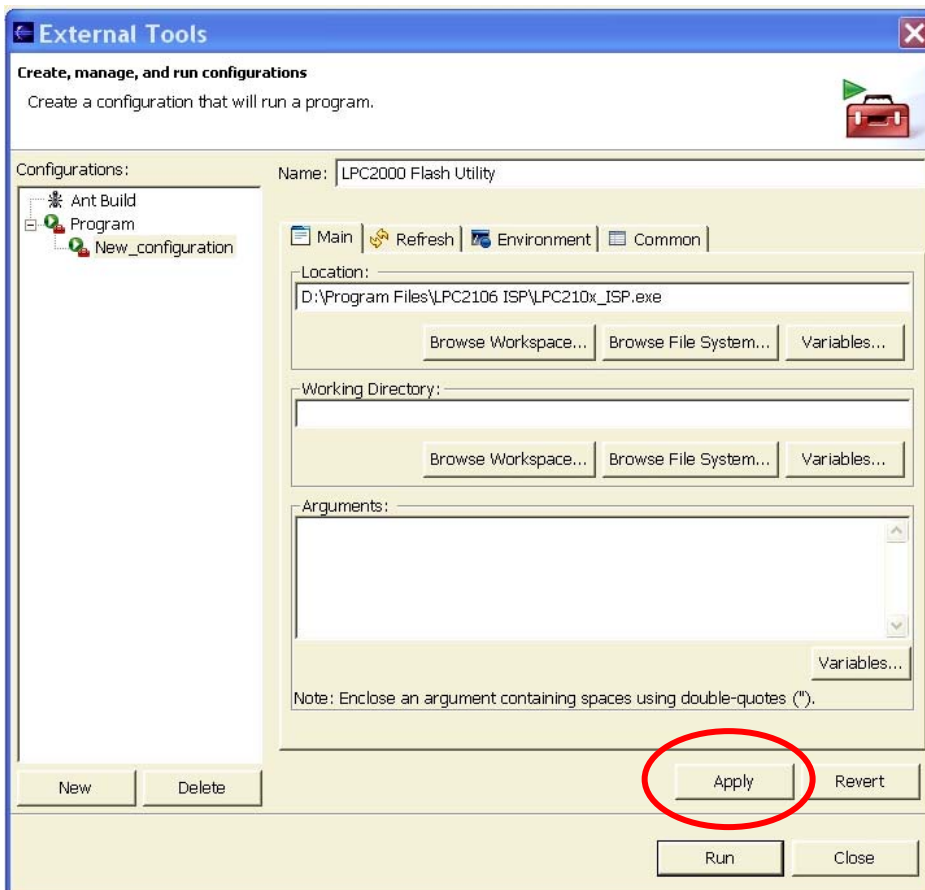
In the **Name** text box, replace **New-configuration** with **LPC2000 Flash Utility**.

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**.

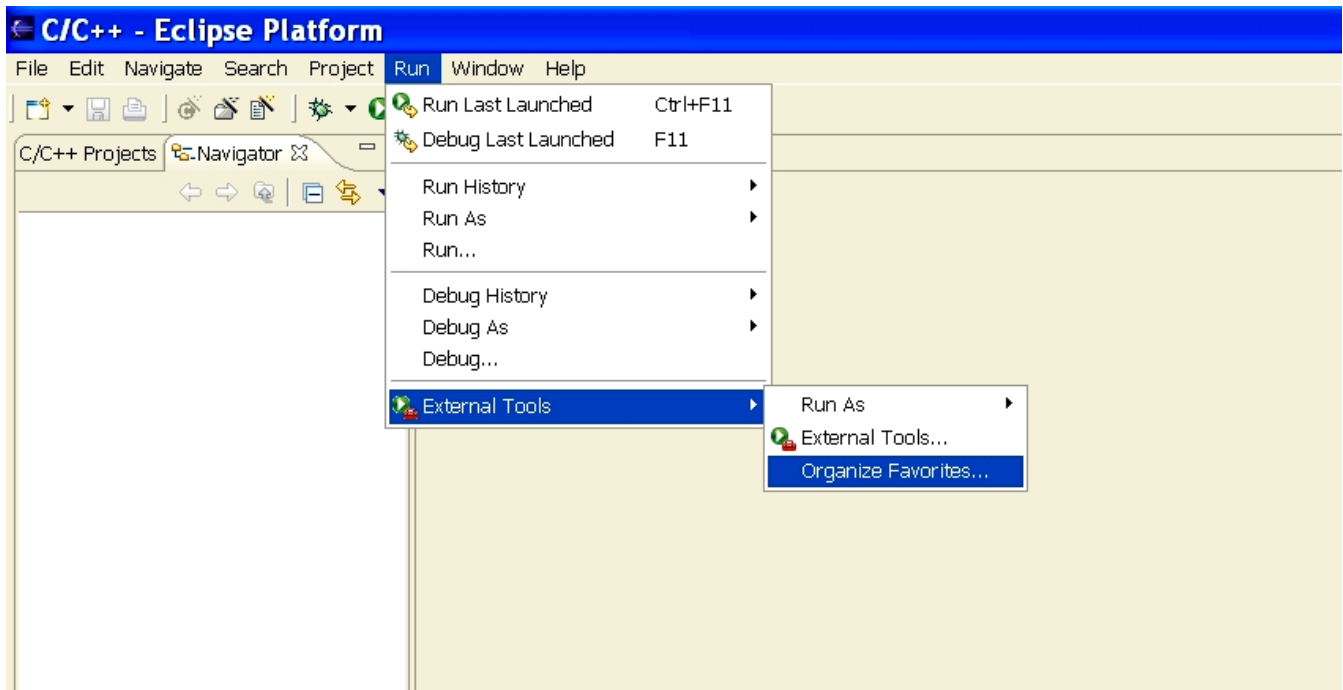
Here's the External Tools window before editing.



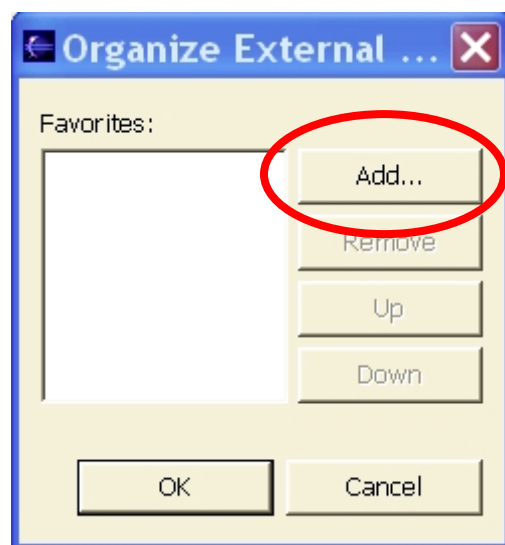
Here's the External Tools window after our modifications. Click on **Apply** to accept.



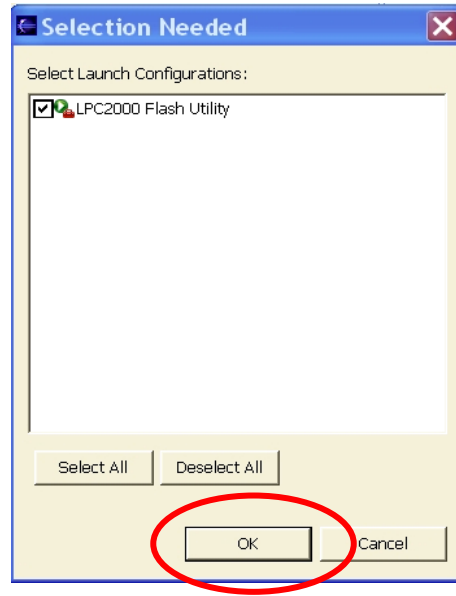
Close everything out and return to the **Run** pull-down menu. Select **Run – External Tools – Organize Favorites**.



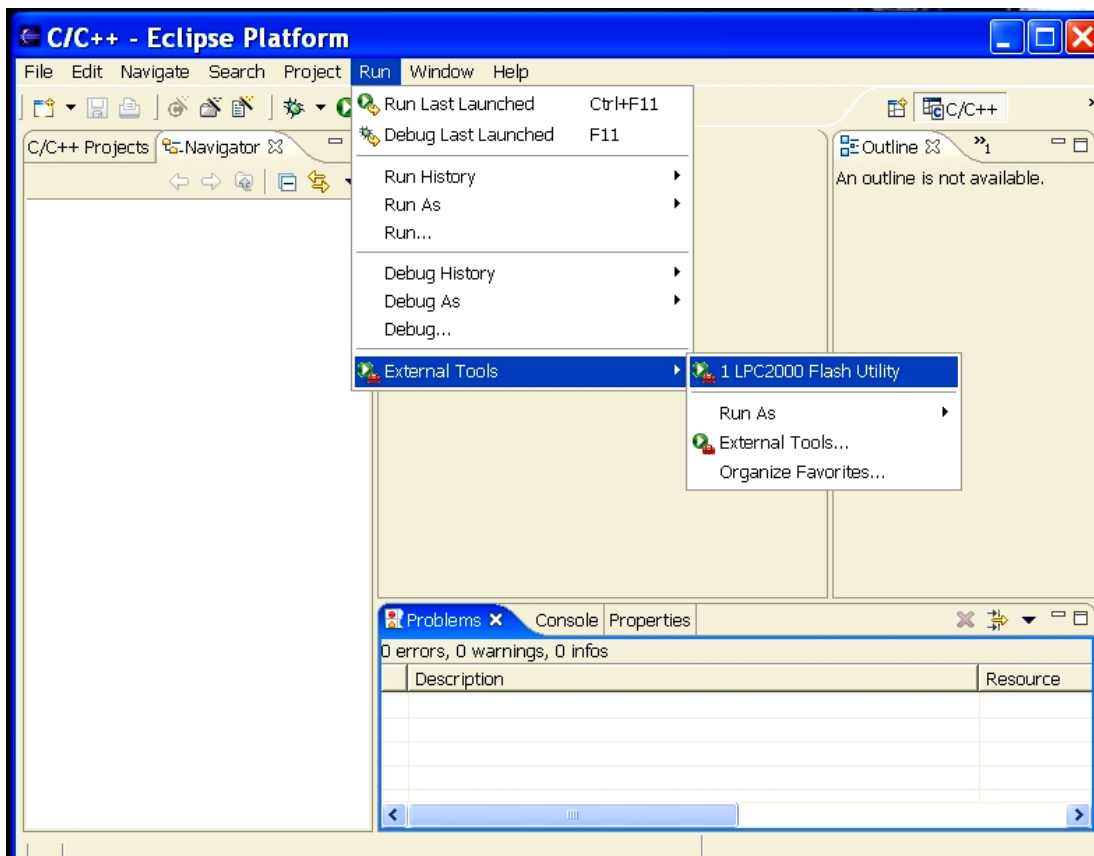
We're now going to put the Philips PLC2000 Flash Utility into the "favorites" list. Click on **Add** in the window below.



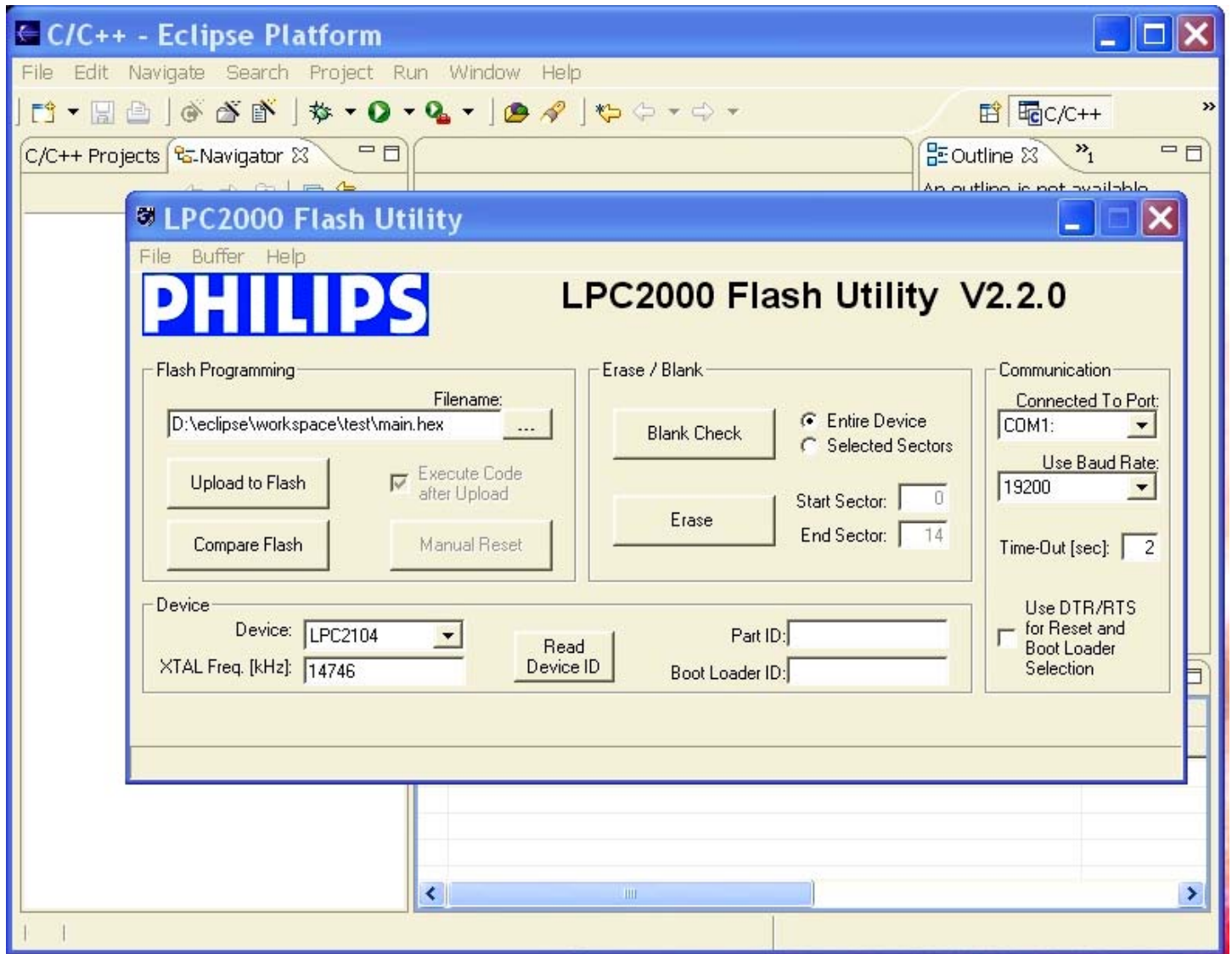
Click the selection box for LPC2000 Flash Utility. This will add it to the favorites list.



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.



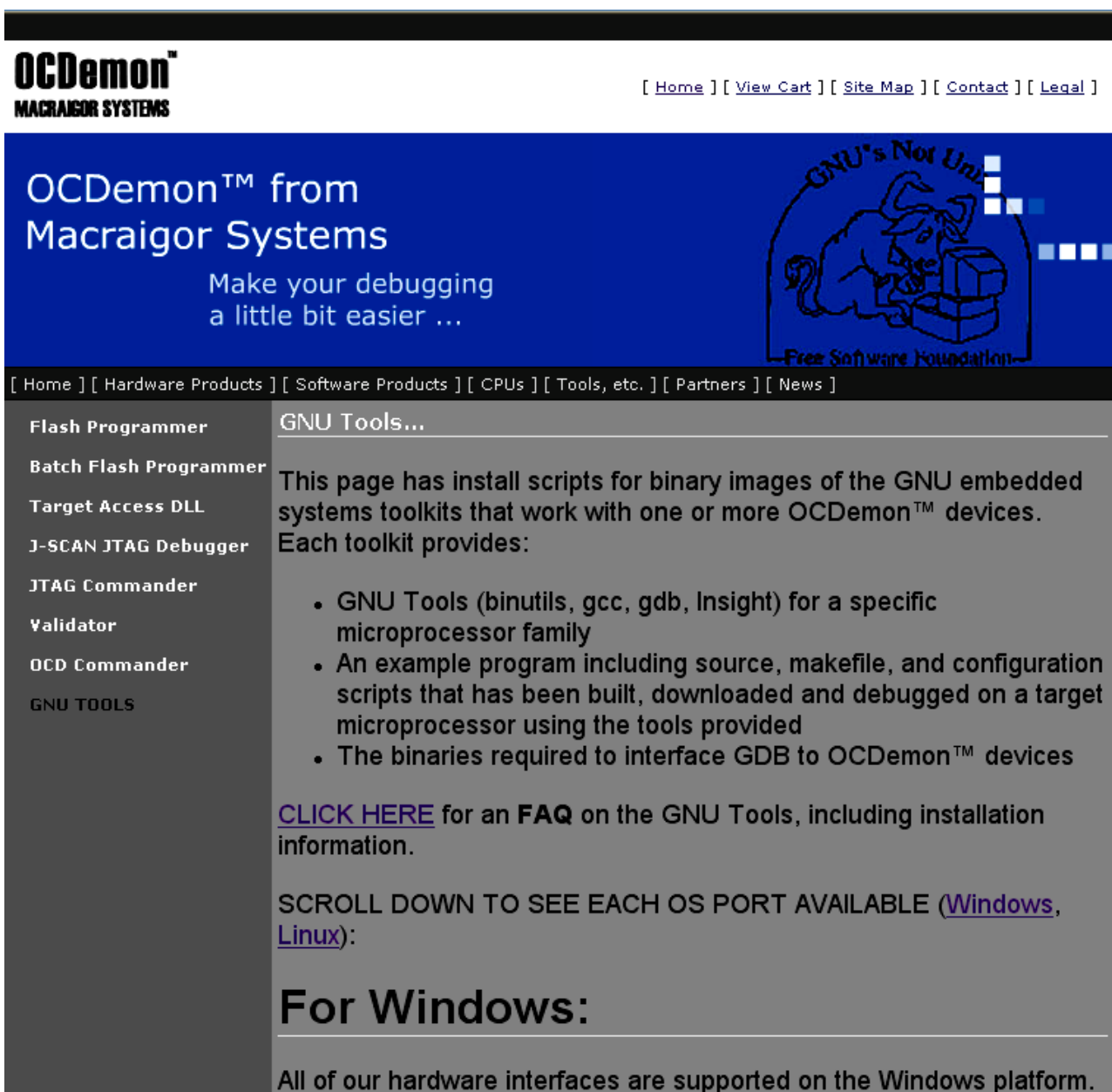
Now cancel the LPC2000 Flash Utility and quit Eclipse.

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:

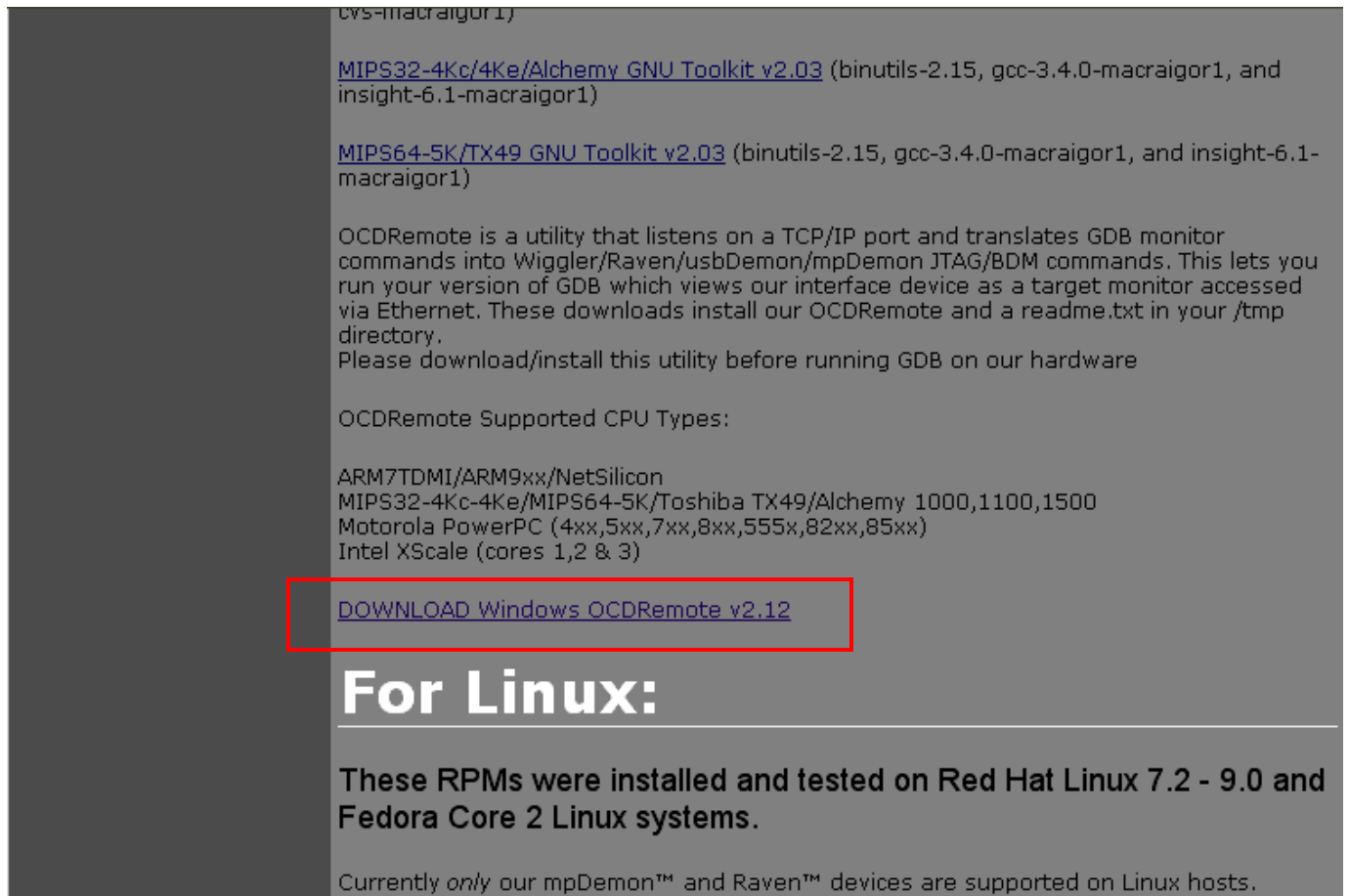
http://www.macraigor.com/full_gnu.htm

You should see the following screen open up.



The screenshot shows the website for OCDemon™ from Macraigor Systems. At the top left is the logo "OCDemon™" with "MACRAIGOR SYSTEMS" underneath. To the right are navigation links: [Home] [View Cart] [Site Map] [Contact] [Legal]. Below this is a blue banner with the text "OCDemon™ from Macraigor Systems" and "Make your debugging a little bit easier ...". On the right of the banner is the GNU logo with the text "GNU's Not Unix!" and "Free Software Foundation". Below the banner is another set of navigation links: [Home] [Hardware Products] [Software Products] [CPUs] [Tools, etc.] [Partners] [News]. A sidebar on the left contains a menu with items: Flash Programmer, Batch Flash Programmer, Target Access DLL, J-SCAN JTAG Debugger, JTAG Commander, Validator, OCD Commander, and GNU TOOLS. The main content area is titled "GNU Tools..." and contains the following text: "This page has install scripts for binary images of the GNU embedded systems toolkits that work with one or more OCDemon™ devices. Each toolkit provides:" followed by a bulleted list: "• GNU Tools (binutils, gcc, gdb, Insight) for a specific microprocessor family", "• An example program including source, makefile, and configuration scripts that has been built, downloaded and debugged on a target microprocessor using the tools provided", and "• The binaries required to interface GDB to OCDemon™ devices". Below the list is a link: "CLICK HERE for an FAQ on the GNU Tools, including installation information." At the bottom of the main content area is the text: "SCROLL DOWN TO SEE EACH OS PORT AVAILABLE (Windows, Linux):" followed by a large heading "For Windows:" and a final line: "All of our hardware interfaces are supported on the Windows platform."

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**”.



...ws-macraigor1)

[MIPS32-4Kc/4Ke/Alchemy GNU Toolkit v2.03](#) (binutils-2.15, gcc-3.4.0-macraigor1, and insight-6.1-macraigor1)

[MIPS64-5K/TX49 GNU Toolkit v2.03](#) (binutils-2.15, gcc-3.4.0-macraigor1, and insight-6.1-macraigor1)

OCDRemote is a utility that listens on a TCP/IP port and translates GDB monitor commands into Wiggler/Raven/usbDemon/mpDemon JTAG/BDM commands. This lets you run your version of GDB which views our interface device as a target monitor accessed via Ethernet. These downloads install our OCDRemote and a readme.txt in your /tmp directory.
Please download/install this utility before running GDB on our hardware

OCDRemote Supported CPU Types:

ARM7TDMI/ARM9xx/NetSilicon
MIPS32-4Kc-4Ke/MIPS64-5K/Toshiba TX49/Alchemy 1000,1100,1500
Motorola PowerPC (4xx,5xx,7xx,8xx,555x,82xx,85xx)
Intel XScale (cores 1,2 & 3)

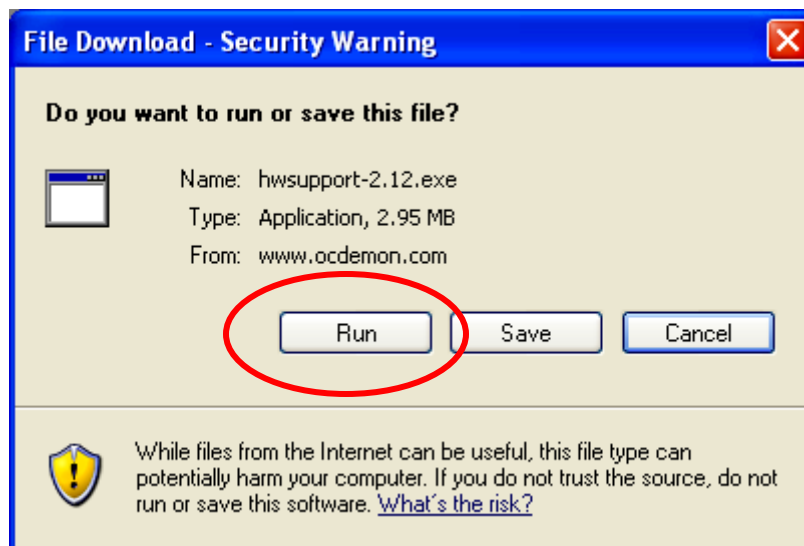
[DOWNLOAD Windows OCDRemote v2.12](#)

For Linux:

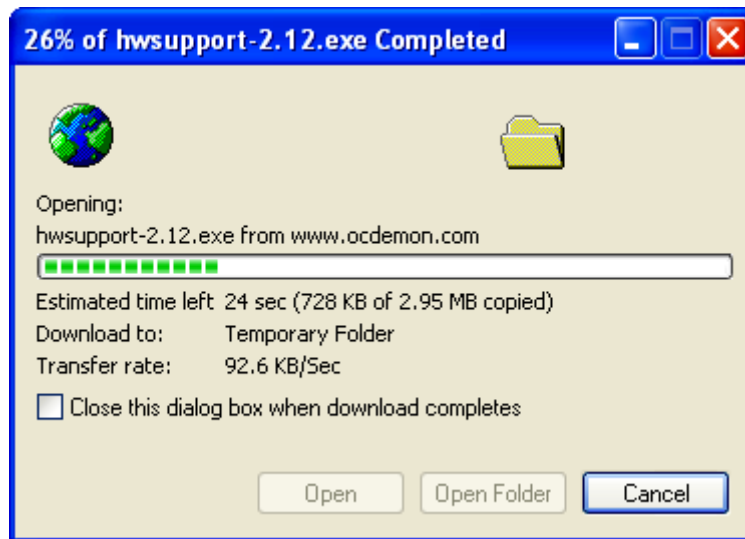
These RPMs were installed and tested on Red Hat Linux 7.2 - 9.0 and Fedora Core 2 Linux systems.

Currently *only* our mpDemon™ and Raven™ devices are supported on Linux hosts.

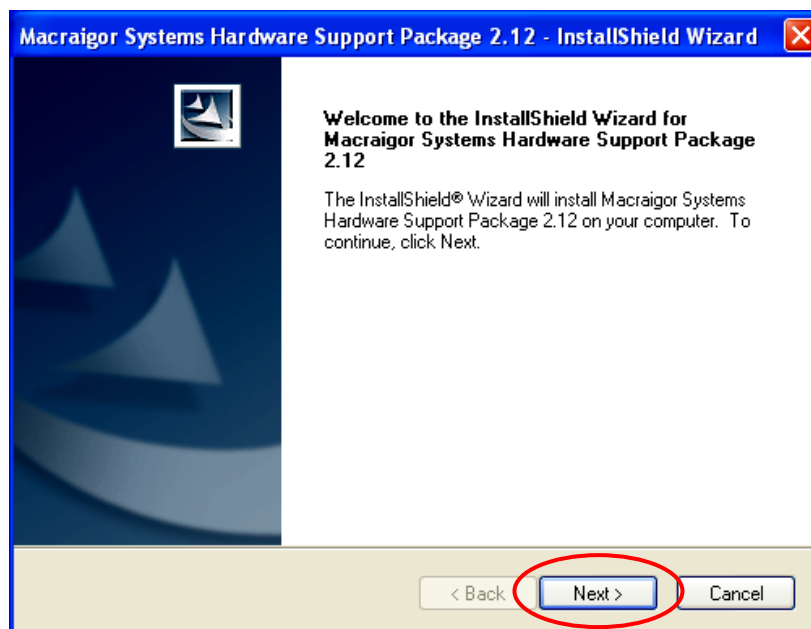
Click on “**Run**” so it will download and immediately install **OCDRemote**.



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



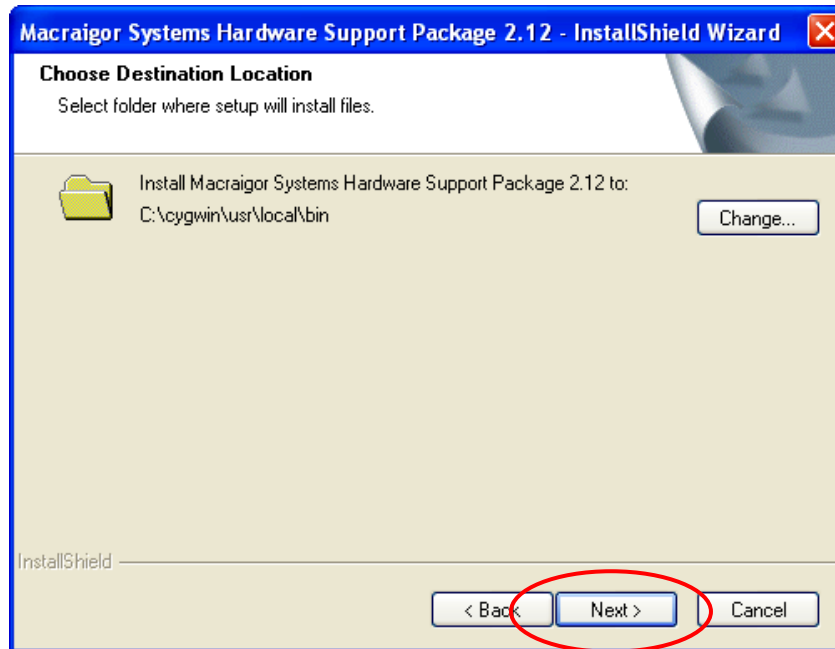
The Macraigor installer should start up; just click “**Next**” to continue.



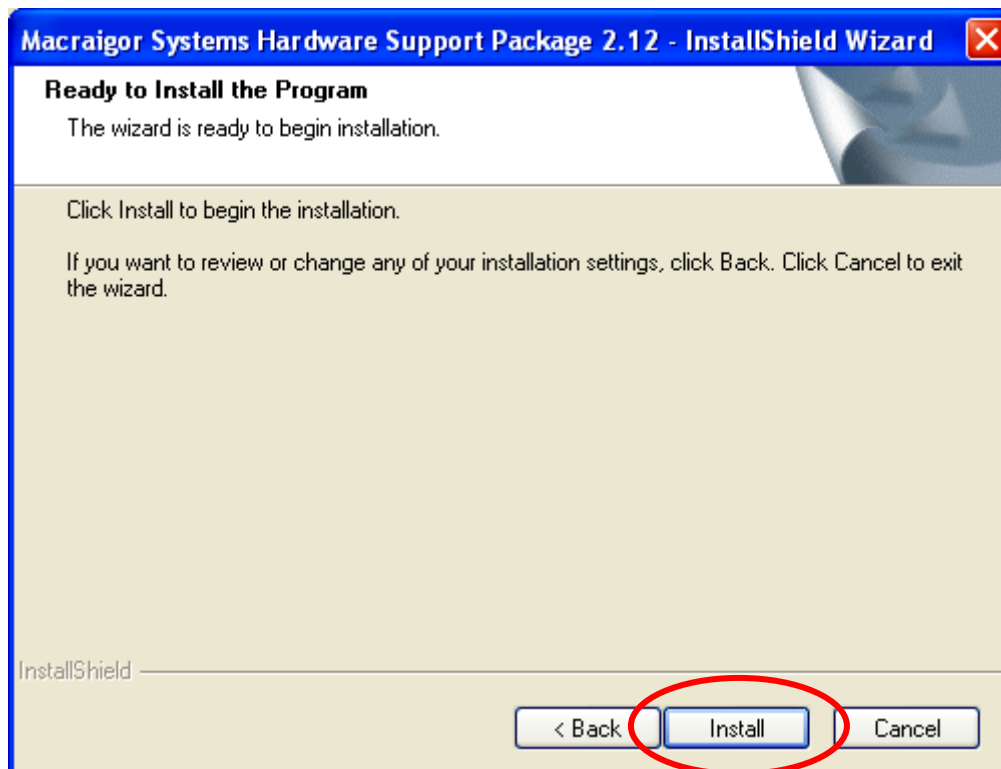
The next screen lets you choose where **OCDRemote** is installed. **OCDRemote** normally installs in **c:/cygwin/usr/local/bin**.

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

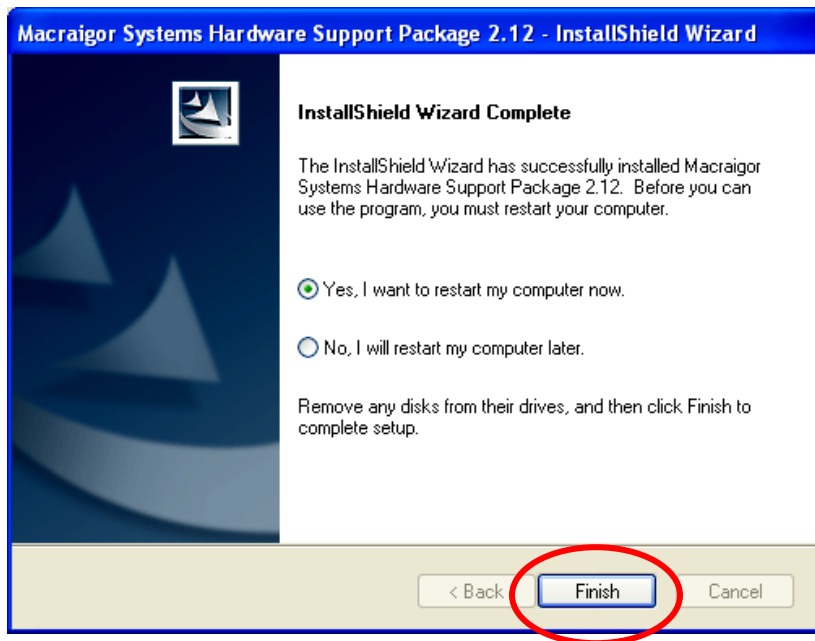
Click on “**Next**” to accept `c:/cygwin/usr/local/bin` as the **OCDRemote** installation directory.



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

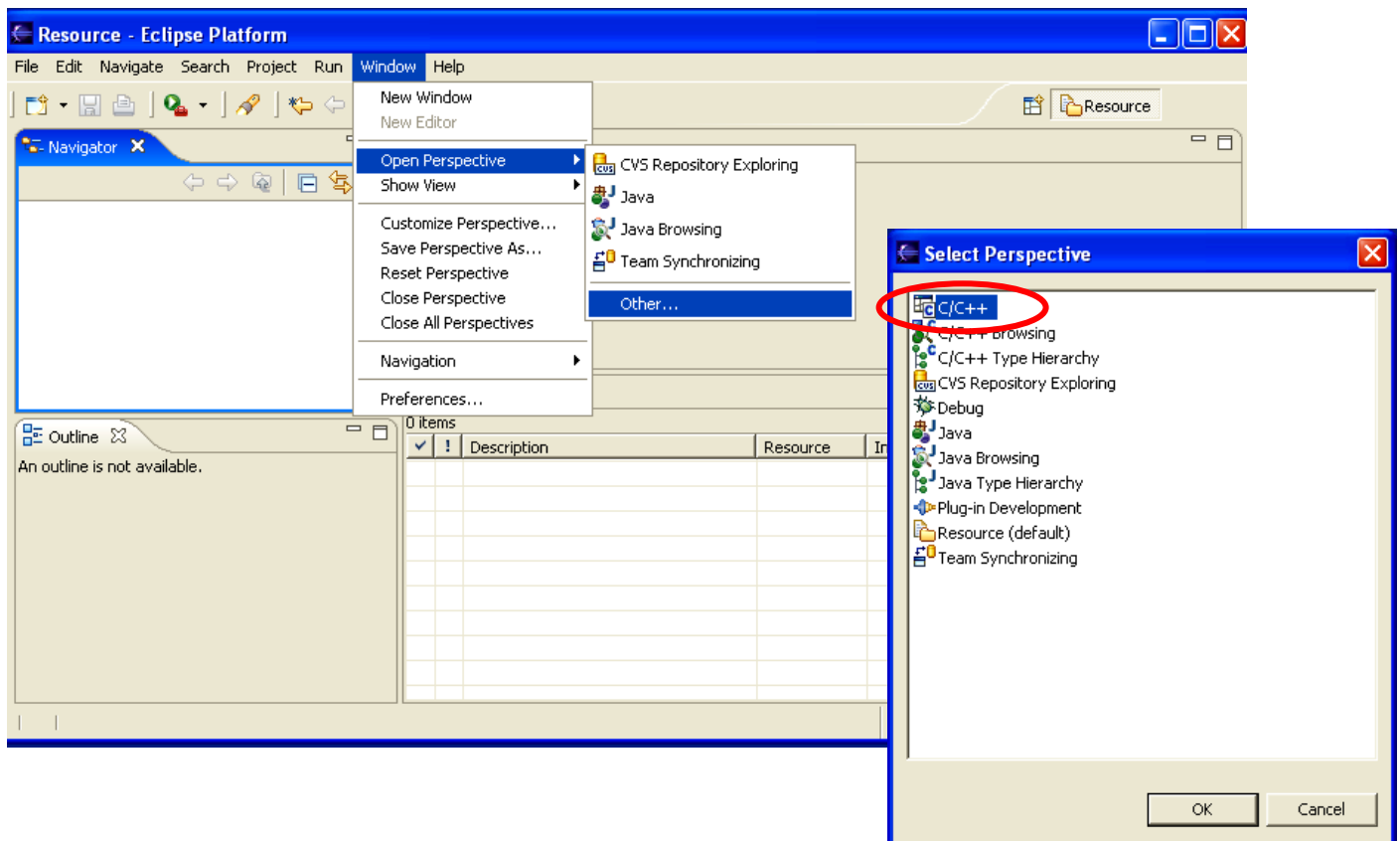


The Wizard completion screen lets you restart your computer to put **OCDRemote** into the Windows registry.

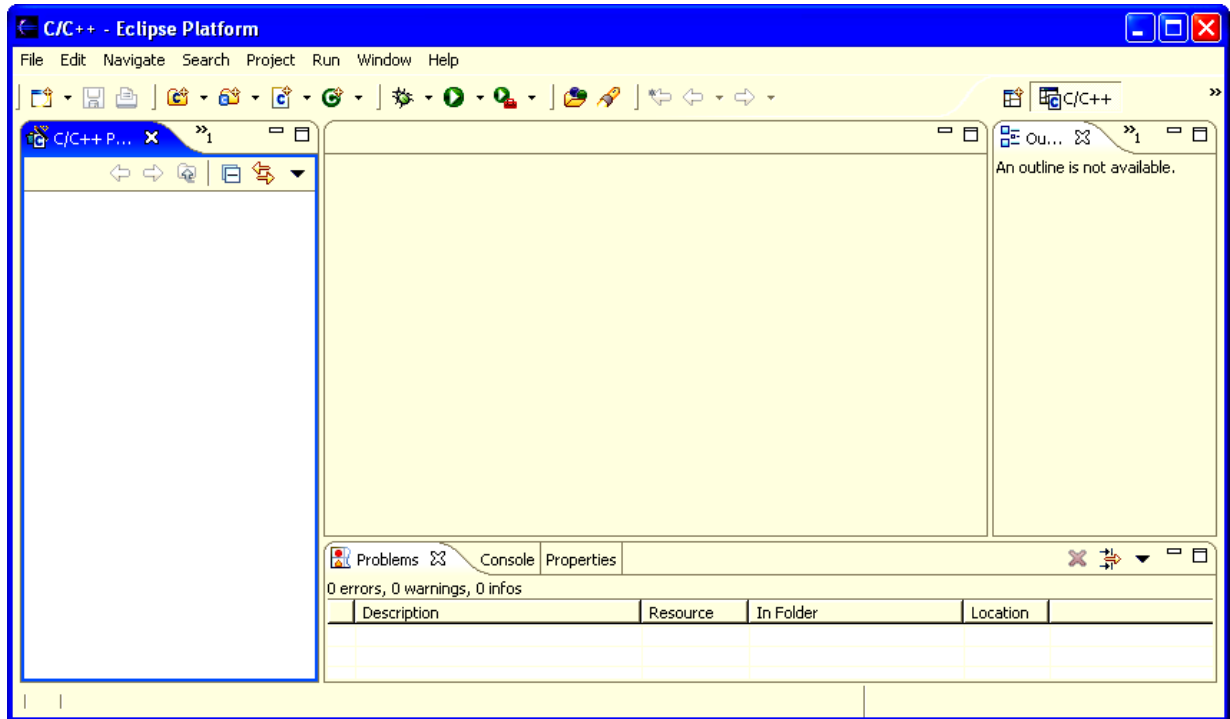


Just like the Philips ISP Flash Utility, we should install the Macraigor **OCDremote** utility as an “external tool” that can be accessed easily from the Eclipse CDT **RUN** pull-down menu.

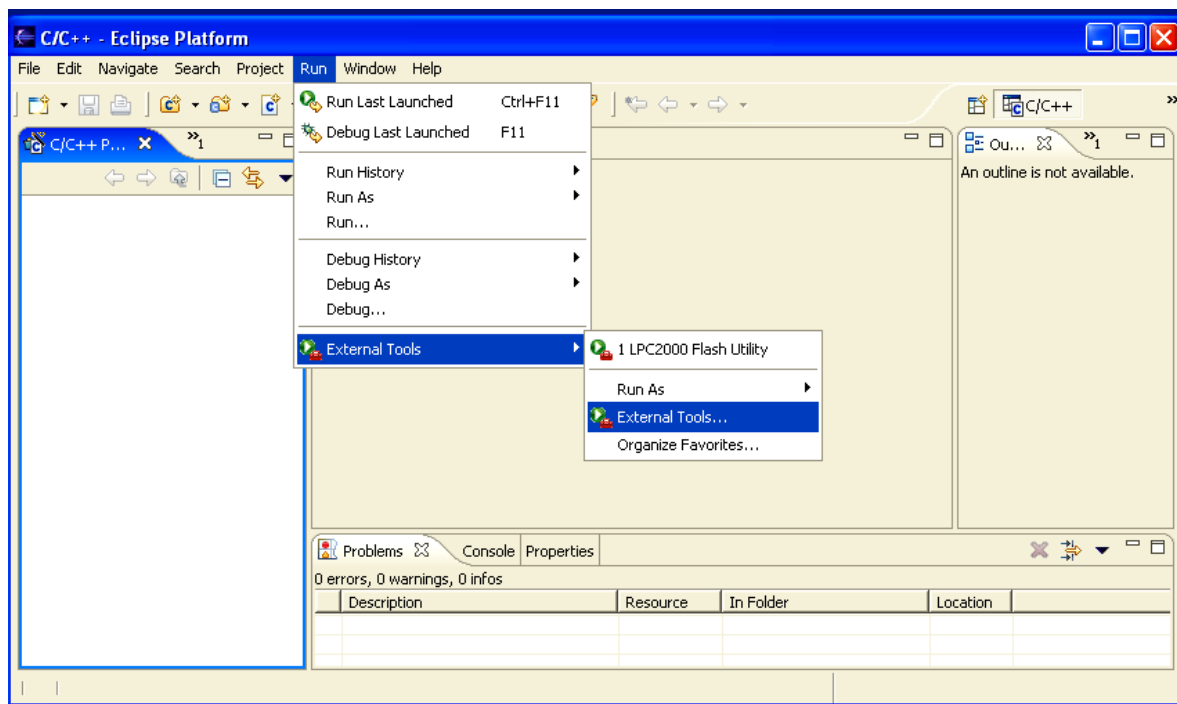
Start up Eclipse and, if necessary, switch to the C/C++ perspective by clicking “**Window – Open Perspective – Other – C/C++.**”



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.

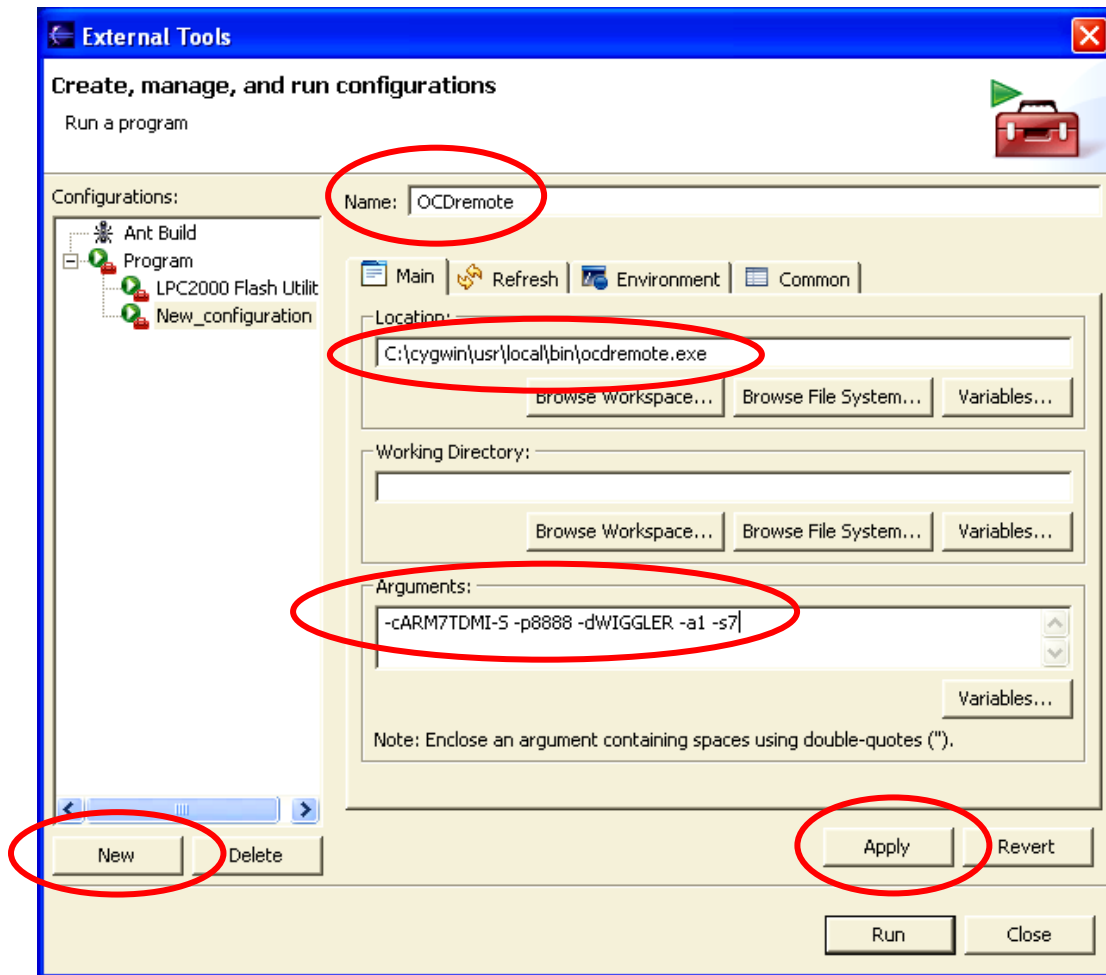


Click on “**New**” and replace the name with **OCDremote**. Use the “**browse file system**” to find it. It should be in the directory **c:/cygwin/usr/local/bin**.

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

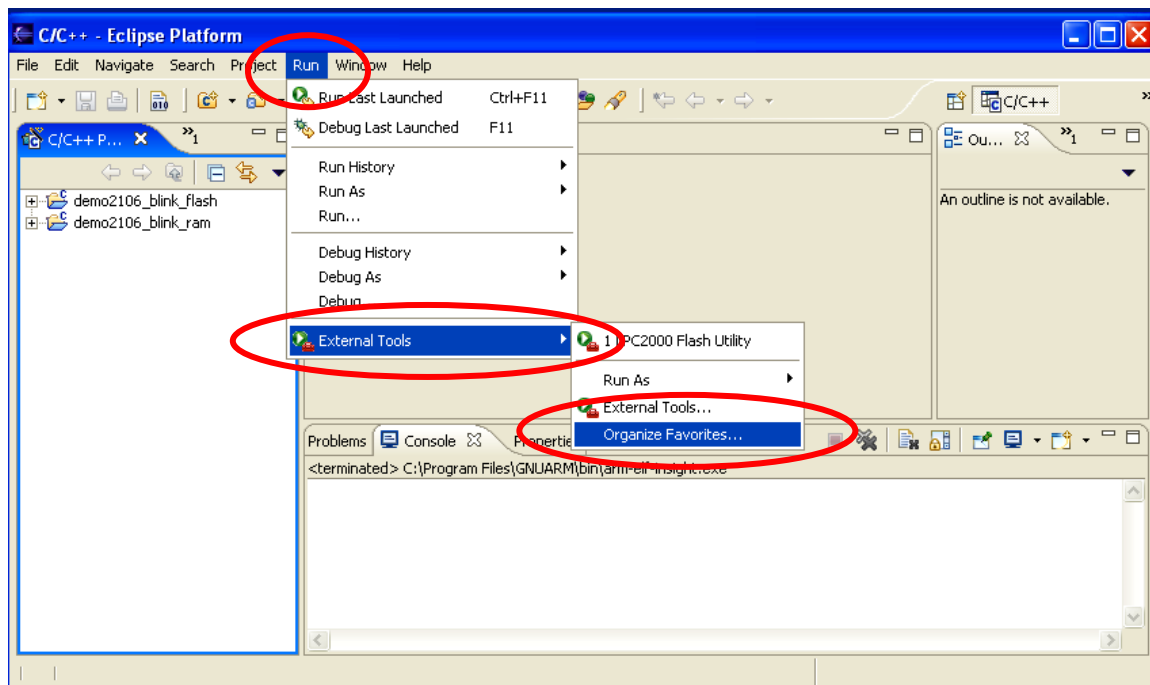
-cARM7TDMI-S	specifies the CPU being accessed
-p8888	specifies the pseudo TCP-IP port being used
-dWIGGLER	specifies the JTAG hardware being used
-a1	specifies LPT1 for the Wiggler
-s7	specifies next-to-slowest speed

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

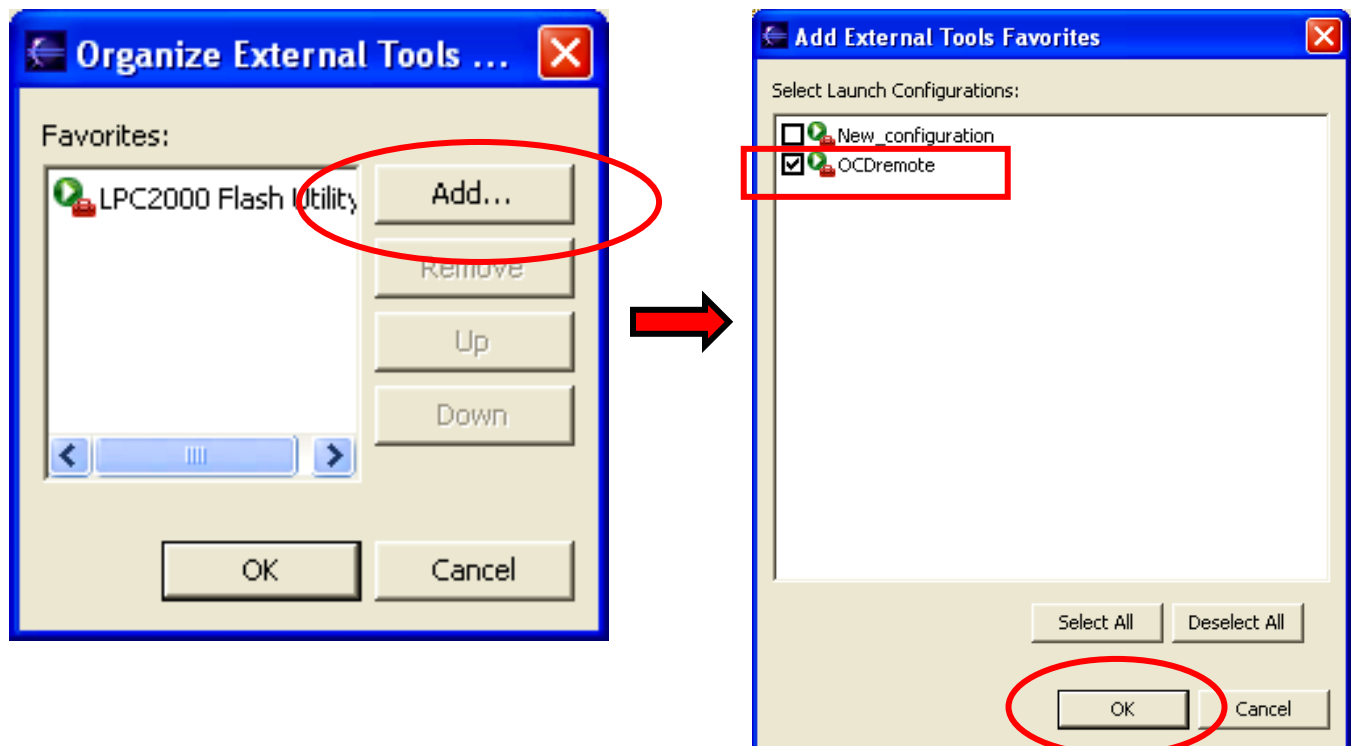


Just like the Philips LPC2000 Flash Utility, we’d like to include the **OCDremote** application in our list of “**favorite**” External Tools. This allows us to quickly start the **OCDremote** JTAG server from within Eclipse.

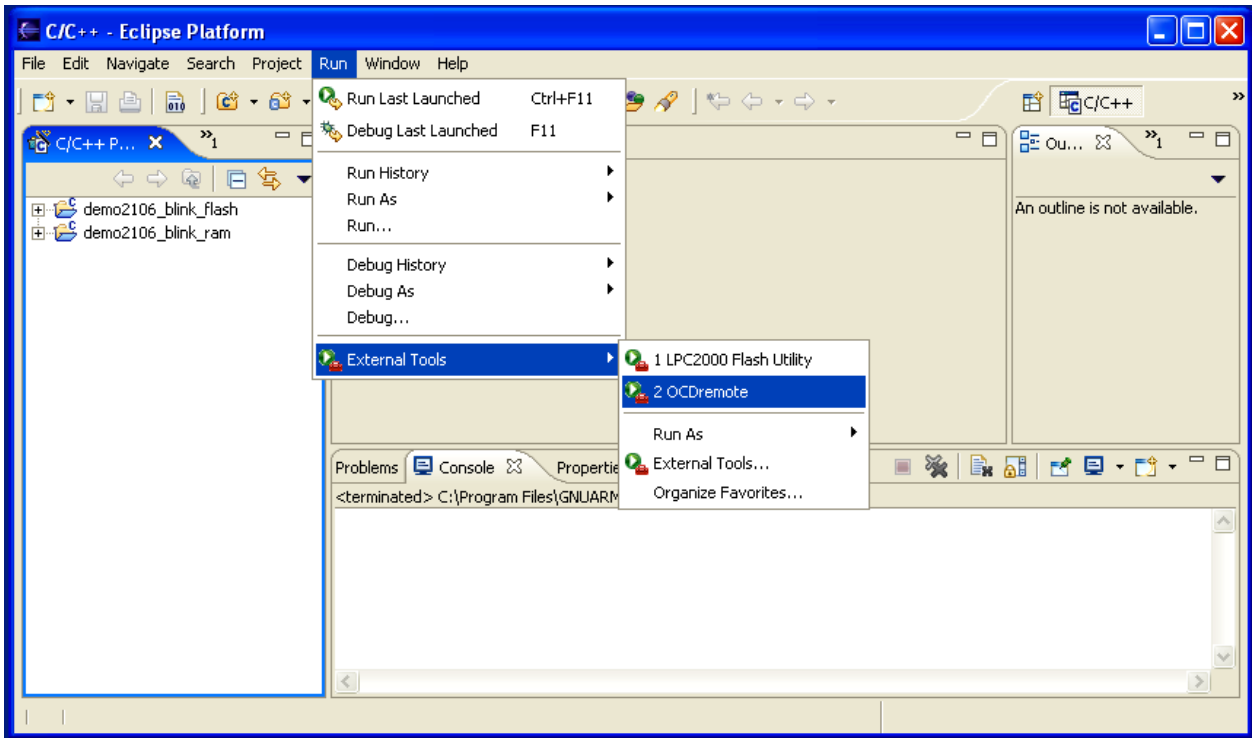
Click on “Run – External Tools – Organize Favorites”



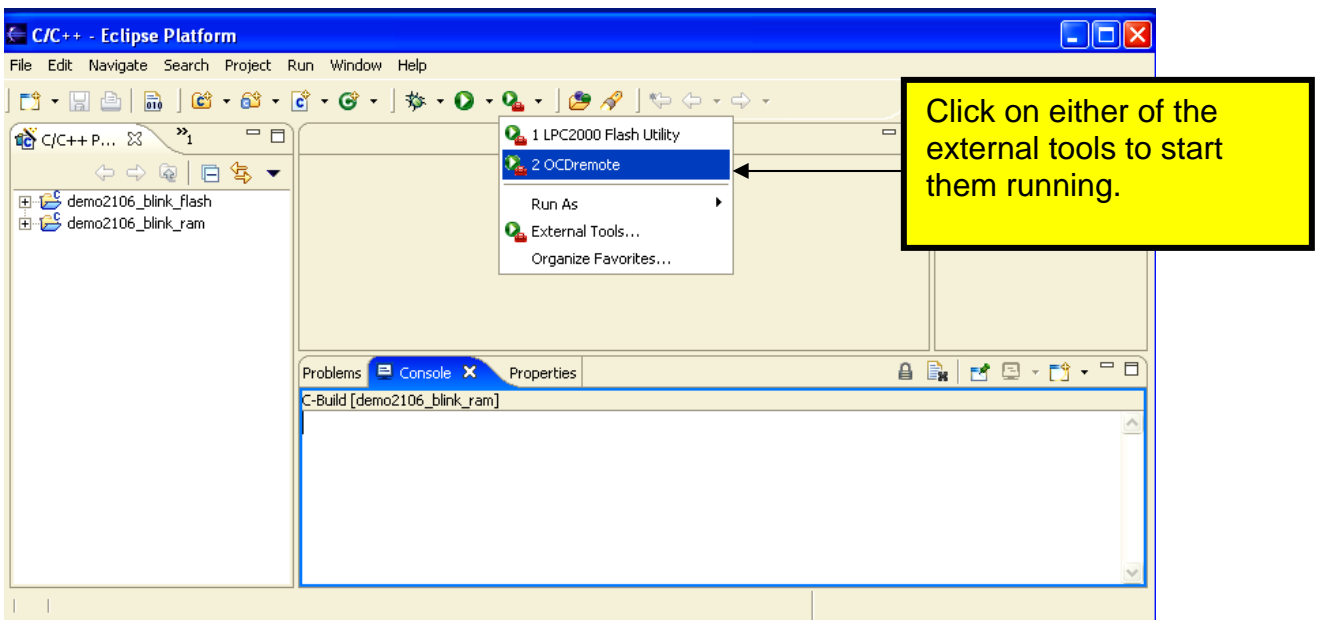
Now click on “Add...” in the Organize External Tools ... window and follow that by checking “OCDremote” in the Add External Tools Configurations: window. Click on “OK” to add the OCDremote to the list of favorites.



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.



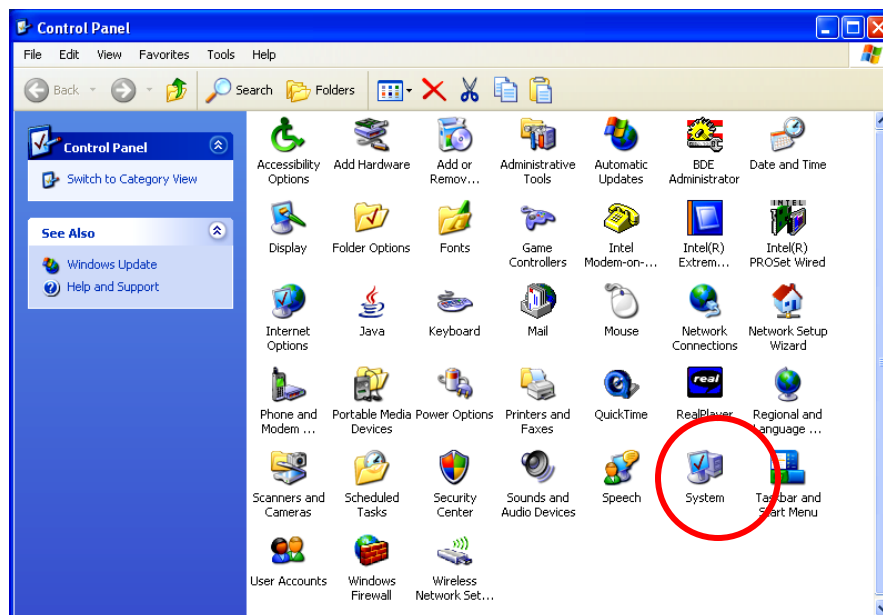
10 Verifying the PATH Settings

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.

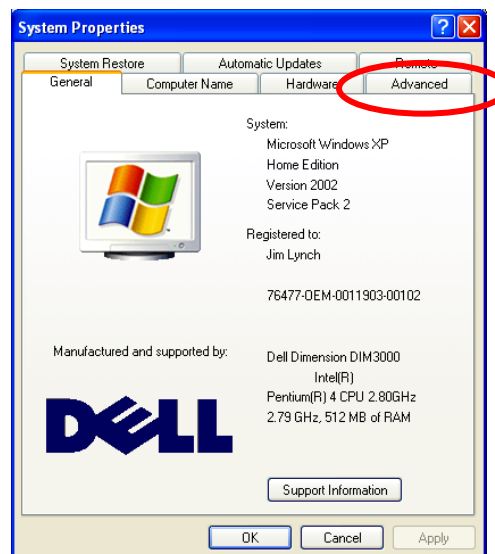
These are the three paths that **must** be present in the Windows environment:

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

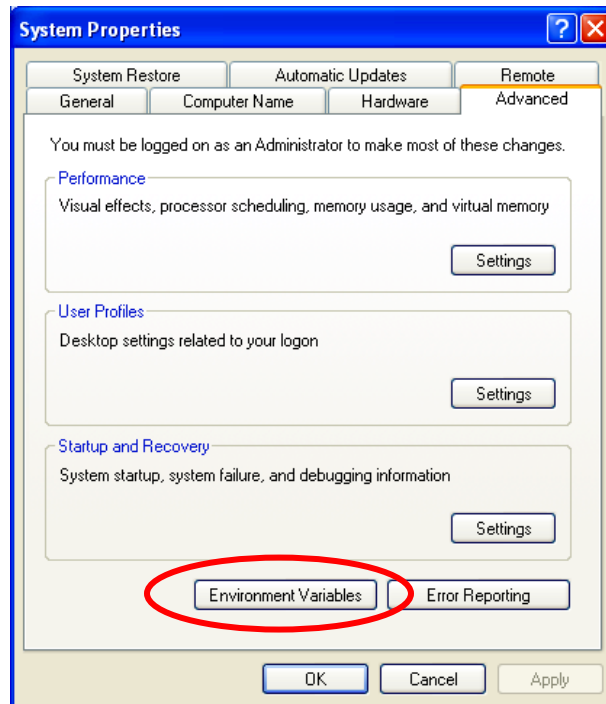
To verify that these paths are present in Windows and to make changes if required, start the Windows Control Panel by clicking “**Start – Control Panel**”.



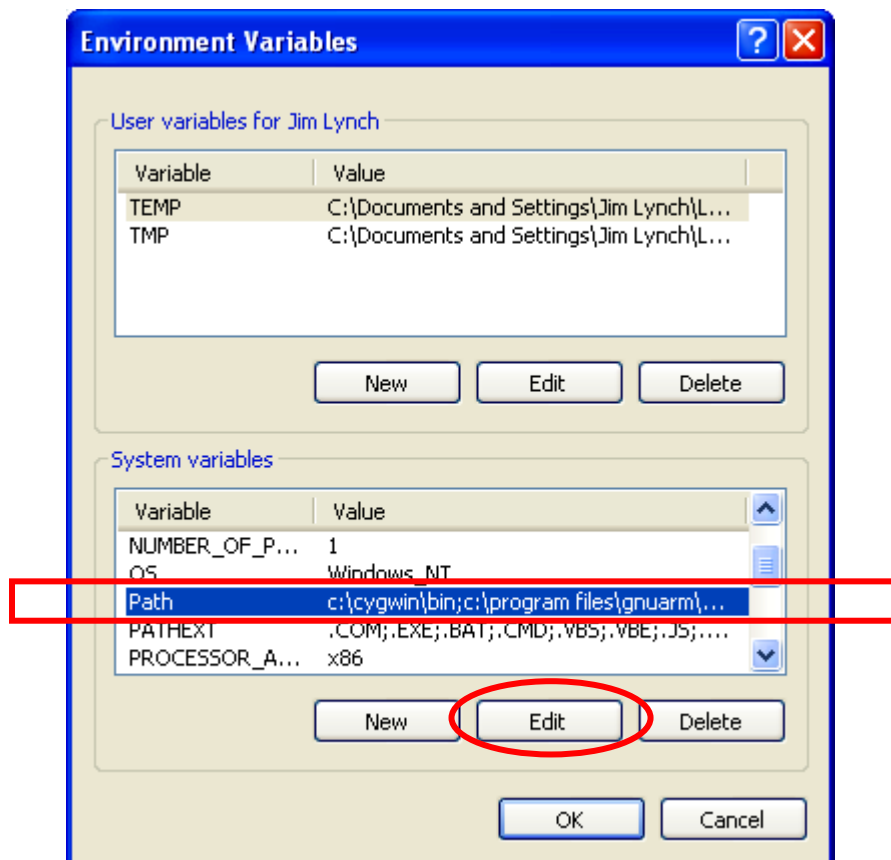
Now click on the “**Advanced**” tab below.



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!

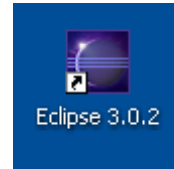
Or as Yoda would say, "***Rewarded soon, your patience will be!***"

11 Creating a Simple Eclipse Project

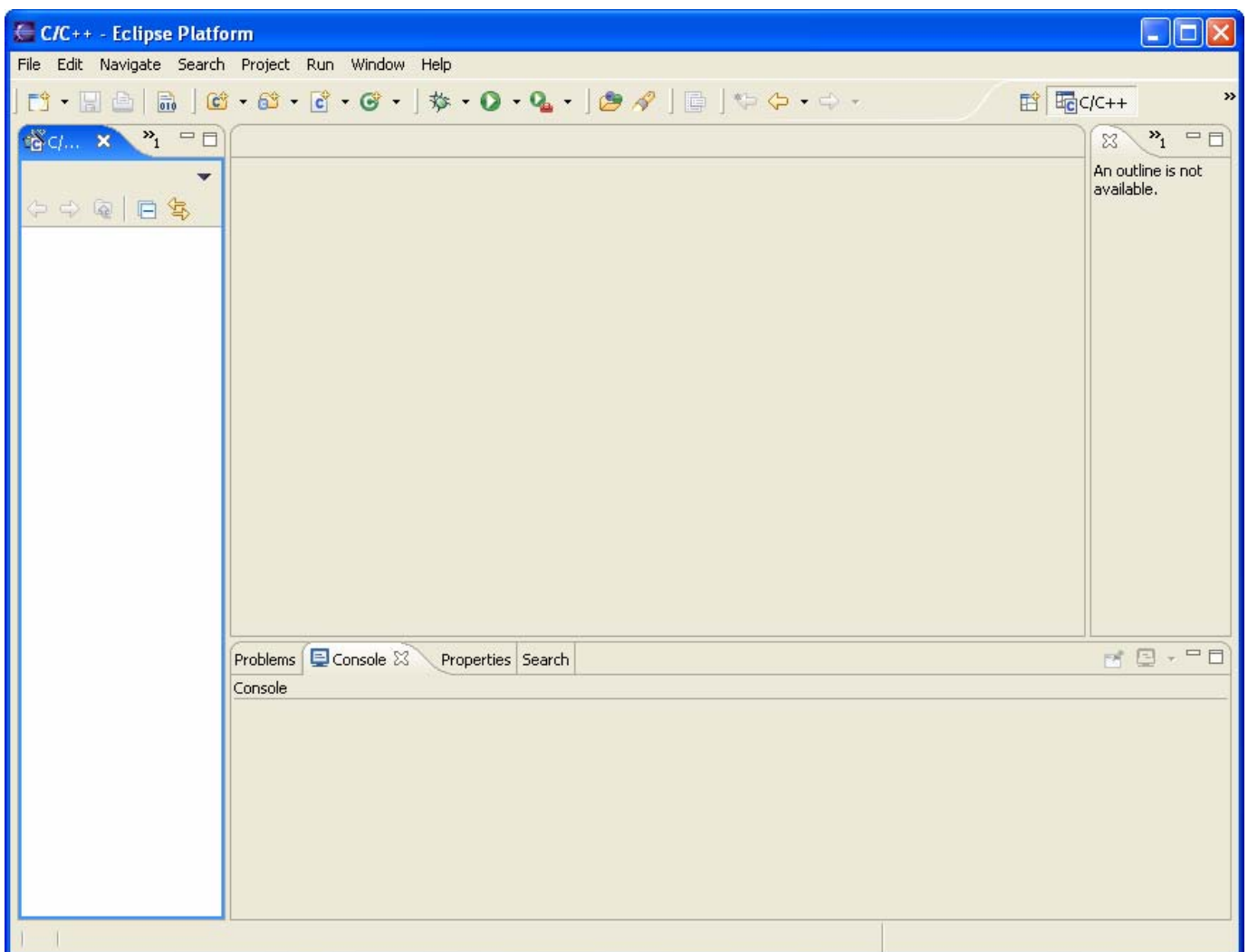
At this point, we have a fully-functioning Eclipse IDE capable of building C/C++ programs for the ARM microprocessor (specifically for the Olimex LPC-P2106 prototype board).

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.

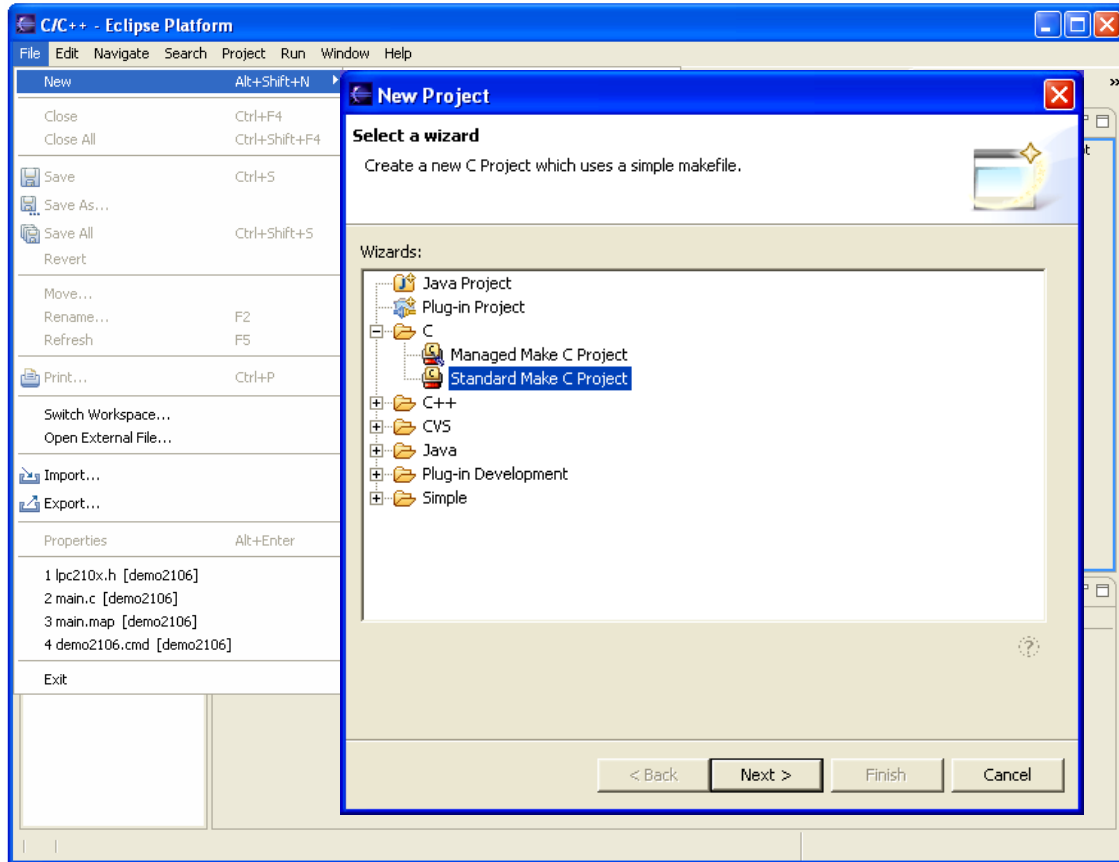
Click on our Eclipse desktop icon to start Eclipse.



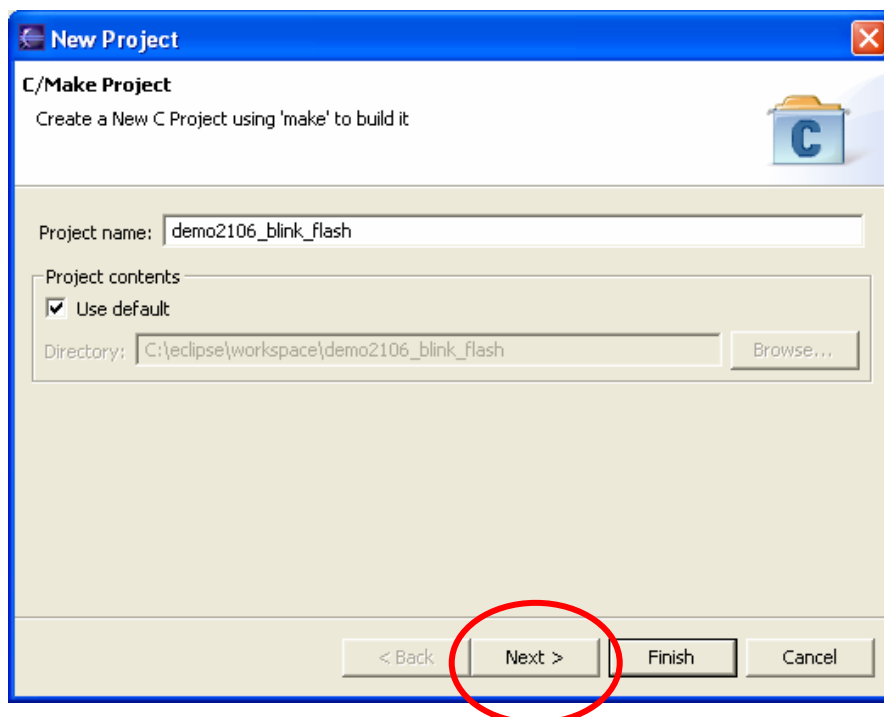
Eclipse should start and present the C/C++ perspective as shown below. If not, select “**Window - Open Perspective – Other - C/C++**” to change to the C++ perspective.



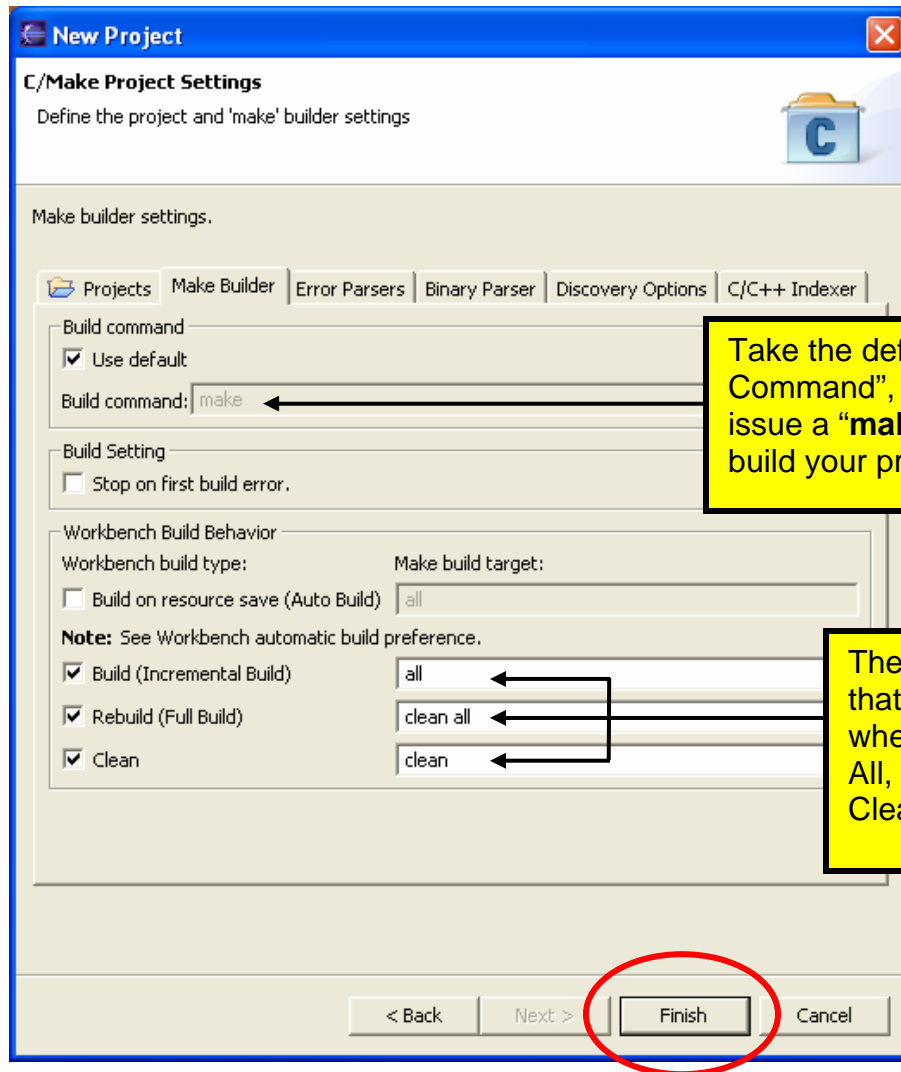
To create a project, select **File – New – New Project - Standard Make C Project** from the File pull-down menu and click **“Next”** to continue.



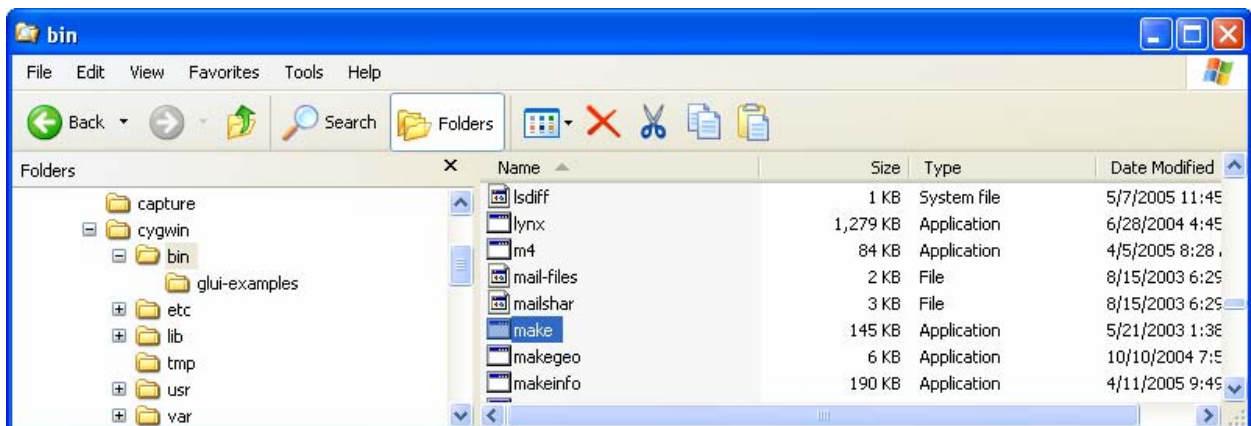
You should see the “New Project” dialog box and enter the project name (**demo2106_blink_flash**) in the box as shown below. Click on **Next** to continue.



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.

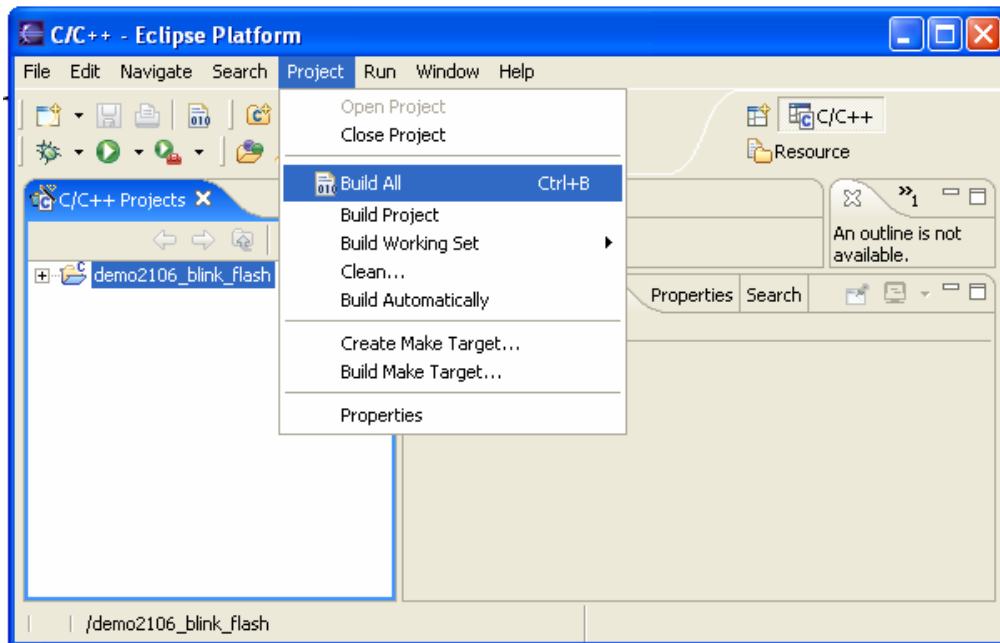


This is a good time to point out the differences between “Build All”, “Build Project” and “Clean.”

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.

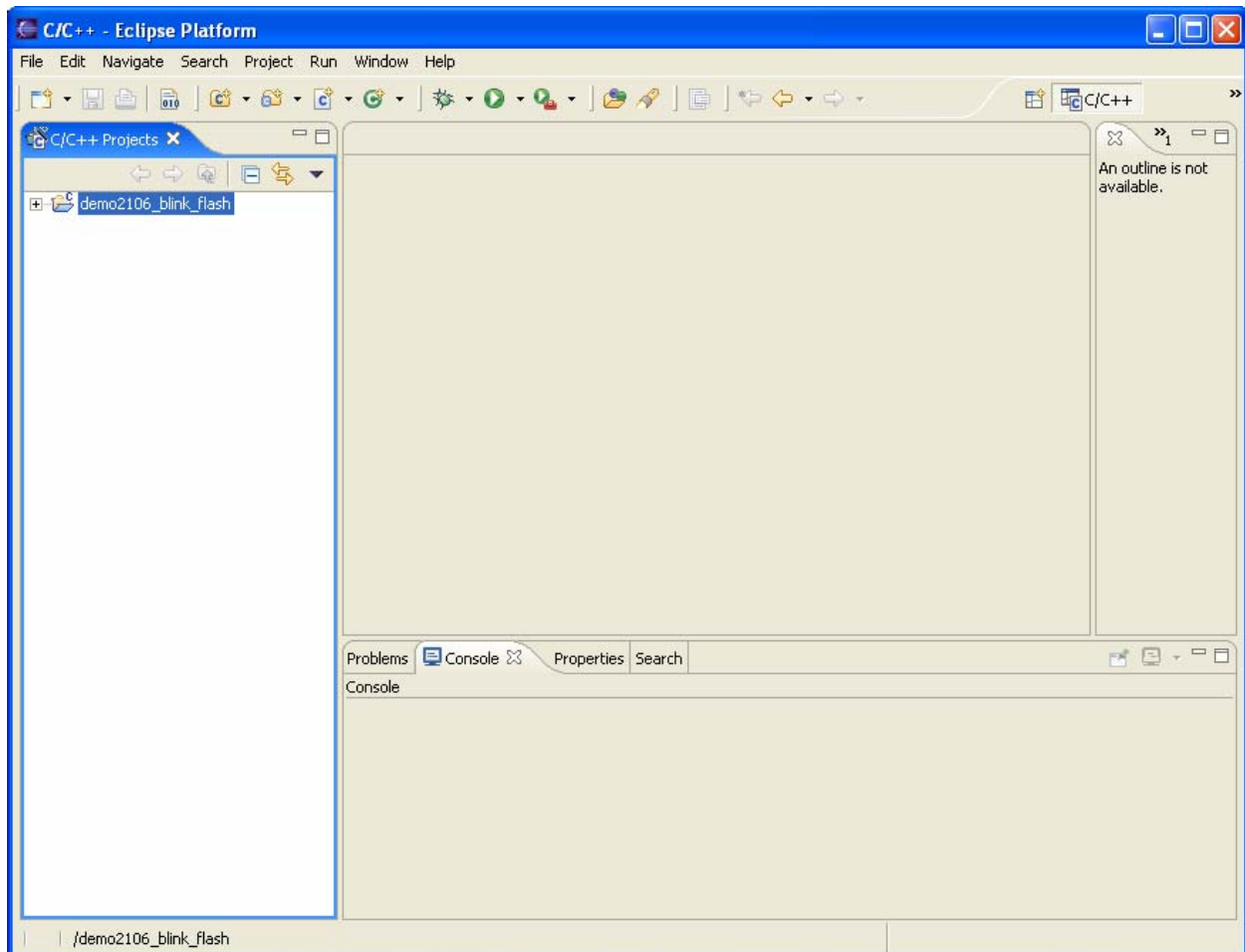


This is no different from opening up a DOS command window and typing the command in directly, such as.

```
> make clean all
```

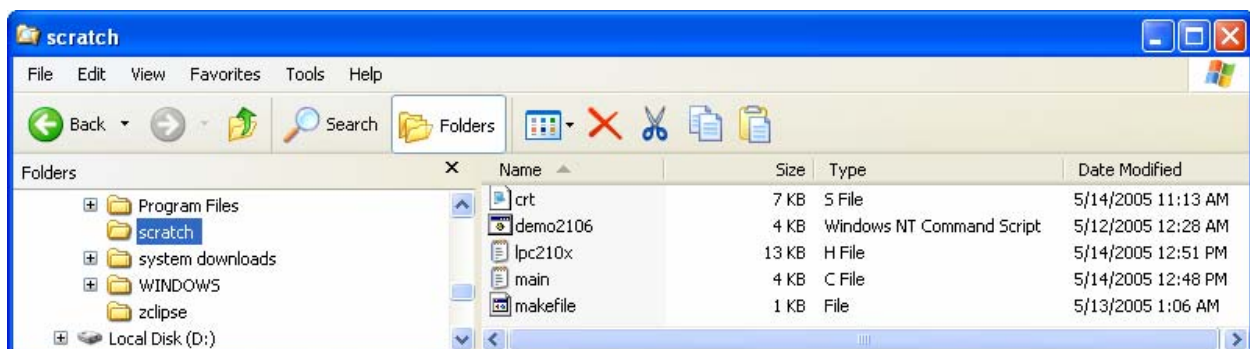
If you click **Finish** on the “New Project” dialog, Eclipse will return to the C/C++ Perspective.

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.

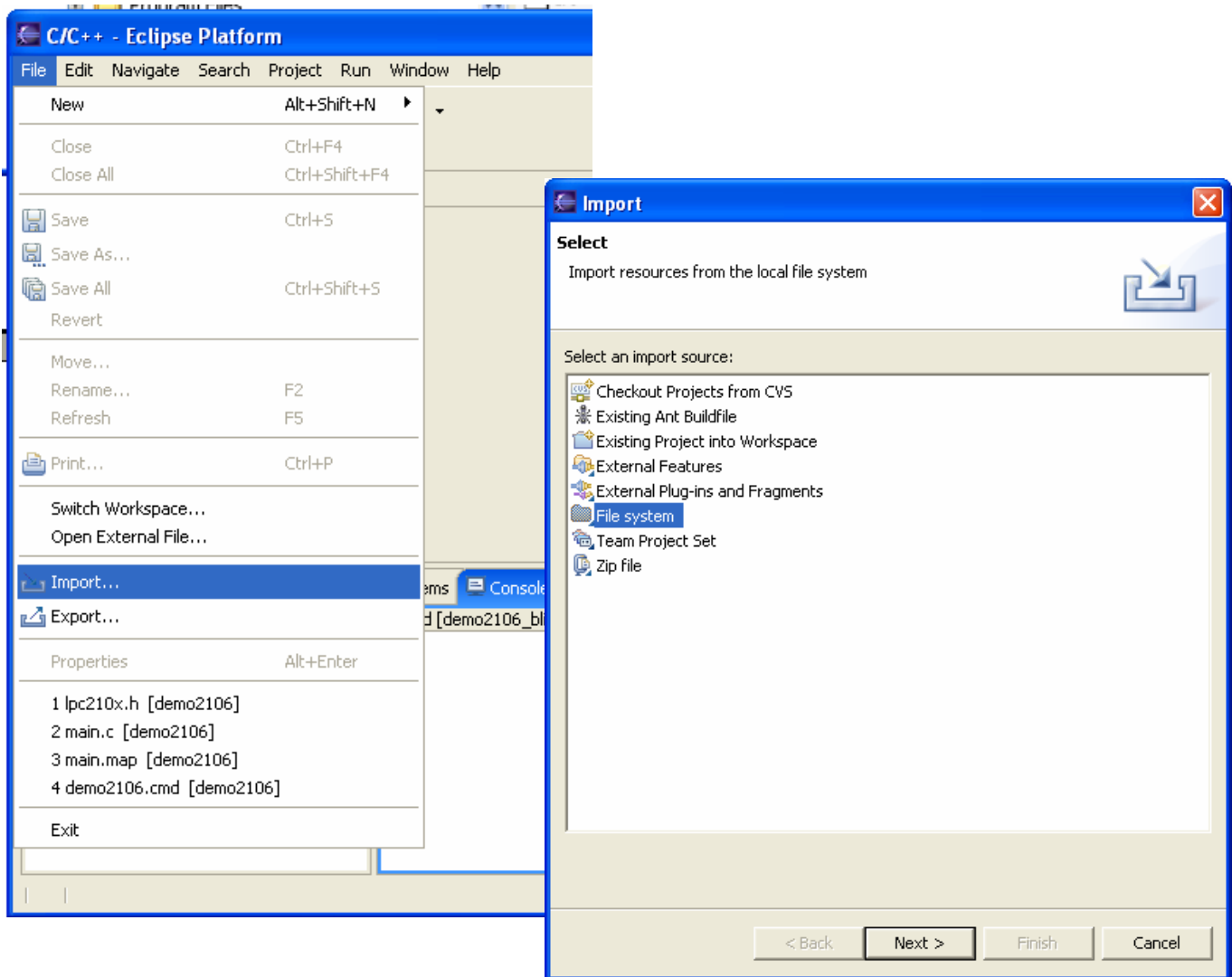


We can now use Eclipse/CDT’s **import** feature to copy the source files into the project.

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.

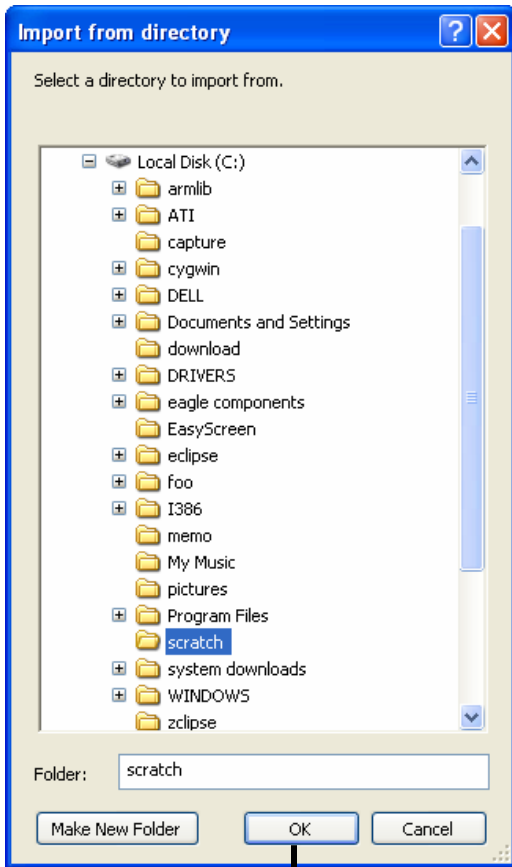


Click on the “**File**” pull-down menu and then click on “**Import.**” Then in the “**Import**” window, click on “**File System.**”

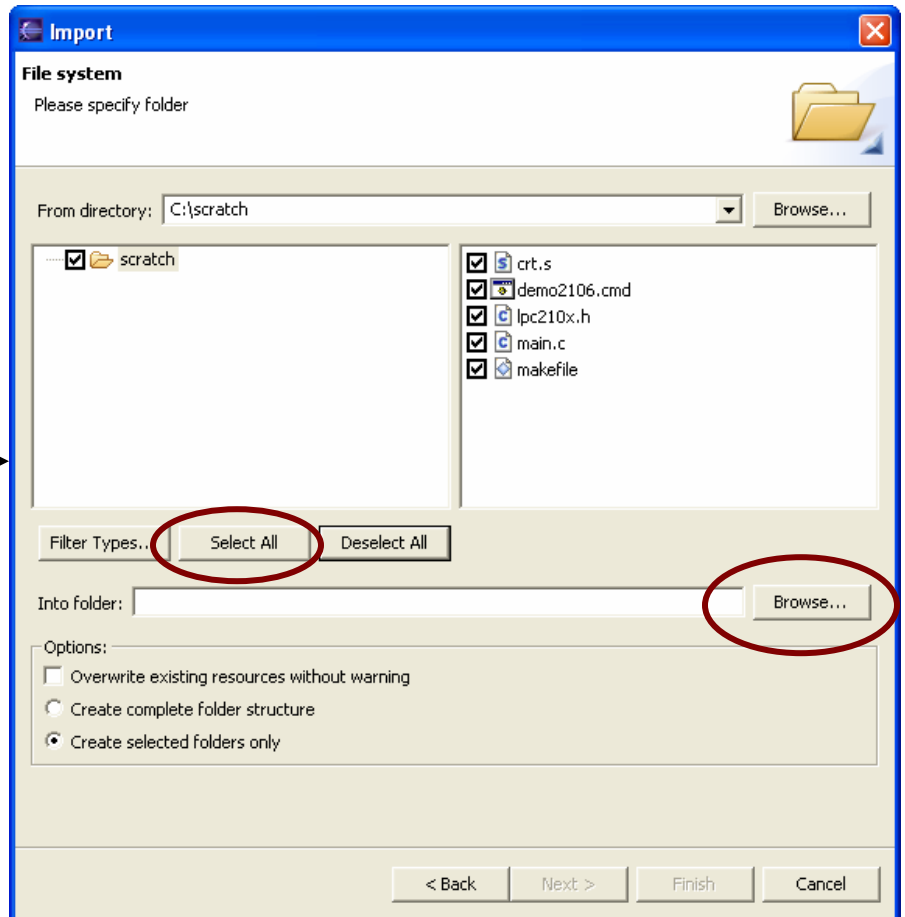


When the “**Import – File System**” window appears, click on the “**Browse**” button. Hunt for the sample project which is stored in the **c:/scratch/** directory.

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



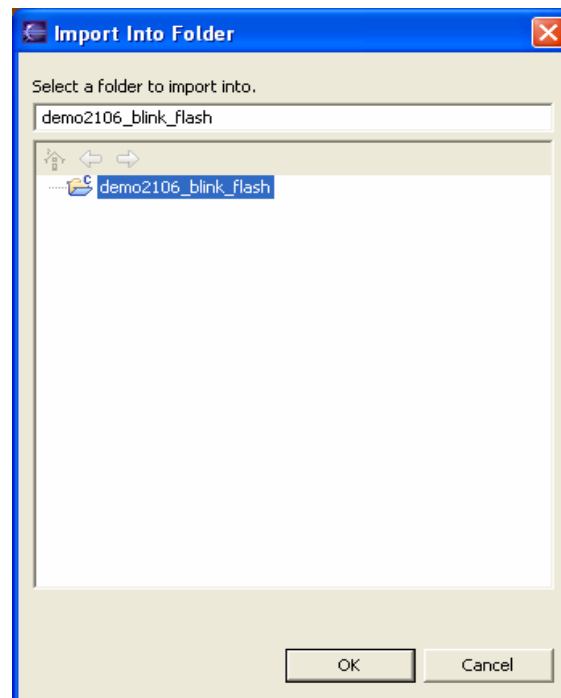
Click on “**Select All**” in the Import window below right to get the source files selected for import into our project.



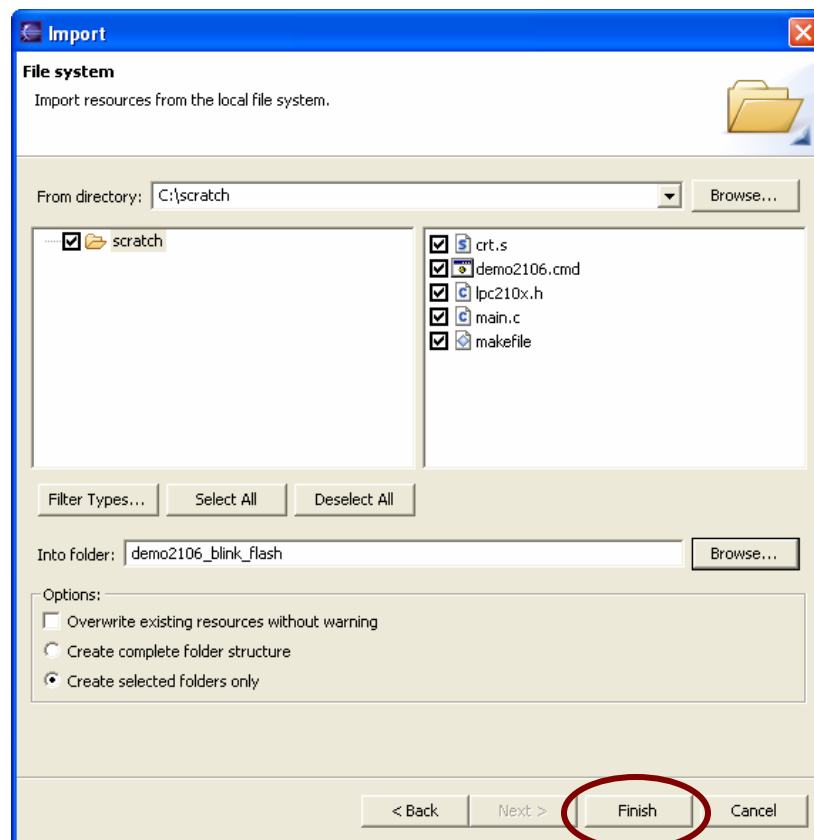
Now we have to indicate the destination for our source files. Click on “**Browse**” on the line to the right that says “**Into Folder.**”

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

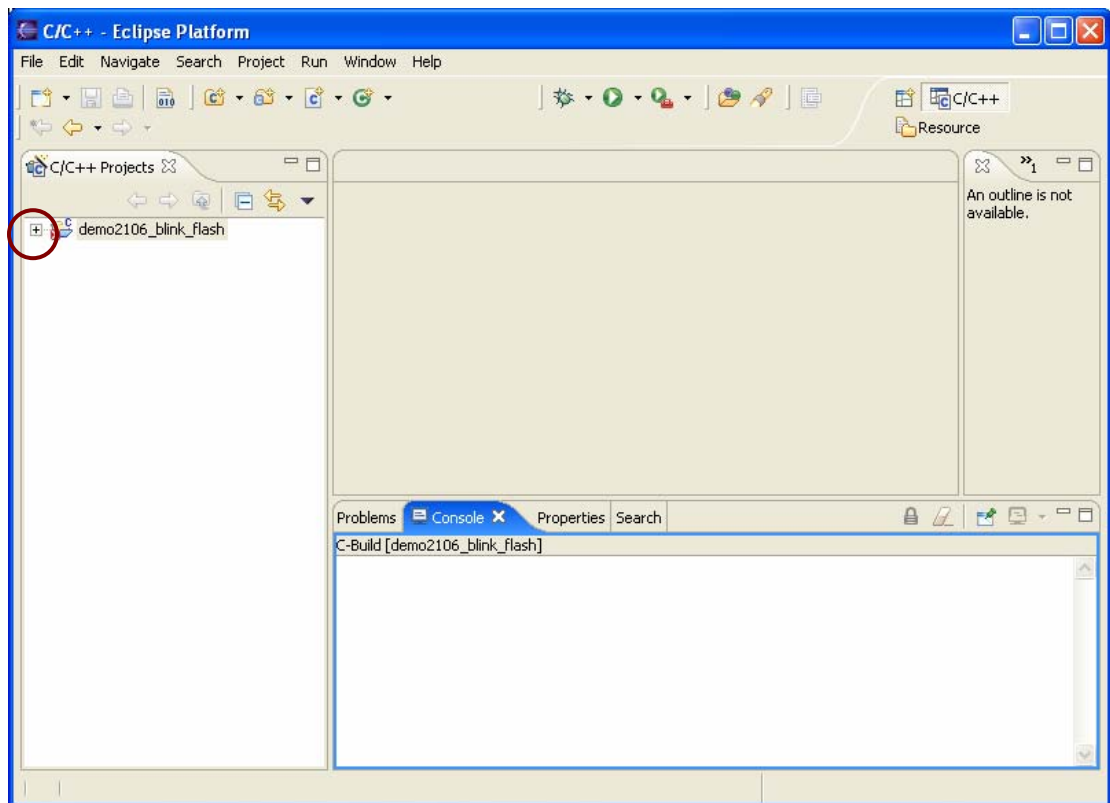
Click on the folder name “**demo2106_blink_flash**” and click “**OK.**” The directory name “demo2106_blink_flash” should appear in the text box.



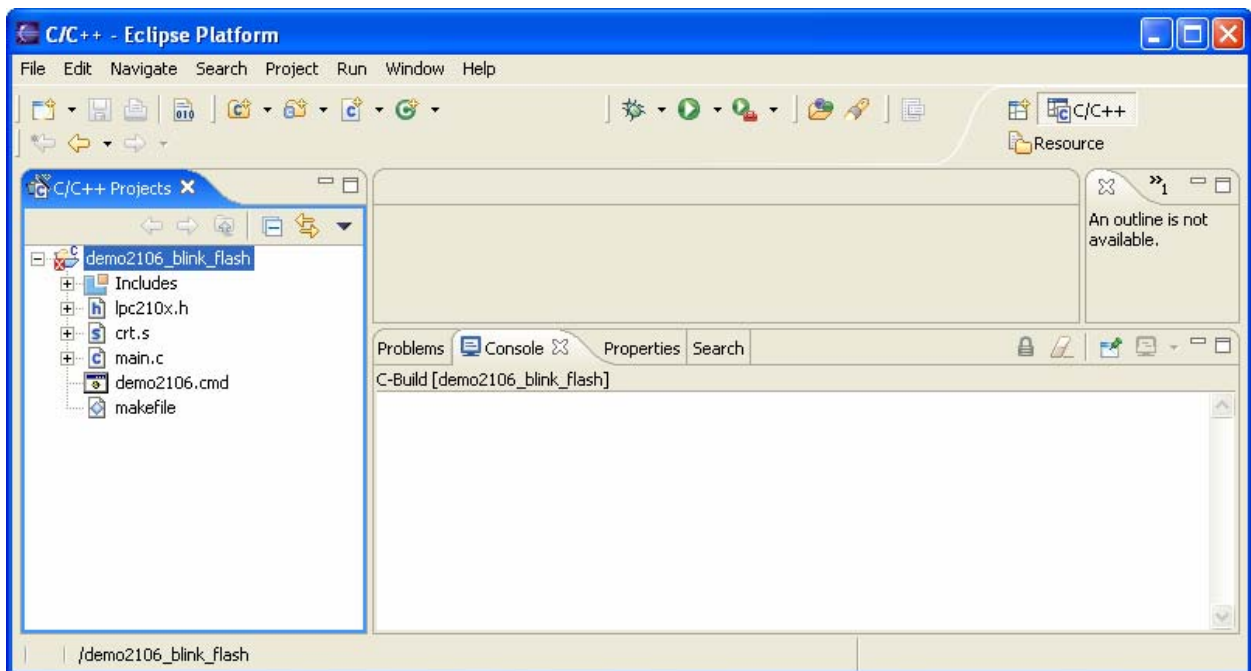
Now the Import dialog is completely filled out; we can click on “finish” to actually import the source files into our project.



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.



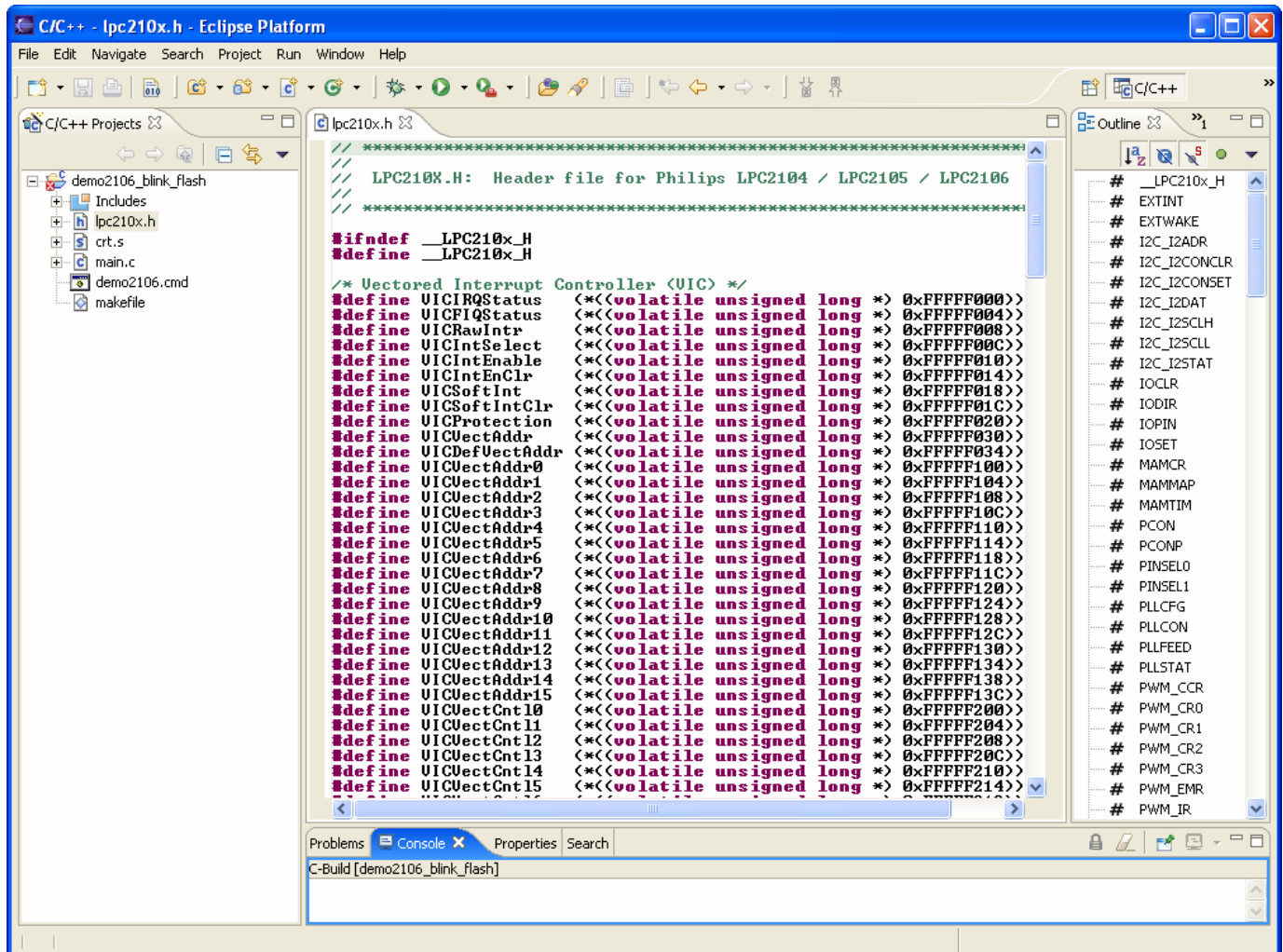
This is a good place to identify the imported source files.

<h2>Description of Project Files</h2>	
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

12 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'

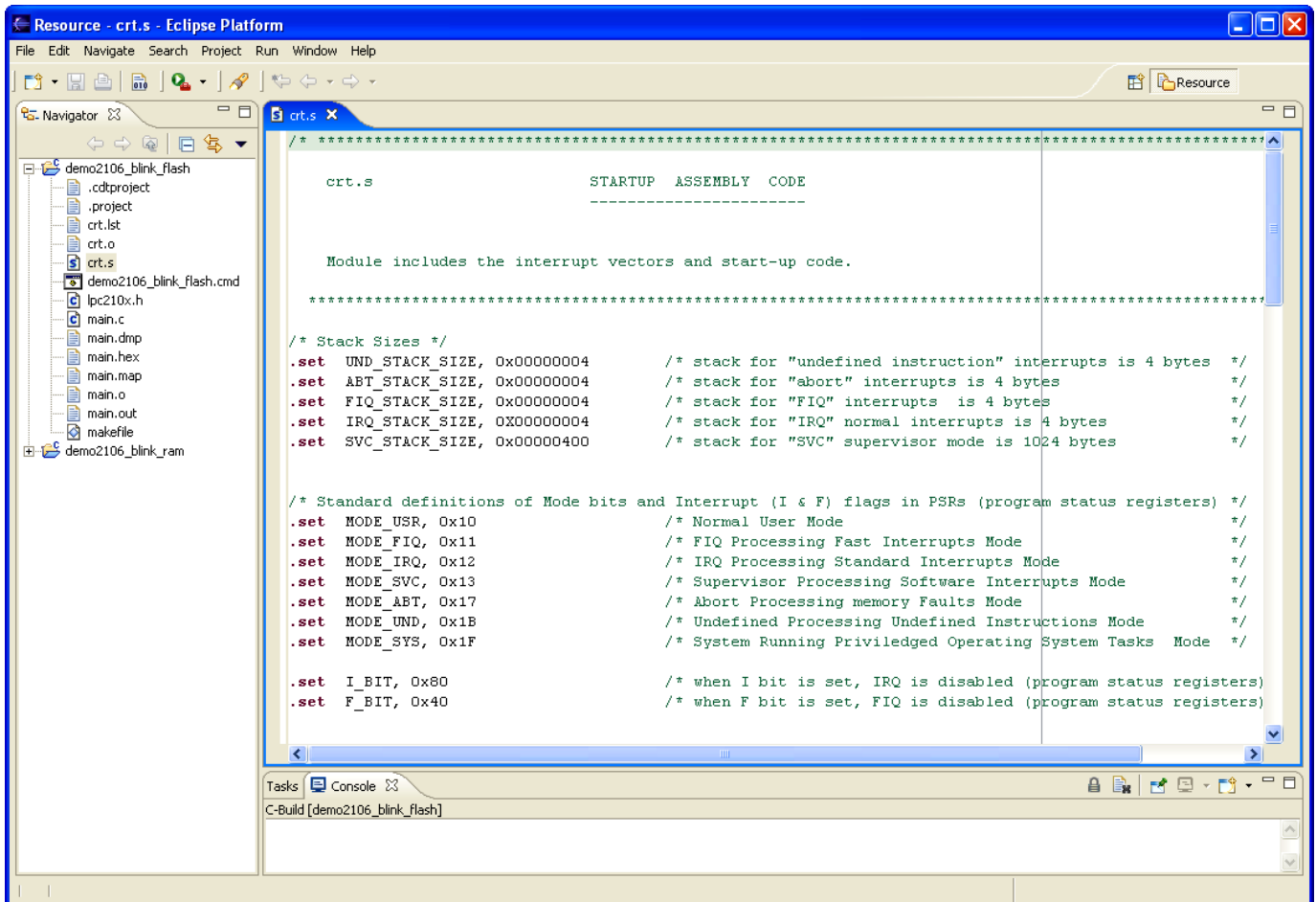
ARM peripherals are memory-mapped, so all I/O registers are defined in this file so you don't have to type in the absolute memory addresses.



13 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.



The screenshot shows the Eclipse IDE interface with the **crt.s** file open. The Navigator on the left shows the project structure for **demo2106_blink_flash**. The main editor displays the following assembly code:

```
Resource - crt.s - 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 STARTUP ASSEMBLY CODE
-----

Module includes the interrupt vectors and start-up code.

*****

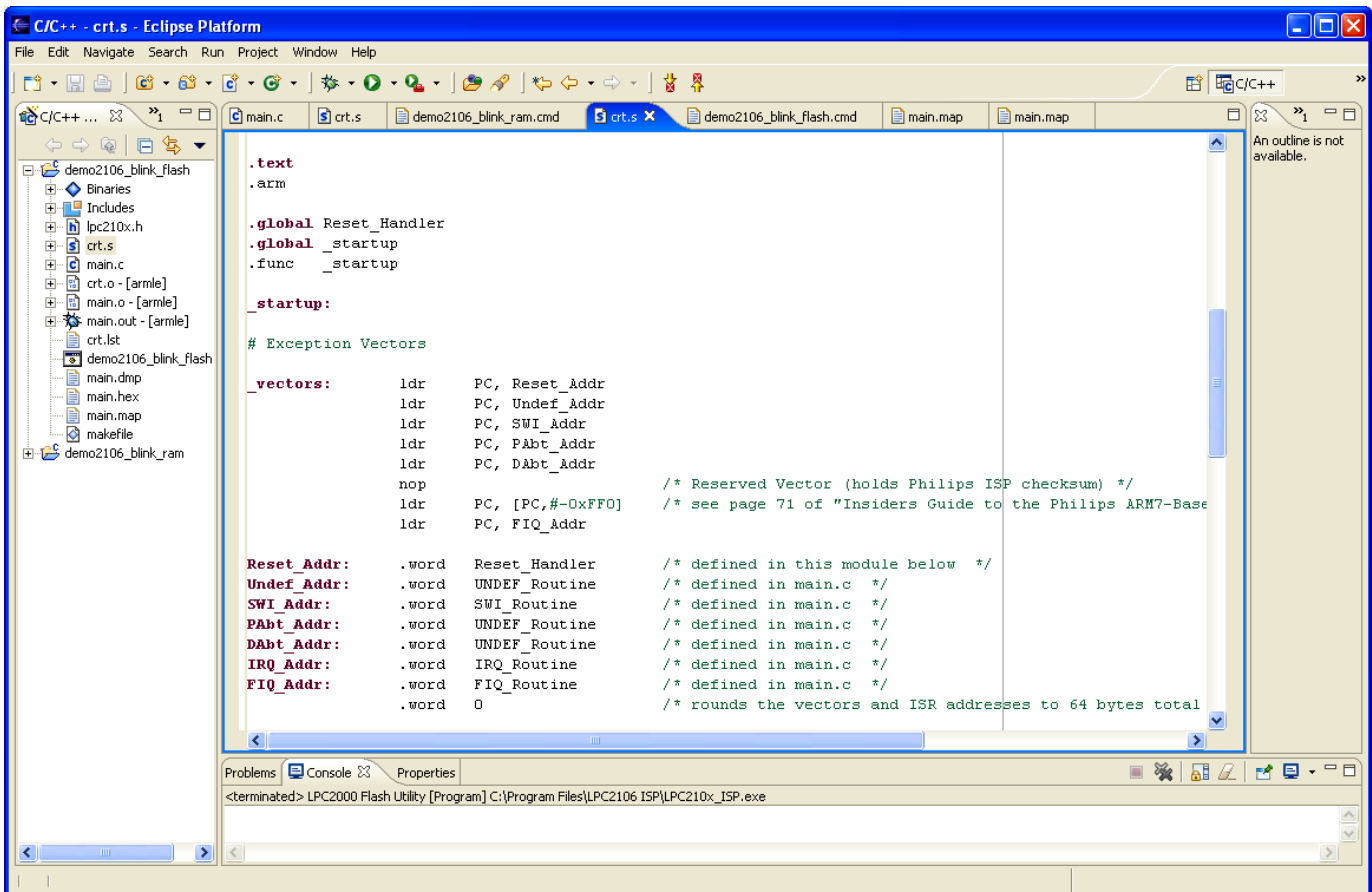
/* Stack Sizes */
.set UND_STACK_SIZE, 0x00000004 /* stack for "undefined instruction" interrupts is 4 bytes */
.set ABT_STACK_SIZE, 0x00000004 /* stack for "abort" interrupts is 4 bytes */
.set FIQ_STACK_SIZE, 0x00000004 /* stack for "FIQ" interrupts is 4 bytes */
.set IRQ_STACK_SIZE, 0x00000004 /* stack for "IRQ" normal interrupts is 4 bytes */
.set SVC_STACK_SIZE, 0x00000400 /* stack for "SVC" supervisor mode is 1024 bytes */

/* Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs (program status registers) */
.set MODE_USR, 0x10 /* Normal User Mode */
.set MODE_FIQ, 0x11 /* FIQ Processing Fast Interrupts Mode */
.set MODE_IRQ, 0x12 /* IRQ Processing Standard Interrupts Mode */
.set MODE_SVC, 0x13 /* Supervisor Processing Software Interrupts Mode */
.set MODE_ABT, 0x17 /* Abort Processing memory Faults Mode */
.set MODE_UND, 0x1B /* Undefined Processing Undefined Instructions Mode */
.set MODE_SYS, 0x1F /* System Running Privileged Operating System Tasks Mode */

.set I_BIT, 0x80 /* when I bit is set, IRQ is disabled (program status registers) */
.set F_BIT, 0x40 /* when F bit is set, FIQ is disabled (program status registers) */

Tasks Console
C-Build [demo2106_blink_flash]
```


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



```
.text
.arm

.global Reset_Handler
.global _startup
.func _startup

_startup:

# Exception Vectors

_vectors:
    ldr    PC, Reset_Addr
    ldr    PC, Undef_Addr
    ldr    PC, SWI_Addr
    ldr    PC, PAbt_Addr
    ldr    PC, DAbt_Addr
    nop
    ldr    PC, [PC, #-0xFF0] /* Reserved Vector (holds Philips ISP checksum) */
    ldr    PC, FIQ_Addr /* see page 71 of "Insiders Guide to the Philips ARM7-Base

Reset_Addr:    .word    Reset_Handler    /* defined in this module below */
Undef_Addr:    .word    UNDEF_Routine /* defined in main.c */
SWI_Addr:      .word    SWI_Routine    /* defined in main.c */
PAbt_Addr:     .word    UNDEF_Routine /* defined in main.c */
DAbt_Addr:     .word    UNDEF_Routine /* defined in main.c */
IRQ_Addr:      .word    IRQ_Routine    /* defined in main.c */
FIQ_Addr:      .word    FIQ_Routine   /* defined in main.c */
               .word    0            /* rounds the vectors and ISR addresses to 64 bytes total
```

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`.

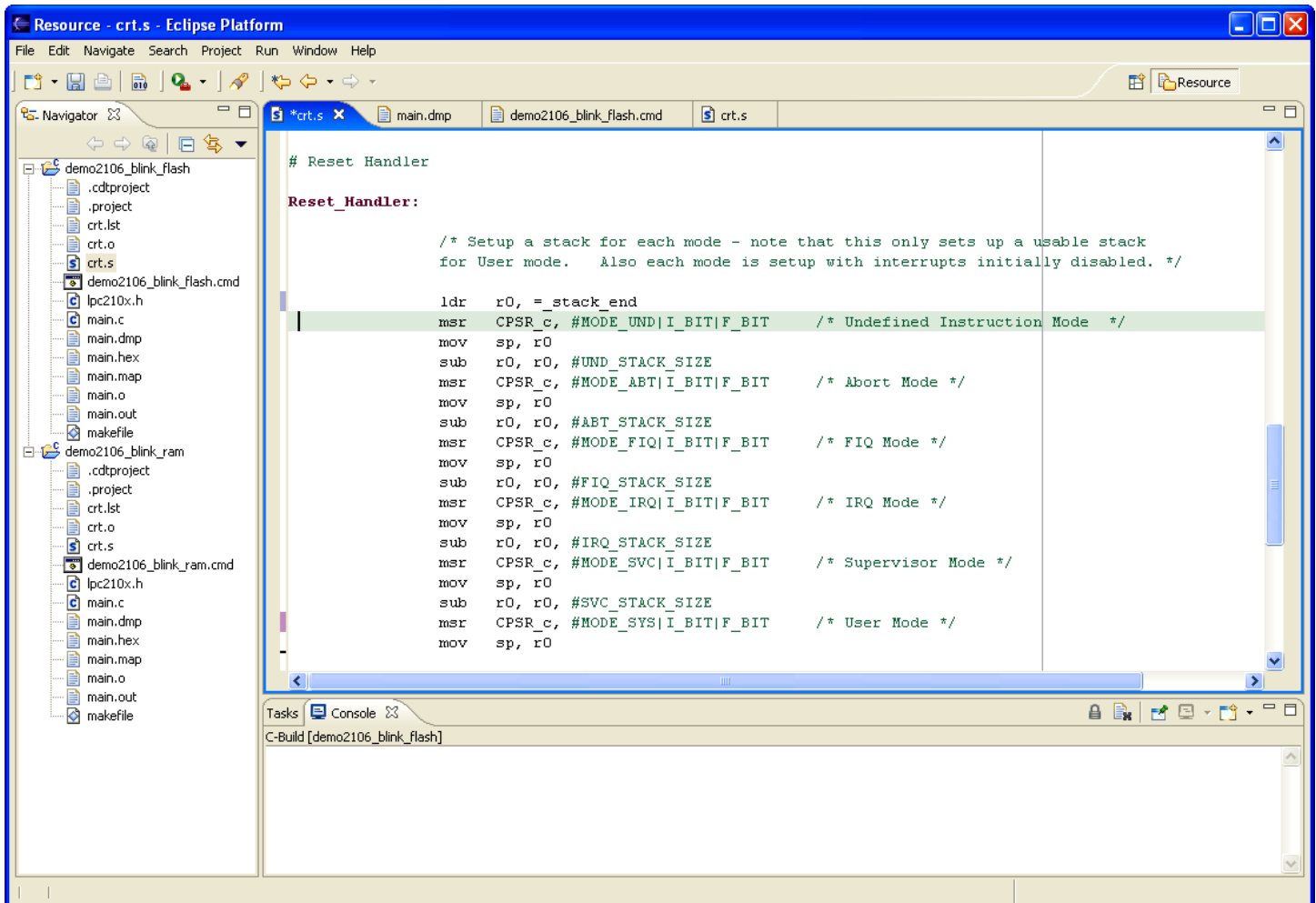
You will see later in this tutorial that the interrupt service routines referenced in the Vector Table are just endless-loop stubs in the `main.c` function and the interrupts are turned off.

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 `cr.t.s` file sets up the various interrupt modes and stacks.



```
# Reset Handler
Reset_Handler:

/* Setup a stack for each mode - note that this only sets up a usable stack
for User mode. Also each mode is setup with interrupts initially disabled. */

ldr r0, =_stack_end
msr CPSR_c, #MODE_UND|I_BIT|F_BIT /* Undefined Instruction Mode */
mov sp, r0
sub r0, r0, #UND_STACK_SIZE
msr CPSR_c, #MODE_ABT|I_BIT|F_BIT /* Abort Mode */
mov sp, r0
sub r0, r0, #ABT_STACK_SIZE
msr CPSR_c, #MODE_FIQ|I_BIT|F_BIT /* FIQ Mode */
mov sp, r0
sub r0, r0, #FIQ_STACK_SIZE
msr CPSR_c, #MODE_IRQ|I_BIT|F_BIT /* IRQ Mode */
mov sp, r0
sub r0, r0, #IRQ_STACK_SIZE
msr CPSR_c, #MODE_SVC|I_BIT|F_BIT /* Supervisor Mode */
mov sp, r0
sub r0, r0, #SVC_STACK_SIZE
msr CPSR_c, #MODE_SYS|I_BIT|F_BIT /* User Mode */
mov sp, r0
```

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 **0x400FEE0**. (Note: $0x40010000 - 0x120 = 0x400FEE0$)

```
/* 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 0x4000FEDC */  
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 0x4000FDE0 */  
sub r0, r0, #IRQ_STACK_SIZE /* R0 is now 0x4000FDE0 */
```

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.

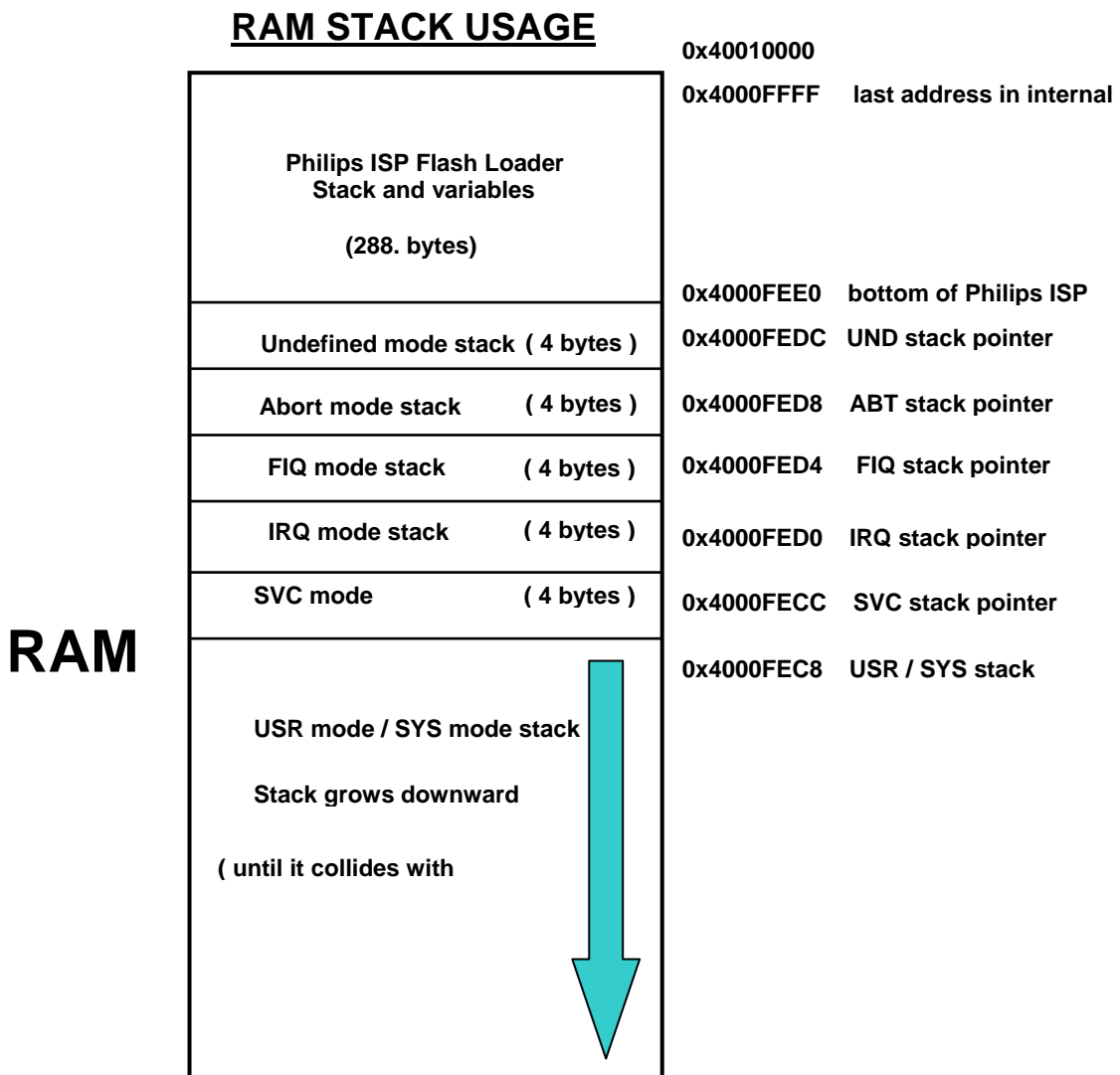
```
msr CPSR_c, #MODE_SVC|I_BIT|F_BIT    /* This puts the CPU in SVC mode */
mov sp, r0                            /* stack pointer for SVC mode is 0x4000FECC */
sub r0, r0, #SVC_STACK_SIZE          /* R0 is now 0x4000FEC8 */
```

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.

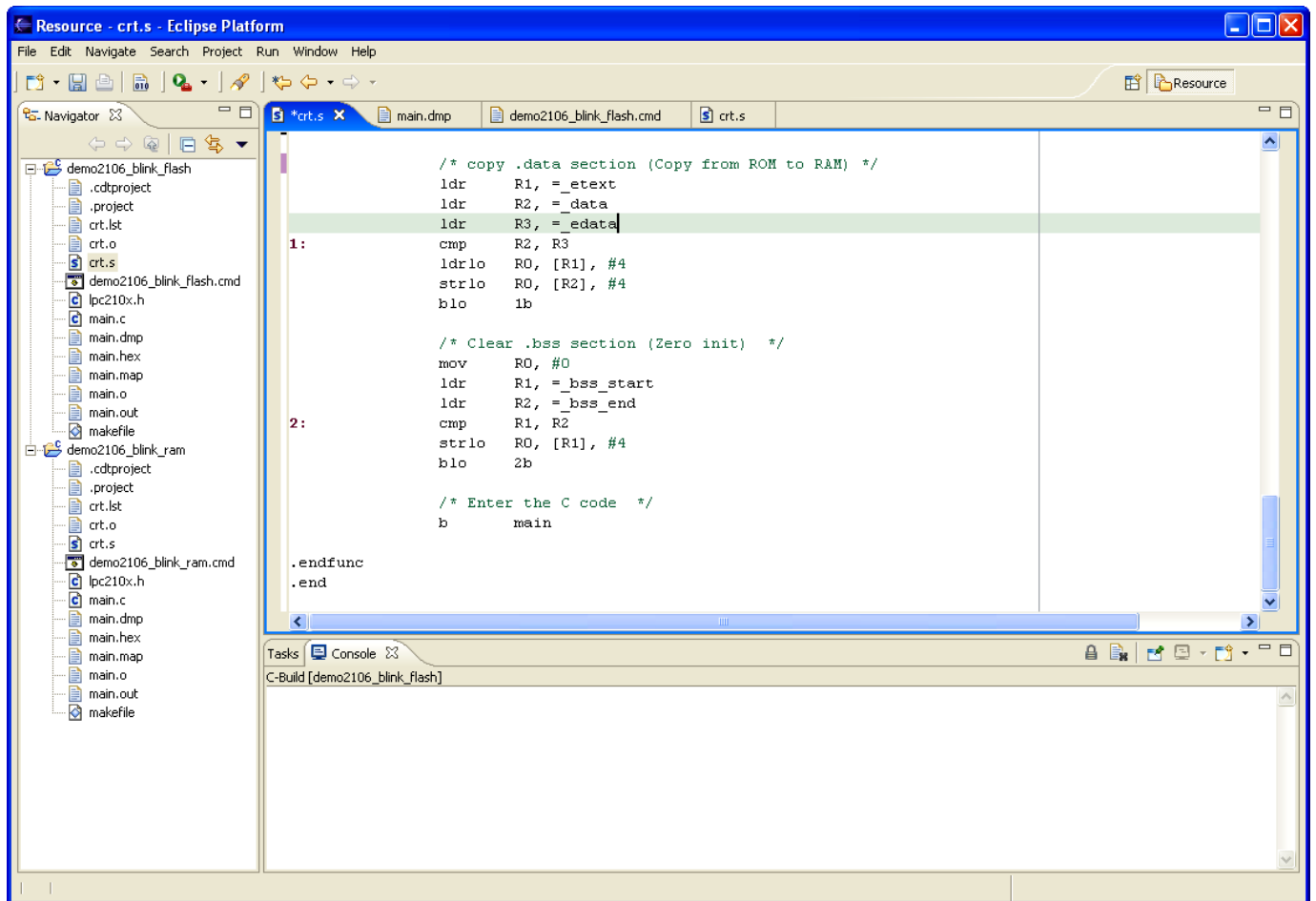
Now we put the ARM into USR (user) mode by setting the MODE_USR bit in the CPSR. We now set the USR mode stack pointer to 0x4000FEC8.

```
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 startup file `cr.t.s` to investigate is the setup of the `.data` and `.bss` sections, as shown below.



```
Resource - cr.t.s - 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
  .cdtproject
  .project
  crt.lst
  crt.o
  crt.s
  demo2106_blink_ram.cmd
  lpc210x.h
  main.c
  main.dmp
  main.hex
  main.map
  main.o
  main.out
  makefile
*cr.t.s x main.dmp demo2106_blink_flash.cmd crt.s
Resource
1:
/* copy .data section (Copy from ROM to RAM) */
ldr R1, =_etext
ldr R2, =_data
ldr R3, =_edata
cmp R2, R3
ldrlo RO, [R1], #4
strlo RO, [R2], #4
blo 1b

/* Clear .bss section (Zero init) */
mov RO, #0
ldr R1, =_bss_start
ldr R2, =_bss_end
cmp R1, R2
strlo RO, [R1], #4
blo 2b

/* Enter the C code */
b main

.endfunc
.end
Tasks Console
C-Build [demo2106_blink_flash]
```

The `.data` section contains all the initialized static and global variables. The GNU linker will create an 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.

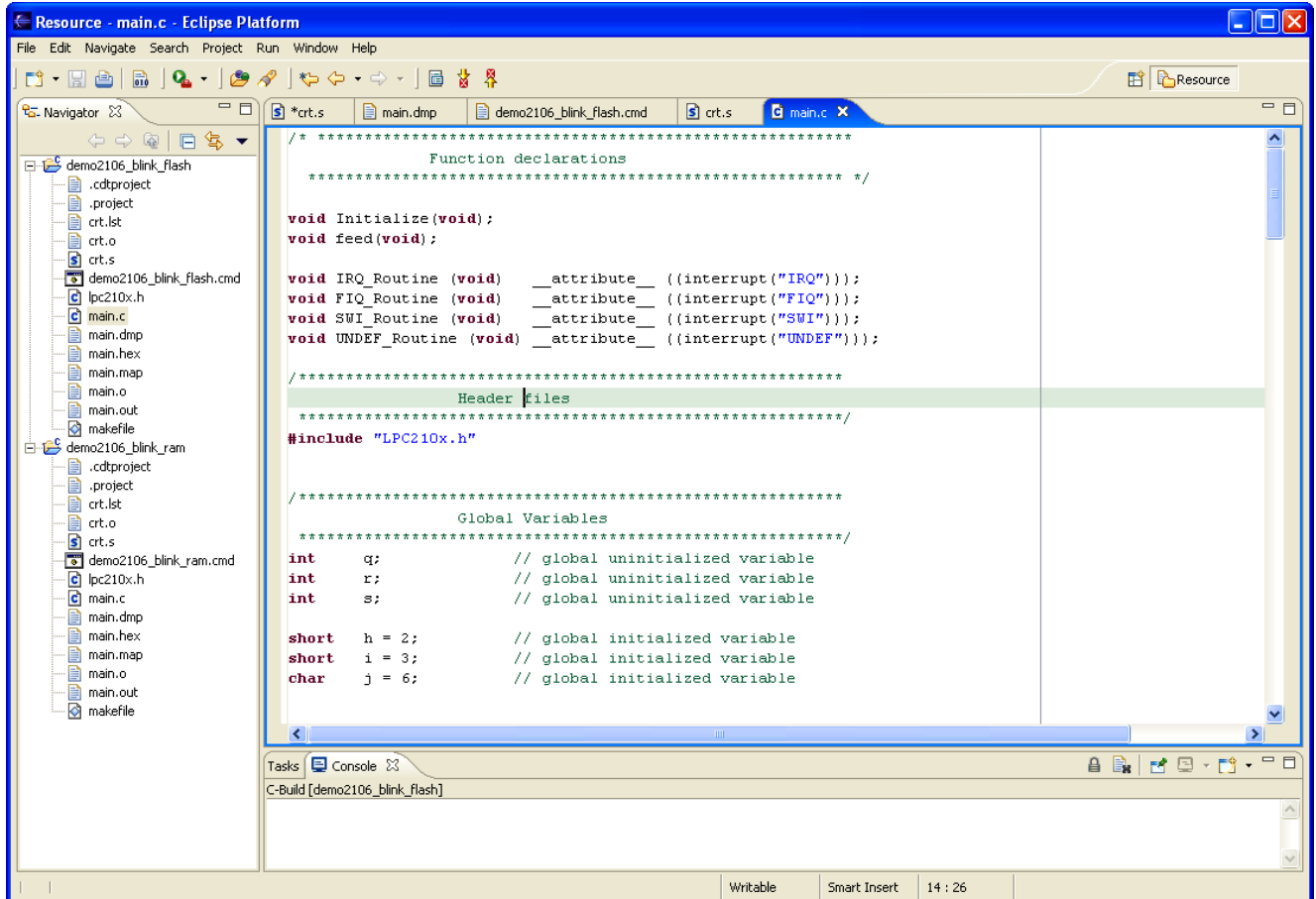
Two simple assembly language loops load the `.data` section in RAM with the initializers in flash and clear out the `.bss` section in RAM.

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.

14 Description of the Main Program main.c

Now let's look at the main program.



```

/* *****
Function declarations
***** */

void Initialize(void);
void feed(void);

void IRQ_Routine (void)  __attribute__ ((interrupt("IRQ")));
void FIQ_Routine (void)  __attribute__ ((interrupt("FIQ")));
void SWI_Routine (void)  __attribute__ ((interrupt("SWI")));
void UNDEF_Routine (void) __attribute__ ((interrupt("UNDEF")));

/*****
Header files
***** */
#include "LPC210x.h"

/*****
Global Variables
***** */

int  q;          // global uninitialized variable
int  r;          // global uninitialized variable
int  s;          // global uninitialized variable

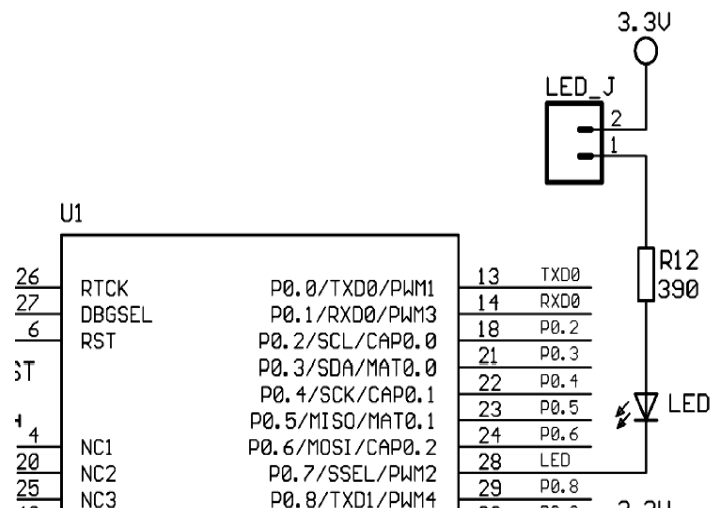
short h = 2;     // global initialized variable
short i = 3;     // global initialized variable
char  j = 6;     // global initialized variable

```

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.

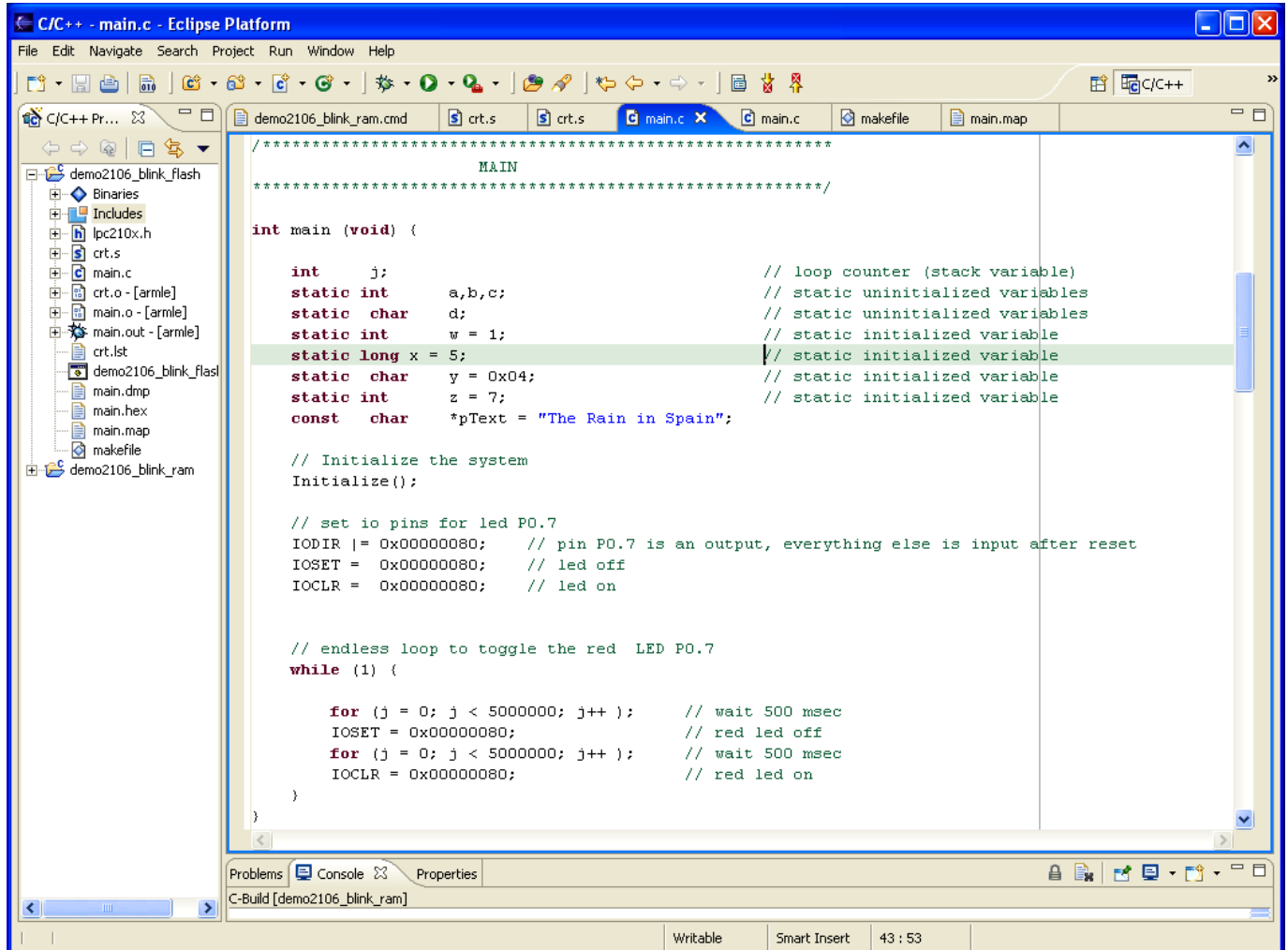
We're going to try to toggle a single I/O bit, specifically P0.7 which is the Olimex red LED.



By the way, with this hardware arrangement:

P0.7 = 1 // turn off LED

P0.7 = 0 // turn on LED



```

/*****
MAIN
*****/

int main (void) {

    int    j;                                // loop counter (stack variable)
    static int    a,b,c;                    // static uninitialized variables
    static char   d;                        // static uninitialized variables
    static int    w = 1;                    // static initialized variable
    static long   x = 5;                    // static initialized variable
    static char   y = 0x04;                 // static initialized variable
    static int    z = 7;                    // static initialized variable
    const char   *pText = "The Rain in Spain";

    // Initialize the system
    Initialize();

    // set io pins for led P0.7
    IODIR |= 0x00000080; // pin P0.7 is an output, everything else is input after reset
    IOSET = 0x00000080; // led off
    IOCLR = 0x00000080; // led on

    // 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
    }
}

```

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
```

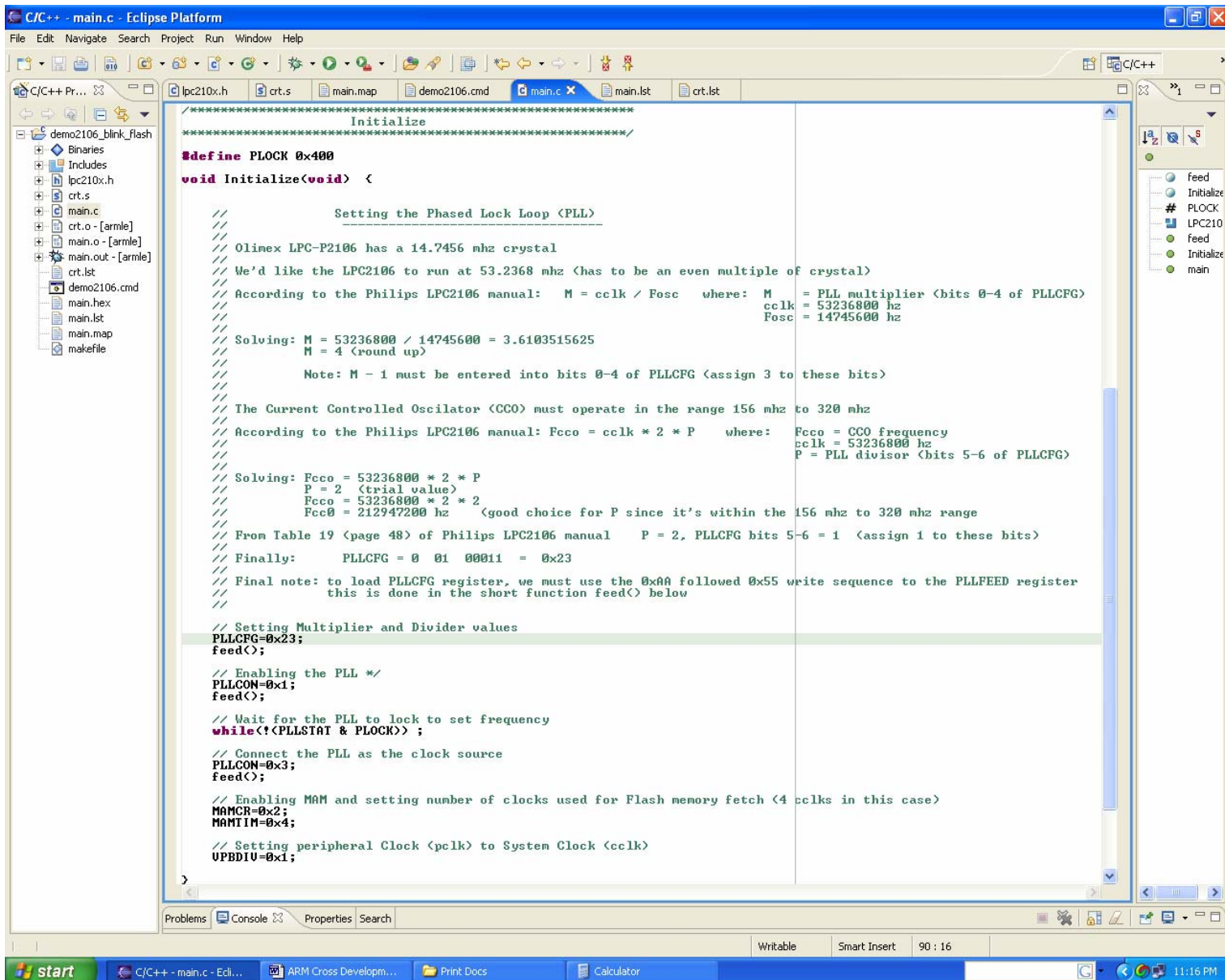
As you can see, it's fairly simple to manipulate I/O bits on the ARM processor.

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.



```
*****
***** Initialize
*****
#define PLOCK 0x400
void Initialize(void) {

    //-----
    // Setting the Phased Lock Loop <PLL>
    //-----
    // Olimex LPC-P2106 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>
    // 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
    //           Fcco = 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
    //-----
    // Setting Multiplier and Divider values
    PLLCFG=0x23;
    feed();

    // Enabling the PLL */
    PLLCON=0x1;
    feed();

    // Wait for the PLL to lock to set frequency
    while(!<PLLSTAT & PLOCK>);

    // Connect the PLL as the clock source
    PLLCON=0x3;
    feed();

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

    // Setting peripheral Clock <pcclk> to System Clock <cclk>
    UPBDIV=0x1;
}
```

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: $M = \text{cclk} / \text{Fosc}$ where: M = PLL multiplier (bits 0-4 of PLLCFG)

$\text{cclk} = 53236800 \text{ hz}$
 $\text{Fosc} = 14745600 \text{ hz}$

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: $F_{cco} = cclk * 2 * P$ where: $F_{cco} = \text{CCO frequency}$

$cclk = 53236800 \text{ hz}$

$P = \text{PLL divisor (bits 5-6 of PLLCFG)}$

Solving: $F_{cco} = 53236800 * 2 * P$
 $P = 2$ (trial value)
 $F_{cco} = 53236800 * 2 * 2$
 $F_{cco} = 212947200 \text{ hz}$ (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.

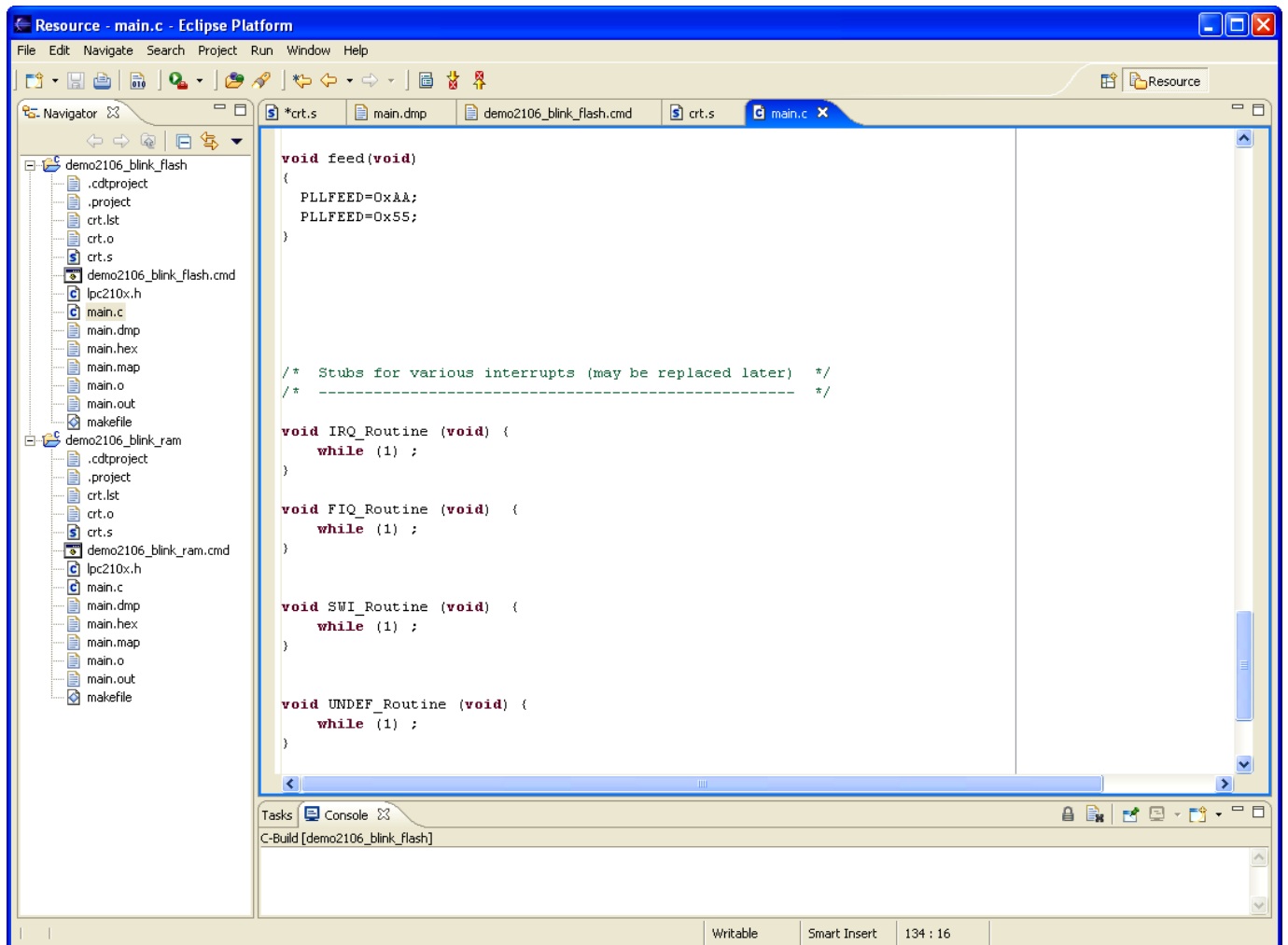
Next we fully enable the Memory Accelerator module and set the Flash memory to run at 1/4 the clock speed. Now you see why some people prefer to execute out of RAM where it's much faster.

```
// 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 intend to allow interrupts in this simple example.



```
Resource - main.c - Eclipse Platform
File Edit Navigate Search Project Run Window Help
* crt.s | main.dmp | demo2106_blink_flash.cmd | crt.s | main.c x
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
    ├── .cdtproject
    ├── .project
    ├── crt.lst
    ├── crt.o
    ├── crt.s
    ├── demo2106_blink_ram.cmd
    ├── lpc210x.h
    ├── main.c
    ├── main.dmp
    ├── main.hex
    ├── main.map
    ├── main.o
    ├── main.out
    └── makefile

void feed(void)
{
    PLLFEED=0xAA;
    PLLFEED=0x55;
}

/* Stubs for various interrupts (may be replaced later) */
/* ----- */

void IRQ_Routine (void) {
    while (1) ;
}

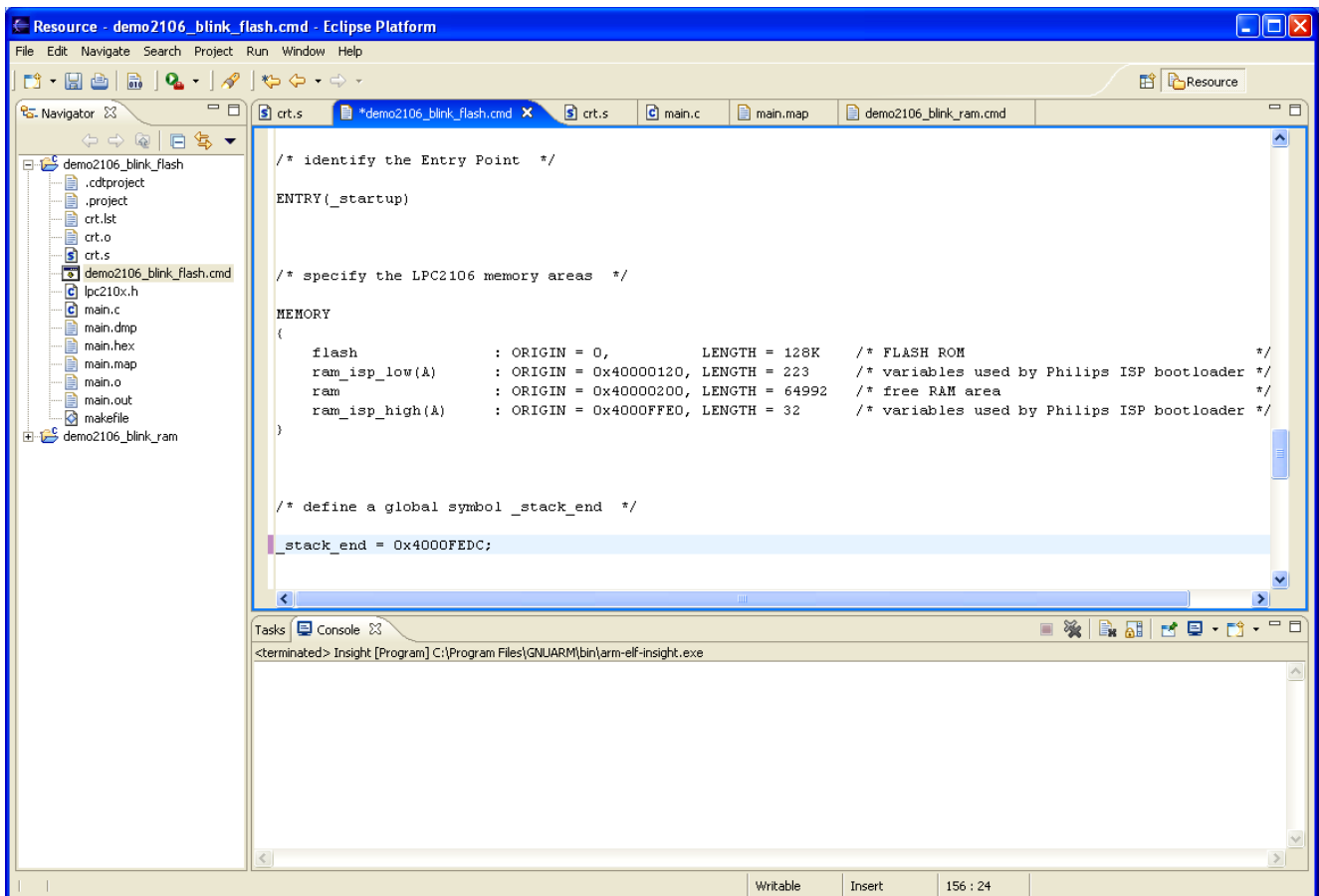
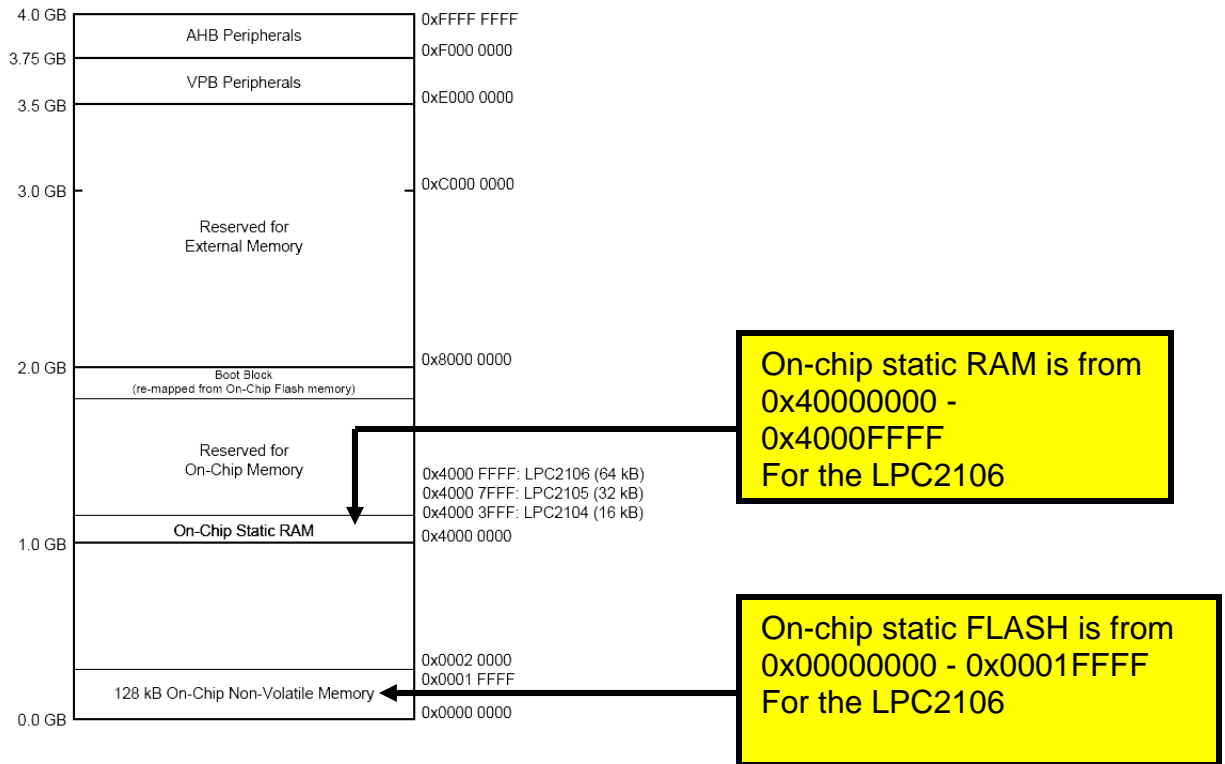
void FIQ_Routine (void) {
    while (1) ;
}

void SWI_Routine (void) {
    while (1) ;
}

void UNDEF_Routine (void) {
    while (1) ;
}

Tasks Console
C-Build [demo2106_blink_flash]
Writable Smart Insert 134 : 16
```


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.

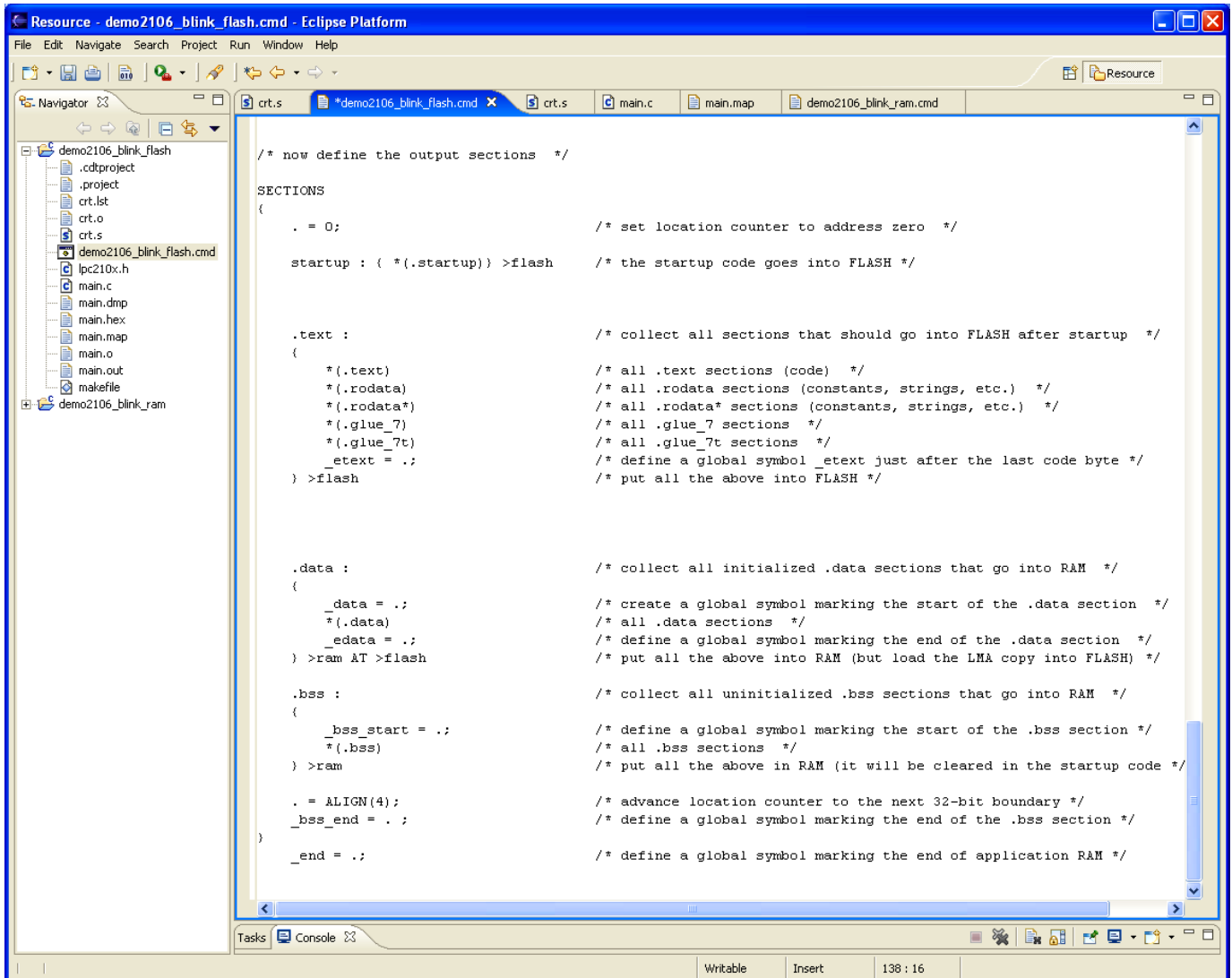
```
_stack_end = 0x4000FEDC;
```

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.



```
Resource - demo2106_blink_flash.cmd - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator crt.s *demo2106_blink_flash.cmd crt.s main.c main.map demo2106_blink_ram.cmd
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

/* now define the output sections */
SECTIONS
{
    . = 0; /* set location counter to address zero */

    startup : { *(.startup) } >flash /* the startup code goes into FLASH */

    .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 just after the last code byte */
    } >flash /* put all the above into FLASH */

    .data : /* collect all initialized .data sections that go into RAM */
    {
        _data = .; /* create a global symbol marking the start of the .data section */
        *(.data) /* all .data sections */
        _edata = .; /* define a global symbol marking the end of the .data section */
    } >ram AT >flash /* put all the above into RAM (but load the LMA copy into FLASH) */

    .bss : /* collect all uninitialized .bss sections that go into RAM */
    {
        _bss_start = .; /* define a global symbol marking the start of the .bss section */
        *(.bss) /* all .bss sections */
    } >ram /* put all the above in RAM (it will be cleared in the startup code) */

    . = ALIGN(4); /* advance location counter to the next 32-bit boundary */
    _bss_end = .; /* define a global symbol marking the end of the .bss section */

    _end = .; /* define a global symbol marking the end of application RAM */
}
```

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.

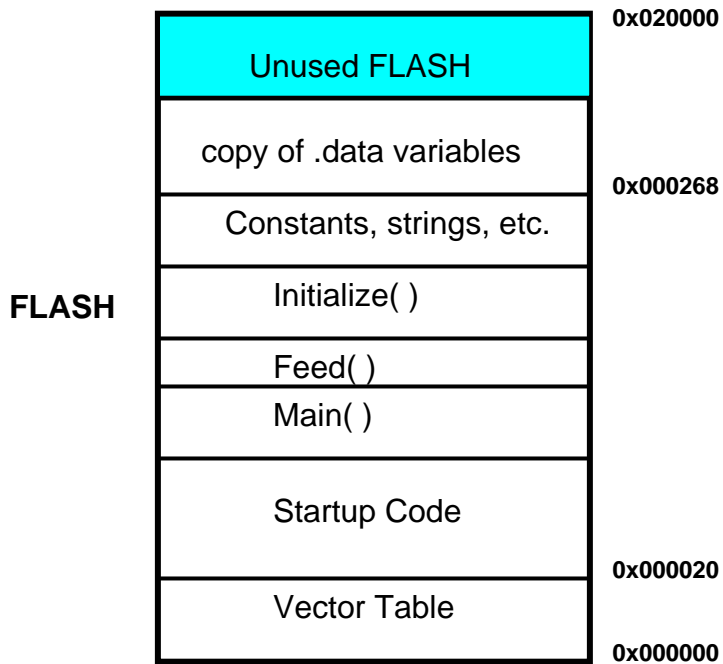
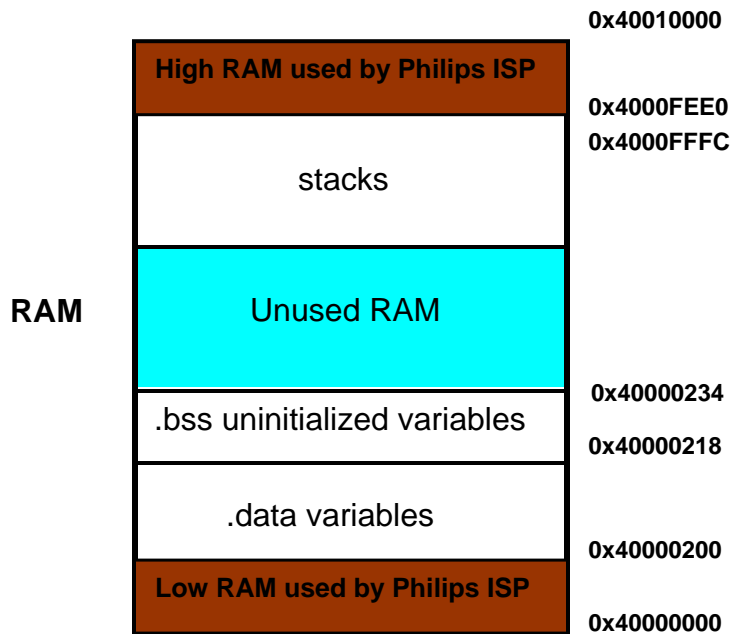
The **.bss** section is all variables that are not initialized. It is loaded into RAM and we create two global symbols **_bss_start** and **_bss_end** to locate the beginning and end for clearing by a loop in the startup code.

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

. = ALIGN(4);
}

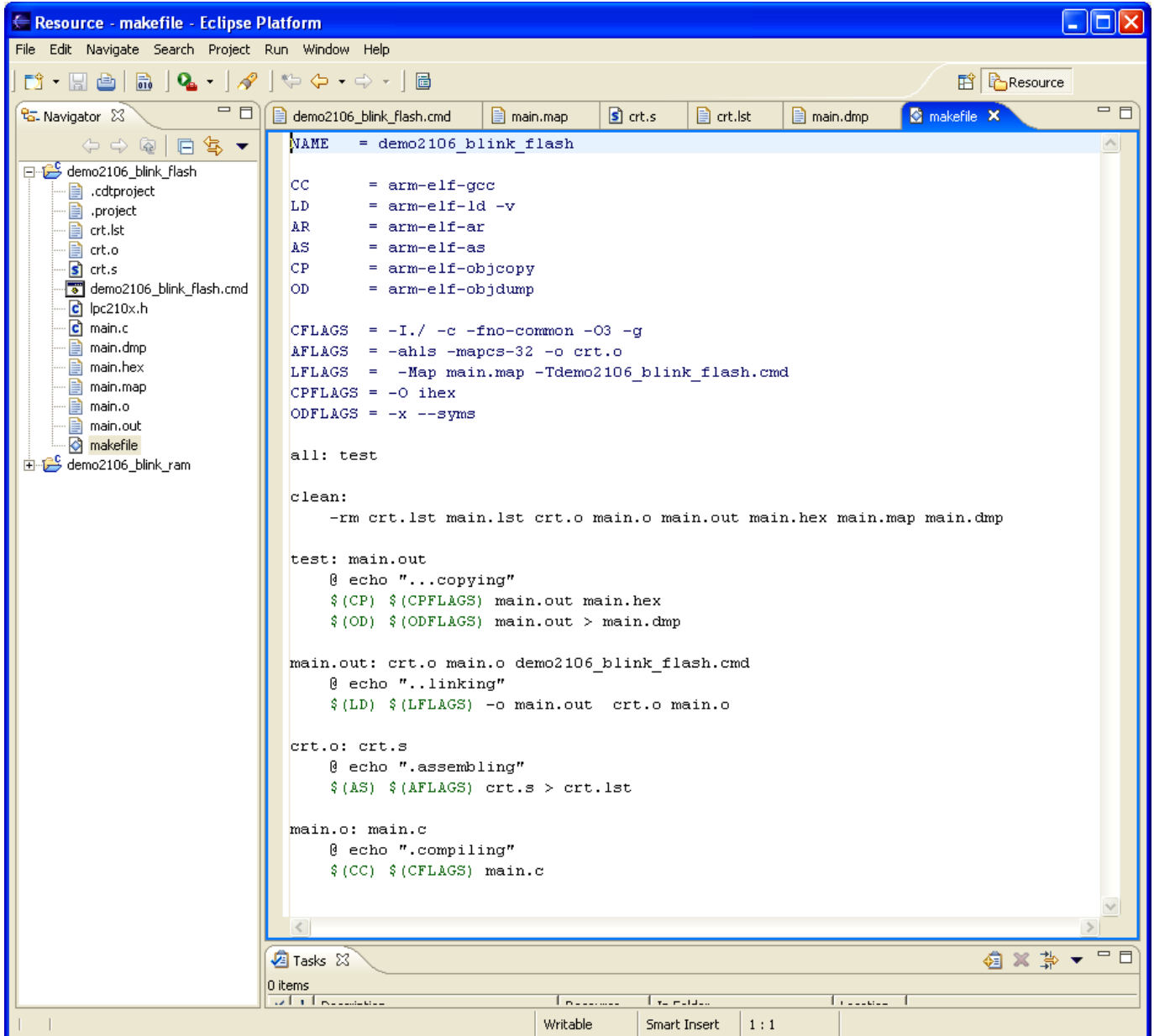
_bss_end = .;
_end = .;
```

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



16 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 screenshot shows the Eclipse IDE interface with a Makefile open in the editor. The Makefile content is as follows:

```
NAME = demo2106_blink_flash

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 -O3 -g
AFLAGS = -ahls -mapcs-32 -o crt.o
LFLAGS = -Map main.map -Tdemo2106_blink_flash.cmd
CPFLAGS = -O ihex
ODFLAGS = -x --syms

all: test

clean:
    -rm crt.lst main.lst crt.o main.o main.out main.hex main.map main.dmp

test: main.out
    @ echo "...copying"
    ${CP} ${CPFLAGS} main.out main.hex
    ${OD} ${ODFLAGS} main.out > main.dmp

main.out: crt.o main.o demo2106_blink_flash.cmd
    @ echo "..linking"
    ${LD} ${LFLAGS} -o main.out crt.o main.o

crt.o: crt.s
    @ echo ".assembling"
    ${AS} ${AFLAGS} crt.s > crt.lst

main.o: main.c
    @ echo ".compiling"
    ${CC} ${CFLAGS} main.c
```

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:

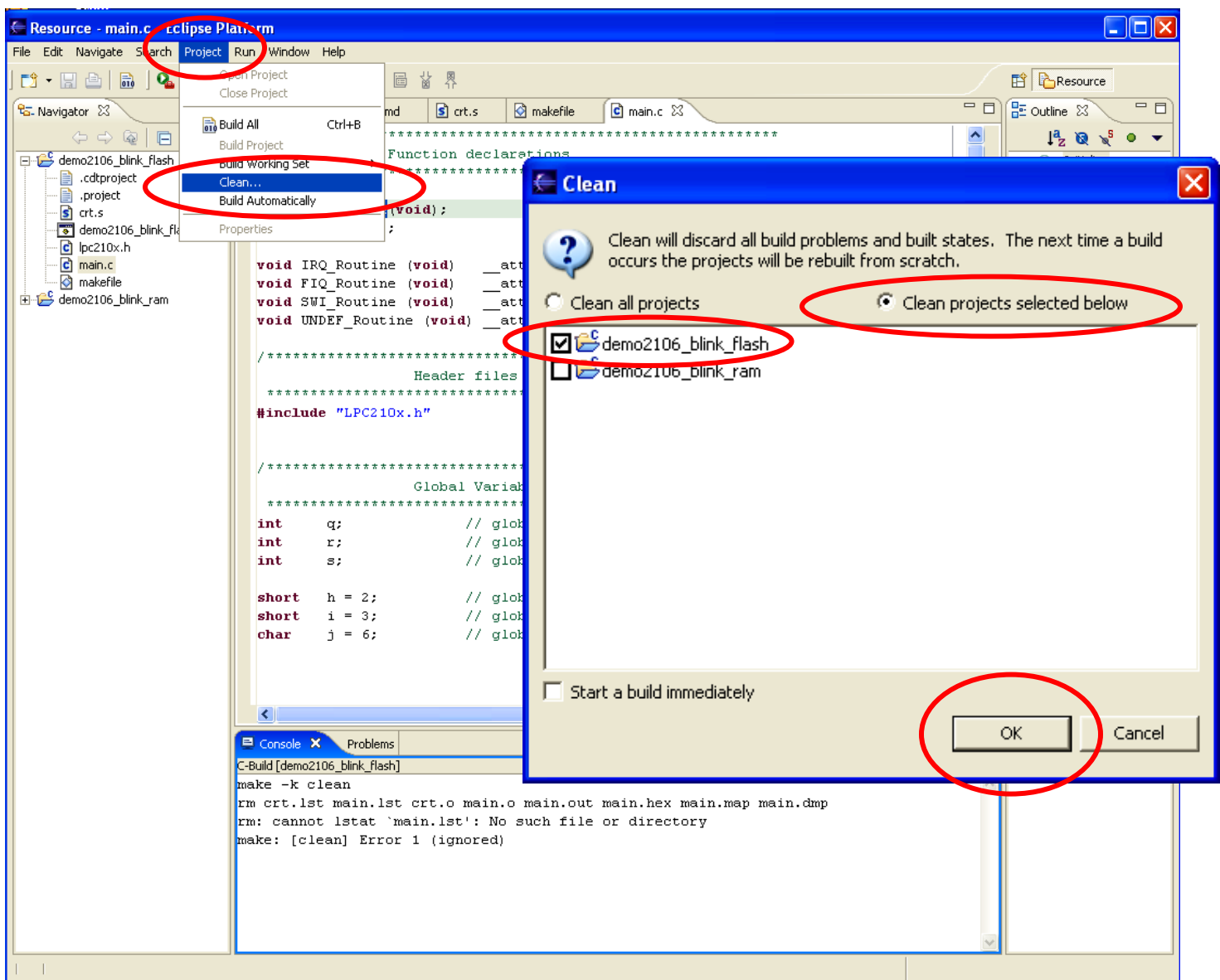
- Map main.map** = creates a map file

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

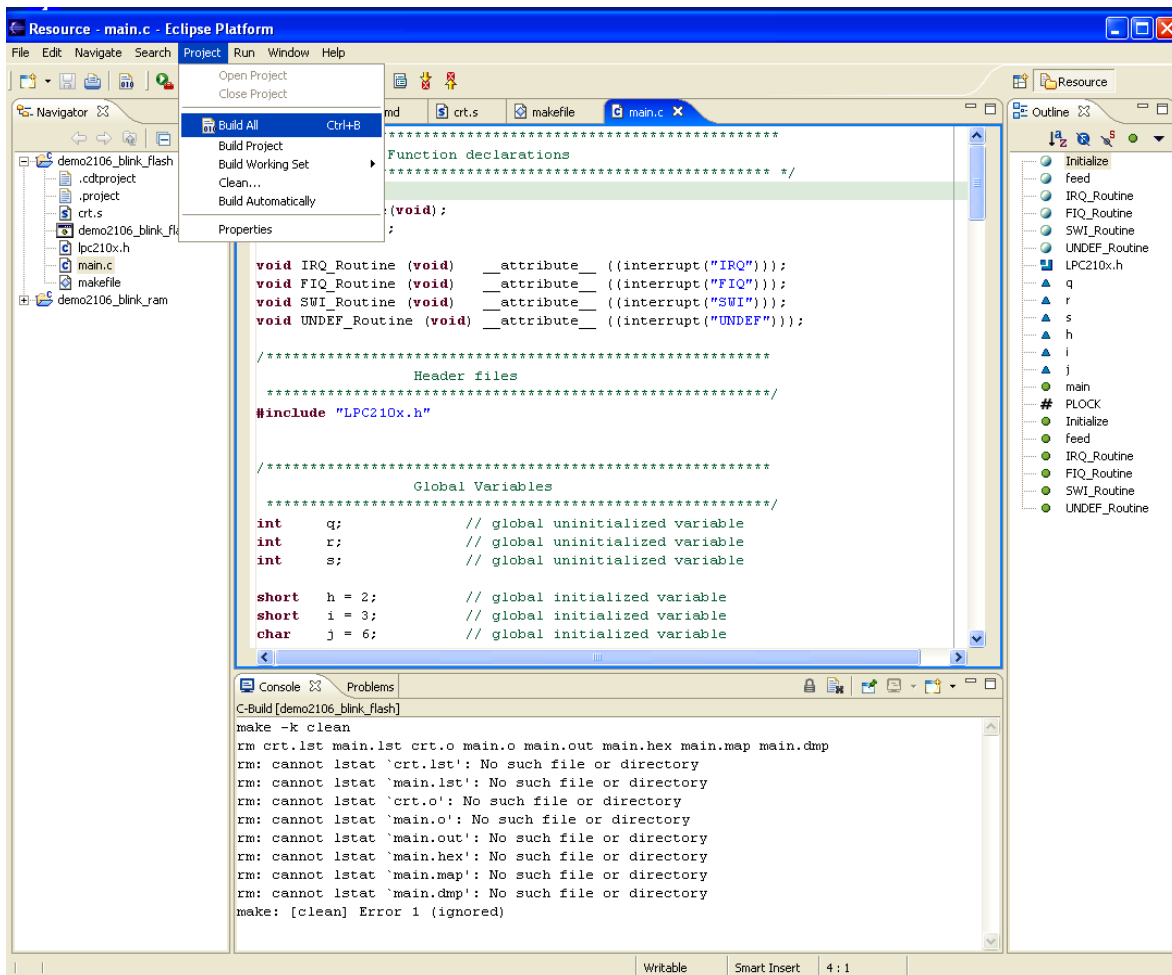
Note that I've kept this GNU makefile as simple as possible. You can clearly see the assembler, C compiler and linker steps. They are followed by the **objcopy** utility that makes the hex file for the Philips ISP boot loader and an **objdump** operation to give a nice file of all symbols, etc.

17 Compiling and Linking the Sample Application

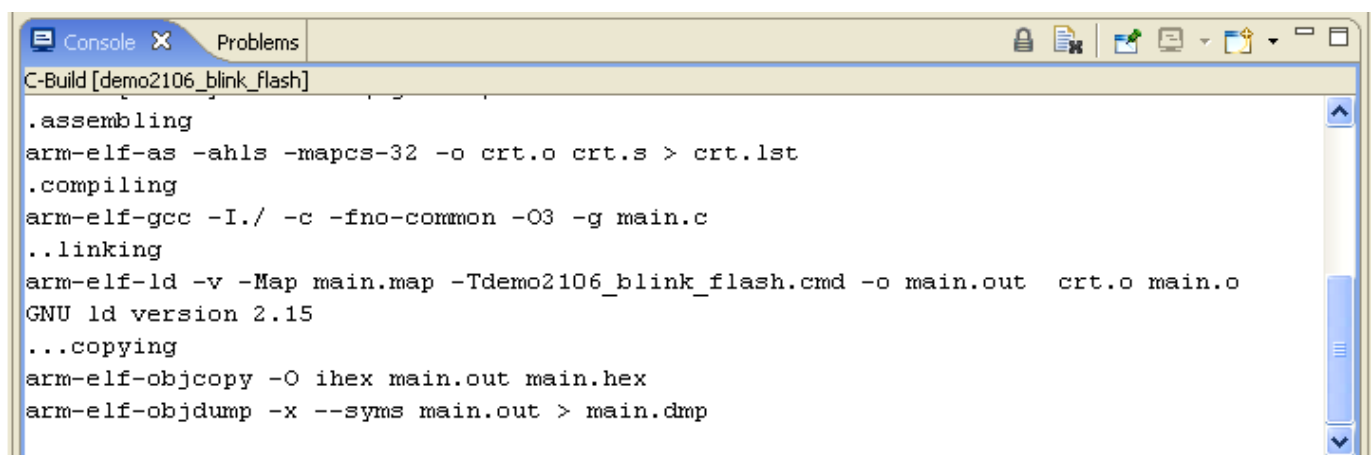
OK, now it's time to actually do something. First, let's "Clean" the project; this gets rid of all object and list files, etc. Click on "Project – Clean ..." and fill out the Clean dialog window.



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.

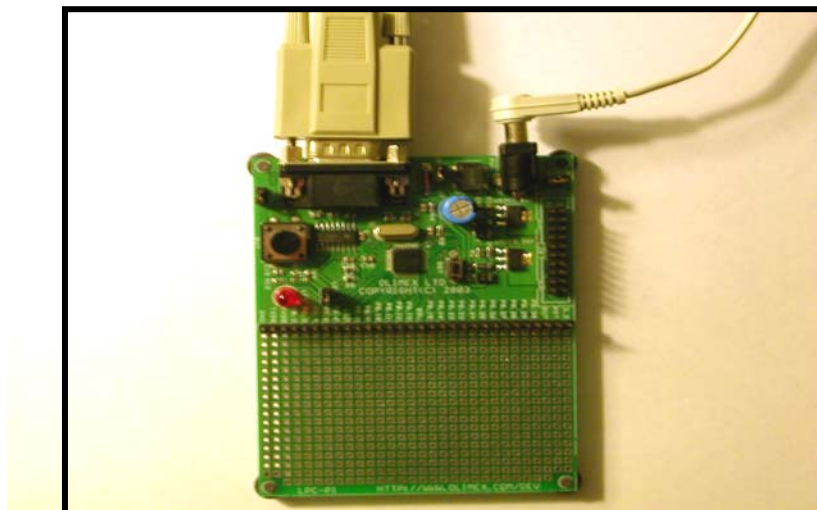
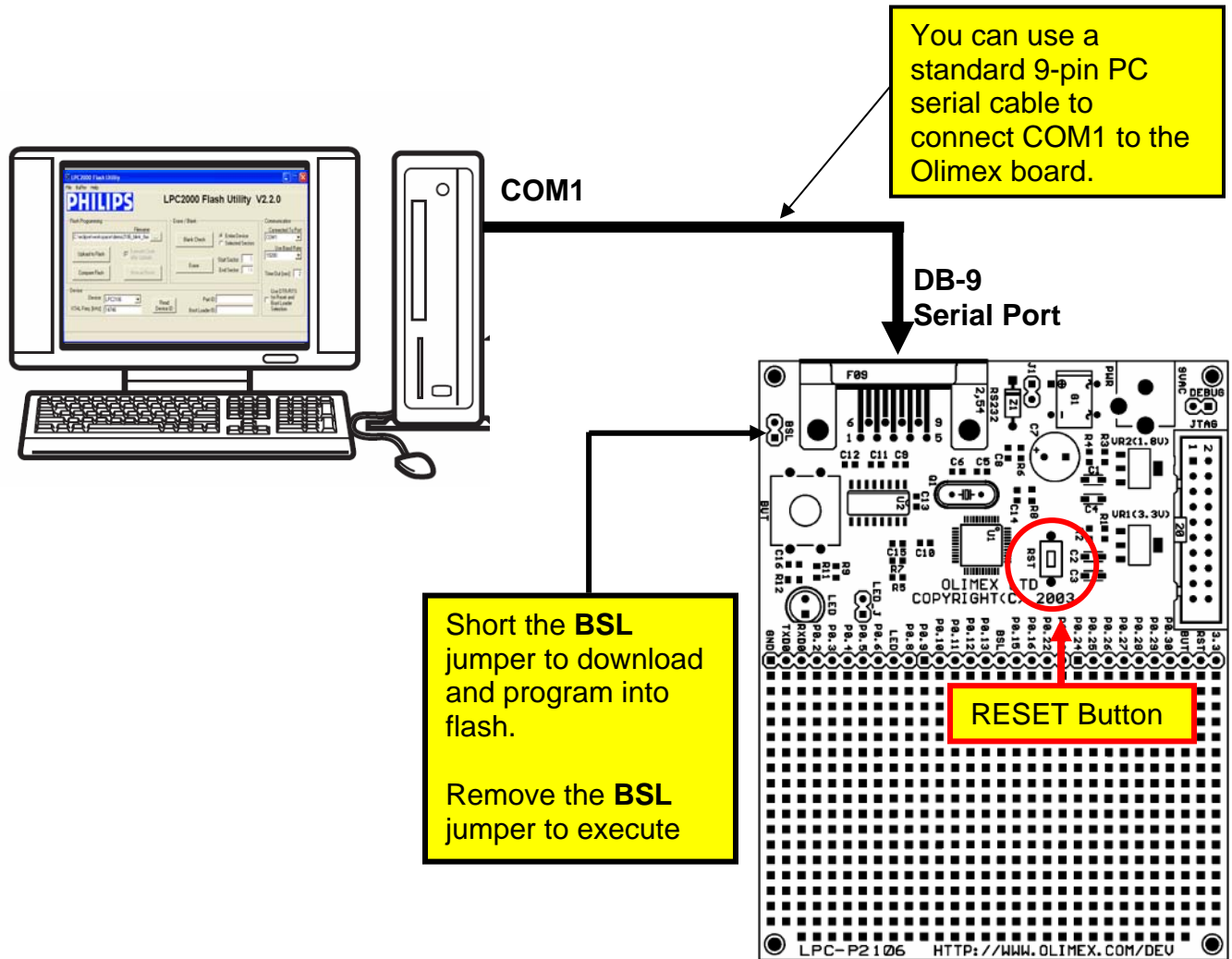


We can see the results in the Console Window at the bottom.

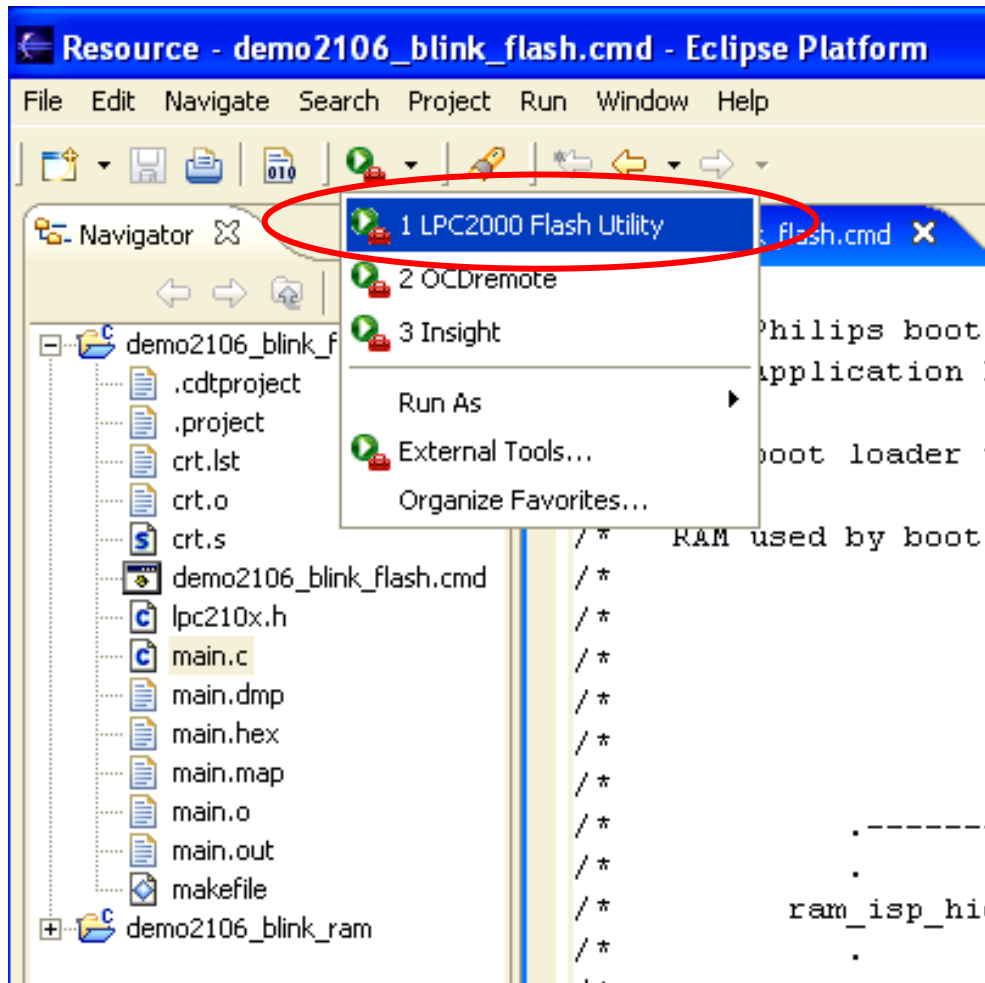


18 Setting Up the Hardware

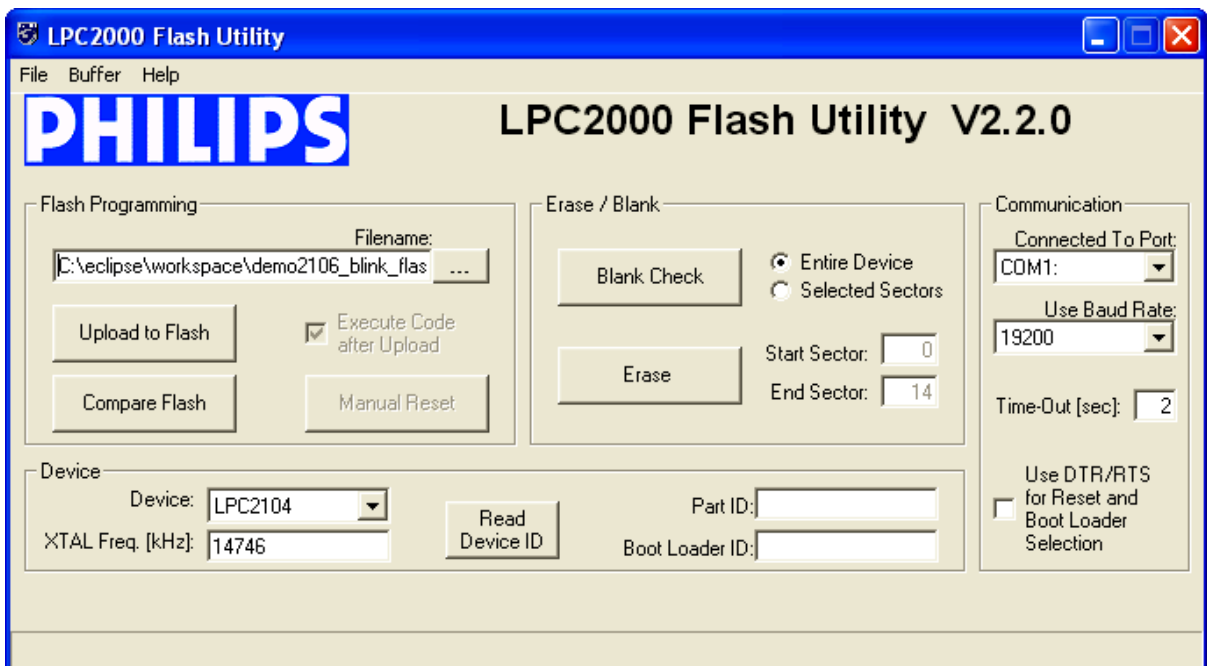
For this tutorial, we'll be using the Olimex **LPC-P2106 Prototype Board**. Connect a straight-through 9-pin serial cable from your computer's COM1 port to the DB-9 connector on the Olimex board. Attach the 9-volt power supply to the PWR connector. Install the BSL jumper and the JTAG jumper.



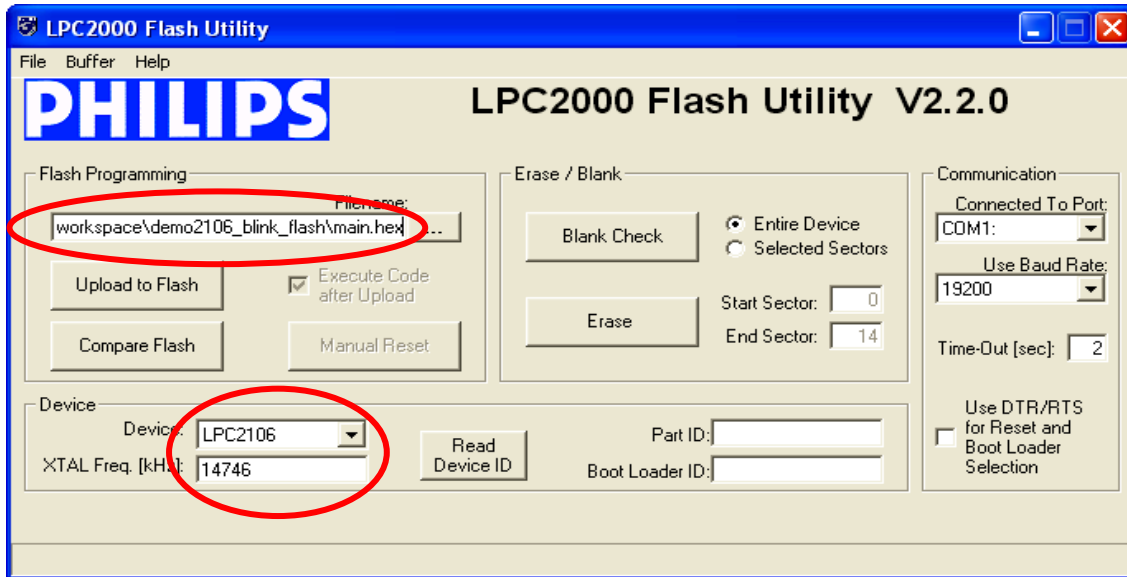
To run the Philips LPC2000 Flash Utility, it's easiest to just click on the “**External Tools**” button and its down arrow to pull-down the available tools. Click on “**LPC2000 Flash Utility**” to start the Philips Boot Loader.



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.

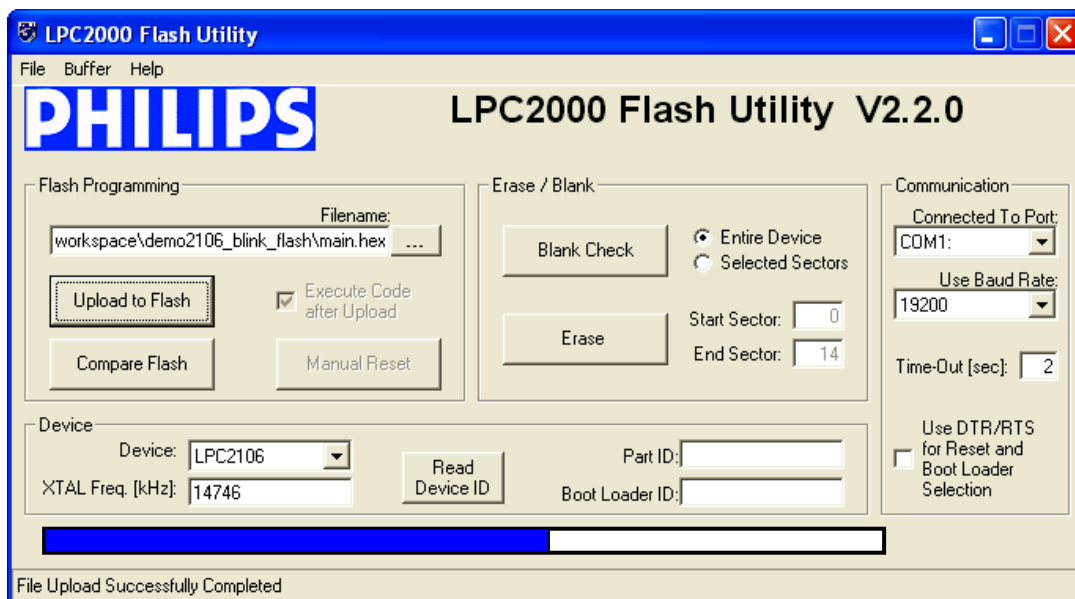


Now click on “**Upload to Flash**” to start the download.

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



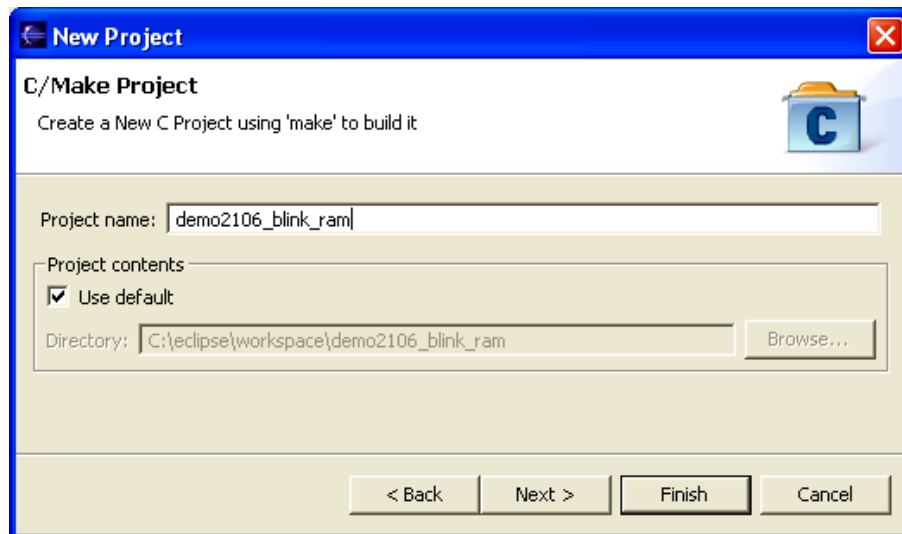
The download will now proceed; you’ll see a blue progress bar at the bottom and then the status line will say “File Upload Successfully Completed”.



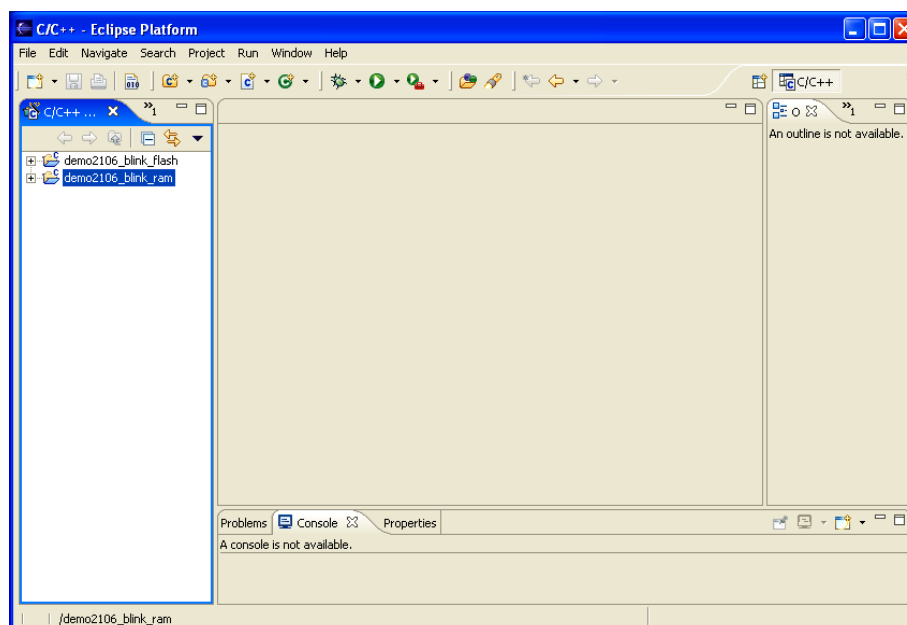
20 Create a New Project to Run the Code in RAM

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 Eclipse/CDT debugger and a Wiggler JTAG interface.

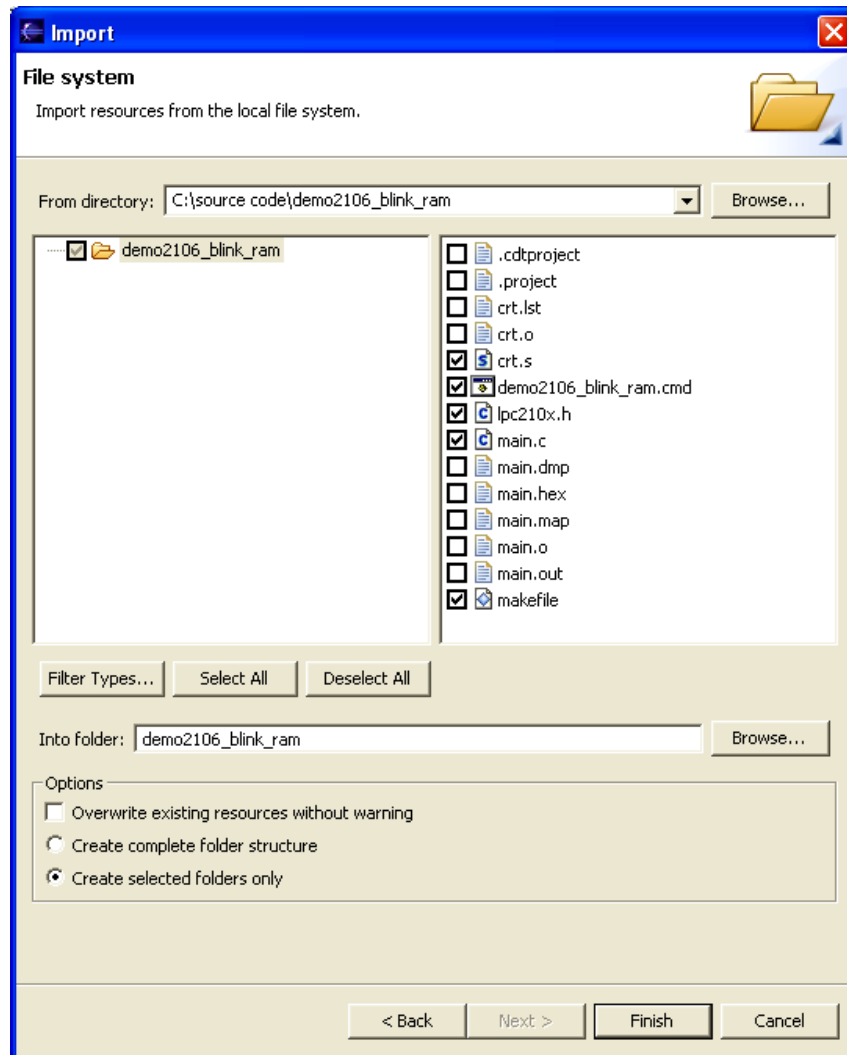
Using the techniques previously discussed, create a new project named **demo2106_blink_ram**.



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

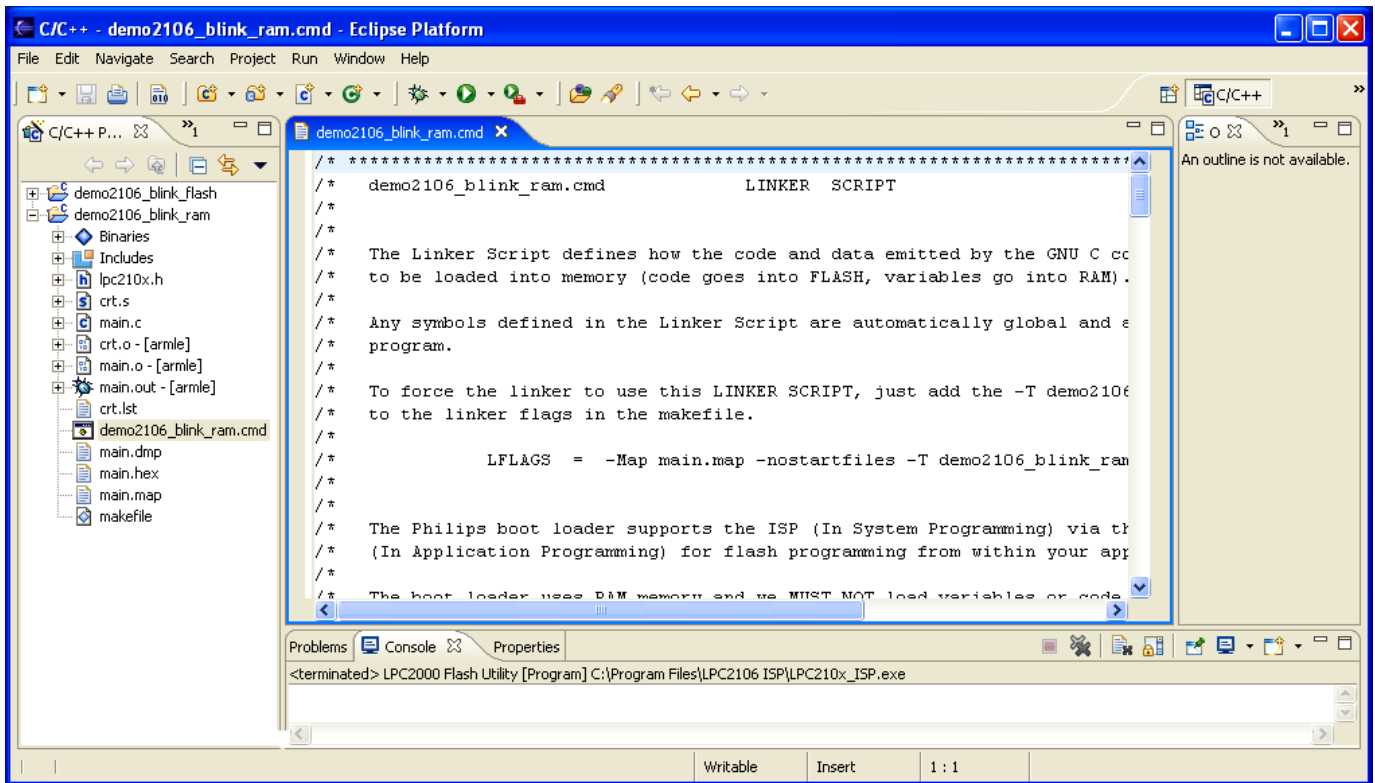


Now using the “**File Import**” procedure described earlier, fetch the source files for the project **demo2106_flash_ram** included in the zip distribution for this tutorial.



The files we import are: **crt.s**
demo2106_blink_ram.cmd
lpc210x.h
main.c
makefile.mak

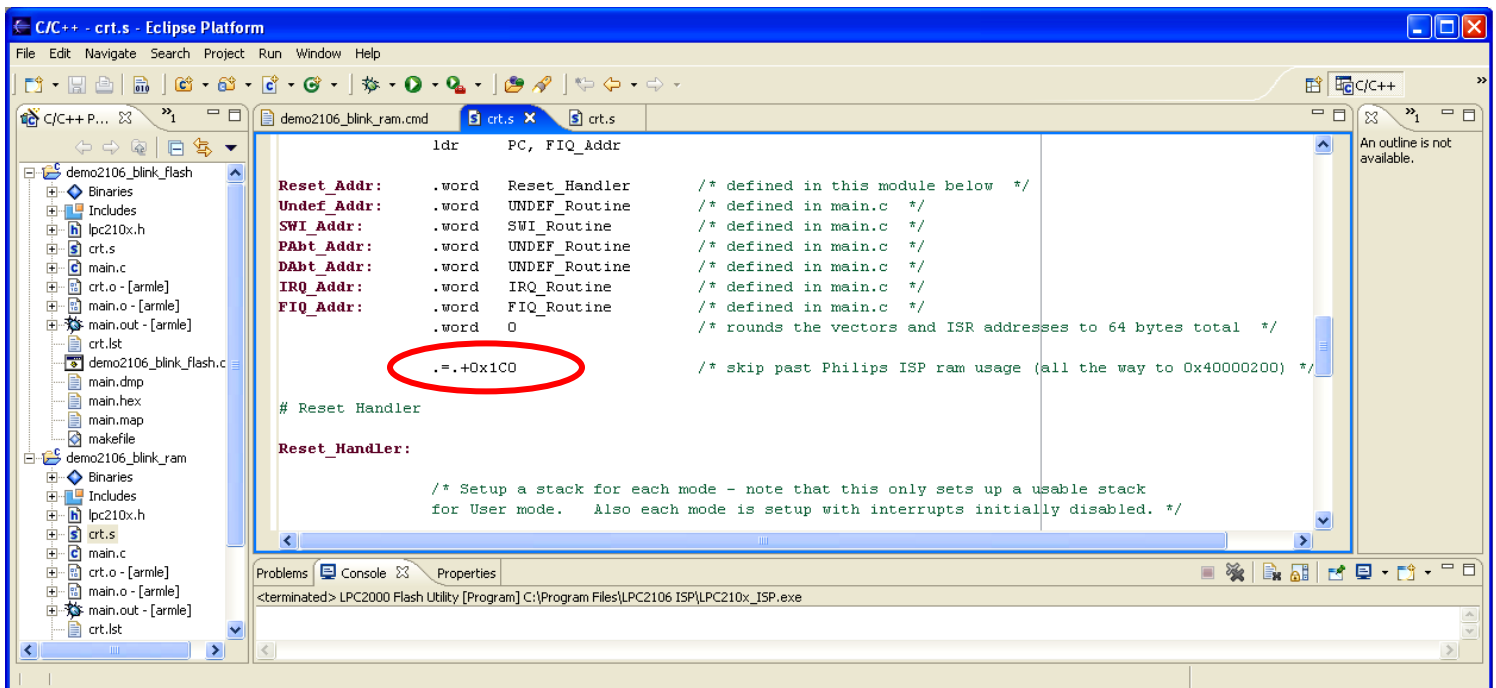
Now if you “Clean and Build” you should see a completed project with all the resultant files, as shown below.



21 Differences in the RAM Version

File CRT.S

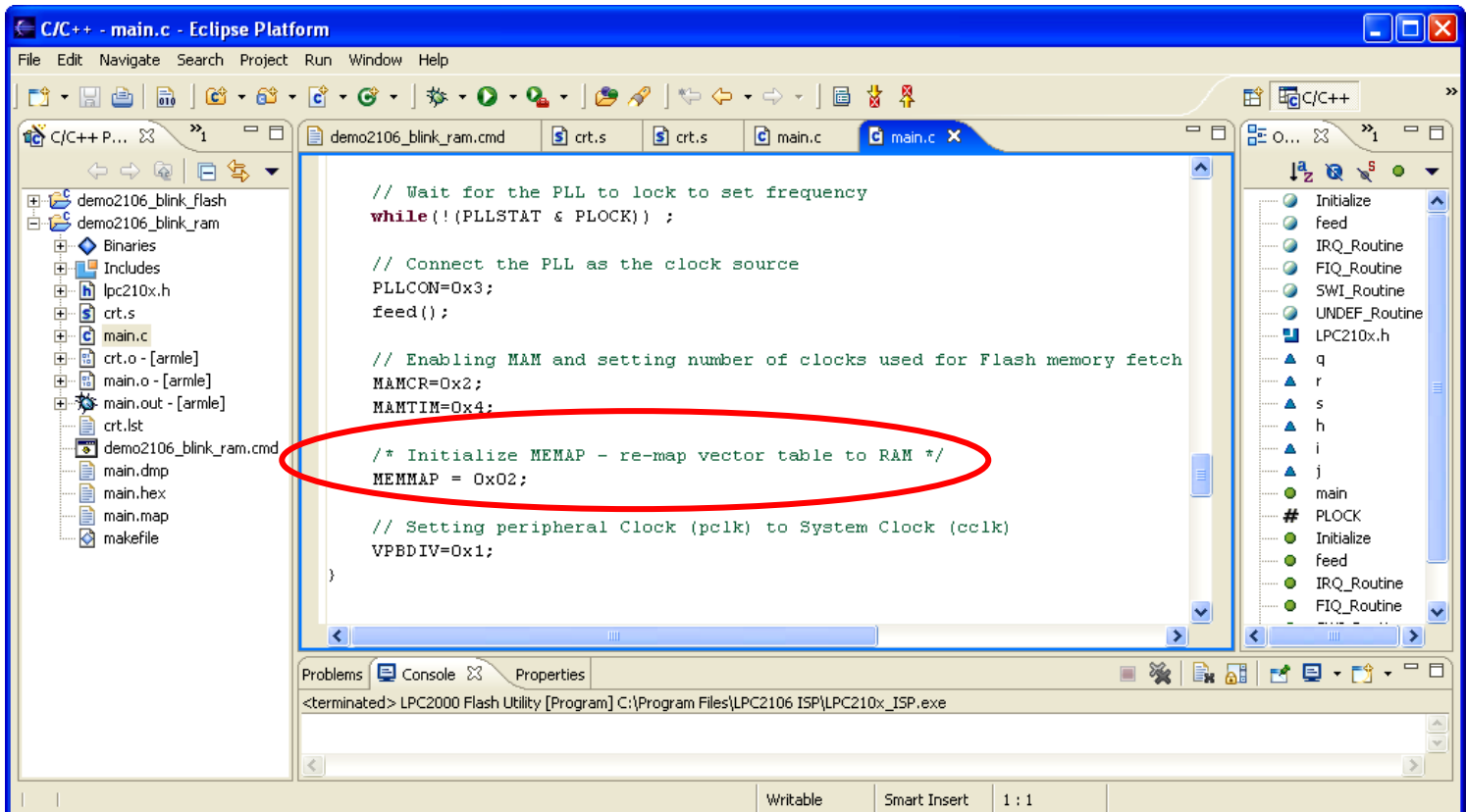
In the startup assembler file, I used a simple trick to move the startup code away from the vectors to ensure that it doesn't encroach on the Philips ISP Flash Loader low RAM area.



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.



```
// Wait for the PLL to lock to set frequency
while (!(PLLSTAT & PLOCK)) ;

// Connect the PLL as the clock source
PLLCON=0x3;
feed();

// Enabling MAM and setting number of clocks used for Flash memory fetch
MAMCR=0x2;
MAMTIM=0x4;

/* Initialize MEMMAP - re-map vector table to RAM */
MEMMAP = 0x02;

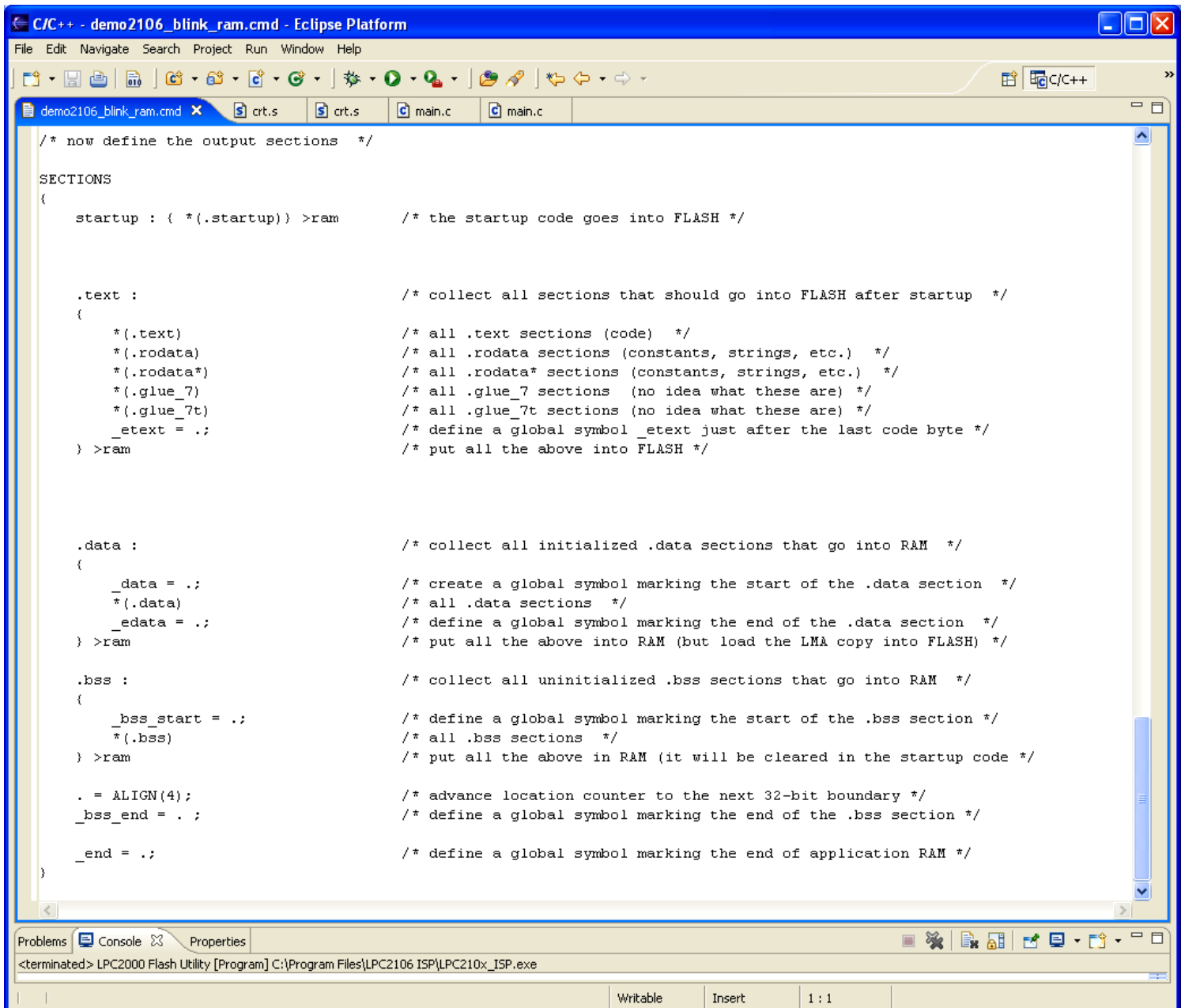
// Setting peripheral Clock (pclk) to System Clock (ccclk)
VPBDIV=0x1;
}
```

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.

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.



```
/* now define the output sections */
SECTIONS
{
    startup : { *(.startup) }>ram      /* the startup code goes into FLASH */

    .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 (no idea what these are) */
        *(.glue_7t)                    /* all .glue_7t sections (no idea what these are) */
        _etext = .;                    /* define a global symbol _etext just after the last code byte */
    }>ram                               /* put all the above into FLASH */

    .data :                             /* collect all initialized .data sections that go into RAM */
    {
        _data = .;                      /* create a global symbol marking the start of the .data section */
        *(.data)                       /* all .data sections */
        _edata = .;                    /* define a global symbol marking the end of the .data section */
    }>ram                               /* put all the above into RAM (but load the LMA copy into FLASH) */

    .bss :                              /* collect all uninitialized .bss sections that go into RAM */
    {
        _bss_start = .;                /* define a global symbol marking the start of the .bss section */
        *(.bss)                        /* all .bss sections */
    }>ram                               /* put all the above in RAM (it will be cleared in the startup code) */

    . = ALIGN(4);                       /* advance location counter to the next 32-bit boundary */
    _bss_end = .;                       /* define a global symbol marking the end of the .bss section */

    _end = .;                           /* define a global symbol marking the end of application RAM */
}
```

Problems Console Properties
<terminated> LPC2000 Flash Utility [Program] C:\Program Files\LPC2106 ISP\LPC210x_ISP.exe
Writable Insert 1:1

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.

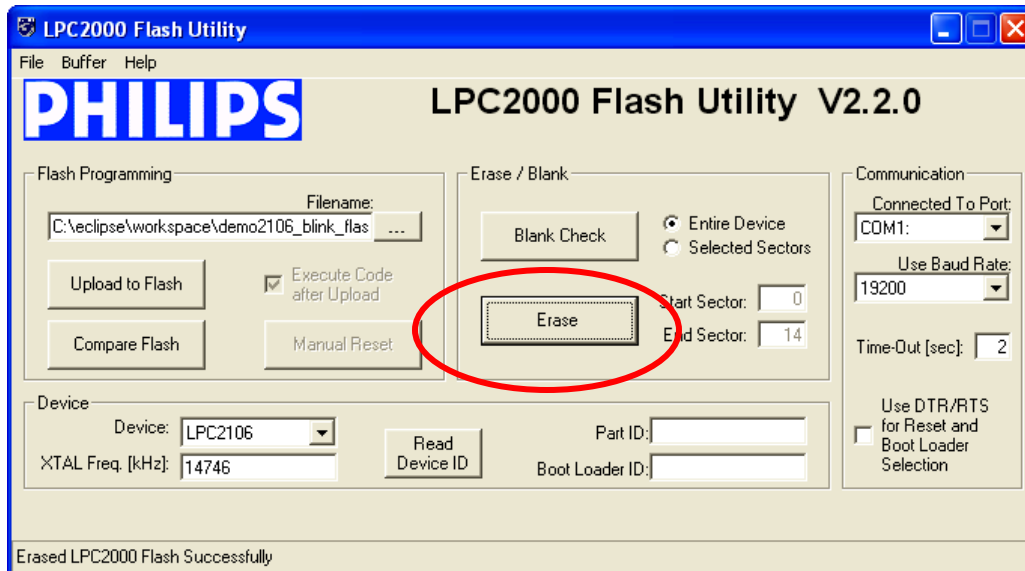
You might also ask, “Do we still clear the **.bss** section?” The answer is absolutely yes, RAM memory powers on into an unknown state. We want all uninitialized variables to

be zero at 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.

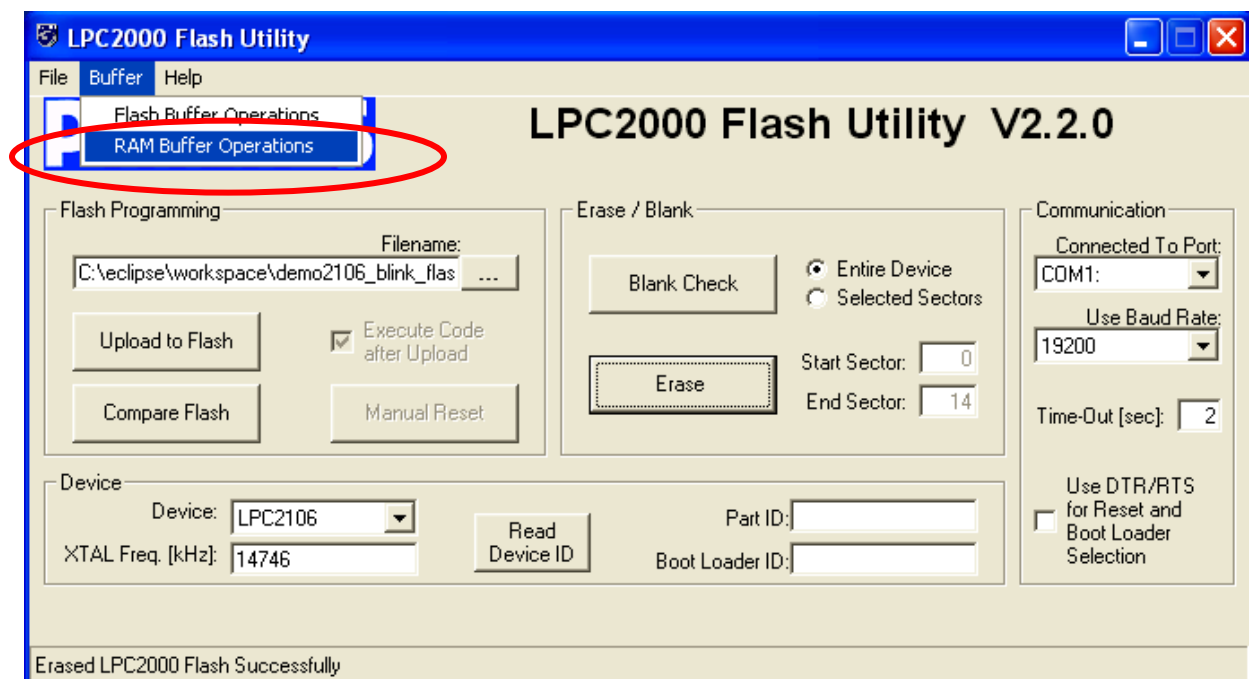
Make sure the BSL jumper is installed.

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.

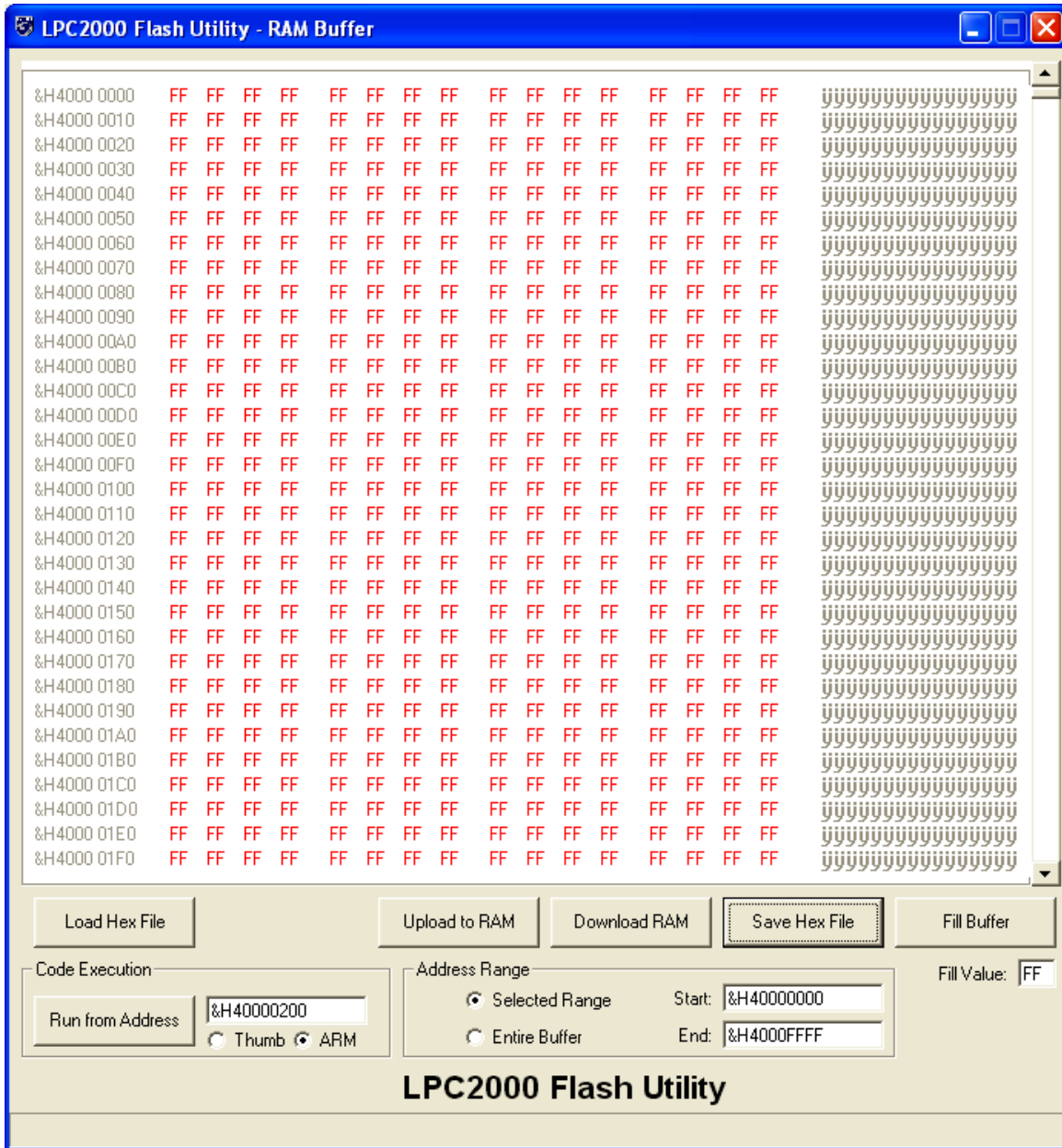


Now we can be sure that the blinking LED is not the Flash application running.

Click on "Buffer – RAM Buffer Operations."

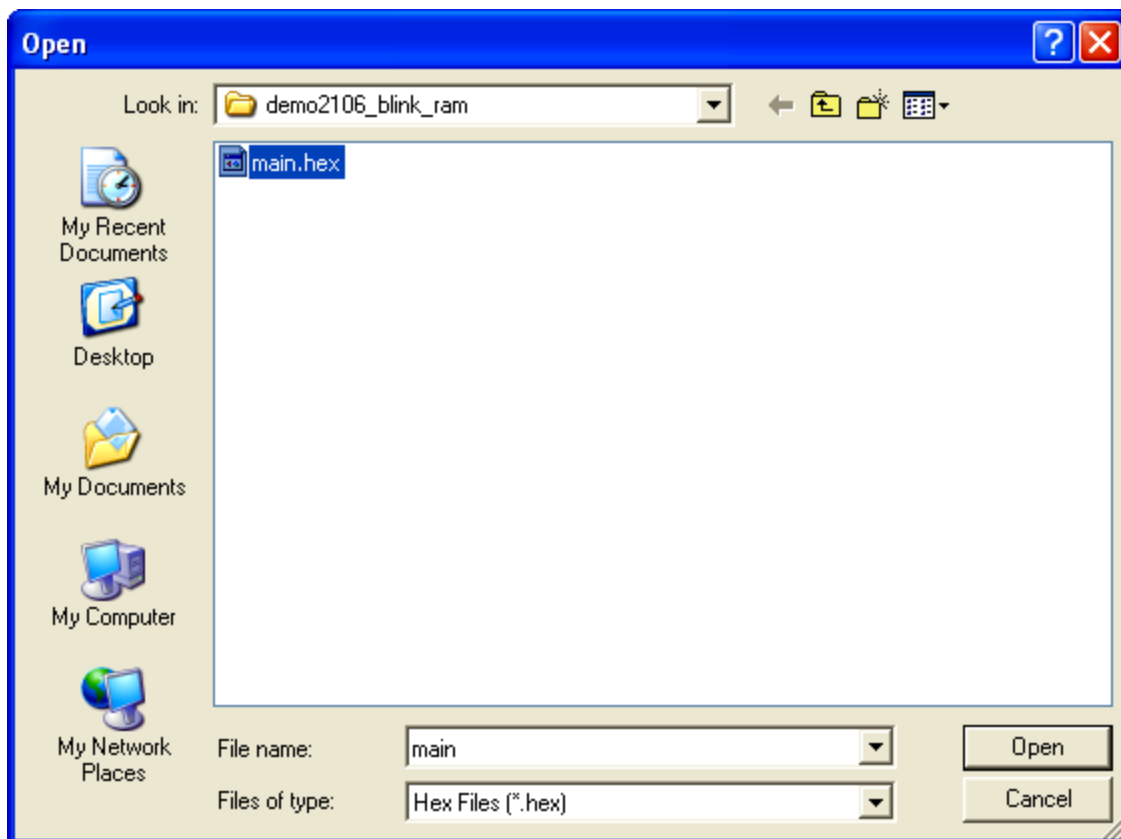


The RAM Buffer screen now appears. Click on “**Load Hex File.**” This is just an operation that fetches the hex file and puts it into the **Philips ISP Flash Utility.**



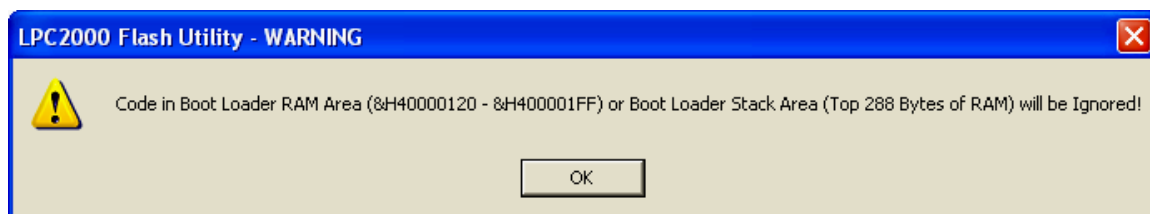
Notice that the button titled “Run from Address” has the value &H40000200 in it. This is thanks to the **ENTRY(Reset_Handler)** directive in the linker command script file. The Philips boot loader will simply load 0x40000200 into the PC register and let her rip!

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.

LPC2000 Flash Utility - RAM Buffer

&H4000 0000	18 F0 9F E5	18 F0 9F E5	18 F0 9F E5	18 F0 9F E5
&H4000 0010	18 F0 9F E5	00 00 A0 E1	14 F0 9F E5	14 F0 9F E5
&H4000 0020	00 02 00 40	C0 02 00 40	BC 02 00 40	C0 02 00 40
&H4000 0030	00 02 00 40	B4 02 00 40	B8 02 00 40	00 00 00 00
&H4000 0040	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0050	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0060	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0070	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0090	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00C0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00D0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00E0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 00F0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0100	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0110	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0120	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0130	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0140	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0150	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0160	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0170	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0180	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 0190	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01C0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01D0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01E0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
&H4000 01F0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

Buttons: Load Hex File, **Upload to RAM**, Download RAM, Save Hex File, Fill Buffer

Code Execution: Run from Address: &H40000200, Thumb, ARM

Address Range: Selected Range, Entire Buffer, Start: &H40000000, End: &H4000043F, Fill Value: FF

LPC2000 Flash Utility

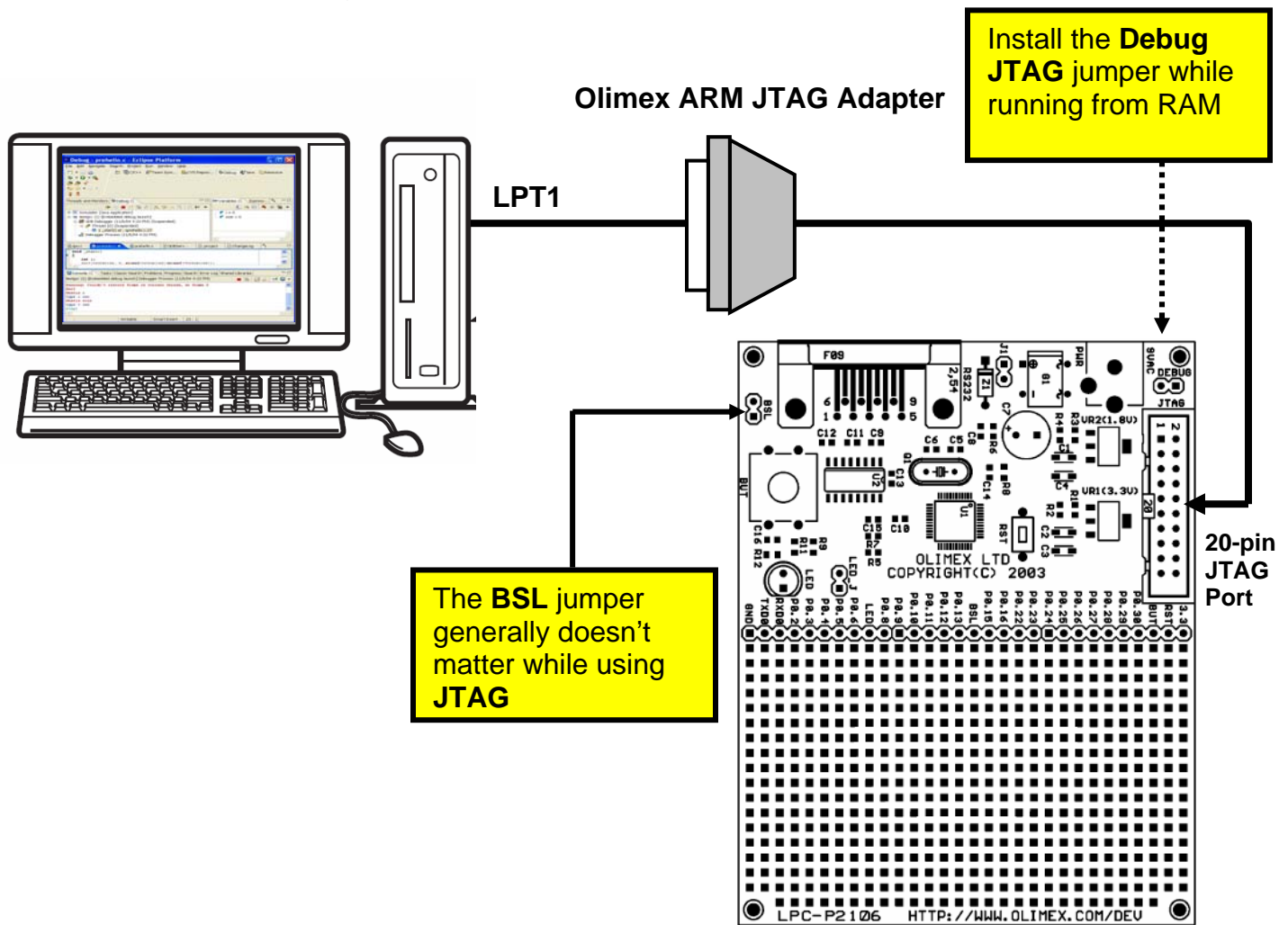
Buffer Upload Successfully Completed

22 Debug the RAM Project

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.

Eclipse/CDT interfaces seamlessly to the GDB debugger that is an integral part of the GNU tool chain. When you click on the “**Debug**” button, you will be able to watch the execution of your program graphically as it goes from breakpoint to breakpoint. You can park the cursor over a variable name and see its current value (assuming that execution has stopped, of course). You’ll be able to look at structured variables, see the ARM registers and have the ability to modify variables and registers. Unfortunately, the inability of the current **Wiggler/OCDremote** hardware to support hardware breakpoints and flash reprogramming limits the debugger to RAM-based 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.

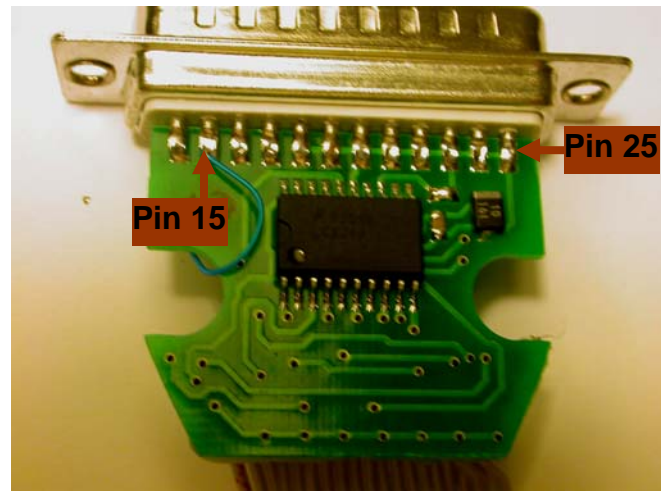
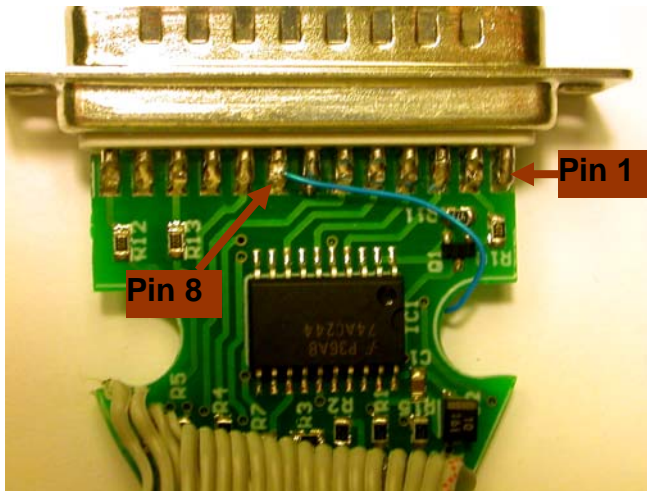


There are several schematic diagrams on the web for the **Wiggler**; notably Leon Heller has one on the LPC2000 message board on Yahoo. You could build your own but I doubt you'd save that much money after paying the shipping from Digikey and the gas to drive to Radio Shack. The Olimex version is a fair 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, **you must connect pin 8 to pin 15.**



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:

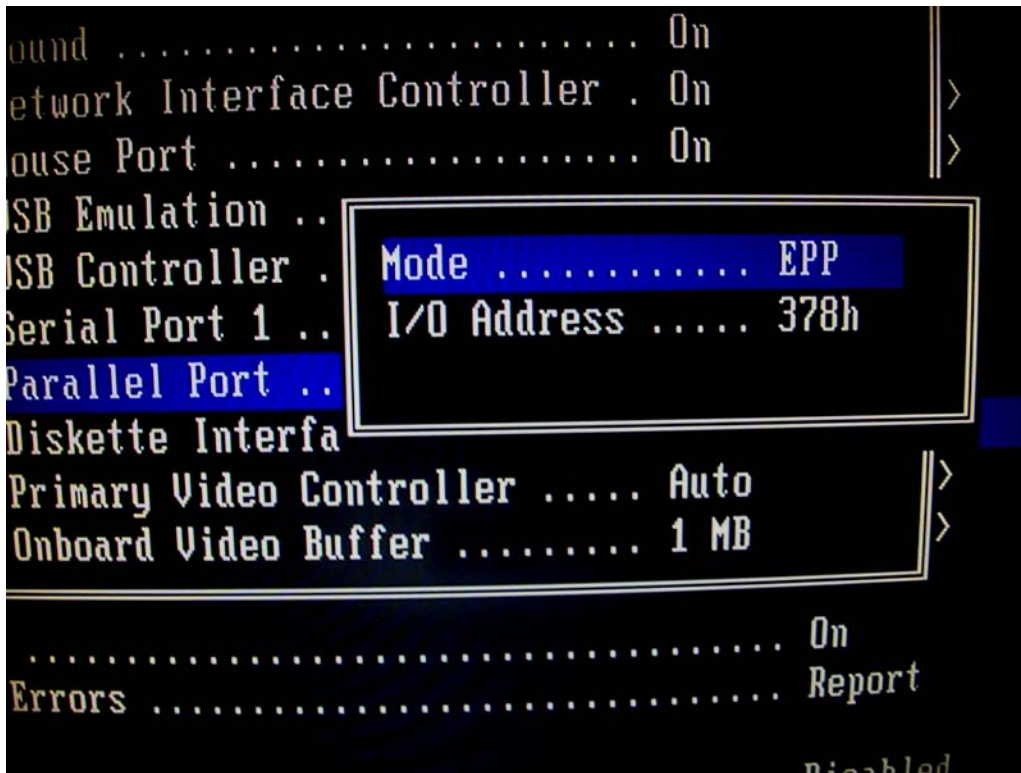
What mode must my parallel port be in?

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.

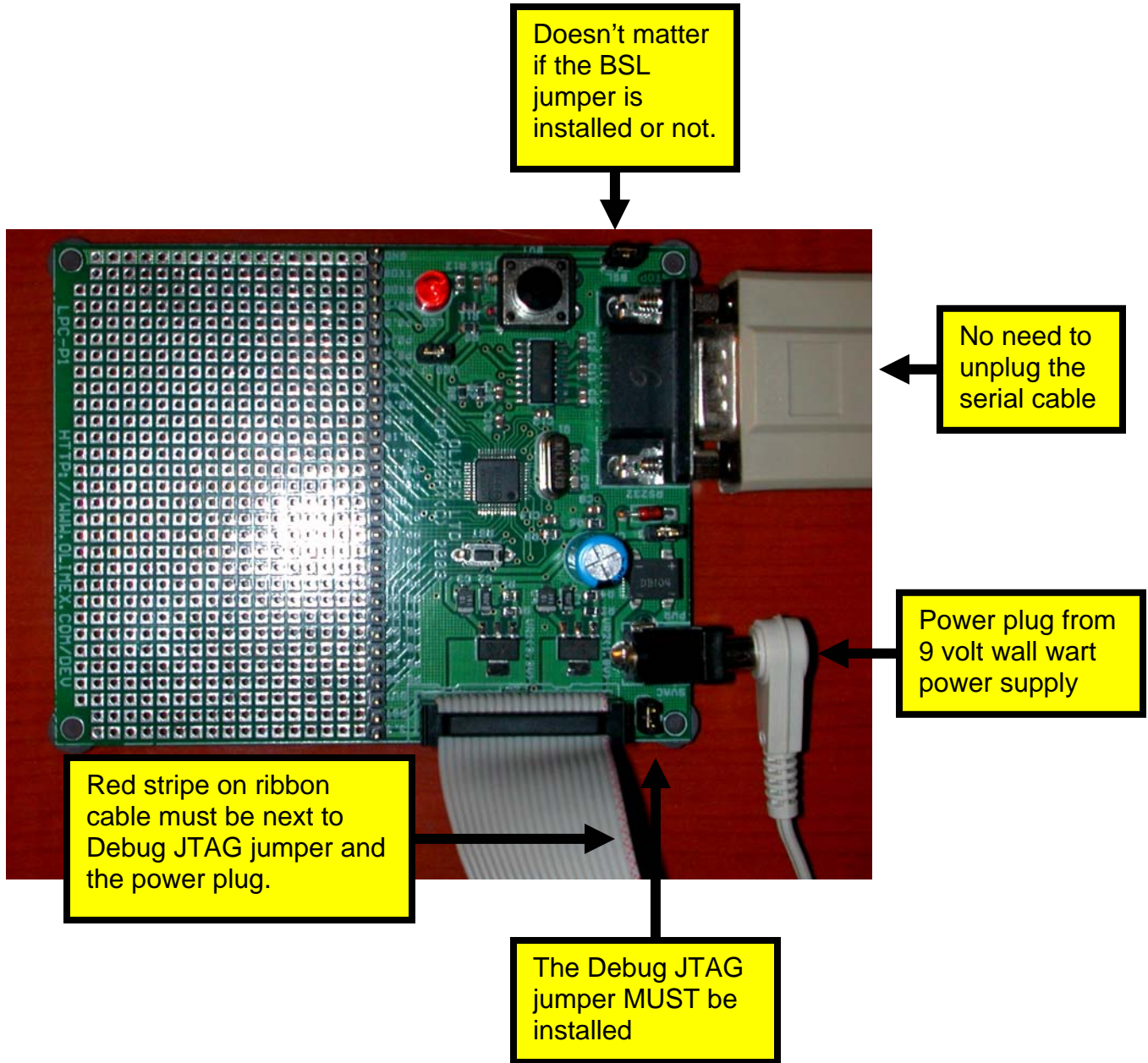
On my Dell Dimension Desktop PC, the CMOS setup can be entered if you hit the **F2** key as the machine boots up. By maneuvering around the CMOS setup, you can find the Parallel Port setup and see what mode it is set up as. If it's **ECP** mode, change it to **EPP** mode, as I did in the screen photograph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

Before we start the **Eclipse** 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

C. Create a Debug Launch Configuration

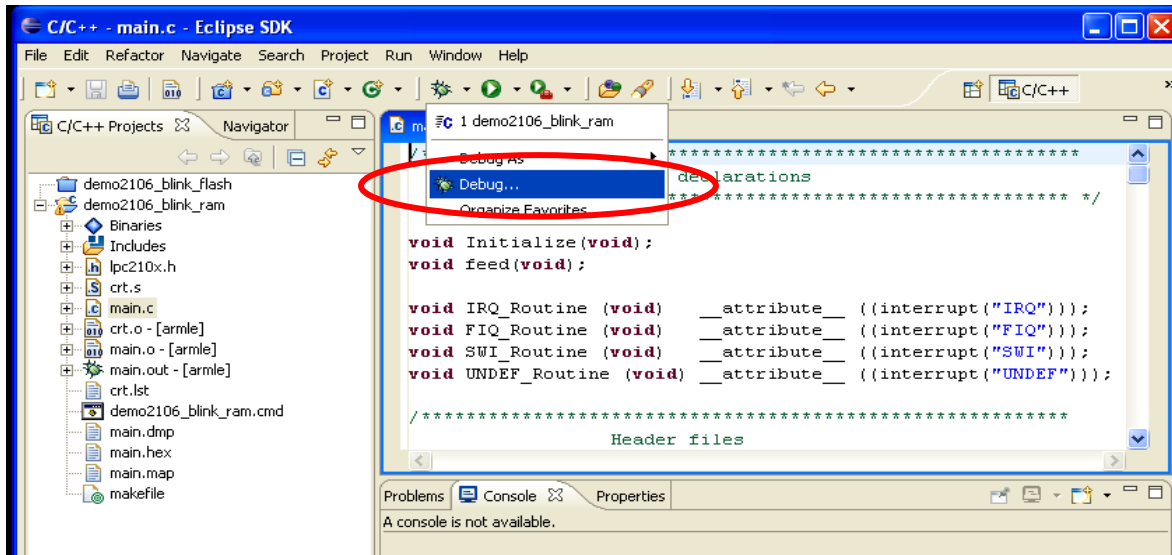
The first order of business is to set up a “**debug launch configuration**.” The quickest way to get to the “**debug launch configuration**” screen is to click on the “insect” button (insect – bug – get it?). Specifically, click on the down arrowhead to bring up the debug pull-down menu.



Click on down arrowhead to get the pull-down menu

Note: Debugging is limited to applications targeted to RAM execution. I suspect that fairly soon an affordable USB/JTAG debug interface and software drivers will be available that will permit use of the two hardware breakpoints available in the ARM JTAG hardware. These two breakpoints will work in flash memory. In this situation, you could use the Philips Flash Utility to download the code into flash and then start the Eclipse debugger.

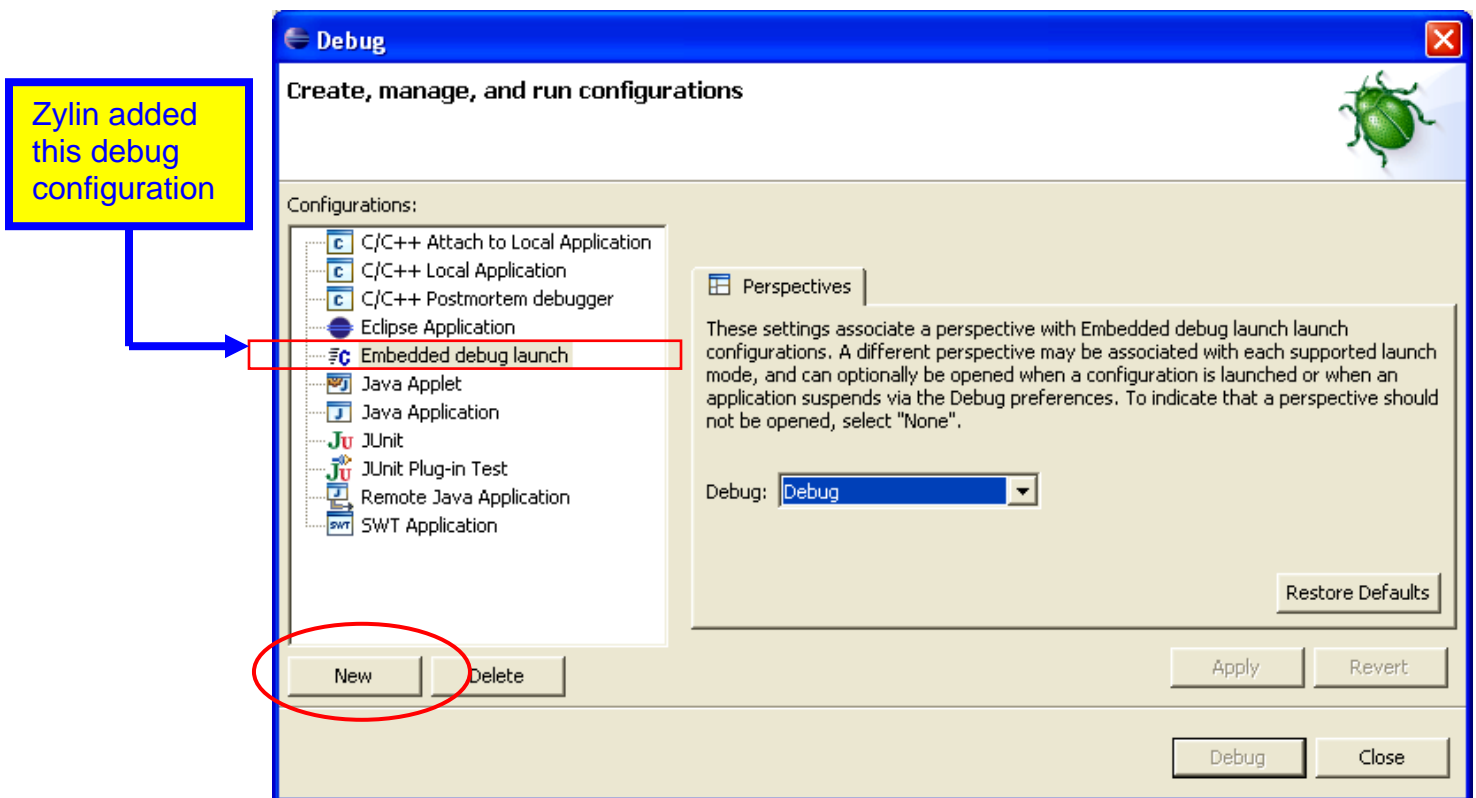
Click on the “**Debug ...**” selection in the debug pull-down list to bring up the Debug configuration screen.



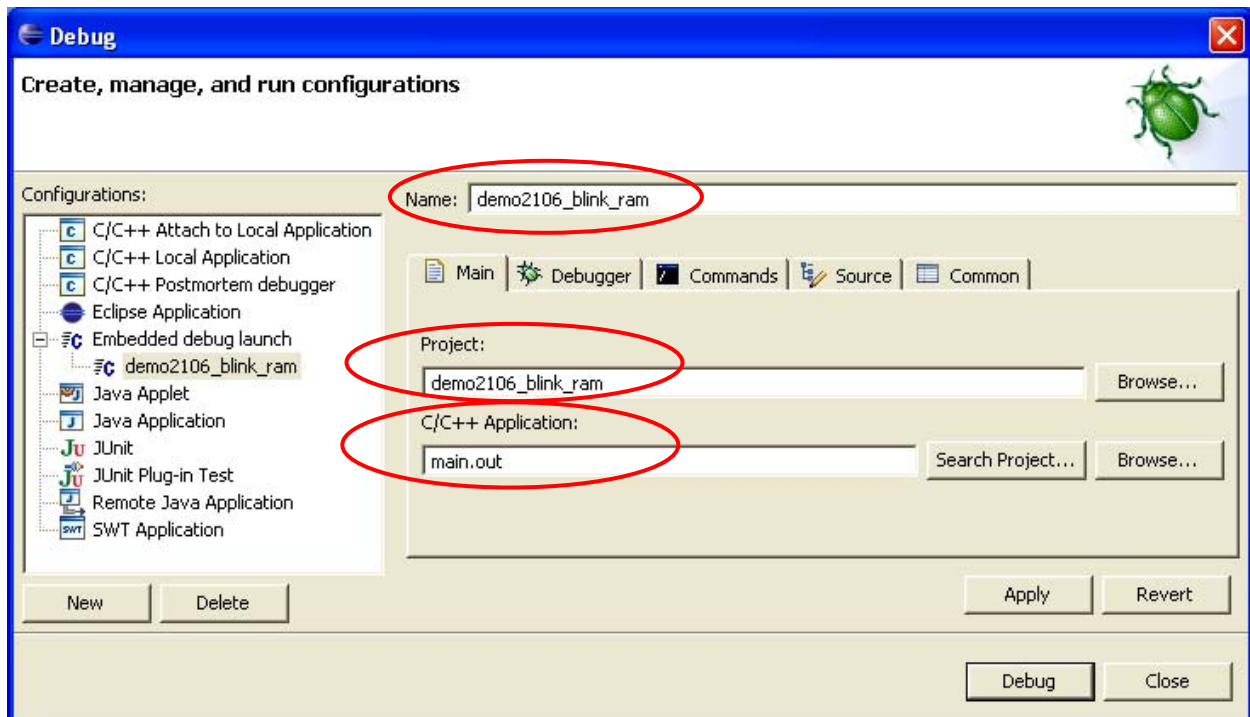
In the “Debug Launch Configuration” screen below, you can see the Zylind modification. Note that one of the possible debug configuration types is now “**Embedded debug launch.**”

You will tend to create a separate “**Embedded debug launch**” configuration for every project you create; it’s very convenient for people who have multiple projects going on at the same time.

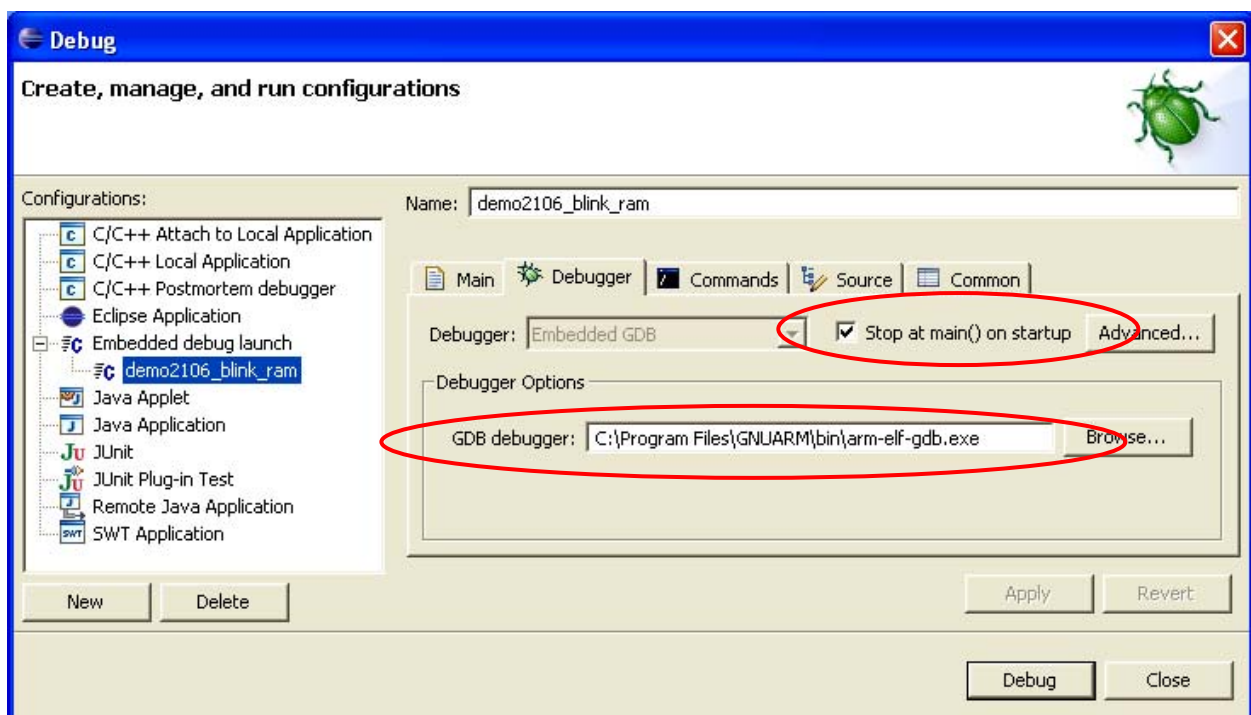
Click on the Zylind “**Embedded debug launch**” configuration and then “**New**” to get started.



In the “Main” tab, set the name to anything you like and the project to “demo2106_blink_ram.” I was, of course, lazy and made the debug configuration name the same as the project. Set the C/C++ Application to “main.out.” Main.out is an arm-elf format file that has the executable and debug information within the file.

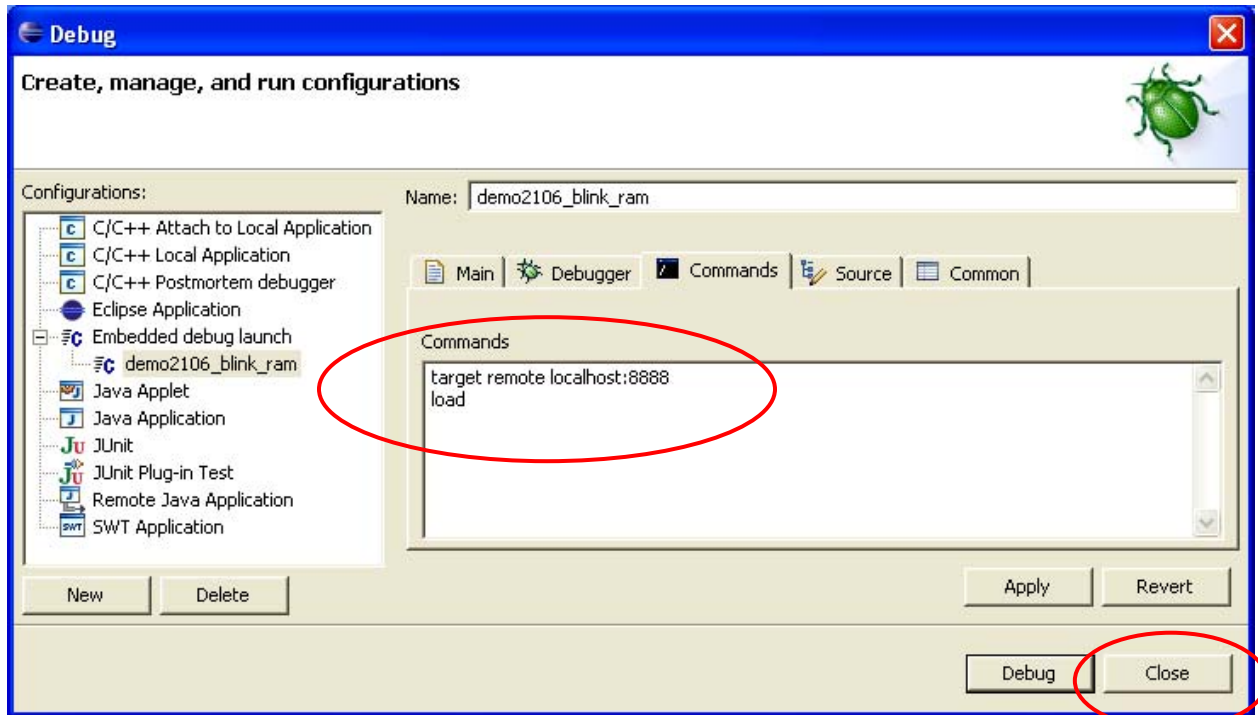


Under the “Debugger” tab, use the “browse” button to set the “GDB debugger” text window to “c:\program files\GNUARM\bin\arm-elf-gdb.exe” and check the box that instructs the debugger to stop at main() on startup.



Under the “**commands**” tab, enter the following two GDB commands to run at startup:

```
target remote localhost:8888
load
```



The “**target remote**” command specifies that the protocol used to talk to the application is “GDB Remote Serial” protocol with the serial device being a internet socket called **localhost:8888** (the default specification for the Macraigor **OCDremote** driver).

Target Remote supports the GDB “load” command; the specific download file (main.out) was specified above in the “main” tab.

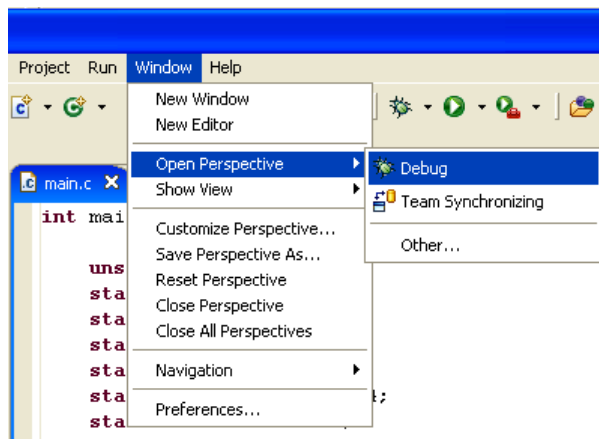
The “**source**” tab can be left at its default settings.

Likewise, the “common” tab can be left at its default setting. Click on “**apply**” and then “**Close**” to complete specification of the debug launch configuration.

D. Switch to Debug Perspective

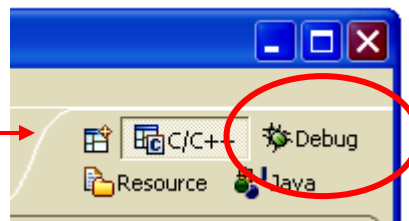
What you see on the screen when using Eclipse is called a “perspective” and up to now, we have been using the “**C/C++**” perspective. Once the application has been built, we must switch to the “**Debug**” perspective to debug it.

One way is to change the perspective in the “**Window**” pull-down menu as shown below.

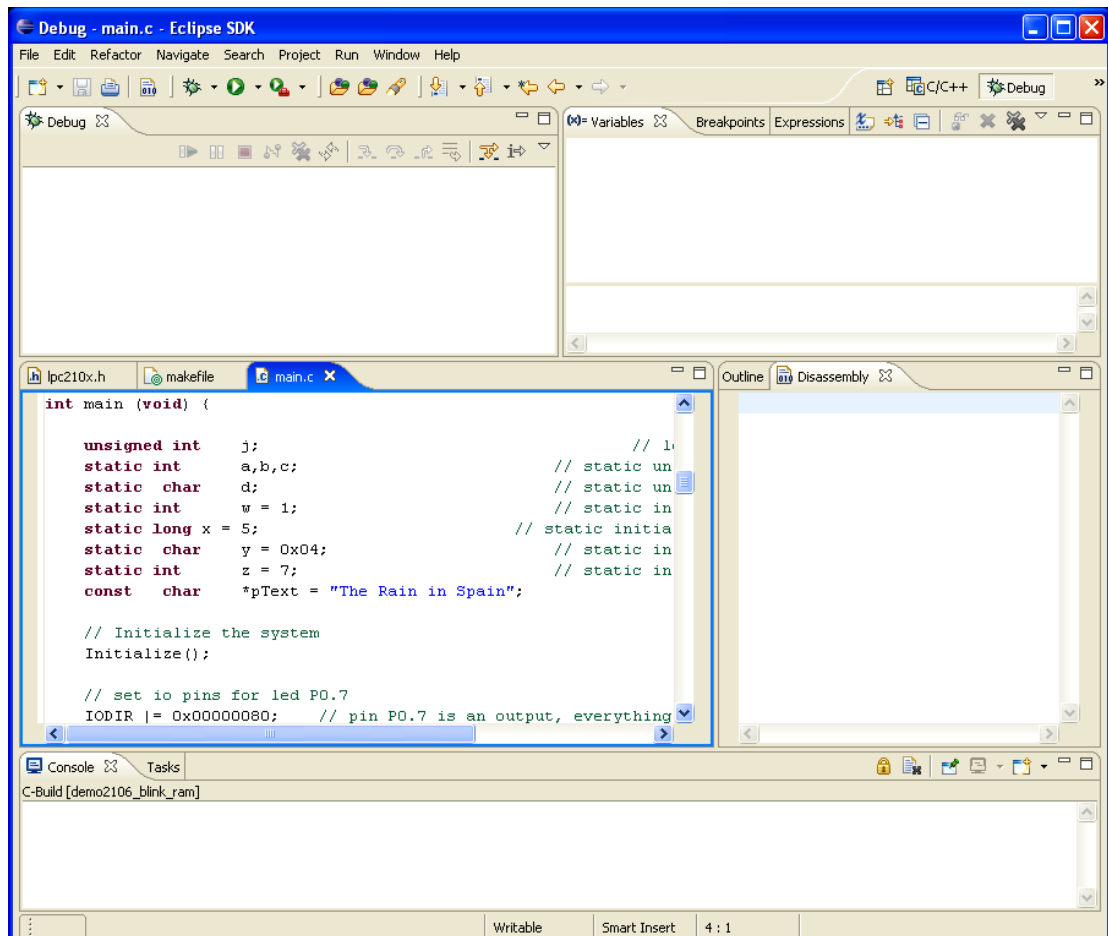


It's also convenient to click on the “**Debug Perspective**” button on the upper right of the Eclipse screen.

You can drag this s-shaped edge to expose all the available perspectives.



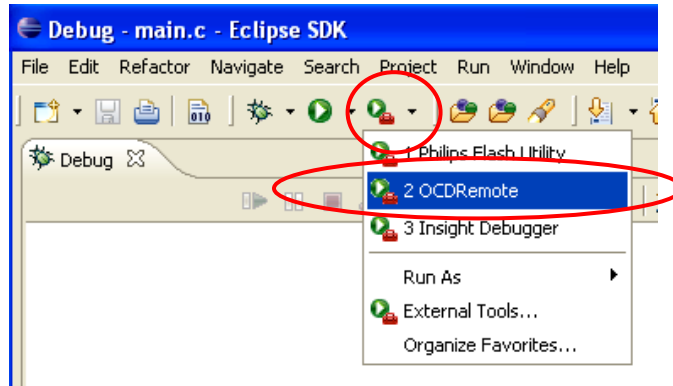
Below is the “Debug” perspective.



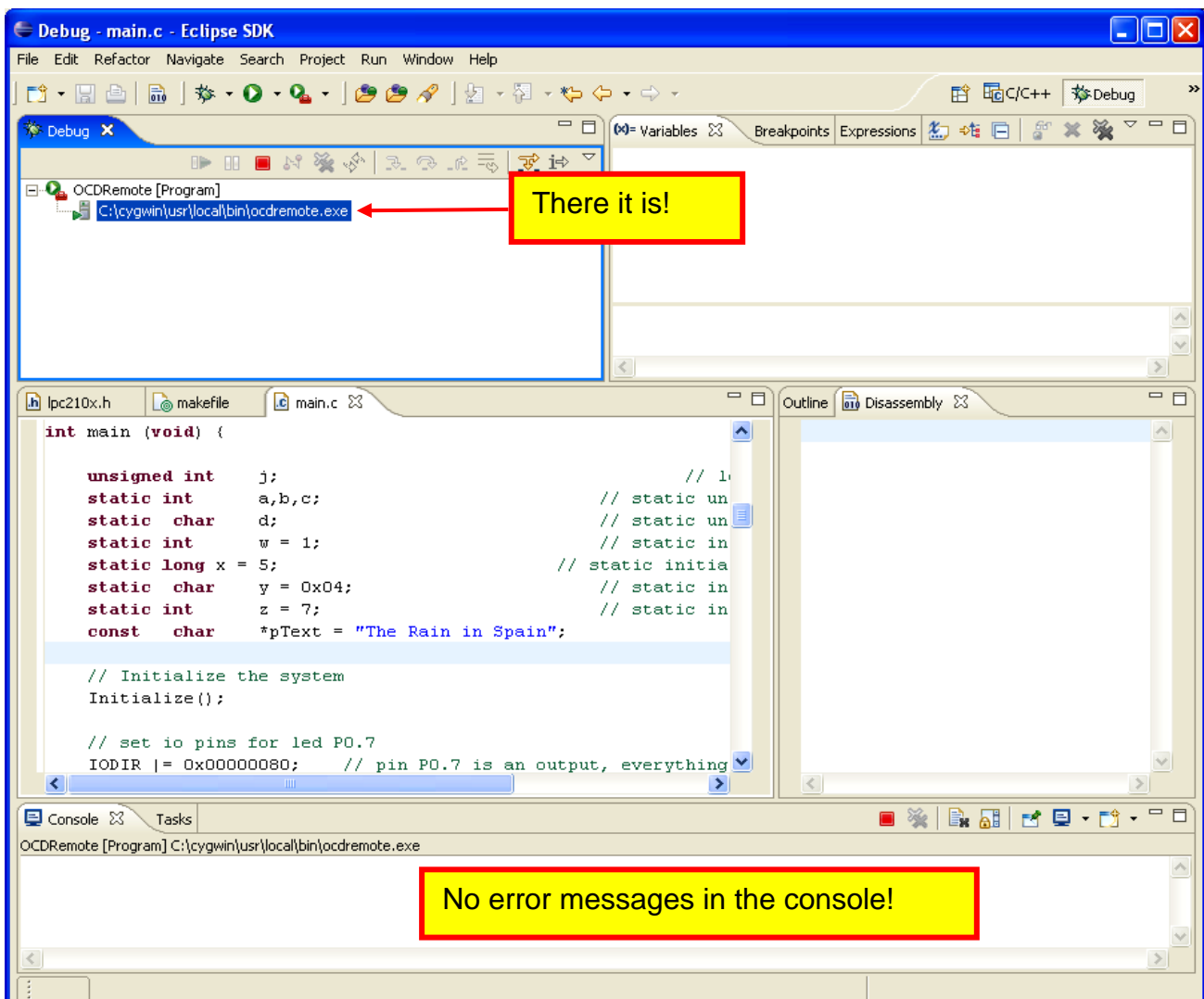
D. Start the OCDRemote utility

The Macraigor **OCDRemote** utility must be started **before** the Debugger is launched.

Remember that we set up the **OCDRemote** as an External Tool. It's easily started by clicking on the pull-down arrow of the External Tool button. Note the little red toolbox on that button.



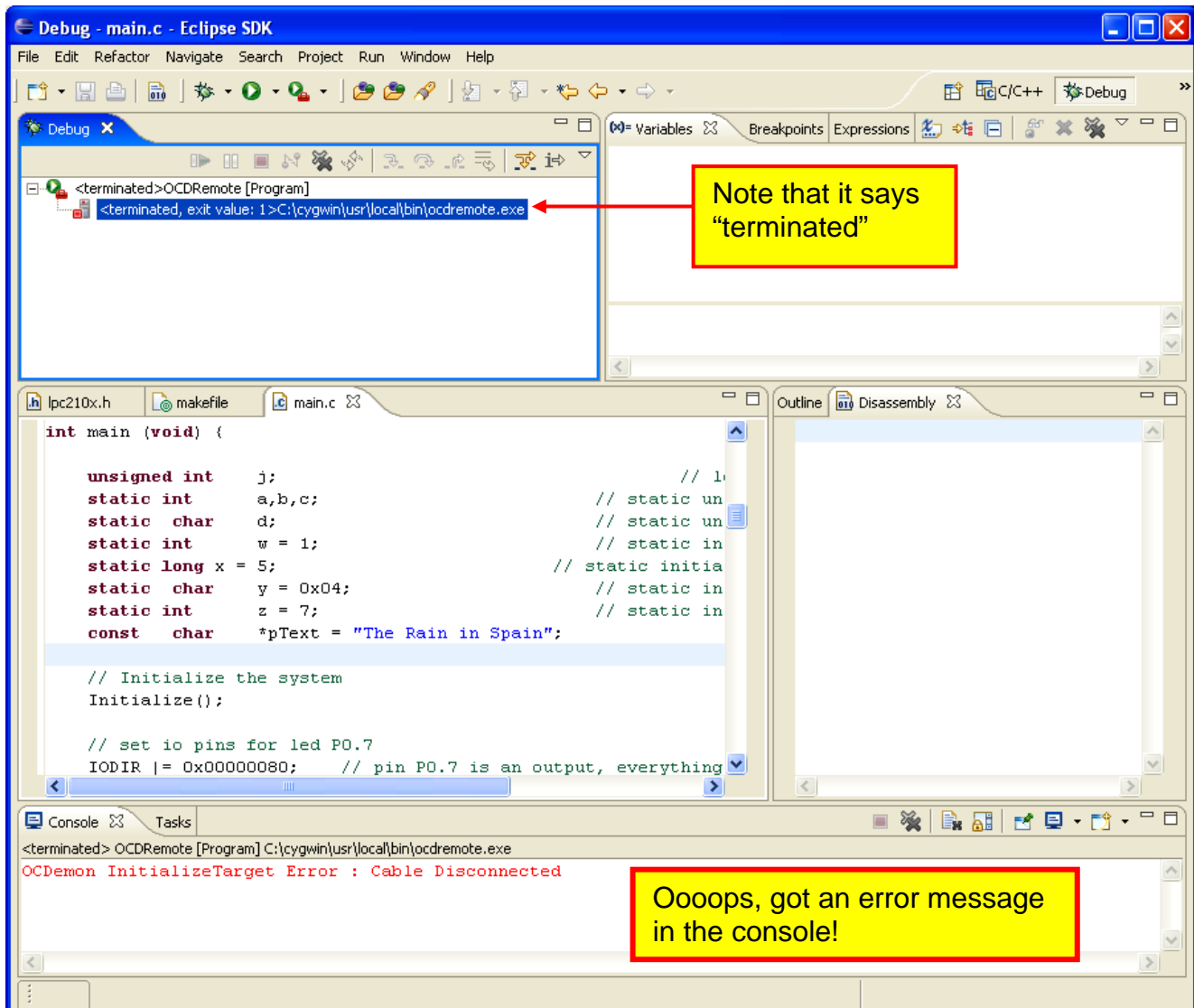
The well-known problem of the Wiggler/**OCDRemote** combination is that it doesn't always start. Below is an example of where it does start properly.



When everything works, the GDB Debugger communicates using the GDB Serial Protocol to an internet socket called localhost:8888, we specified this in our Embedded Debug Launch Configuration (the “command” tab).

The Macraigor **OCDRemote** DLL intercepts the GDB Serial Protocol via the internet socket and converts it into JTAG signals on the LPT1 printer port connector. The Wiggler device merely translates the JTAG signals to 3.3 volts for use with the Philips LPC2106 microprocessor.

Here is an example of **OCDRemote** failing.



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

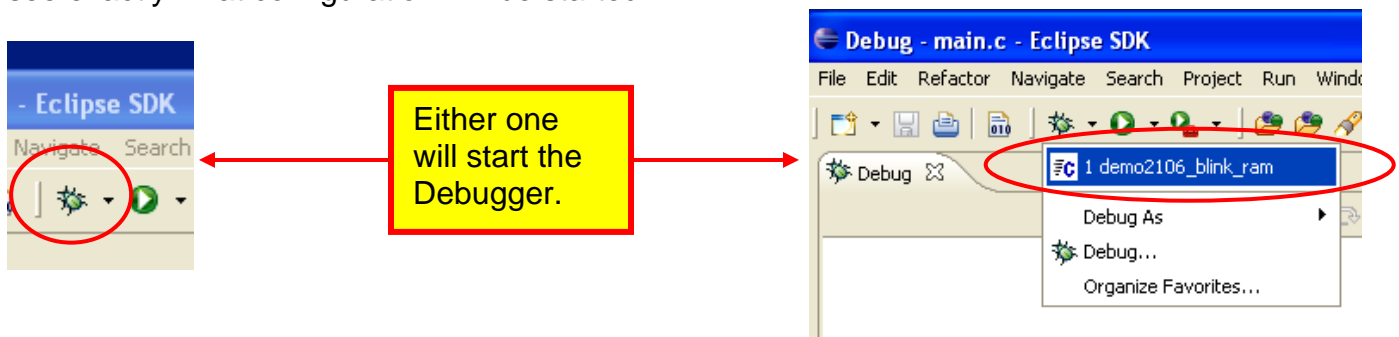
- You may have accidentally started multiple copies of **OCDremote**. Bring up the Windows Task Manager (**ctrl-alt-del**) and search the list of running tasks. If there are multiples of **ocdremote.exe**, terminate all of them and start over.

- Keep trying; I've clicked it ten times before it started (this is simply Voodoo).
- Go to bed; let it win tonight.

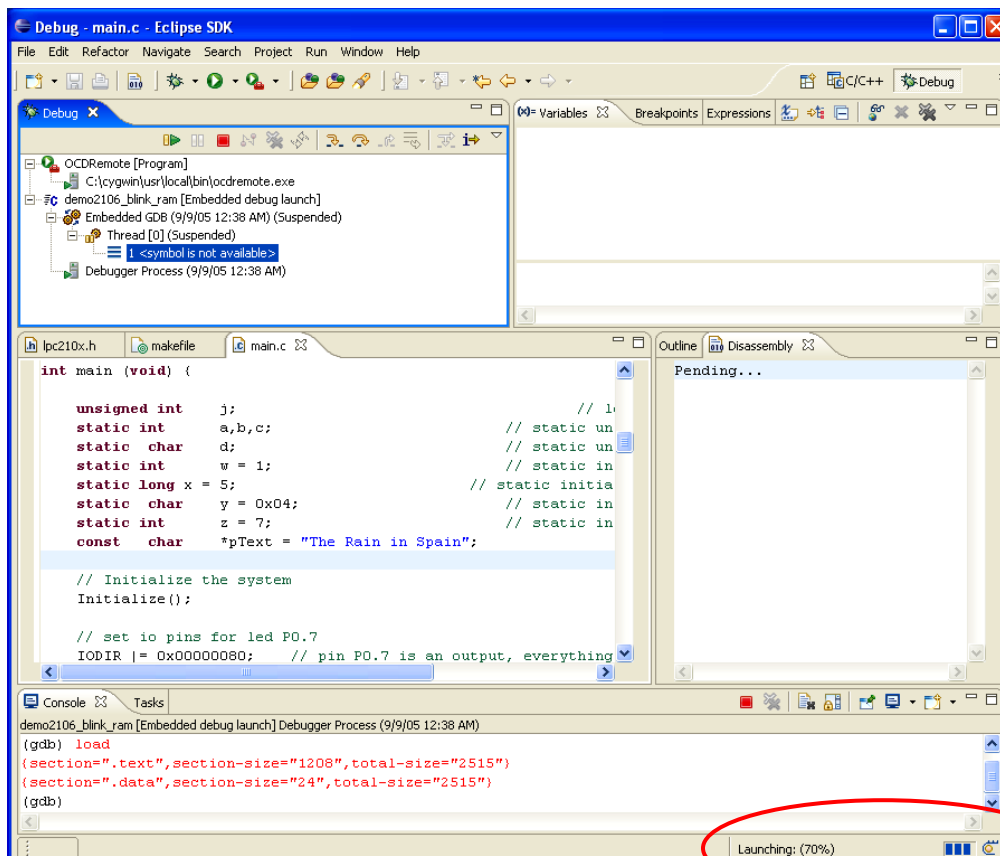
E. Start the Debugger

Our “Debug Configuration” has been defined and we’ve switched to the Debug perspective. We started the **OCDRemote** utility and verified that it’s working.

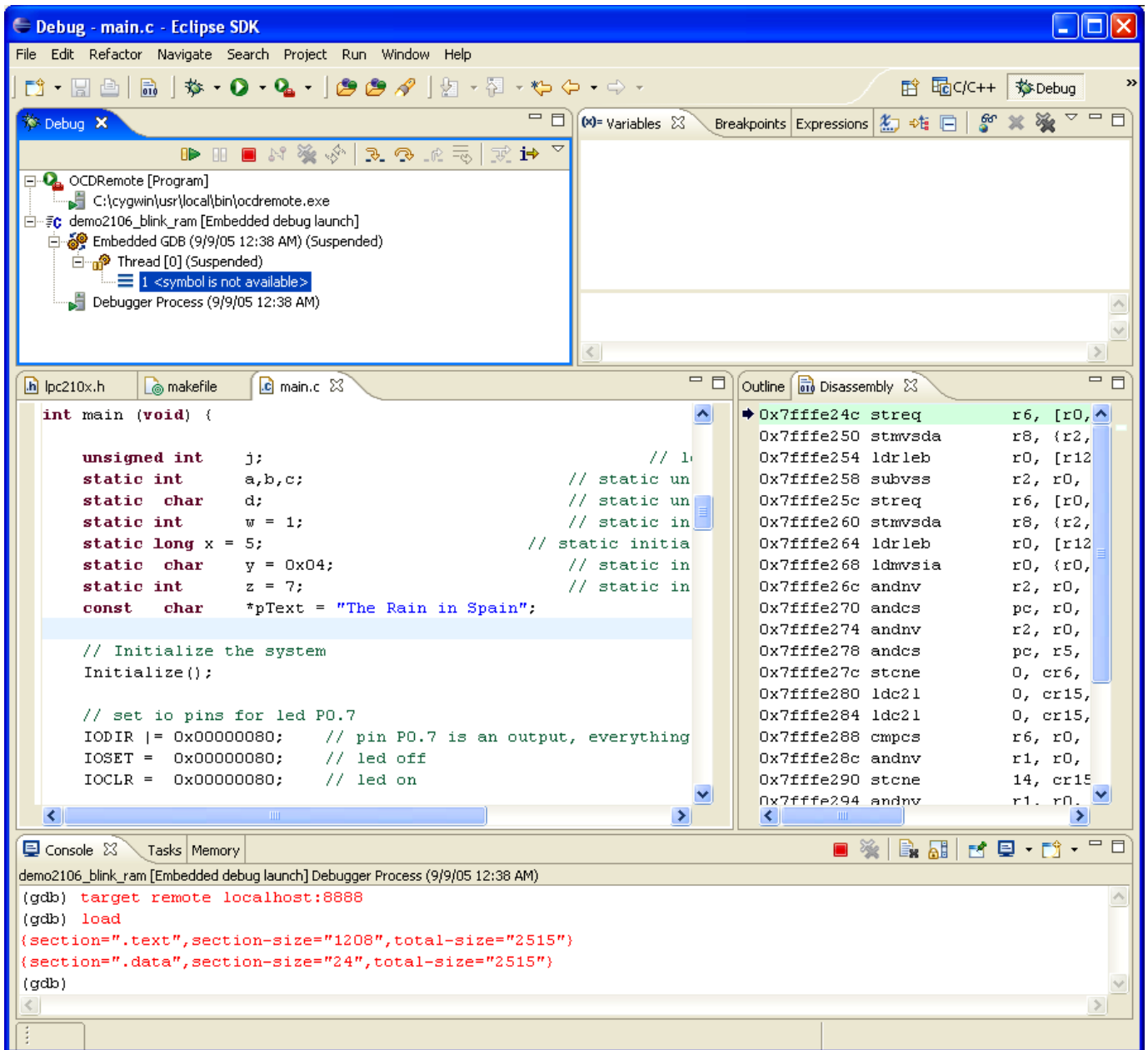
Now is the time to start the debugger. Since the “Embedded Debug Launch” configuration “**demo2106_blink_ram**” was the last configuration accessed above, clicking on the “Bug” button will suffice. If you’re not sure, use the pull-down” arrow to see exactly what configuration will be started.



The Debugger will start up and execute the two commands specified earlier. It will connect to the target via JTAG and start a download of the application. You can watch the progress bars at the lower right of the screen.



When downloading completes, the Debugger is in “idle” mode with the executable code loaded into RAM.



You can see above in the “console” view that the debugger executed our two commands specified in the launch configuration earlier. It followed that with the download of the `.text` and `.data` sections.

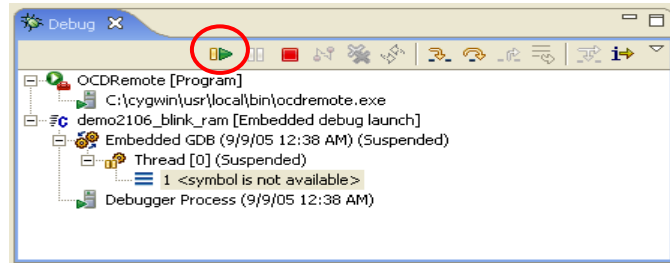
The downloading is very slow. We specified the next to slowest speed 7. Attempting a faster speed with the Olimex wiggler will probably prove futile.

The debugger is “idle”, waiting for you to issue a command.

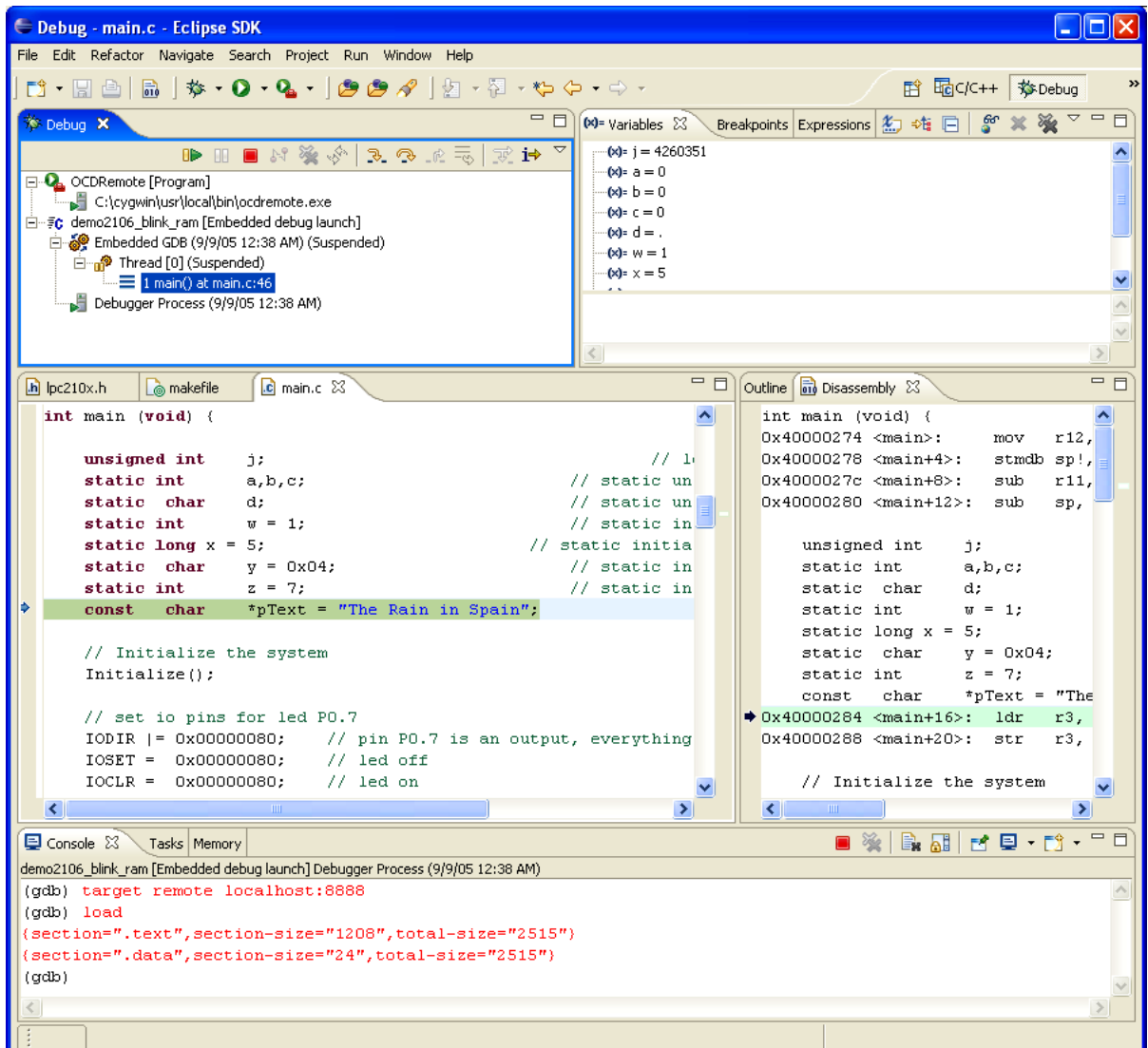
F. Run to Main

The first move is to start the application. It will stop at the `main()` program; we specified this earlier in our launch configuration setup.

In the Debug view, click on the green arrow to start execution of the application..



The application will start, run all of our ARM initialization code and stop at the start of `main()`. Note that in the Debug view, the **Thread[0]** is suspended at line 46 of `main`. With embedded cross development, we only have one execution thread. Code targeted for the PC platform can have multiple threads of execution.



G. Components of the DEBUG Perspective

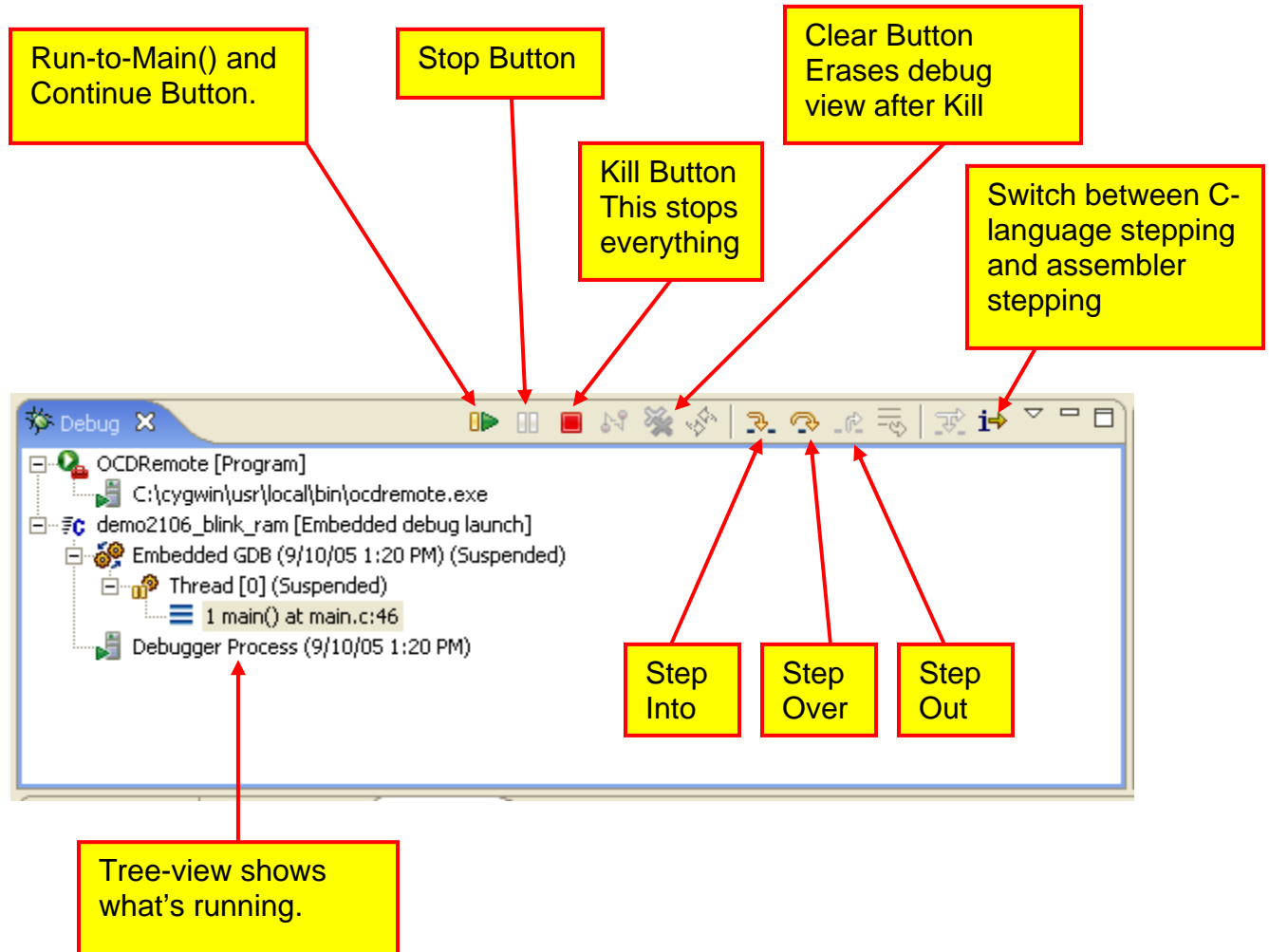
Before operating the Eclipse debugger, let's review the components of the Debug perspective.

The screenshot displays the Eclipse IDE in the Debug perspective, showing the following components:

- Debug Control:** Located in the top-left corner, it shows the current debug session details, including the program being debugged (OCDRemote [Program]), the target (demo2106_blink_ram), and the current thread (Thread [0]).
- Variable display:** Located in the top-right corner, it shows the current values of variables (a, b, c, d, w, x) and registers (j, r11, r12, r2, r3).
- C Code Display:** Located in the bottom-left corner, it shows the source code of the program being debugged, including function declarations, header files, and global variables.
- Assembler Display:** Located in the bottom-right corner, it shows the assembly code corresponding to the source code, including instructions like `ldr r3, [pc, #196]` and `str r3, [r11, #-16]`.
- GDB Debugger Command Window:** Located at the bottom, it shows the output of the GDB debugger, including the current thread (demo2106_blink_ram) and the current command (`(gdb)`).

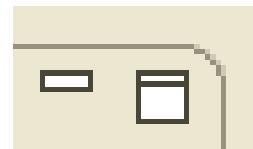
H. Debug Control

The Debug view should be on display at all times. It has the **Run**, **Stop** and **Step** buttons. The tree-structured display shows what is running; in this case it's the **OCDRemote** utility and our application, shown as **Thread[0]**.



Notes: When you resume execution by clicking on the **Run/Continue** button, many of the buttons are “grayed out.” Click on “**Thread[0]**” to highlight it and the buttons will re-appear. This is due to the possibility of multiple threads running simultaneously and you must choose which thread to pause or step. In our ARM development system, we only have one thread.

All of these views, such as the Debug Control view above, can be maximized to full-screen, minimized or returned to multi-pane by the “maximize” and “minimize” buttons at the upper right corner.

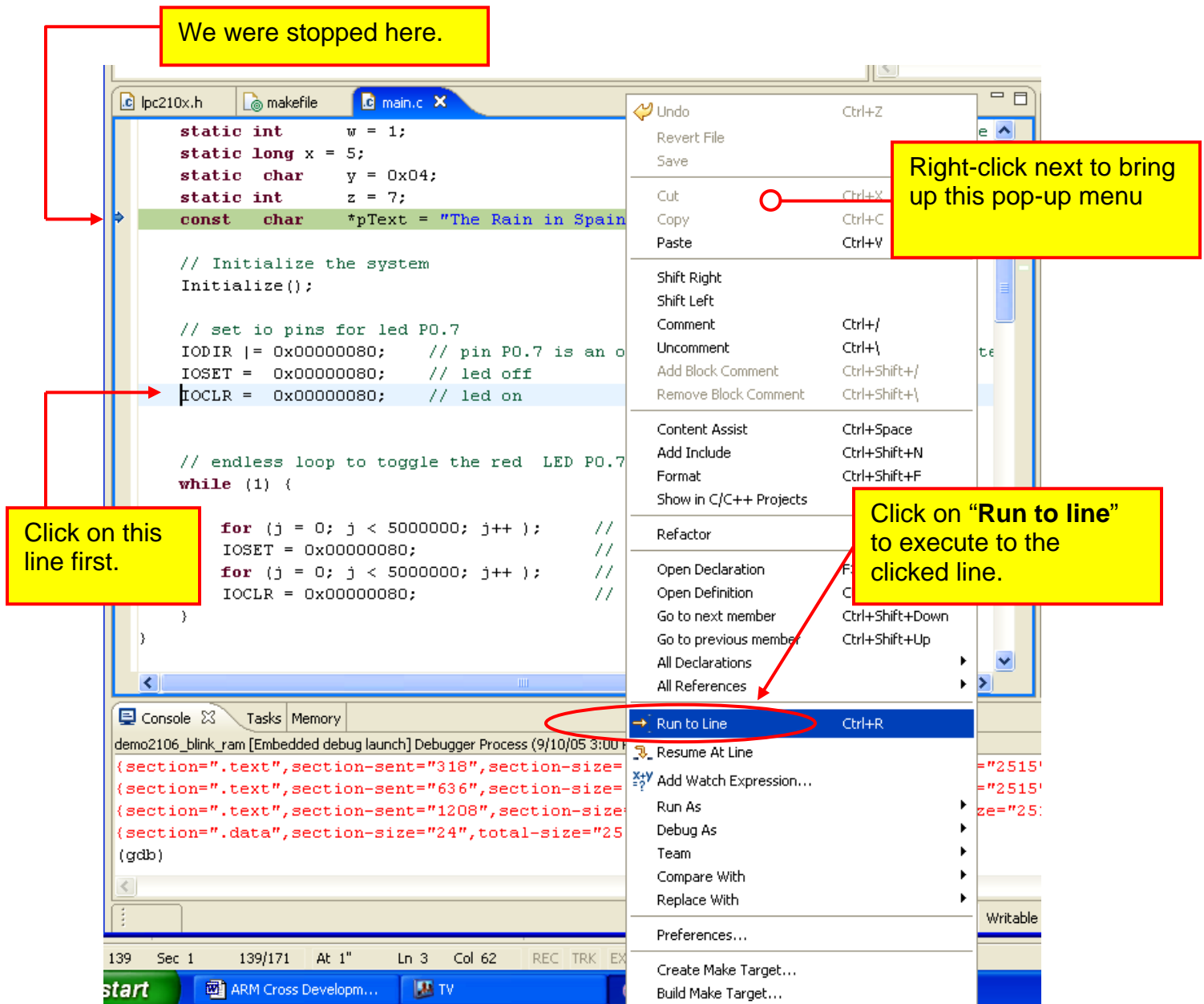


I. Run and Stop with the Right-Click Menu

The easiest method of running is to employ the right-click menu. In the example below, the blue arrowhead cursor indicates where the program is currently stopped.

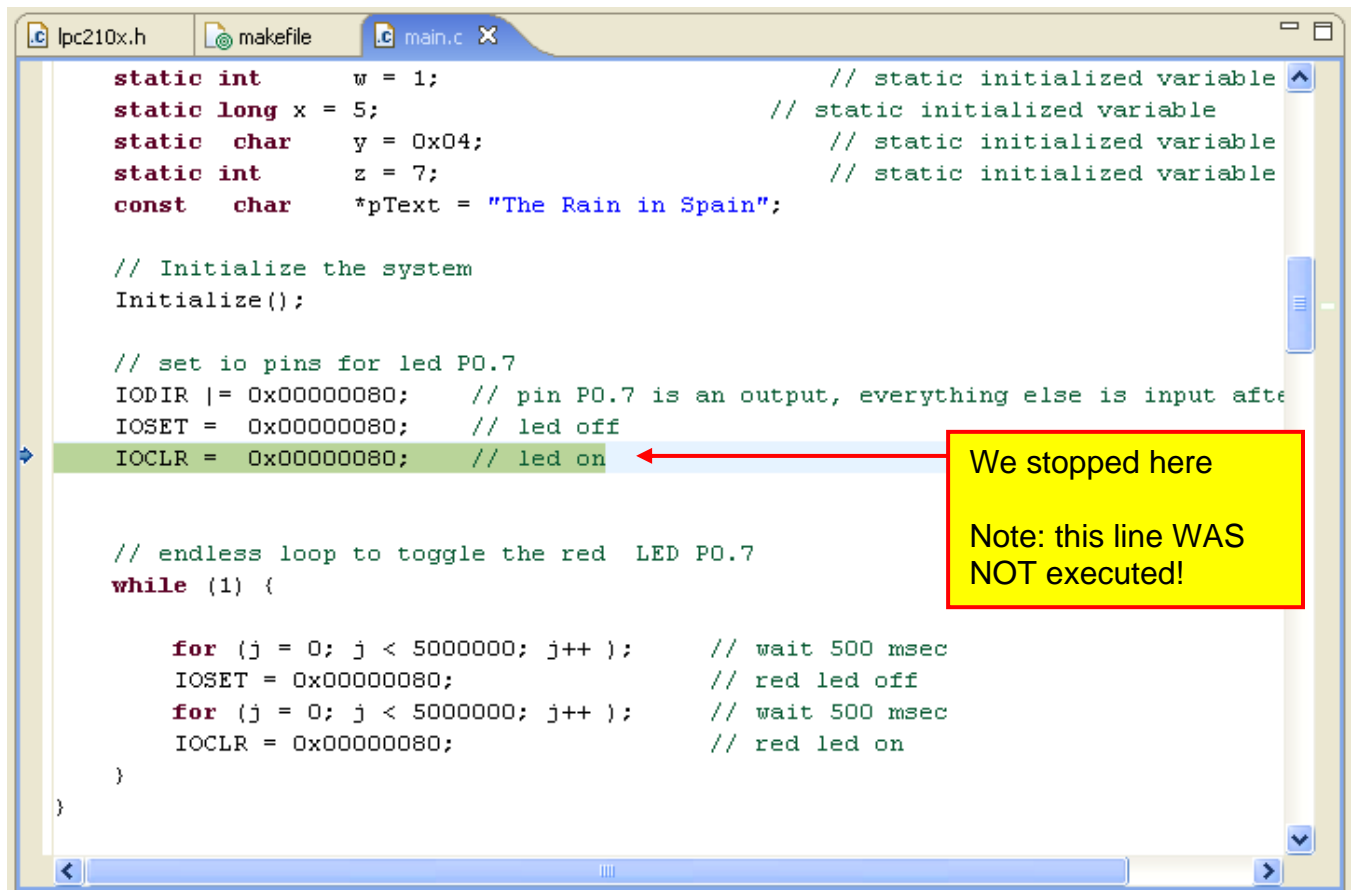
To go to the `IOCLR = 0x00000080;` statement several lines away, click on the line where you want to go (this should highlight the line and place the cursor there).

Now **right click** on that line. Notice that the rather large pop-up menu has a “**Run to Line**” option.



When you click on the “**Run to line**” choice, the program will execute to the line the cursor resides on and then stop (N.B. it will not execute the line).

You can right-click the “Resume at Line” choice to continue execution from that point. If there are no other breakpoints set, then the Blink application will start blinking continuously.



```
lpc210x.h  makefile  main.c X
static int    w = 1;           // static initialized variable
static long  x = 5;           // static initialized variable
static char   y = 0x04;       // static initialized variable
static int    z = 7;         // static initialized variable
const char   *pText = "The Rain in Spain";

// Initialize the system
Initialize();

// set io pins for led PO.7
IODIR |= 0x00000080; // pin PO.7 is an output, everything else is input after
IOSET = 0x00000080; // led off
IOCLR = 0x00000080; // led on

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

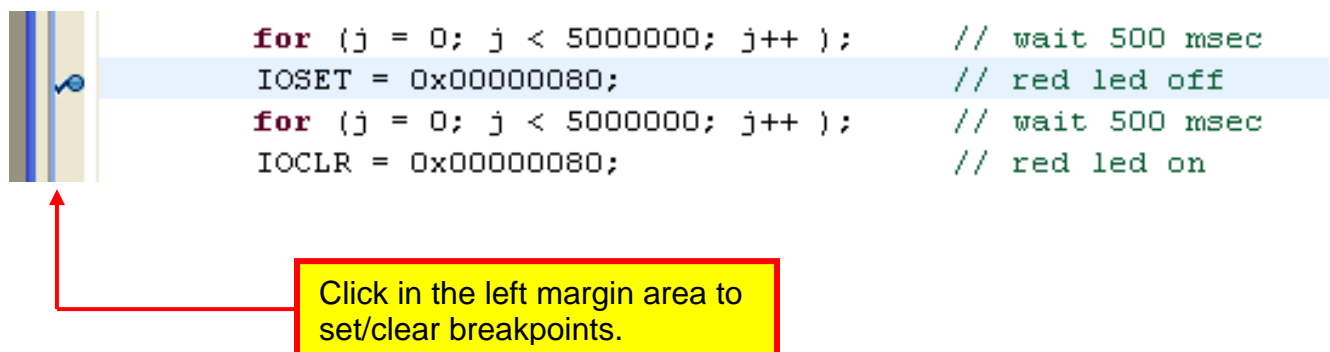
    for (j = 0; j < 5000000; j++ ); // wait 500 msec
    IOSET = 0x00000080; // red led off
    for (j = 0; j < 5000000; j++ ); // wait 500 msec
    IOCLR = 0x00000080; // red led on
}
}
```

We stopped here

Note: this line WAS NOT executed!

J. Setting a Breakpoint

Setting a breakpoint is very simple; just double-click on the far left edge of the line. Double-clicking on the same spot will remove it.



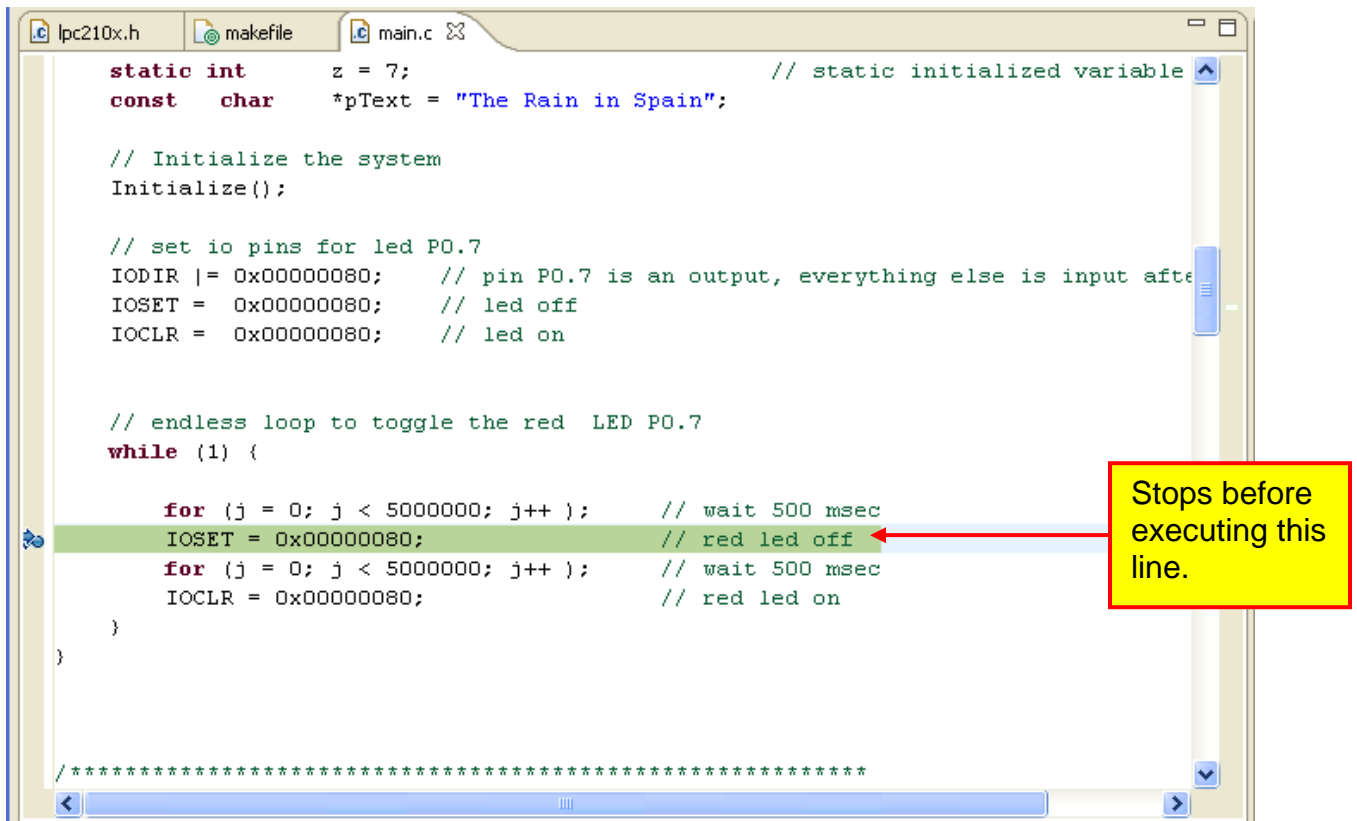
```
for (j = 0; j < 5000000; j++ ); // wait 500 msec
IOSET = 0x00000080; // red led off
for (j = 0; j < 5000000; j++ ); // wait 500 msec
IOCLR = 0x00000080; // red led on
```

Click in the left margin area to set/clear breakpoints.

Now click on the “Run/Continue” button in the Debug view.

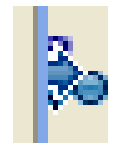


Assuming that this is the only breakpoint set, the program will execute to the breakpoint line and stop.

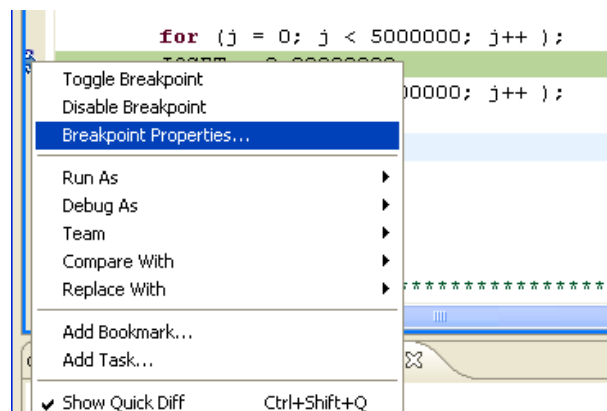


Since this is a RAM application and breakpoints are “software” breakpoints, there can be a nearly unlimited number of breakpoints set.

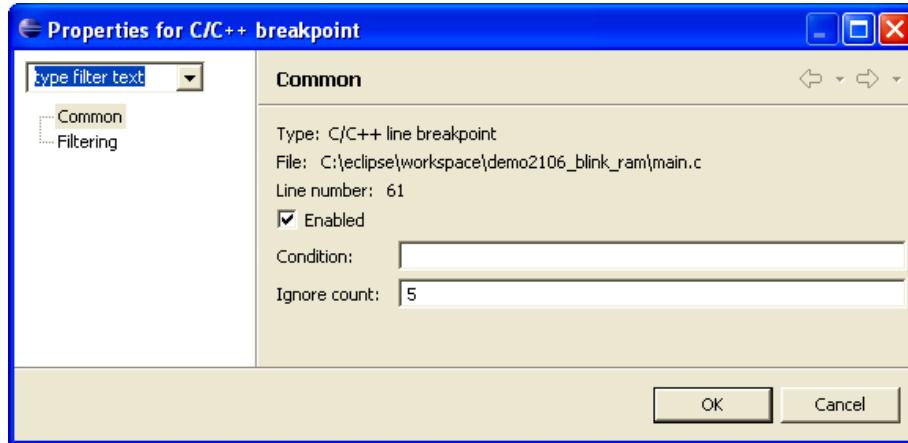
The breakpoints can be more complex. For example, to ignore the breakpoint 5 times and then stop, right-click on the breakpoint symbol on the far left.



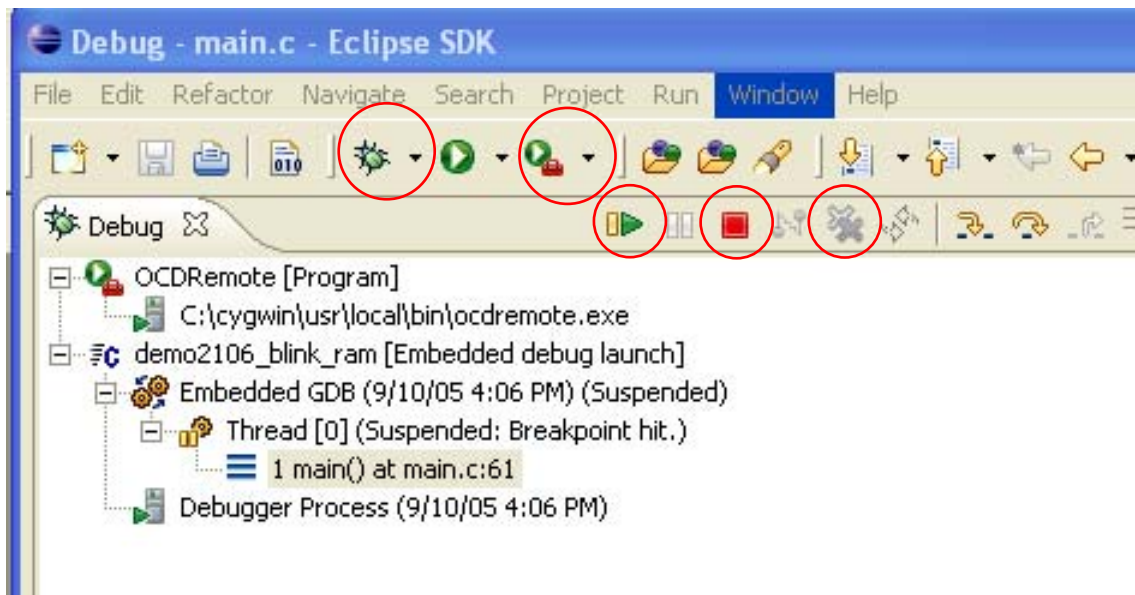
This brings up the pop-up menu below and click on “Breakpoint Properties ...”.



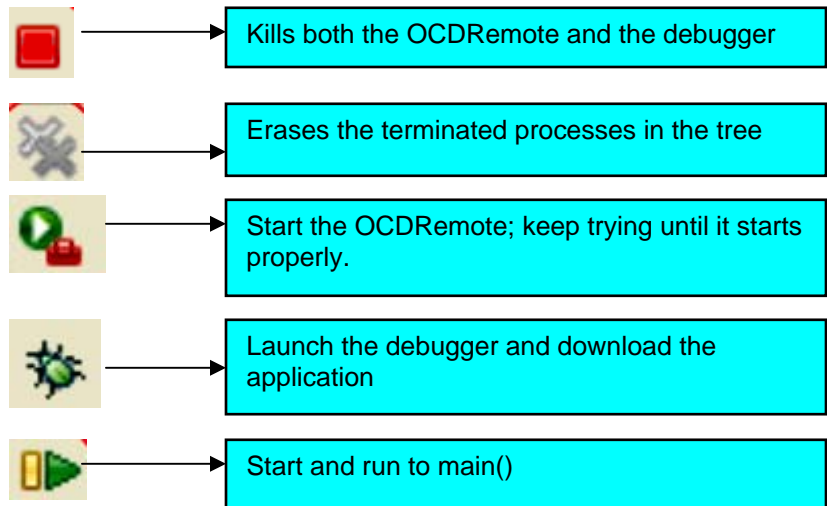
In the “**Properties for C/C++ breakpoint**” window, set the **Ignore Count** to 5. This means that the debugger will ignore the first five times it encounters the breakpoint and then stop.



To test this setup, we must terminate and re-launch the debugger.



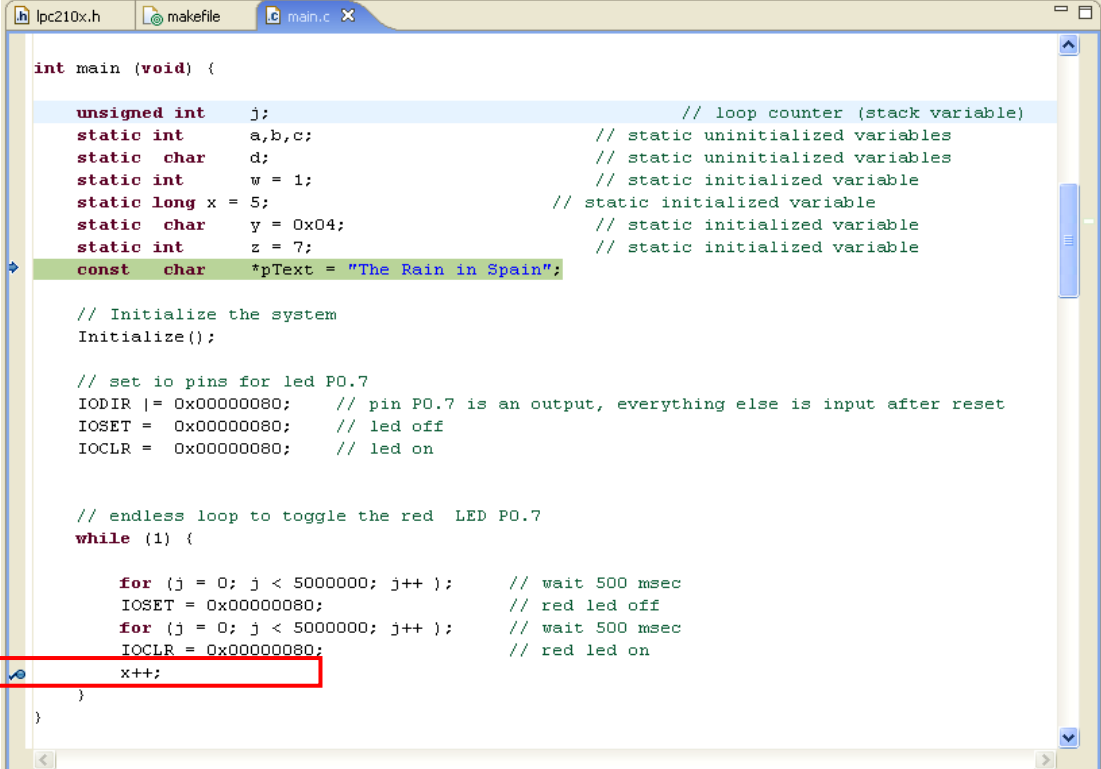
Get used to this sequence:



Now when you hit the **Run/Continue** button again, the program will blink 5 times and stop. Don't expect this feature to run in real-time. Each time the breakpoint is encountered the debugger will automatically continue until the "ignore" count is reached. This involves quite a bit of debugger communication at a very slow baud rate.

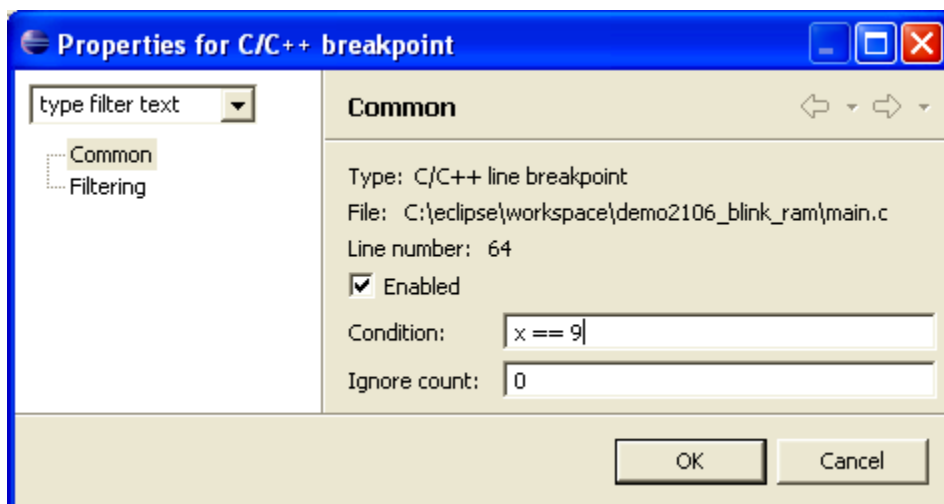
In addition to specifying a "ignore" count, the breakpoint can be made **conditional** on an expression. The general idea is that you set a breakpoint and then specify a conditional expression that must be met before the debugger will stop on the specified source line.

In this example, a line has been added to the blink loop that increments a variable "x". Double-click on that line to set a breakpoint.



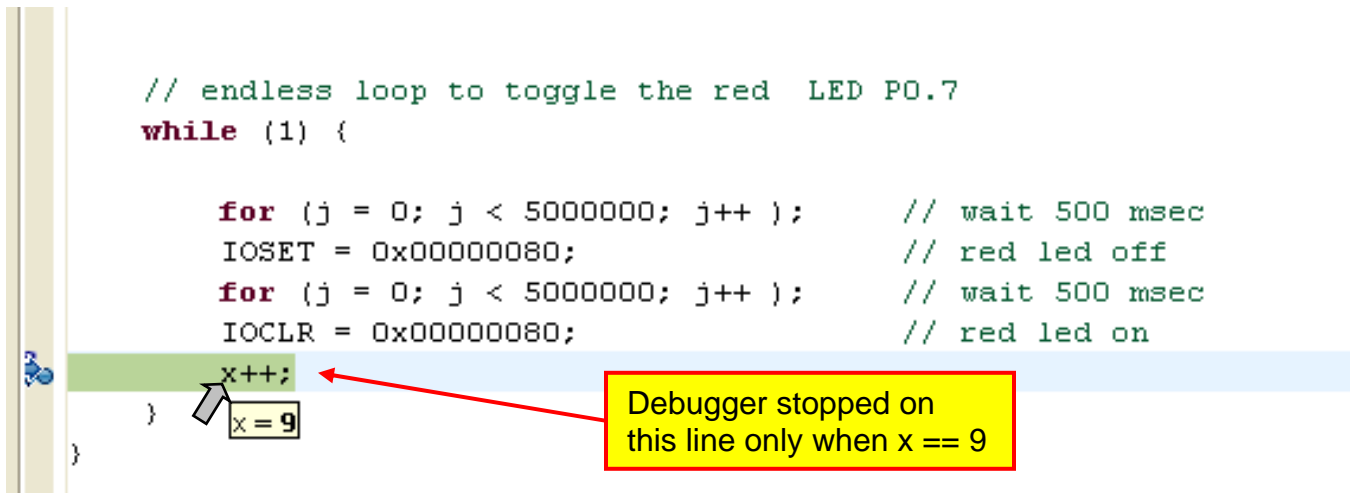
```
int main (void) {  
    unsigned int    j;                // loop counter (stack variable)  
    static int      a,b,c;           // static uninitialized variables  
    static char     d;               // static uninitialized variables  
    static int      w = 1;           // static initialized variable  
    static long     x = 5;           // static initialized variable  
    static char     y = 0x04;        // static initialized variable  
    static int      z = 7;           // static initialized variable  
    const char     *pText = "The Rain in Spain";  
  
    // Initialize the system  
    Initialize();  
  
    // set io pins for led P0.7  
    IODIR |= 0x00000080; // pin P0.7 is an output, everything else is input after reset  
    IOSET = 0x00000080; // led off  
    IOCLR = 0x00000080; // led on  
  
    // 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  
        x++;  
    }  
}
```

Right click on the breakpoint symbol and select "**Breakpoint Properties**". In the Breakpoint Properties window, set the condition text box to "**x == 9**".



A nice feature of Eclipse debugging is that you can edit the source file within the debugger and rebuild the application without leaving the debugger. Of course, you need to **kill the OCDRemote and the Debugger and restart the download** after the build; as specified above. This is necessary for this release of CDT because the “Restart” button appears inoperative. The advantage is that you don’t have to change the Eclipse perspective – just stay in the Debug perspective.

Start the application and it will stop on the breakpoint line (this will take a long time, 9 seconds on my Dell computer). If you park the cursor over the variable x after the program has suspended on the breakpoint, it will display that the current value is 9.



If you specify that it should break when `x == 50000`, you will essentially wait forever. The way this works, the debugger breaks on the selected source line every pass through that source line and then queries via JTAG for the current value of the variable `x`. When `x==50000`, the debugger will stop. Obviously, that requires a lot of serial communication at a very slow baud rate. Sadly, the OCDRemote and wiggler combination cannot service a hardware breakpoint (the ARM supports two hardware breakpoints that can stop on an address and/or data value). Still, you may find some use for this feature.

In the Breakpoint Summary view, shown directly below, you can see all the breakpoints you have created and the right-click menu lets you change the properties, remove or disable any of the breakpoints, etc. The example below shows one conditional breakpoint that will stop on source line 64 only if the variable `x` is equal to 9.



K. Single Stepping

Single-stepping is the single most useful feature in any debugging environment. The debug view has three buttons to support this.



Step Into



If the cursor is at a function call, this will step **into** the function. It will stop at the first instruction inside the function.

If cursor is on any other line, this will execute one instruction.

Step Over



If the cursor is at a function call, this will step **over** the function. It will execute the entire function and stop on the next instruction after the function call.

If cursor is on any other line, this will execute one instruction

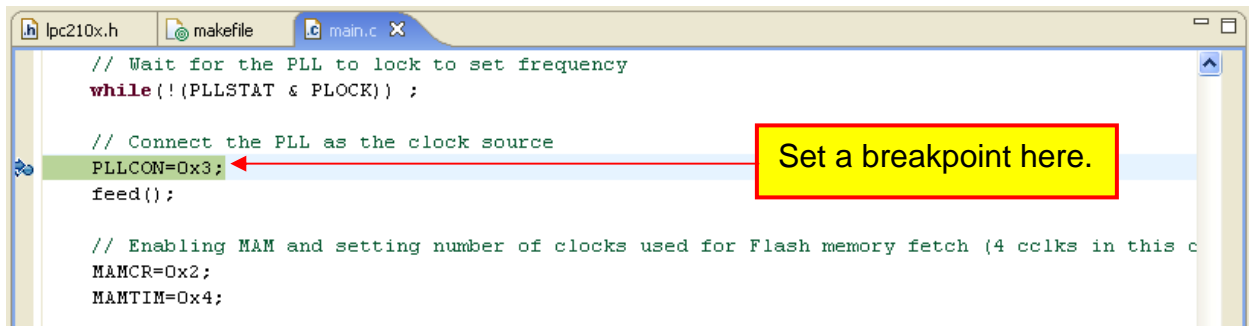
Step Out Of



If the cursor is within a function, this will execute the remaining instructions in the function and stop on the next instruction after the function call.

This button will be “grayed-out” if cursor is not within a function.

As a simple example, restart the debugger and set a breakpoint on a line in the **Initialize()** function. Hit the **Start** button to go to that breakpoint.



```
lpc210x.h | makefile | main.c x
// Wait for the PLL to lock to set frequency
while (!(PLLSTAT & PLOCK)) ;

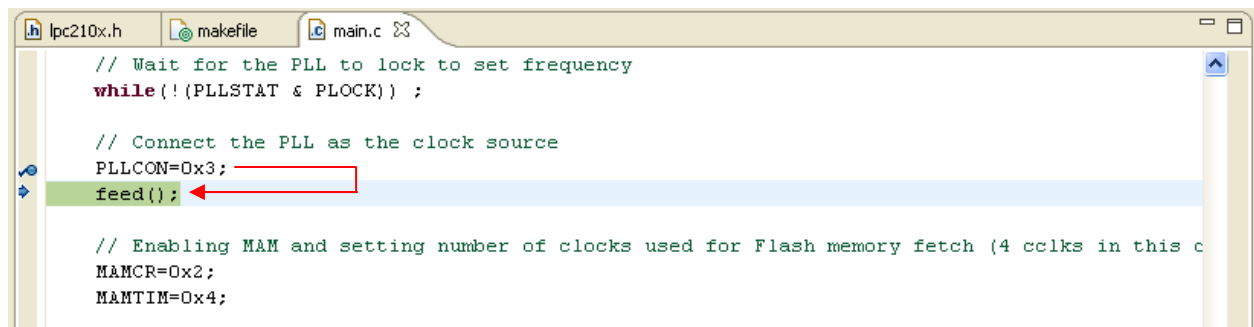
// Connect the PLL as the clock source
PLLCON=0x3;
feed();

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

Click the “**Step Over**” button



The debugger will execute one instruction.



```
lpc210x.h | makefile | main.c x
// Wait for the PLL to lock to set frequency
while (!(PLLSTAT & PLOCK)) ;

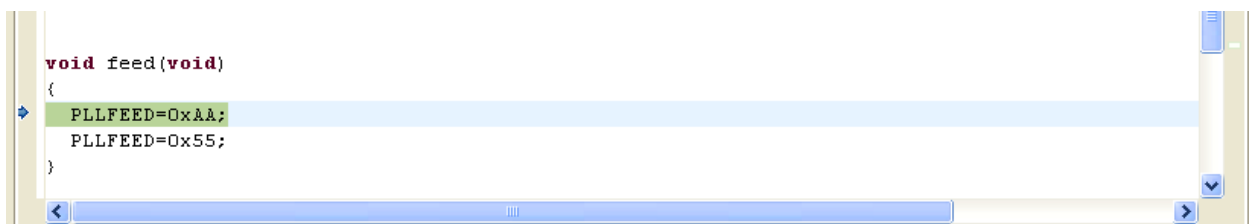
// Connect the PLL as the clock source
PLLCON=0x3;
feed();

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

Click the “**Step Into**” button

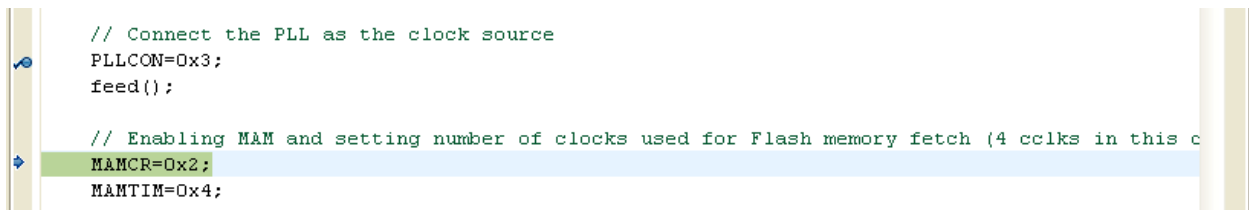


The debugger will enter the feed() function.



```
void feed(void)
{
PLLFEED=0xAA;
PLLFEED=0x55;
}
```

Notice that the “**Step Out Of**” button is illuminated. Click the “**Step Out Of**” button. The debugger will execute the remaining instructions in feed() and return to just after the function call.



```
// Connect the PLL as the clock source
PLLCON=0x3;
feed();

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

L. Inspecting and Modifying Variables

Before proceeding on this topic, let's add a couple of structured variables to the simple blinker test program. After rebuilding the application and re-launching the debugger, we can inspect variables once a breakpoint has been encountered.

```
/* *****
   Function declarations
   ***** */

void Initialize(void);
void feed(void);

void IRQ_Routine (void)  __attribute__ ((interrupt("IRQ")));
void FIQ_Routine (void)  __attribute__ ((interrupt("FIQ")));
void SWI_Routine (void)  __attribute__ ((interrupt("SWI")));
void UNDEF_Routine (void) __attribute__ ((interrupt("UNDEF")));

/*****
   Header files
   *****/
#include "LPC210x.h"

/*****
   Global Variables
   *****/
int    q;           // global uninitialized variable
int    r;           // global uninitialized variable
int    s;           // global uninitialized variable

short  h = 2;       // global initialized variable
short  i = 3;       // global initialized variable
char   j = 6;       // global initialized variable

struct comms {
    int    nbytes;
    char*  pBuf;
    char   buffer[32];
} channel = {5, &channel.buffer[0], {"Faster than a speeding bullet"}};

/*****
   MAIN
   *****/

int main (void) {

    unsigned int    j;           // loop counter (stack variable)
    static int      a,b,c;       // static uninitialized variables
    static char     d;           // static uninitialized variables
    static int      w = 1;       // static initialized variable
    static long     x = 5;       // static initialized variable
    static char     y = 0x04;    // static initialized variable
    static int      z = 7;       // static initialized variable
    const char     *pText = "The Rain in Spain";

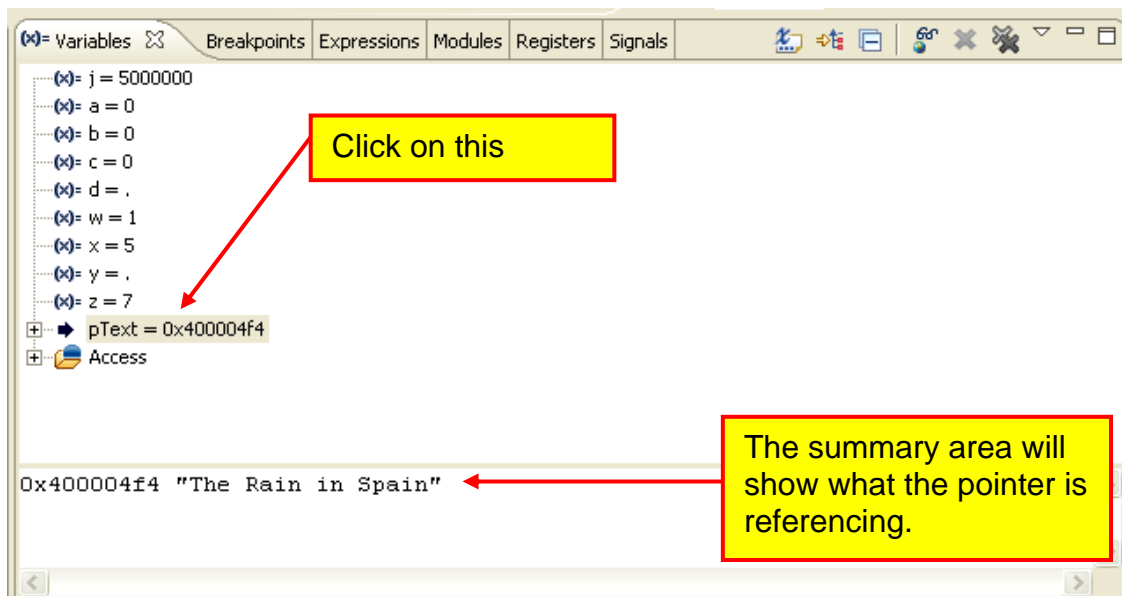
    struct EntryLock {
        long        key;
        int         nAccesses;
        char        name[16];
    } Access = {14705, 0, "Spiderman"};

    // Initialize the system
    Initialize();

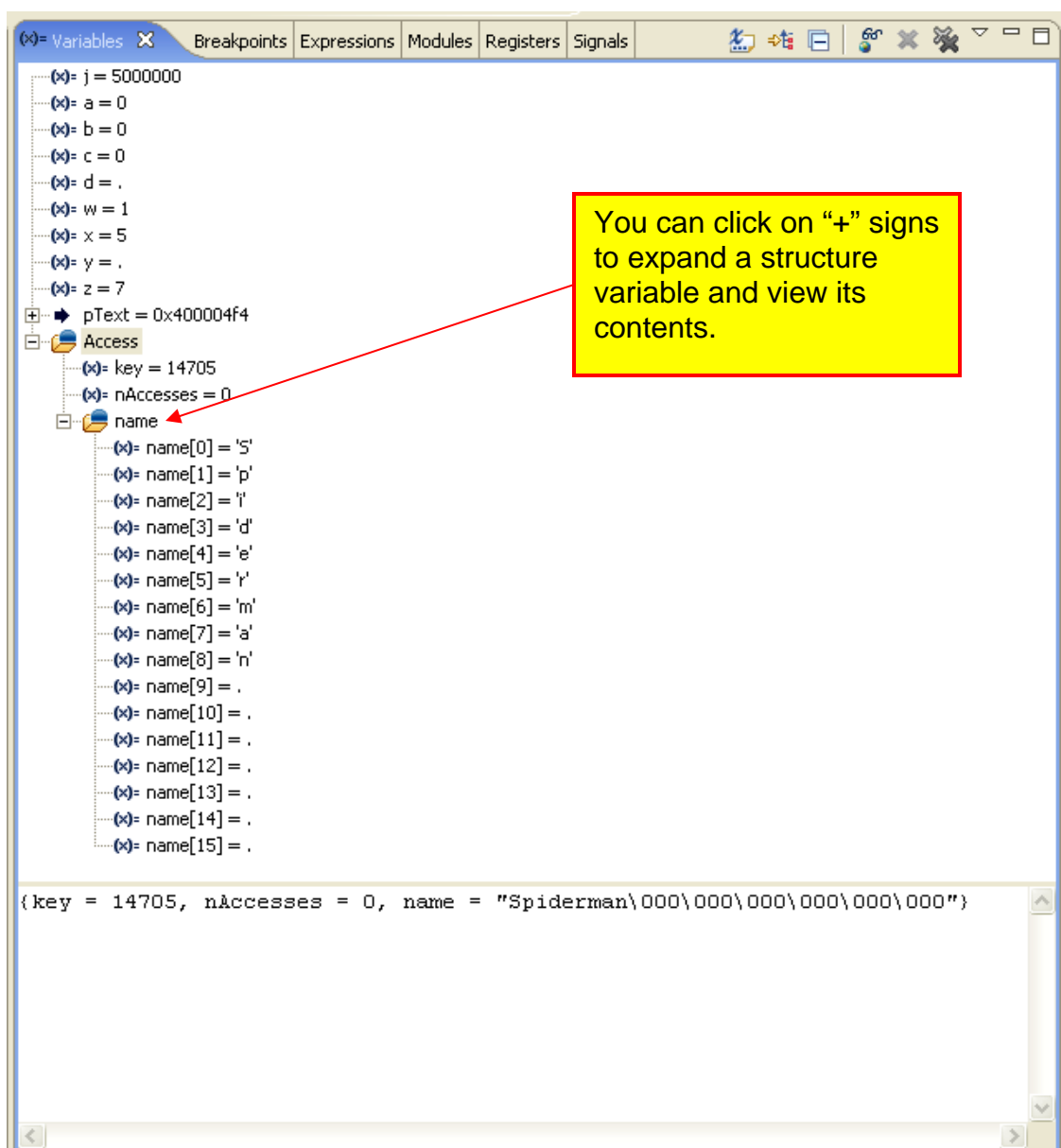
    // set io pins for led PO.7
    IODIR |= 0x00000080; // pin PO.7 is an output, everything else is input after reset
    IOSET = 0x00000080; // led off
    IOCLR = 0x00000080; // led on

    // endless loop to toggle the red LED PO.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
        x = x + 1;
    }
}
```

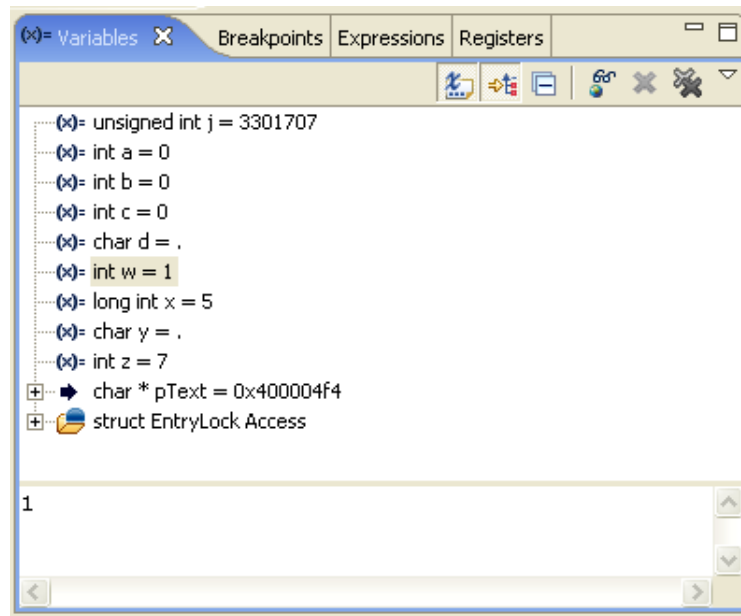
The Variables view can also expand structures. Just click on any “+” signs you see to expand the structure and view its contents.



If you click on the “Show Type Names” button,



each variable name will be



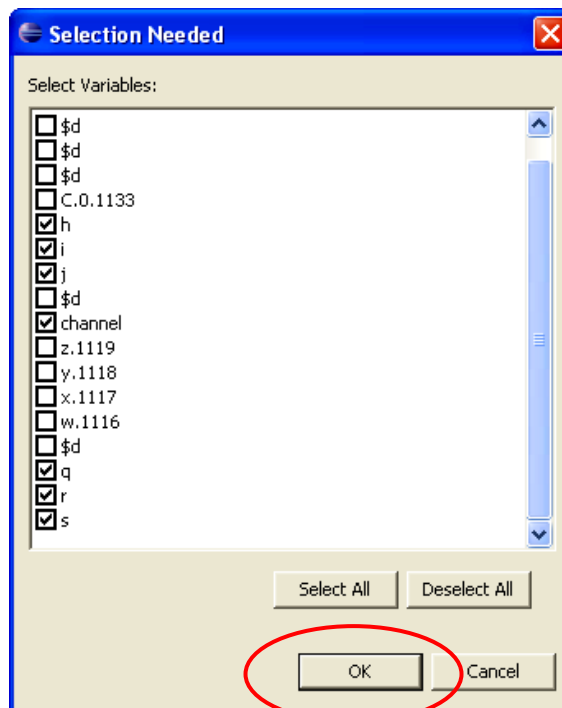
Global variables have to be individually selected for display within the “Variables” view.

Use the “Add Global Variables” button



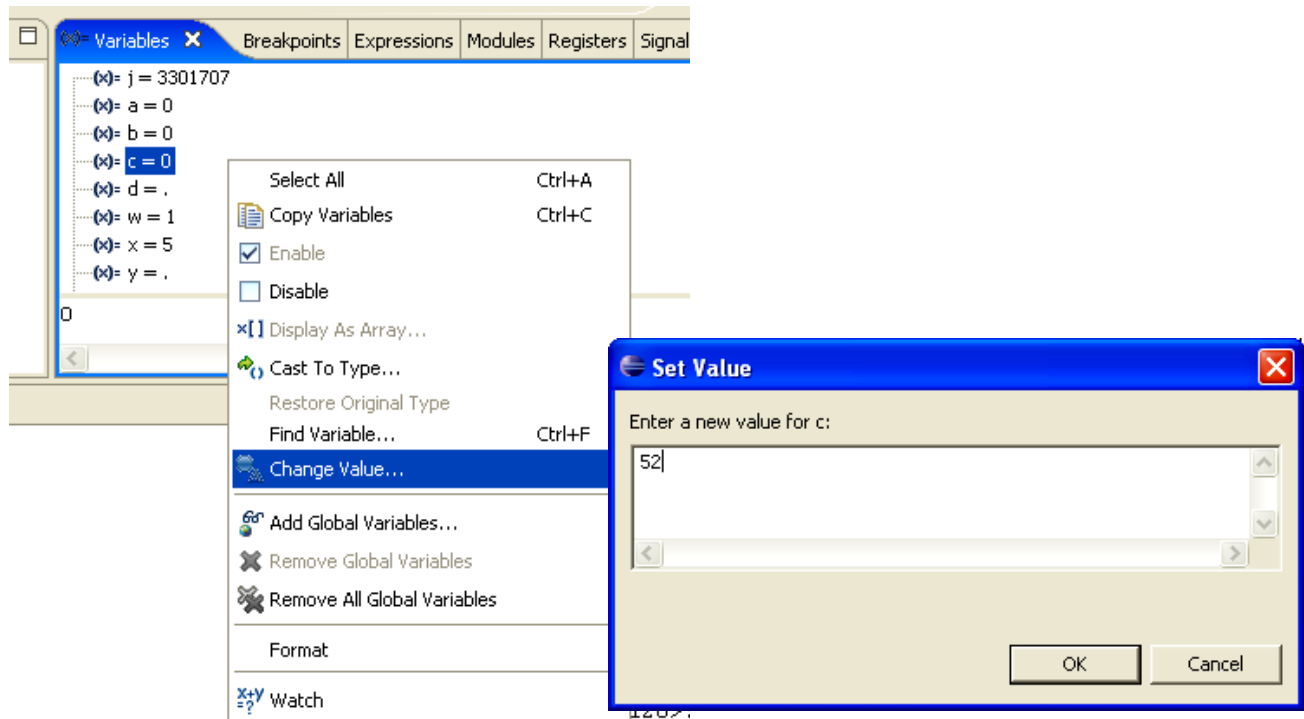
to open the selection dialog.

Check the variables you want to display and then click “OK” add them to the **Variables** view,

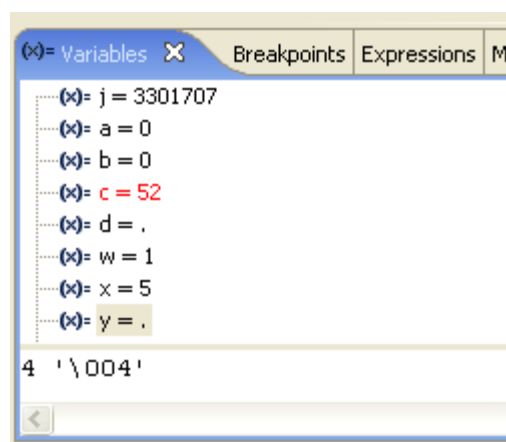


Note: not sure what the extra variables are. Might be a CDT bug?

You can easily change the value of a variable at any time. Assuming that the debugger has stopped, click on the variable you wish to change and right click. In the right-click menu, select “**Change Value...**” and enter the new value into the pop-up window as shown below. In this example, we change the variable “c” to 52.



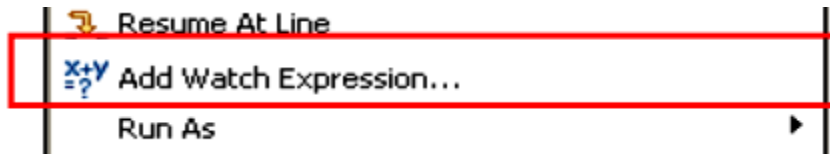
Now the “**Variables**” view should show the new value for the variable “c”. Note that it has been colored **red** to indicate that it has been changed.



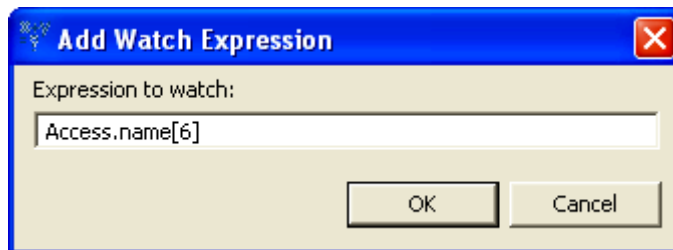
M. Watch Expressions

The “Expressions” view can display the results of expressions (any legal C Language expression). Since it can pick any local or global variable, it forms the basis of a customizable variable display; showing only the information you want.

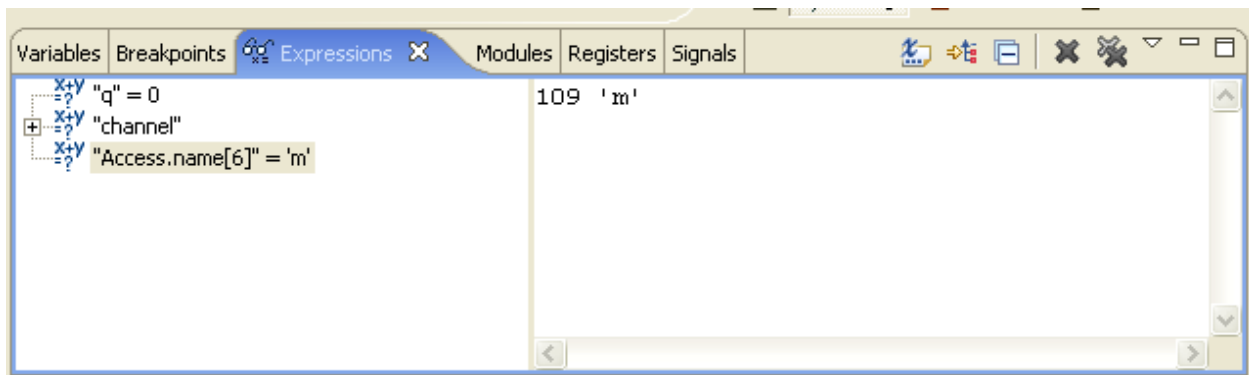
For example, to display the 6th character of the name in the structured variable “**Access**”, bring up the right-click menu and select “**Add Watch Expression...**”.



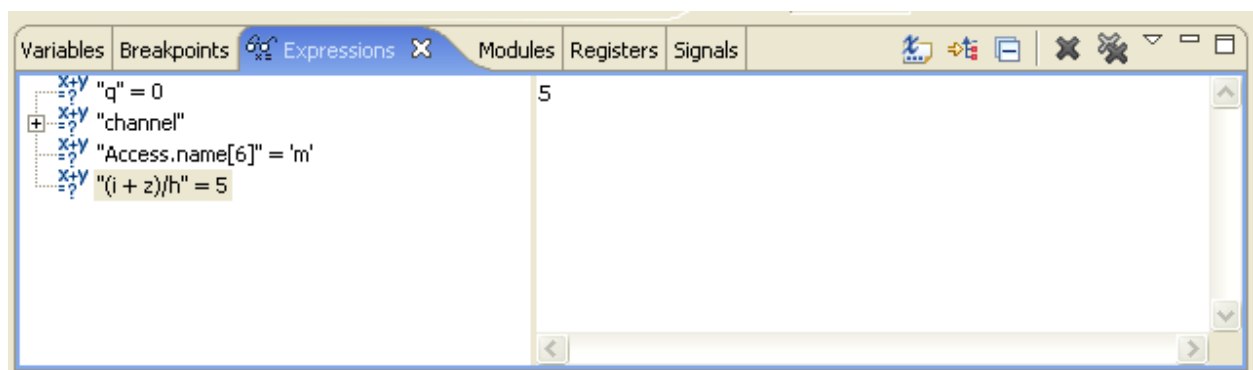
Enter the fully qualified name of the 6th character of the name[] array.



Note that it now appears in the “Expressions” view.



You can type in very complicated expressions. Here we defined the expression $(i + z)/h$



N. Assembly Language Debugging

The Debug perspective includes an Assembly Language view.



If you click on the Instruction Stepping Mode toggle button in the Debug view, the assembly language window becomes active and the single-step buttons apply to the assembler window. The single-step buttons will advance the program by a single assembler instruction. Note that the “Disassembly” tab lights up when the assembler view has control.

Note that the debugger is currently stopped at the assembler line at address 0x400003f0.

```
Outline Disassembly X
0x400003d8 <Initialize+76>: mov r3, r3, lsl #16
0x400003dc <Initialize+80>: mov r3, r3, lsr #16
0x400003e0 <Initialize+84>: mov r3, r3, lsr #10
0x400003e4 <Initialize+88>: and r3, r3, #1 ; 0x1
0x400003e8 <Initialize+92>: cmp r3, #0 ; 0x0
0x400003ec <Initialize+96>: beq 0x400003c8 <Initialize+60>

// Connect the PLL as the clock source
PLLCON=0x3;
0x400003f0 <Initialize+100>: mov r3, #-536870912 ; 0xe0000000
0x400003f4 <Initialize+104>: add r3, r3, #2080768 ; 0x1fc000
0x400003f8 <Initialize+108>: add r3, r3, #128 ; 0x80
0x400003fc <Initialize+112>: mov r2, #3 ; 0x3
0x40000400 <Initialize+116>: strb r2, [r3]
feed();
0x40000404 <Initialize+120>: bl 0x40000454 <feed>
```



If we click the “Step Over” button in the Debug view, the debugger will execute one assembler line.

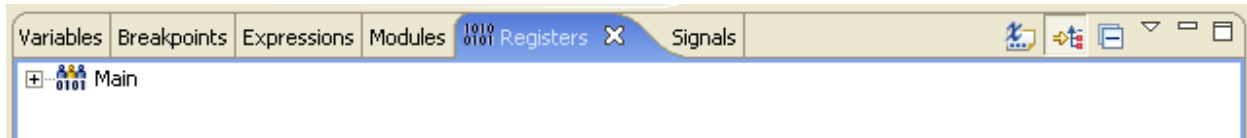
```
Outline Disassembly X
0x400003d8 <Initialize+76>: mov r3, r3, lsl #16
0x400003dc <Initialize+80>: mov r3, r3, lsr #16
0x400003e0 <Initialize+84>: mov r3, r3, lsr #10
0x400003e4 <Initialize+88>: and r3, r3, #1 ; 0x1
0x400003e8 <Initialize+92>: cmp r3, #0 ; 0x0
0x400003ec <Initialize+96>: beq 0x400003c8 <Initialize+60>

// Connect the PLL as the clock source
PLLCON=0x3;
0x400003f0 <Initialize+100>: mov r3, #-536870912 ; 0xe0000000
0x400003f4 <Initialize+104>: add r3, r3, #2080768 ; 0x1fc000
0x400003f8 <Initialize+108>: add r3, r3, #128 ; 0x80
0x400003fc <Initialize+112>: mov r2, #3 ; 0x3
0x40000400 <Initialize+116>: strb r2, [r3]
feed();
0x40000404 <Initialize+120>: bl 0x40000454 <feed>
```

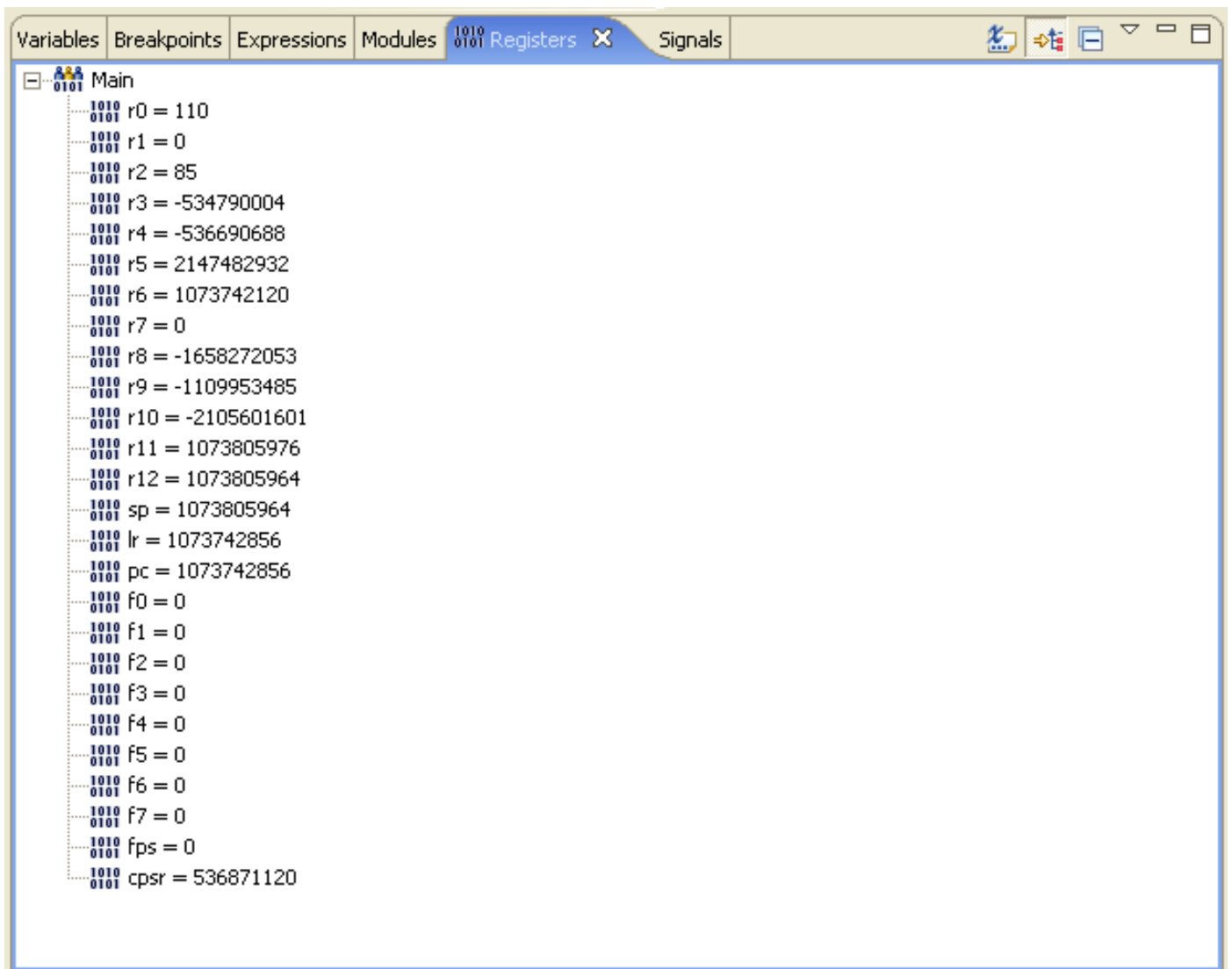
The “Step Into” and “Step Out Of” buttons work in the same way as for C code.

O. Inspecting Registers

Unfortunately, parking the cursor over a register name (R3 e.g.) does not pop up its current value. For that, you can refer to the “Registers” view.

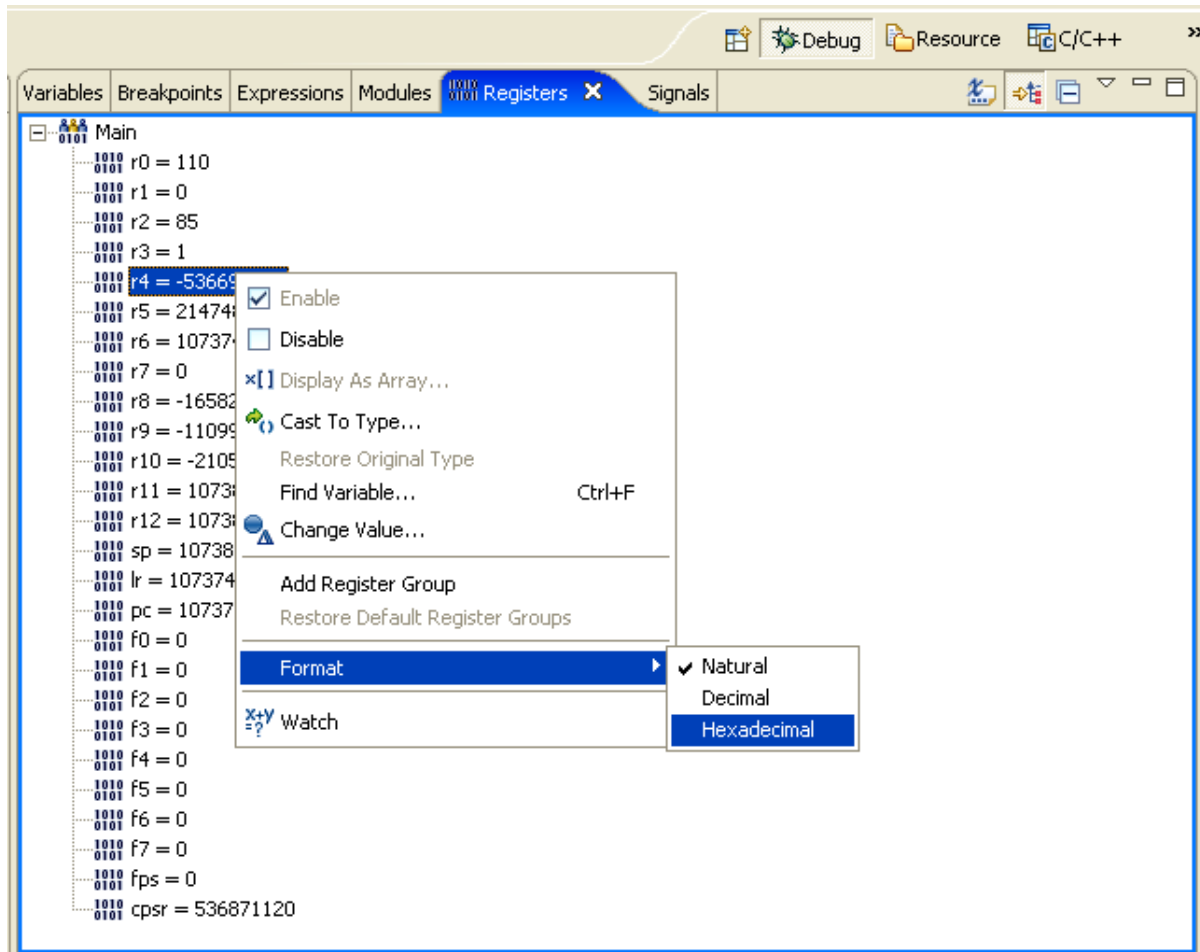


Click on the “+” symbol next to Main and the registers will appear. The Philips LPC2106 doesn’t have any floating point registers so registers F0 through FPS are not applicable.

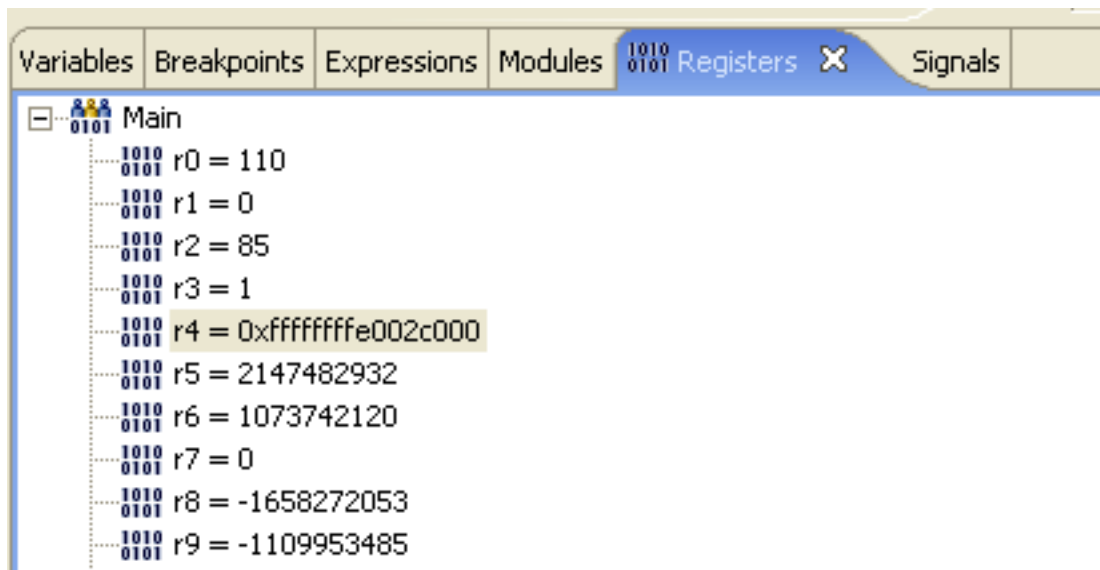


If you don’t like a particular register’s numeric format, you can click to highlight it and then bring up the right-click menu.

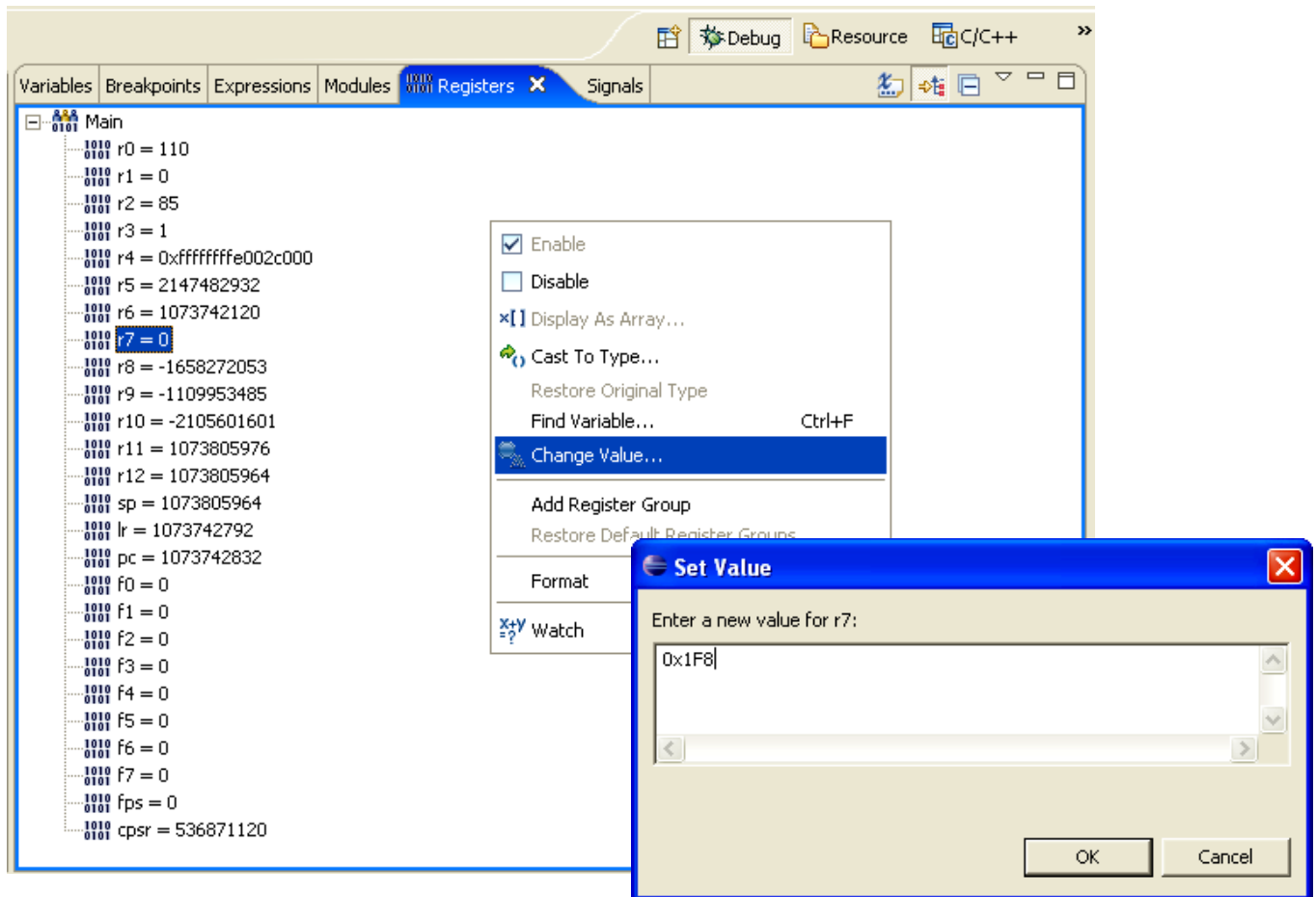
The “**Format**” option permits you to change the numeric format to hexadecimal, for example.



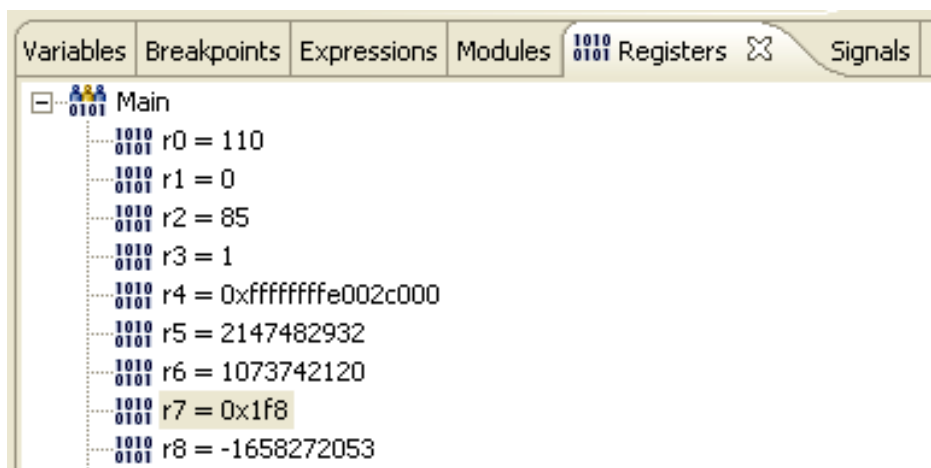
Now the register display shows r4 in hexadecimal format.



Of course, the right click menu lets you change the value of any register. For example, to change **r7** from **zero** to **0x1F8**, just select the register, right-click and select “**Change Value...**”



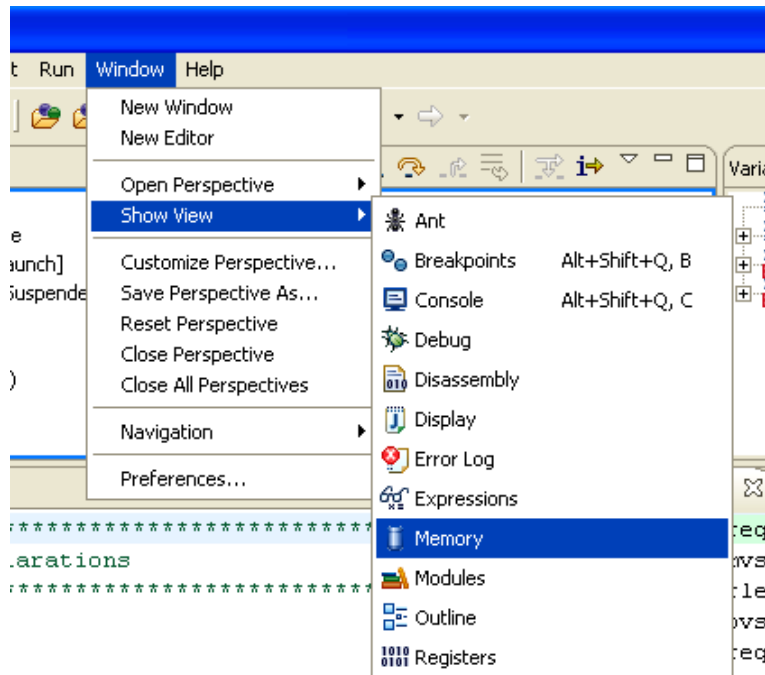
Now the value for **r7** has been changed to **0x1F8**.



It goes without saying that you had better use this feature with great care! Make sure you know what you are doing before tampering with the ARM registers.

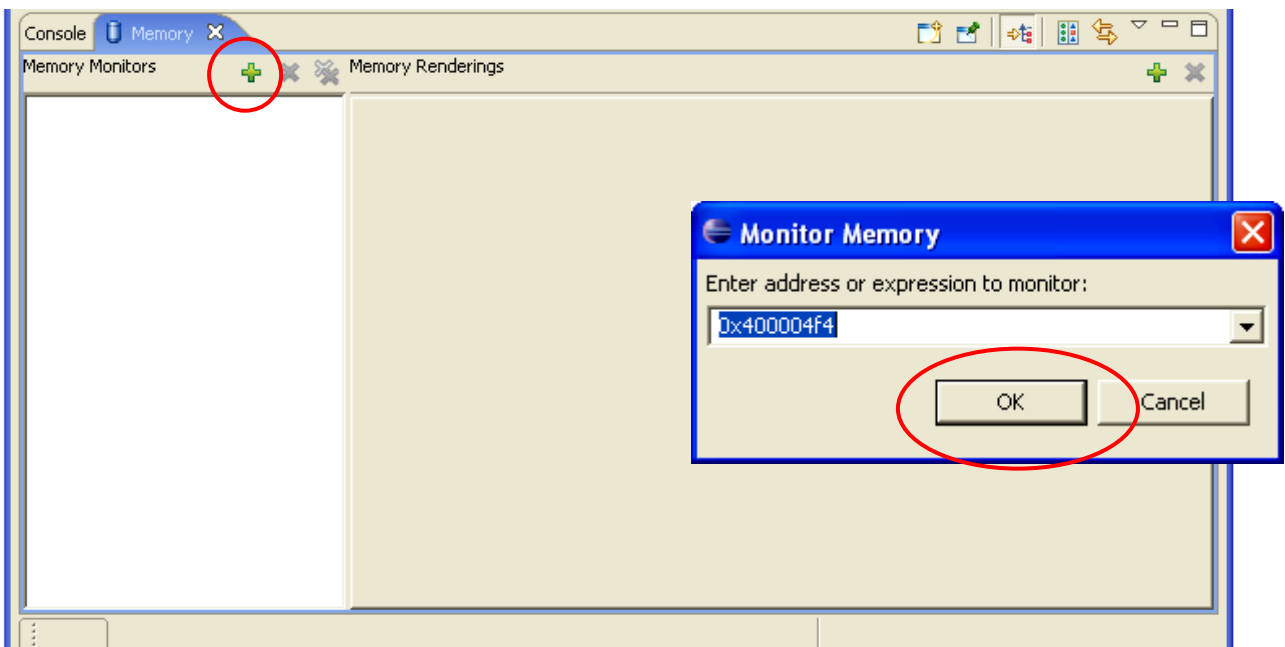
P. Inspecting Memory

Viewing memory is a bit complex in Eclipse. First, the memory view is not part of the default debug launch configuration. You can add it by clicking “**Window – Show View – Memory**” as shown below.



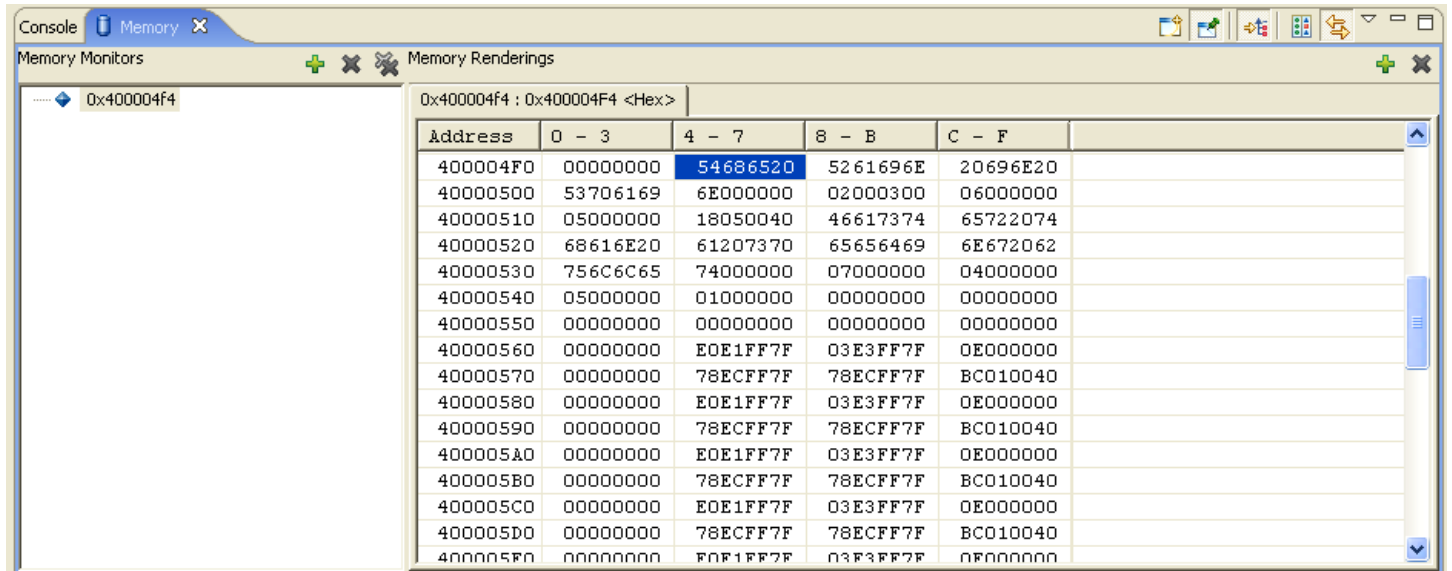
The memory view appears in the “**Console**” view at the bottom of the Debug perspective. At this point, nothing has been defined. Memory is displayed as one or more “**memory monitors**”. To create a memory monitor, click on the “**+**” symbol.

Enter the address **0x400004f4** (address of the string “The Rain in Spain”) in the dialog box.



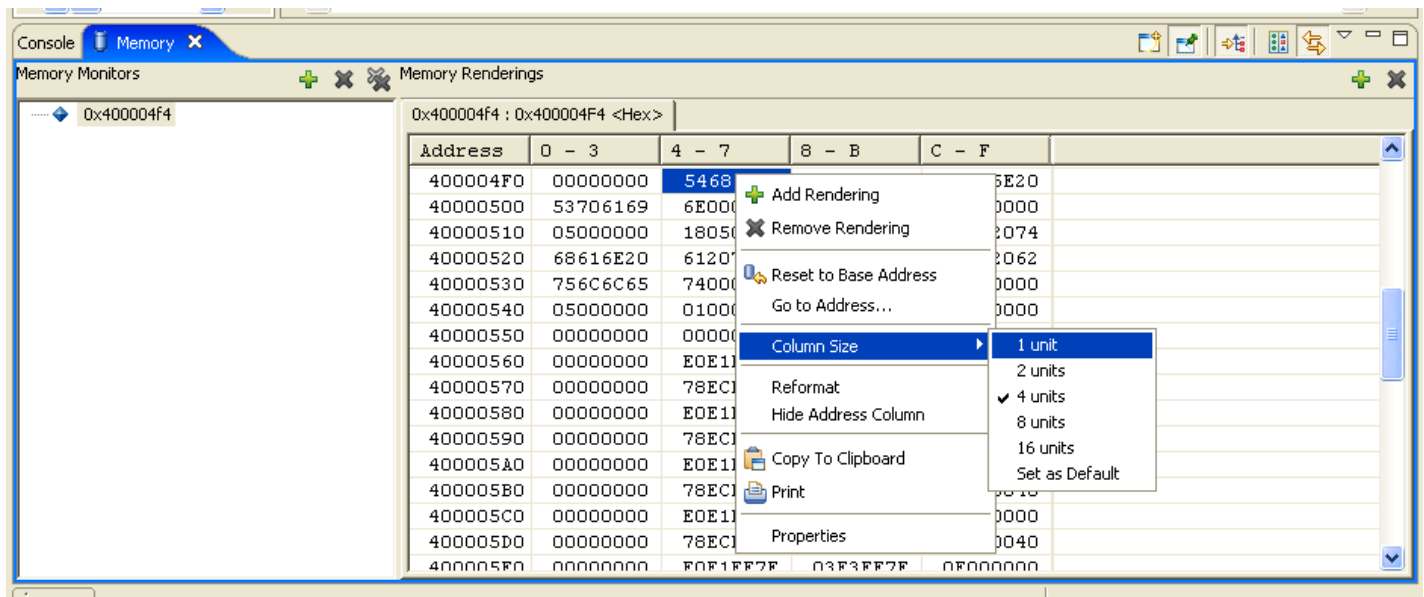
The memory monitor is created, although it defaults to 4-byte display mode. The display of the address columns and the associated memory contents is called a “**Rendering**”.

The address **0x40004F4** is called the Base Address; there’s a right-click menu option “**Reset to Base Address**” that will automatically return you to this address if you scroll the memory display.

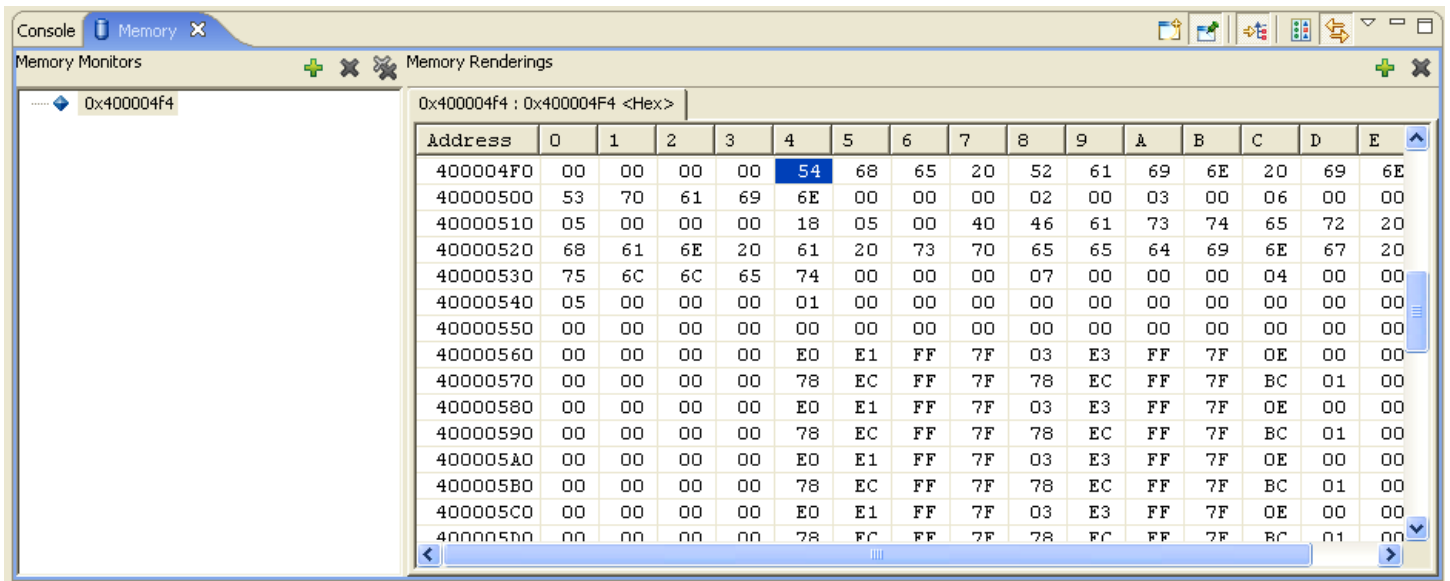


There’s also a “**Go to Address...**” right-click menu option that will jump all over memory for you.

By right-clicking anywhere within the memory rendering (display area), you can select “**Column Size – 1 unit**”.

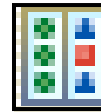


This will repaint the memory rendering in Byte format.



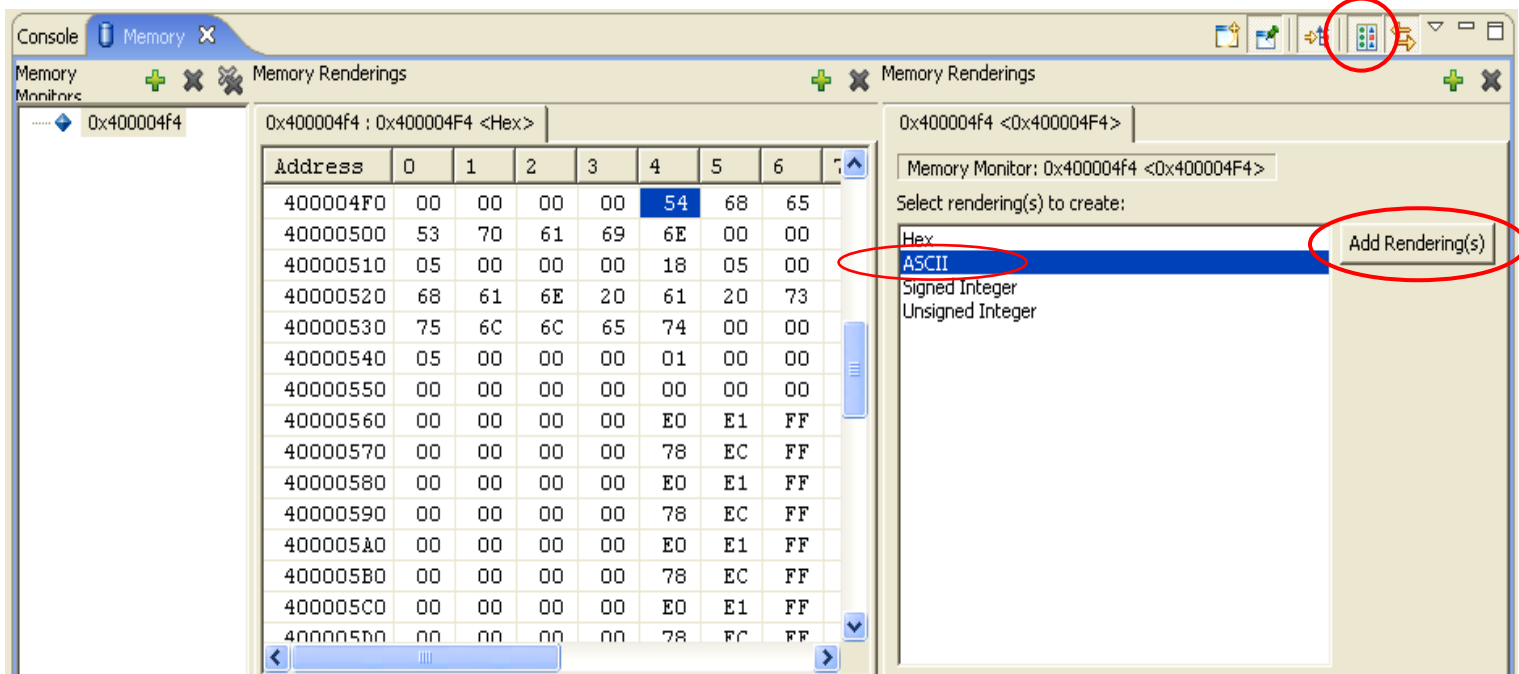
Now we will add a second rendering that will display the memory monitor in ASCII.

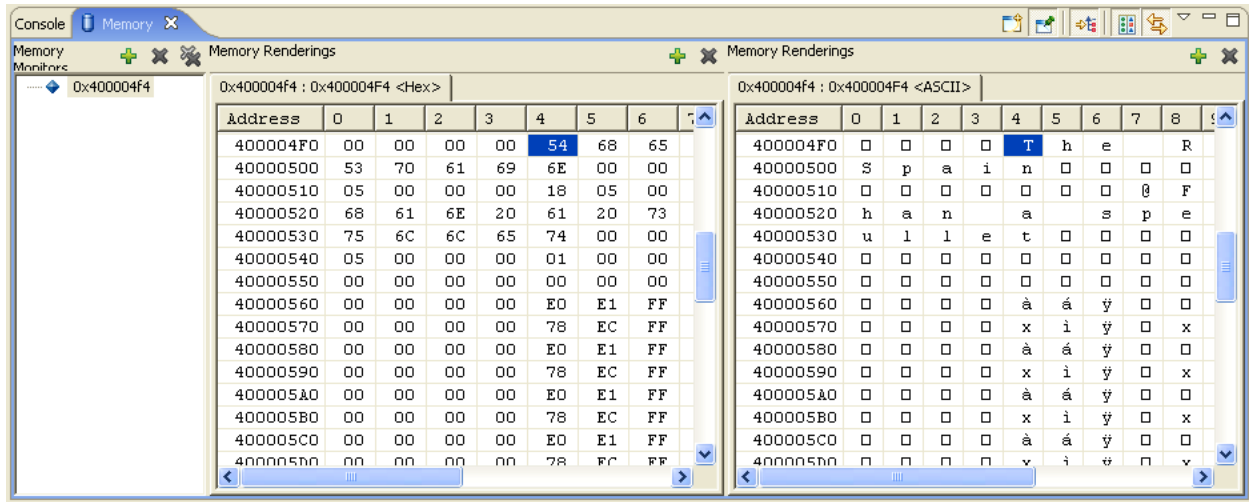
Click on the “Toggle Split Pane” button to create a second rendering pane.



Pick “ASCII” display for the new rendering.

Click on the “Add Rendering(s)” button to create an additional ASCII memory display.



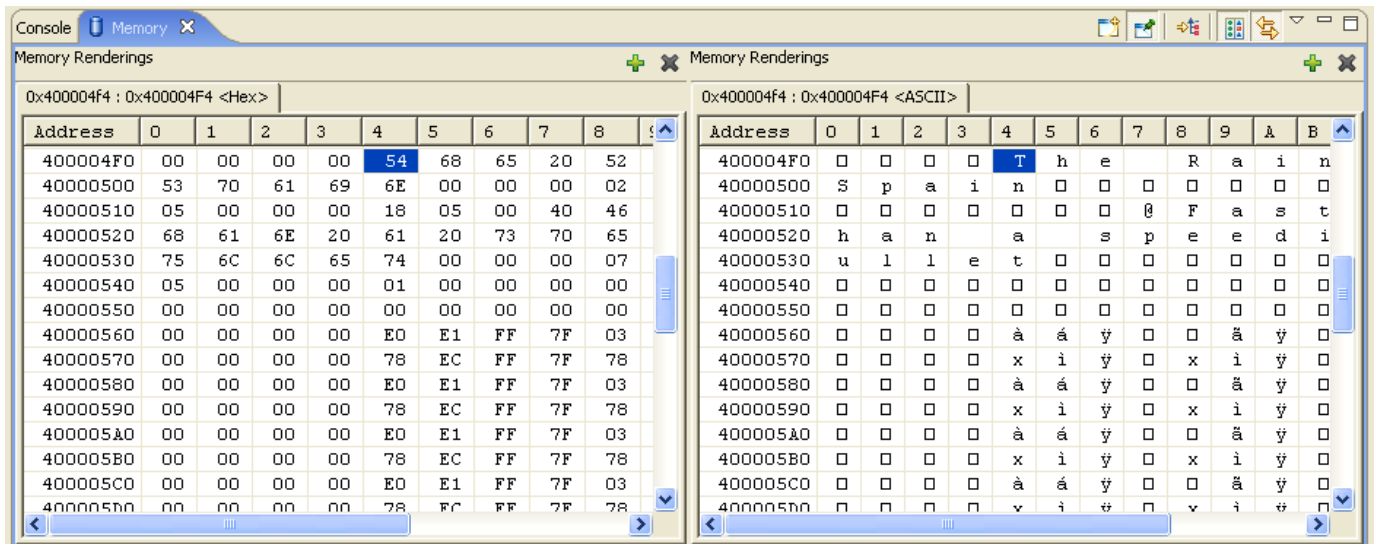


Click on the “Link Memory Rendering Panes”  button.

This means that scrolling one memory rendering will automatically scroll the other one in synchronism.

Click on the “Toggle Memory Monitors Pane”  button.

This will expand the display erasing the “memory monitors” list on the left.



Personally, I think this Eclipse memory display is a bit complex. However, it allows you to define many “memory monitors” and clicking on any one of them pops up the renderings instantly. It’s like so many things in life, once you learn how to do it; it seems easy!

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/Eclipse CDT Debugger 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.

The world will beat a path to the first vendor that markets an inexpensive USB-based debugger for the ARM microprocessor. The new Philips LPC2148 ARM chip with onboard USB looks like the perfect vehicle for such an application. The ARM JTAG circuits allow two "hardware" breakpoints and this should be adequate for a student or hobbyist who wishes to debug flash-based applications. The "student" version of a ARM USB-based debugger could be just a little circuit board with a cheap wall-wart power supply. Olimex has a USB-based JTAG debugger project in the works, so we can all keep our collective fingers crossed that they will be successful.

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

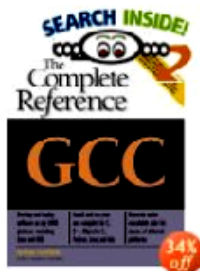
24 Some Books That May Be Helpful

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%)**

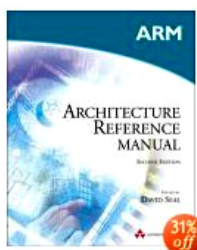
Availability: Usually ships within 24 hours. Ships from and sold by Amazon.com.

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](#)

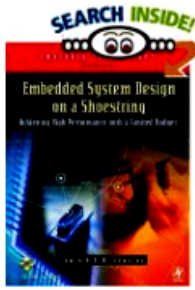
64 used & new available from **\$20.00**

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](#), [gdb 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!