Using Open Source Tools for AT91SAM7S Cross Development

Revision 1

Author:

James P. Lynch
Grand Island, New York, USA

June 6, 2006

TABLE OF CONTENTS

Introduction

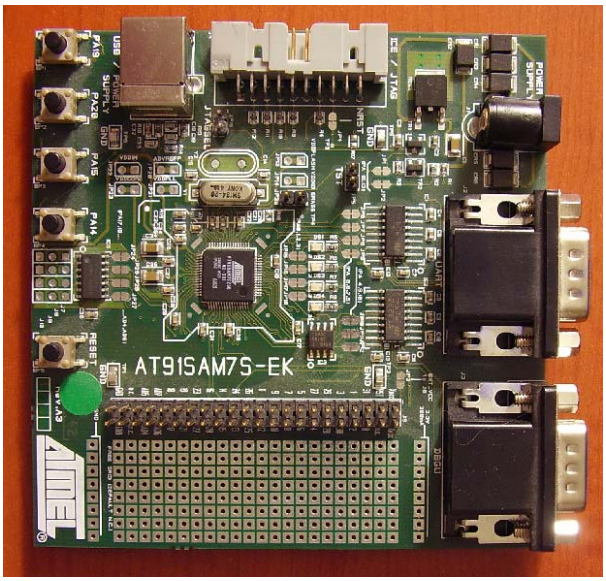
For those on a limited budget, use of open source tools to develop embedded software for the Atmel AT91SAM7S family of microcontrollers may be a very attractive approach. Professional software development packages from Keil, IAR, Rowley Associates, etc are convenient, easy to install, well-supported and fairly efficient. The problem is their price ($900 US and up) which is a roadblock for the student, hobbyist, or engineer with limited funding.

Using free open source tools currently available on the web, a very acceptable cross development package can be assembled in an afternoon's work. It does require a high-speed internet connection and quite a lot of patience.

This article guides the reader through the maze of open-source tools, web sites and setups required to build an Eclipse-based system for Atmel AT91SAM7 embedded software development.

Hardware Setup

As a hardware platform to exercise our ARM cross development tool chain, we will be using the Atmel AT91SAM7S-EK evaluation board, shown directly below.



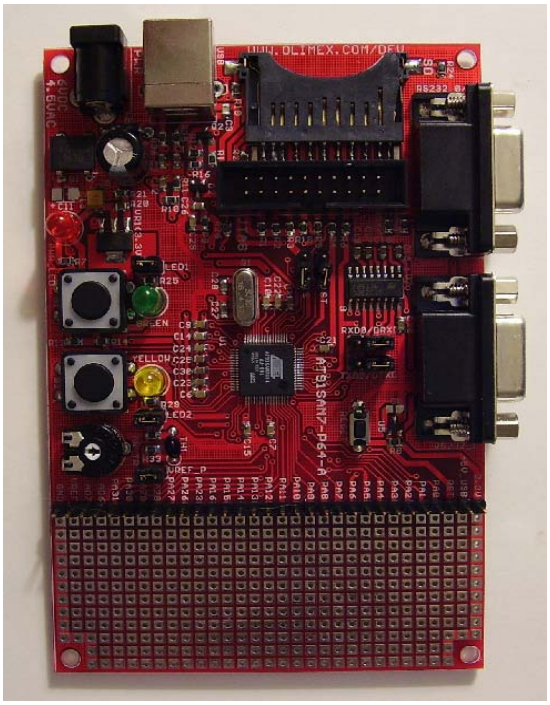
This board includes two serial ports, a USB port, an Atmel Crypto memory, JTAG connector, four buffered analog inputs, four pushbuttons, four LEDs and a prototyping area.

The Atmel AT91SAM7S256 ARM microcontroller includes 256 Kbytes of on chip FLASH memory and 64 Kbytes of on chip RAM.

The board may be powered from either the USB channel or an external DC power supply (7v to 12v).

This board is available from Digikey and retails for $149.00 www.digikey.com

There are numerous third party AT91SAM7 boards available. Notable is the Olimex AT91SAM7-P64-A shown directly below.



This board includes two serial ports, a USB port, expansion memory port, two pushbuttons, two LEDs, one analog input with potentiometer and a prototyping area.

The Atmel AT91SAM7S64 ARM microcontroller includes 64 Kbytes of on chip FLASH memory and 16 Kbytes of on chip RAM.

The board may be powered from either the USB channel or an external DC power supply (7v to 12v).

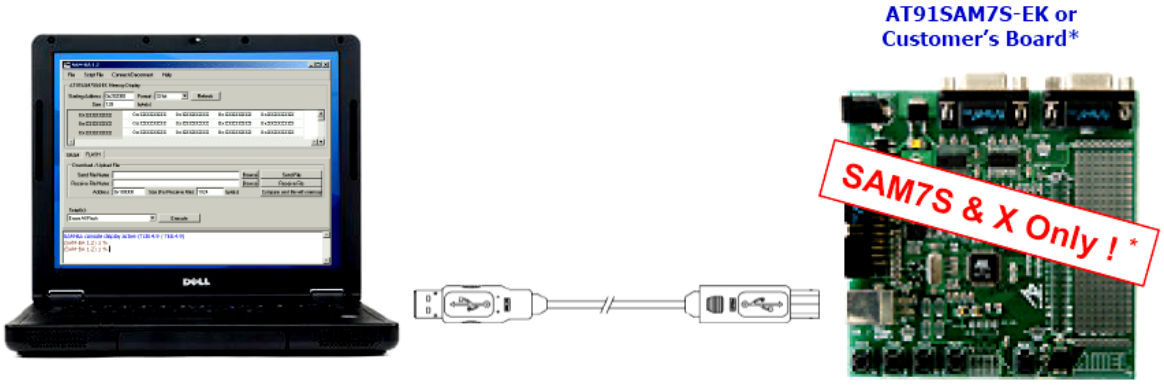
This board is available from Olimex, Spark Fun Electronics and Microcontrollershop; it retails for $59.95

www.olimex.com
www.sparkfun.com
www.microcontrollershop.com

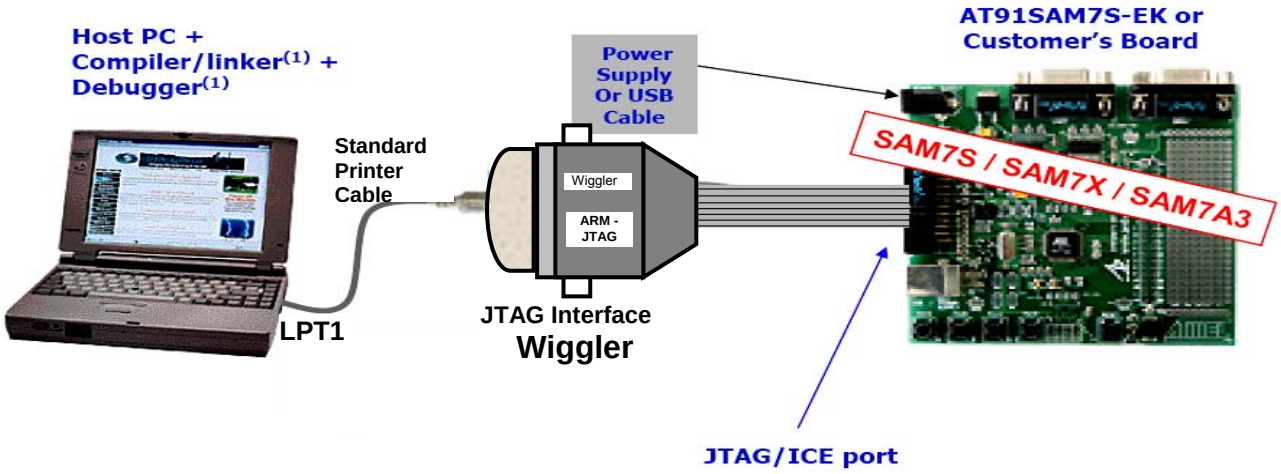
For the rest of this tutorial, we will concentrate on the Atmel AT91SAM7S-EK evaluation board.

The Olimex board can be substituted but the reader must then make minor adjustments to the memory map in the Linker script file since it has only 64K of FLASH memory and 16K of RAM memory.

The onboard FLASH memory may be programmed using the Atmel Boot Assistant (SAM-BA) utility and a standard USB cable, as shown below. The USB cable also powers the board.



If debugging is required, then an inexpensive Olimex **ARM-JTAG** interface can be installed from the PC's printer port to the board's 20-pin JTAG connector as shown below. The USB cable can be left in to power the board. If execution from onboard RAM memory is desired, the JTAG cable can be used to load the executable file into the RAM memory for execution and debug.



Open Source Tools Required

The following components are required to set up a functional ARM cross development environment.

SUN Java Runtime

CYGWIN Linux API emulation layer for Windows

Eclipse IDE

Eclipse CDT Plug-in for C++/C Development (Zylin custom version)

GNUARM GNU C++/C Compiler for ARM Targets

Atmel SAM-BA Flash Programmer for AT91SAM7 Family CPUs

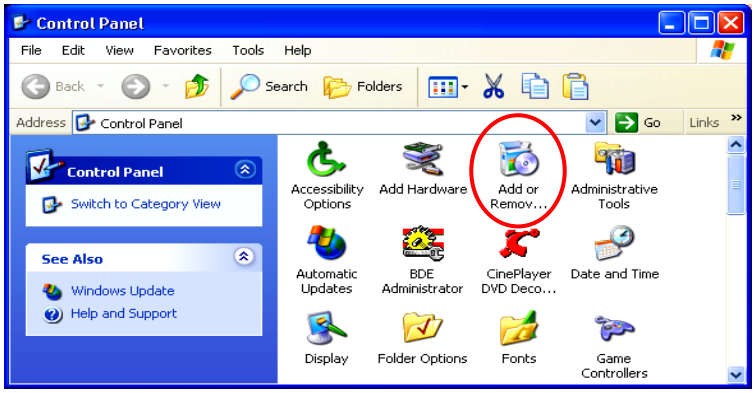
OpenOCD for JTAG debugging

Note: The Eclipse/CDT does NOT run on Windows 98 or Windows ME

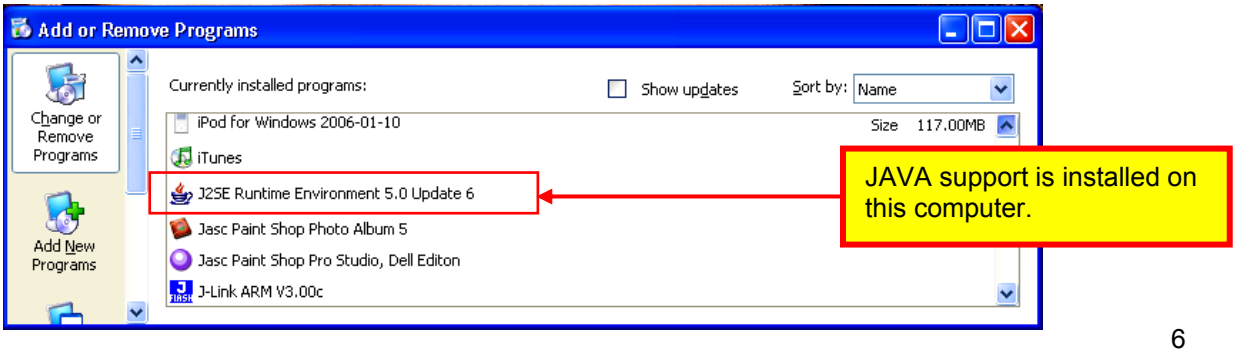
Install JAVA Support

Since the Eclipse Integrated Development Environment (IDE) is written partially in JAVA, we must have JAVA support on our computer to run it. With the recent peace treaty between Microsoft and Sun Microsystems, most recent desktop PCs running Windows 2000 or Windows XP already have JAVA installed.

To check this, go to the Windows Control Panel and click "**Add or Remove …**"



Now scroll through the list of installed programs and verify if JAVA support is present (J2SE Runtime Environment). If it's already installed, go to the next section of the tutorial. If not, follow the instructions for installation of the JAVA runtime environment below.



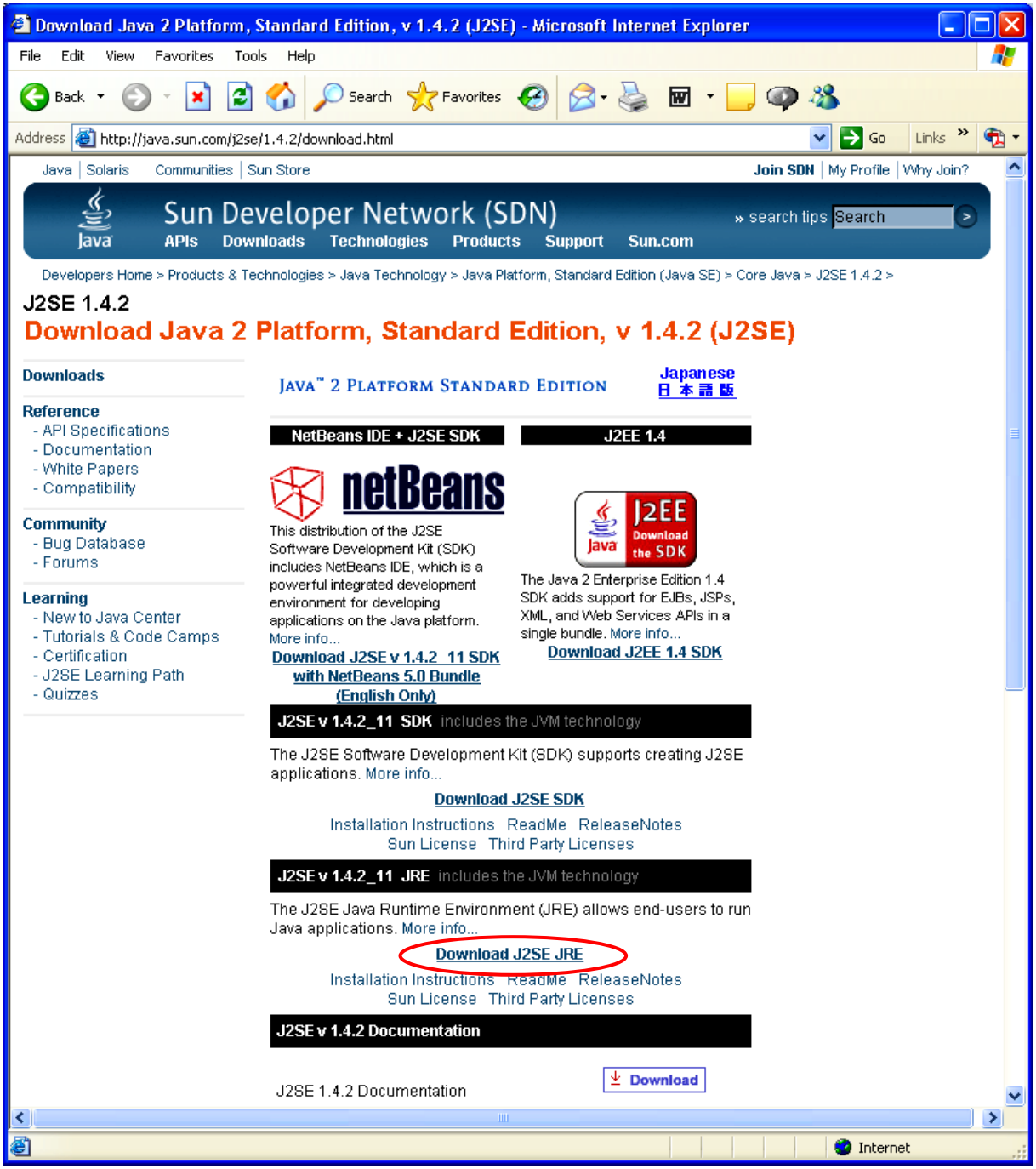
To install the JAVA Runtime Environment, go to the SUN web site and download it.

http://java.sun.com/j2se/1.4.2/download.html

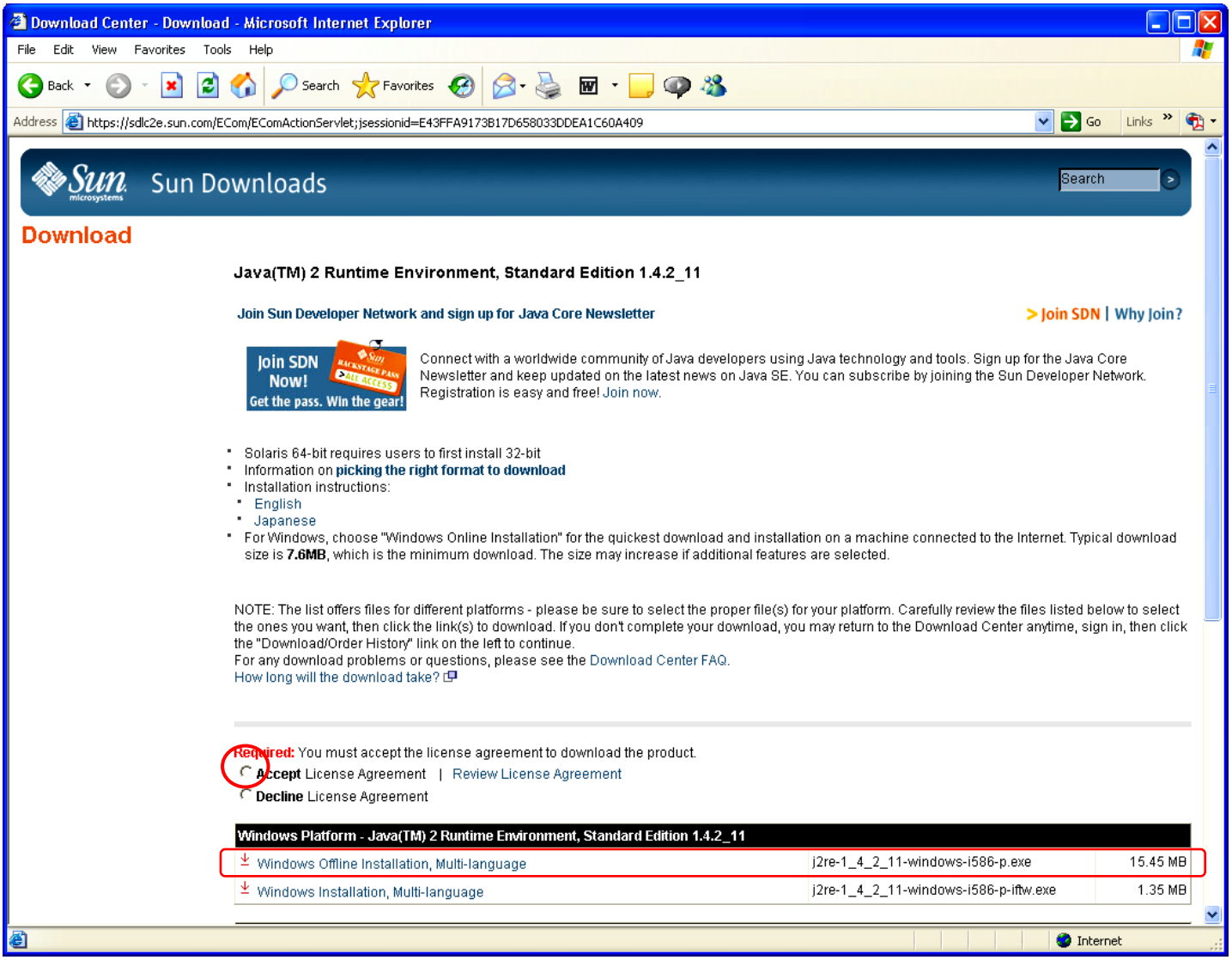
Java is not Open Source but Sun has always allowed free downloads of the JAVA package. If you build a commercial product that uses JAVA and the Java Virtual Machine, they may expect royalties. This would not typically be the case in using Eclipse in a cross-development environment.

The Sun JAVA web site is very dynamic so don't be surprised if the JAVA run time download screens differ slightly from this tutorial.

To support Eclipse, we just need the Sun JAVA Runtime Environment (JRE). Click on "**Download J2SE JRE**" as shown below.



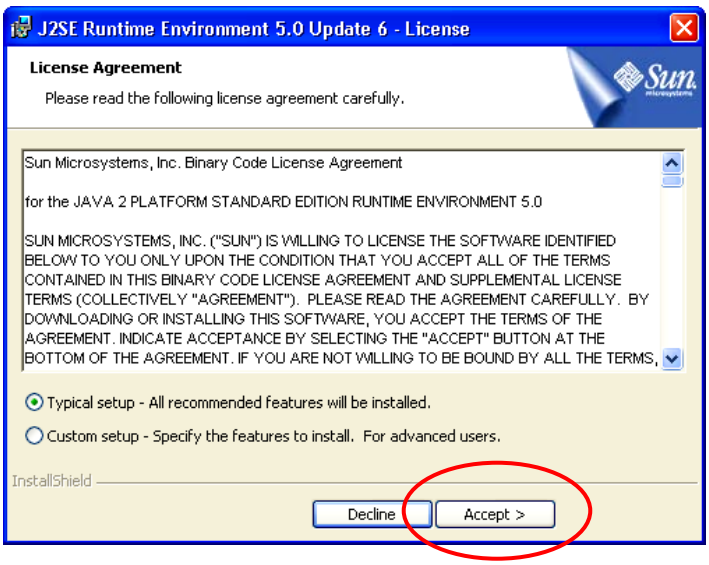
In the next download screen, shown below, click the radio button "**Accept License Agreement**" and then click on "**Windows Offline Installation, Multi-language**".



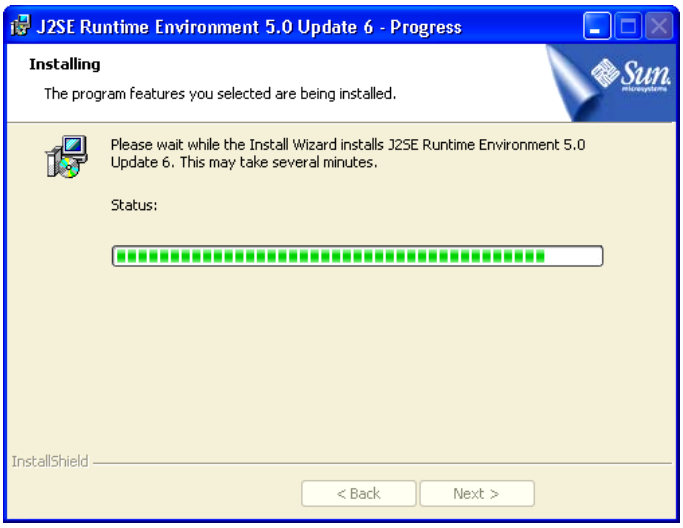
Start the JAVA installation by clicking on "**Run**".

Now the Sun JAVA runtime installation engine will start.

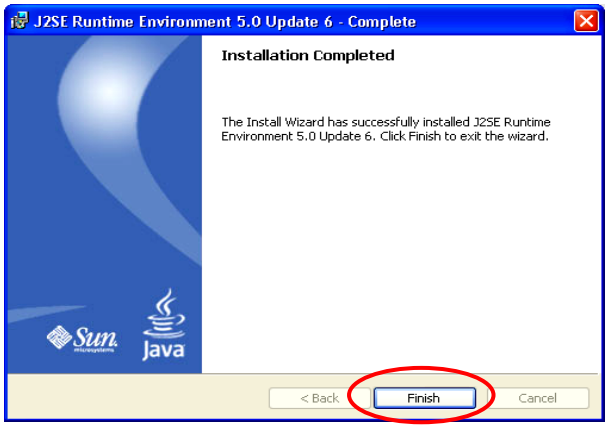
Click on the “Typical Setup” radio button and then accept the license terms. JAVA is free; if you build some fabulous product with JAVA integrated into it, then you may have to pay Sun royalties.



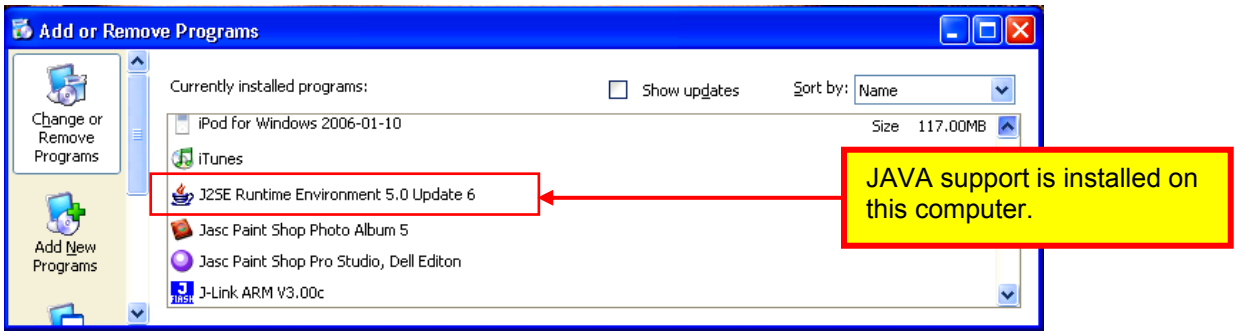
A series of installation progress screens will appear. Installation only takes a couple of minutes.



When JAVA runtime installation completes, click on “Finish” to exit the installer.



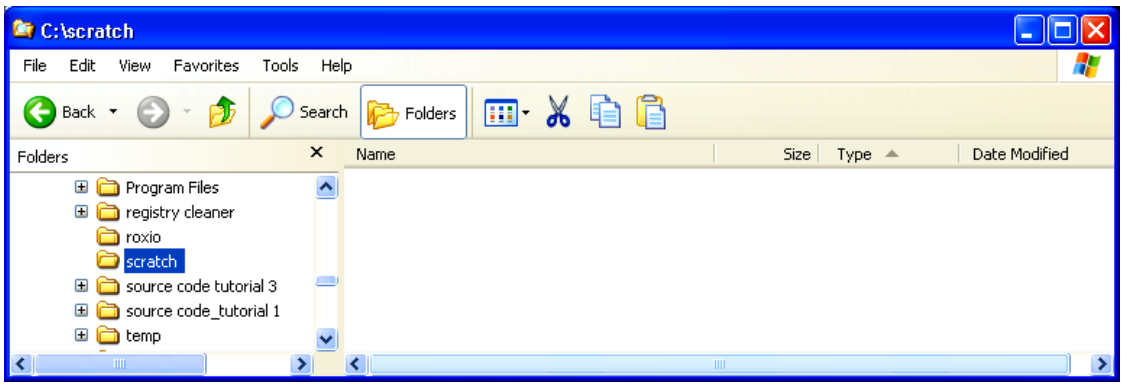
Now go back to the Windows Control Panel and use “**Add or Remove …**” to determine that the JAVA runtime has been installed successfully.



Downloading the Software Components

To make installation more convenient and fool-proof, all the necessary software downloads have been combined into a single 165 Mbyte zip file. There are two very good reasons to do this. First, these selected software components are known to work well together and thus guarantees a working ARM cross development system. Second, it is extremely efficient to have everything you need in one place.

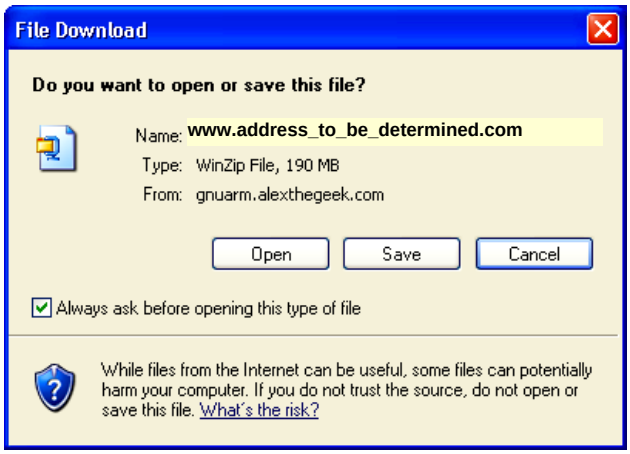
Using Windows Explorer, create an empty **c:\scratch** folder to hold the large download containing our software components.



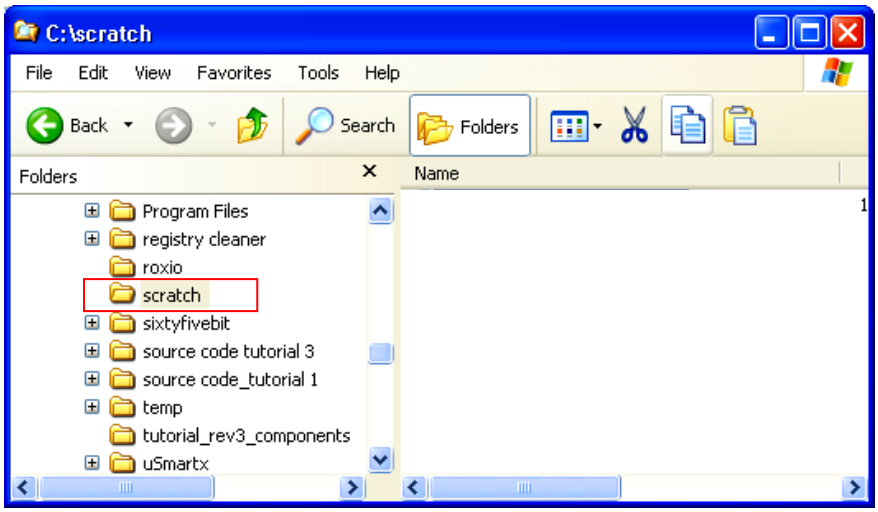
The software components you need to create an Open Source ARM cross-development system are stored as a very large zip file at the following link.

www.address_to_be_determined.com

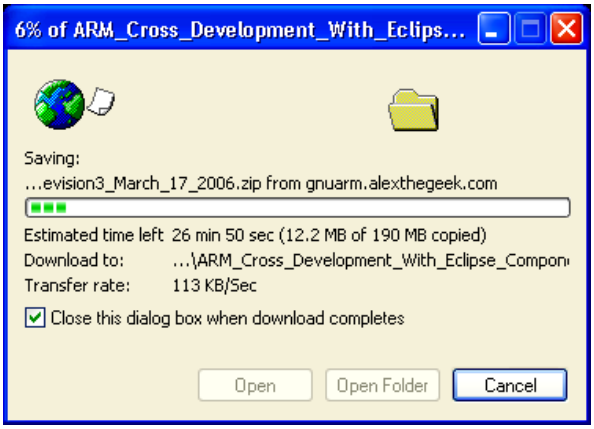
Now click on "**Save**".



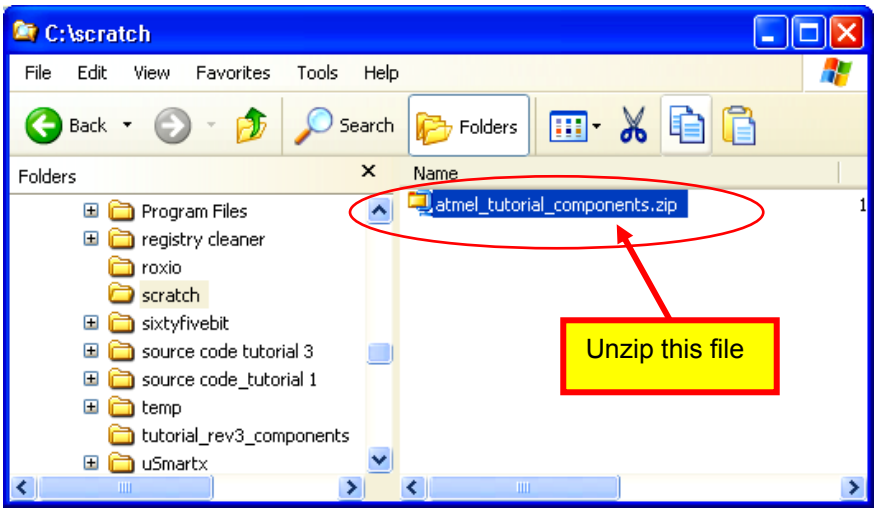
In the "**Save As**" window below, browse to the **c:/scratch** folder. This sets the destination for the large download. Click on "**Save**" to start the download process.



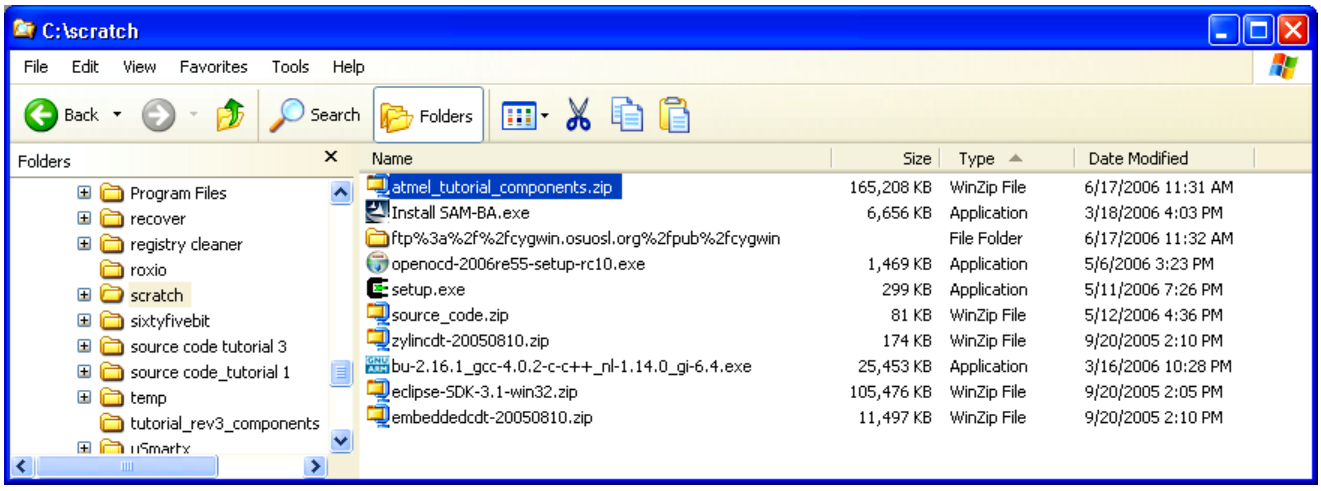
The download may take fifteen minutes to over an hour, depending on traffic.



When completed, you should see the following zip file **(atmel_tutorial_components.zip)** in **c:/scratch**.



Unzip the above file to reveal the various components. We're going to assume that you know how to unzip a compressed file, using Windows XP explorer or a tool such as WinZip (http://www.winzip.com/).



CYGWIN GNU Toolset for Windows

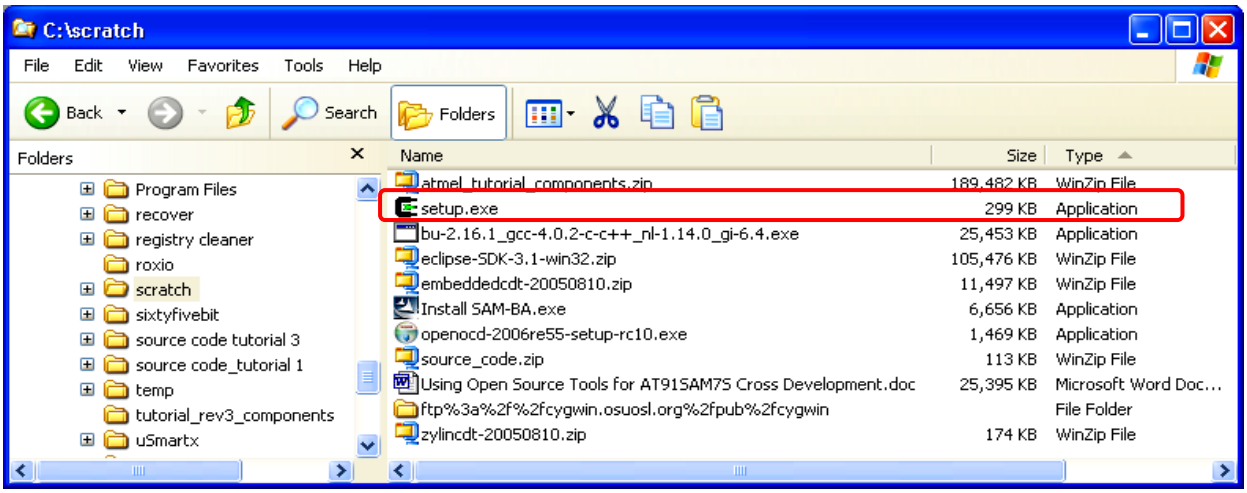
The GNU toolset is an open-source implementation of a universal compiler suite; it provides C, C++, ADA, FORTRAN, JAVA, and Objective C. All these language compilers can be targeted to most of the modern microcomputer platforms (such as the ARM 32-bit RISC microcontrollers) as well as the ubiquitous 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 for the GNU operating system. To the rescue came Cygwin, a company that created a set of Windows dynamic link libraries that enable the GNU compiler toolset to run on a Windows platform.

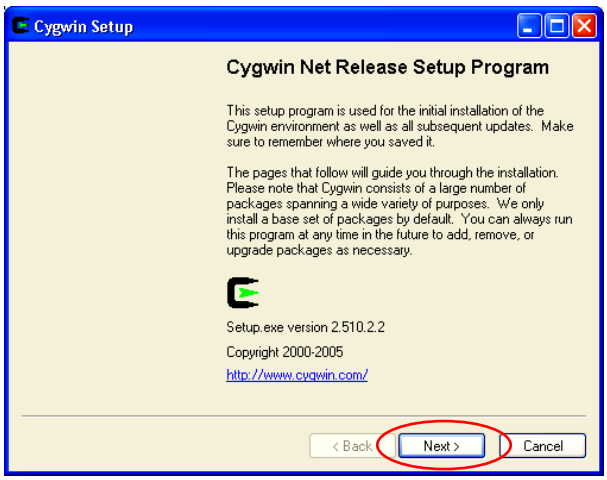
We will be installing a very minimal version of Cygwin; it will not contain a compiler tool chain for the Intel / Windows platform. It will not include an ARM tool chain either. Fortunately, there are quite a few pre-built tool chains on the internet; one such tool chain is GNUARM which gives you a complete set of ARM compilers, assemblers and linkers. We will use this later in the tutorial.

The GNUARM tool chain doesn’t include the crucial **MAKE** utility, it’s in the Cygwin tool kit we’re about to install. Therefore, we will tailor the Cygwin installation to install just the **MAKE** utility and the DLLs necessary to support the GNUARM compiler tool chain – we don’t need the ability to compile for PC targets. This will greatly reduce the size of the Cygwin component.

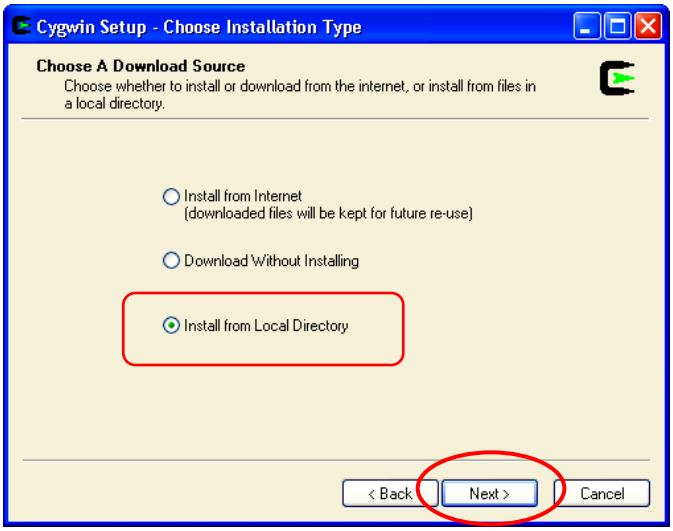
Double-click on the application “**setup.exe**” in the scratch directory as shown below.



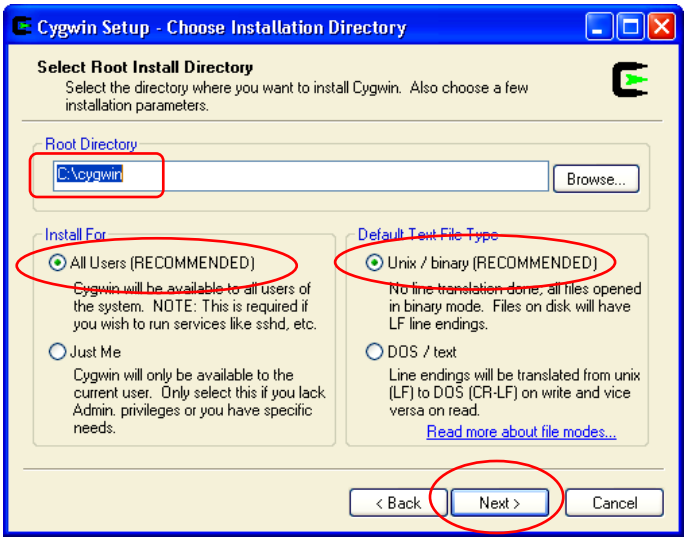
The Cygwin setup program will start. Click on “**Next**” to continue.



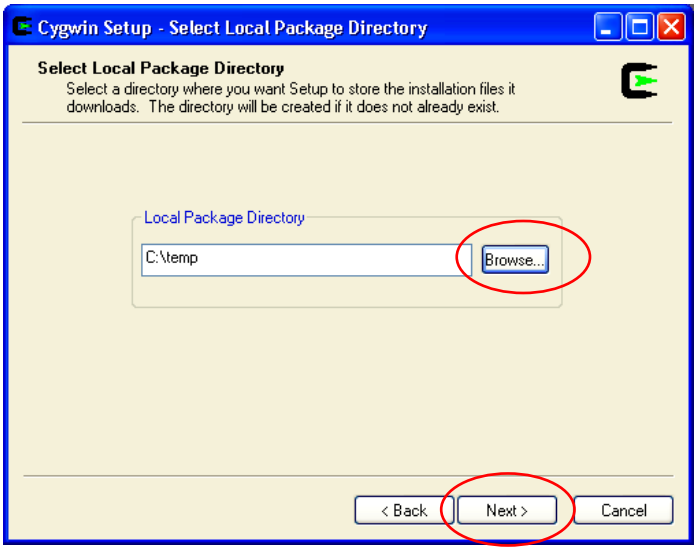
Choose “**Install from Local Directory**” and then click “**Next**.”



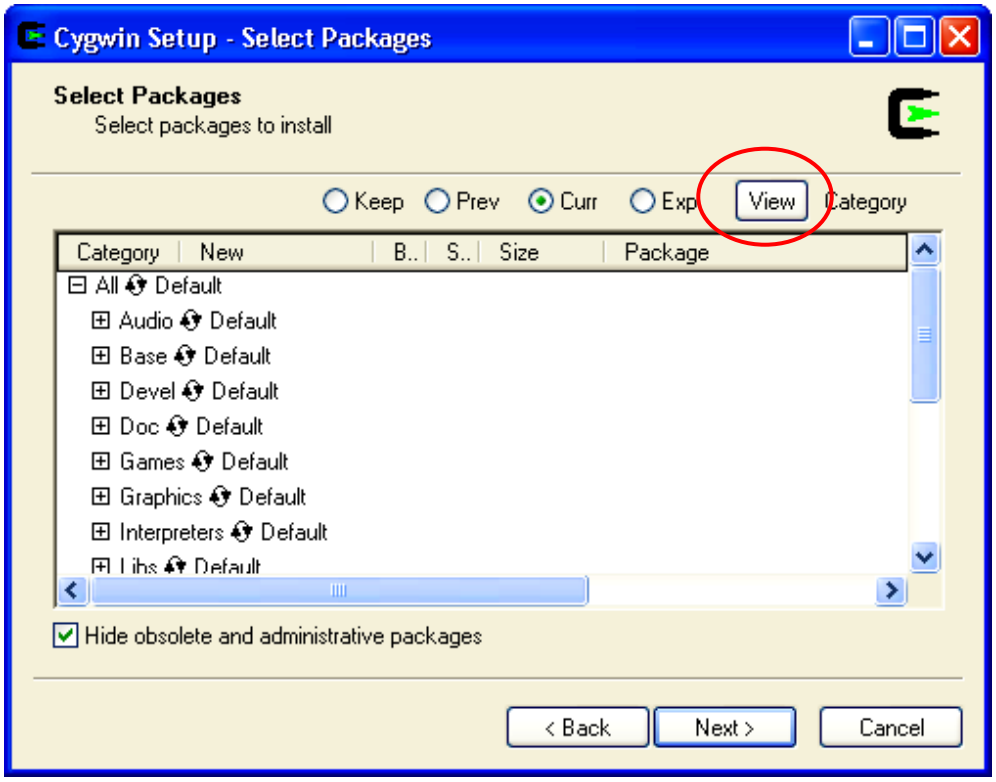
In the “**Choose Installation Directory**” screen below, take the default directory **c:\Cygwin**. Also, the default settings “**All Users**” and “**Unix / Binary**” are appropriate. Click “**Next**” to continue.



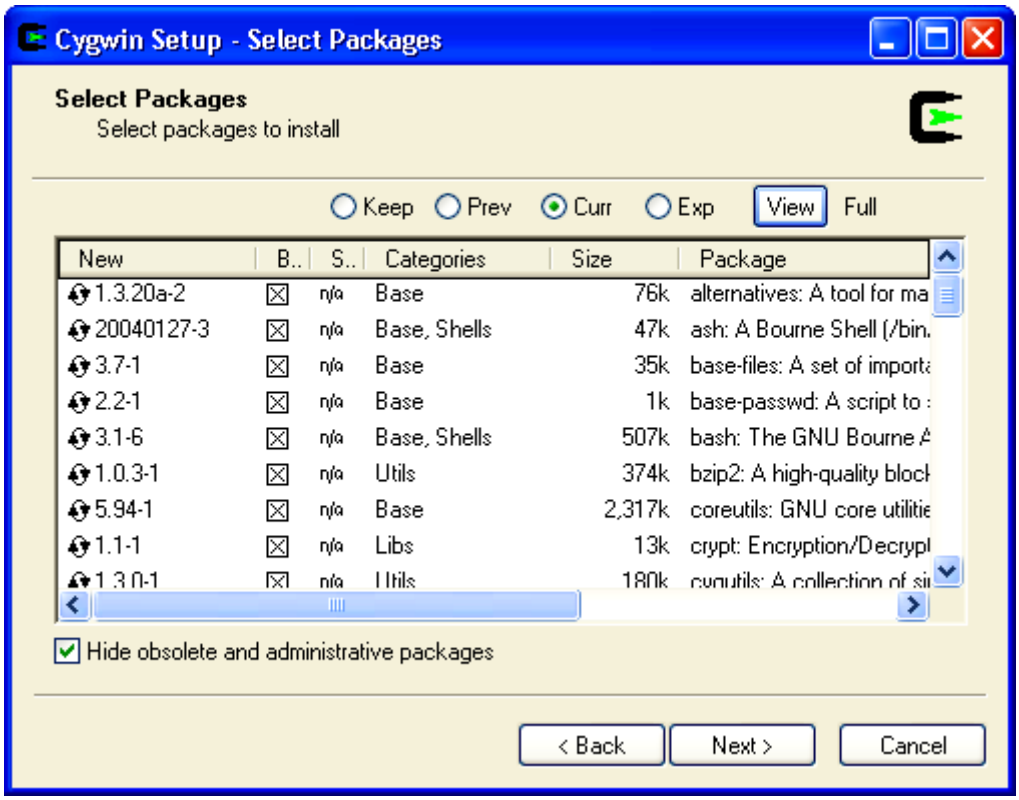
Use the “**Browse**” button to find a temporary folder to hold any temporary files that Cygwin uses. Here we selected the folder “**c:\temp**”. Click “**Next**” to continue.



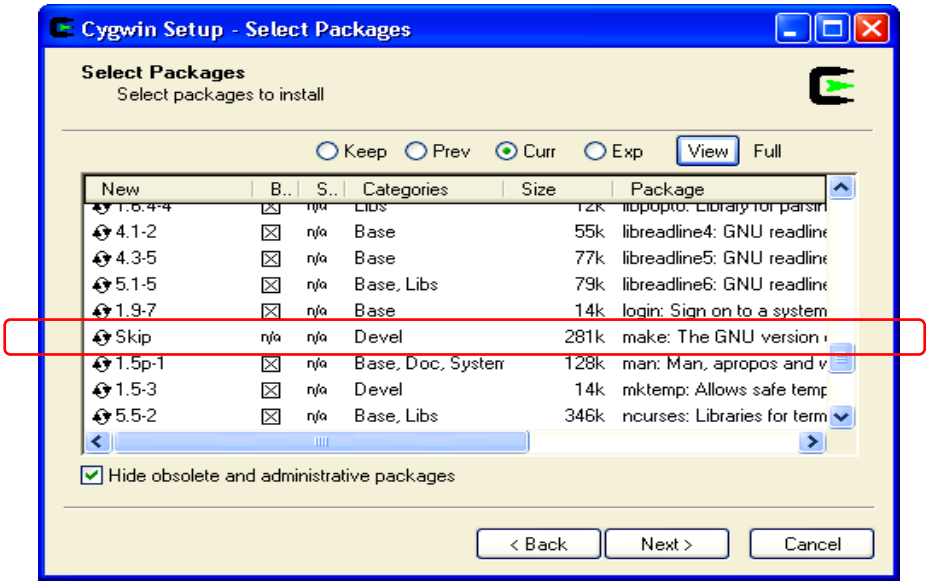
When the “**Select Packages**” screen appears, click on “**View**” to show the individual packages available.



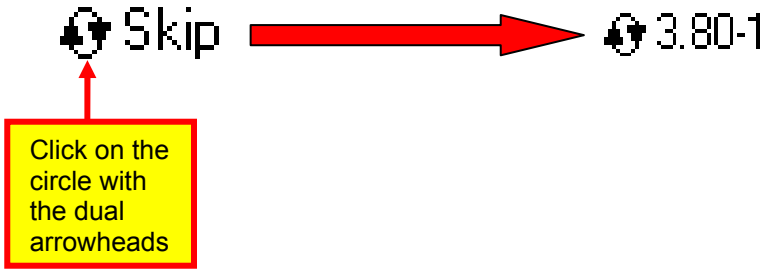
Now the “Select Packages” display is rearranged



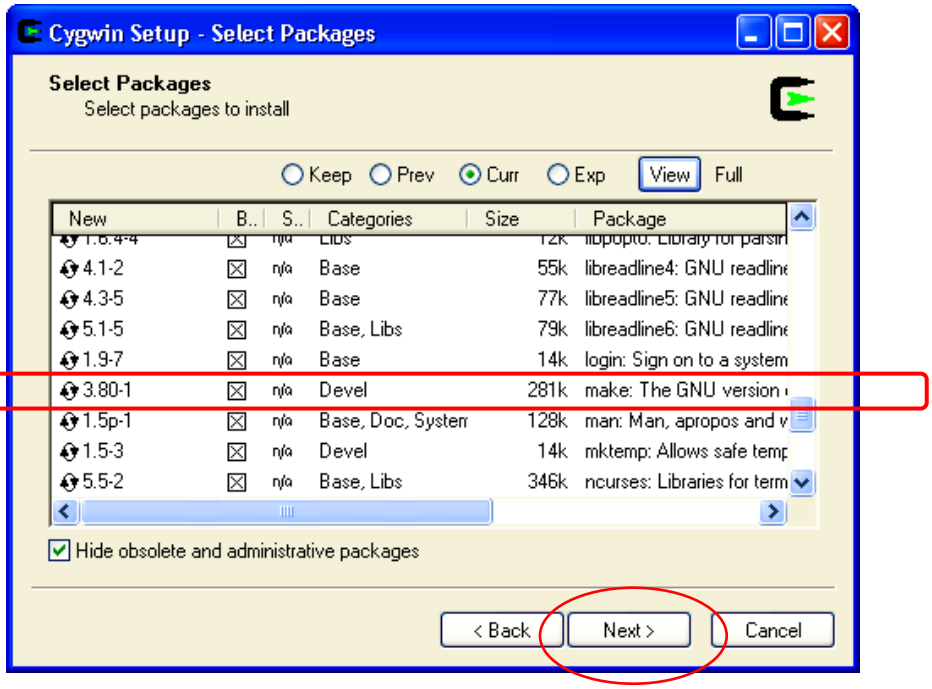
Scroll down the list until you see the “**Make**” utility.



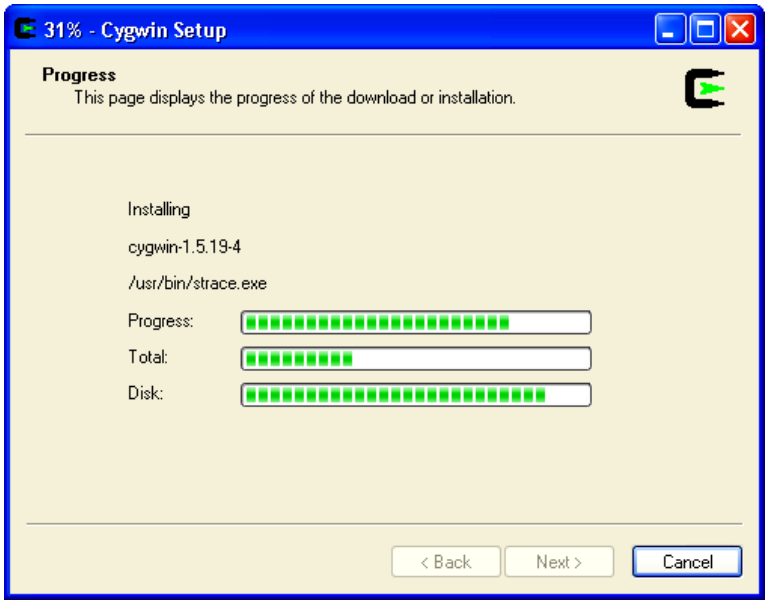
Click on the small circle with the two arrowheads until it changes from “Skip” to “3.80-1”.



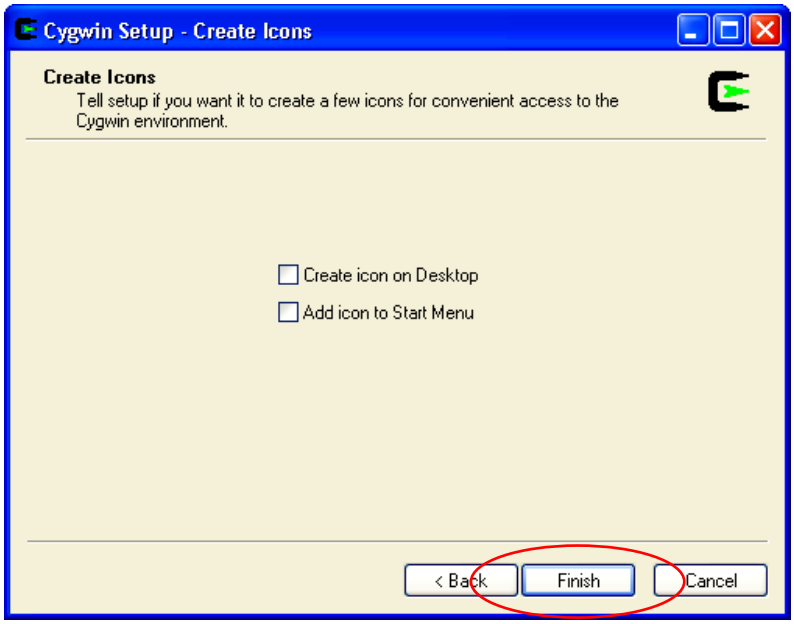
The “**Select Packages**” screen should look as shown directly below. **Make** has now been selected. Click on “**Next**” to start the Cygwin installation.



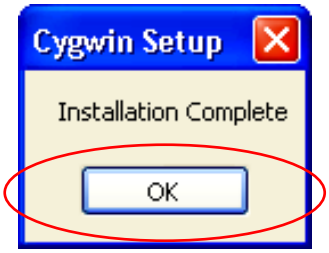
Now Cygwin will start the installation process; this only takes a couple of minutes.



Do not check "**Create icon on Desktop**" and "**Add icon to Start Menu**". Click "**Finish**" to continue.



Click "**OK**" when Cygwin completes. We are now finished with the CYGWIN installation. It runs silently in the background and you should never have to think about it again.



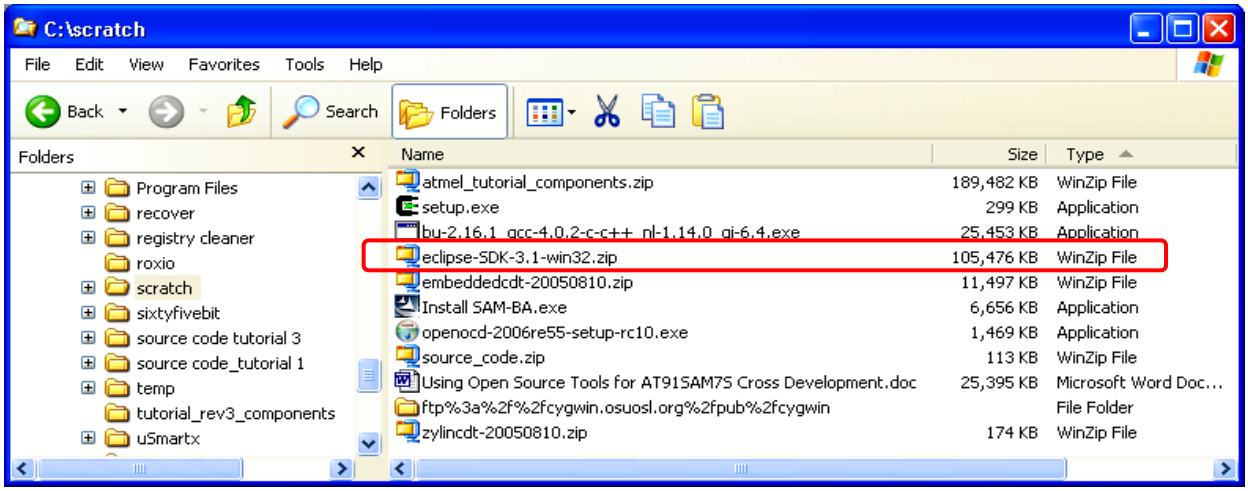
Install Eclipse IDE

IBM has been a competitor in recent years to Microsoft and at one time was building an alternative to Microsoft's Visual Studio (specifically for the purpose of developing JAVA software). This effort was called the Eclipse Project and in 2004 IBM donated Eclipse to the Open Software movement, created an independent Eclipse Foundation to support it and invited programmers worldwide to contribute to it. The result has been an avalanche of activity that has catapulted Eclipse from a simple JAVA editor to a multi-platform tool for developing just about any language, including C/C++ projects.

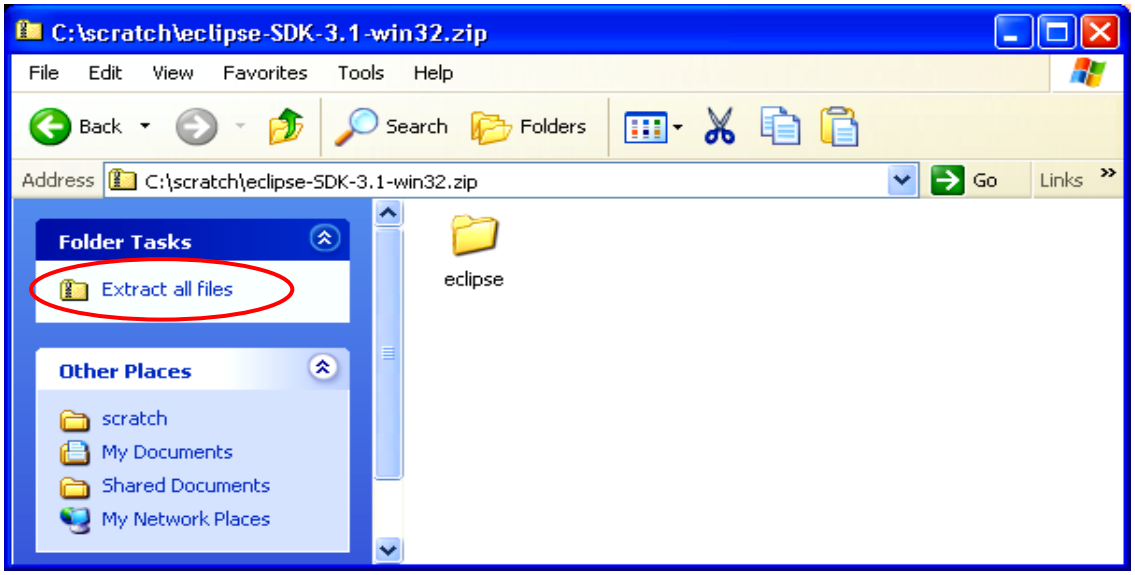
Be sure to visit the Eclipse web site: **www.eclipse.org**

To install Eclipse, we just need to unzip the file "**eclipse-SDK-3.1-win32.zip**" to the **c:** root directory. At the writing of this tutorial, Eclipse Version 3.1 is the official "latest release". The danger of selecting the Eclipse "stream" releases (work-in-progress) is that it may not be compatible with the CDT plug-in. The components downloaded into the **c:\scratch** folder were chosen for their compatibility with each other.

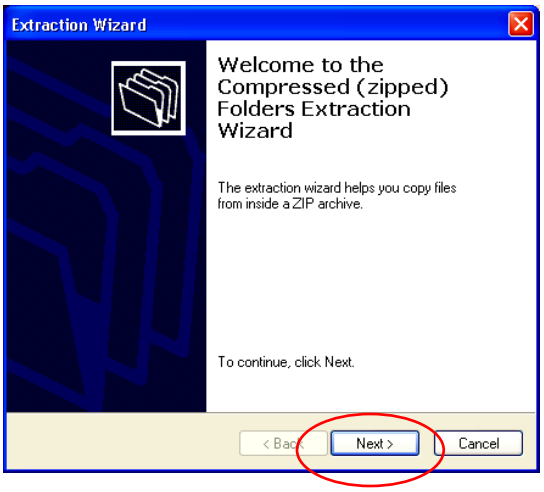
For the Eclipse installation, we will show the unzip operation in more detail.



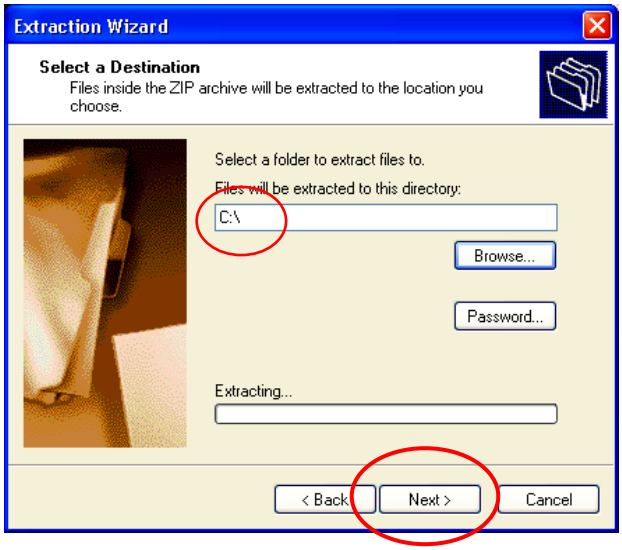
Using the Windows XP unzip facility, double-click on the Eclipse zip file above and select "**Extract all files**" as shown below.



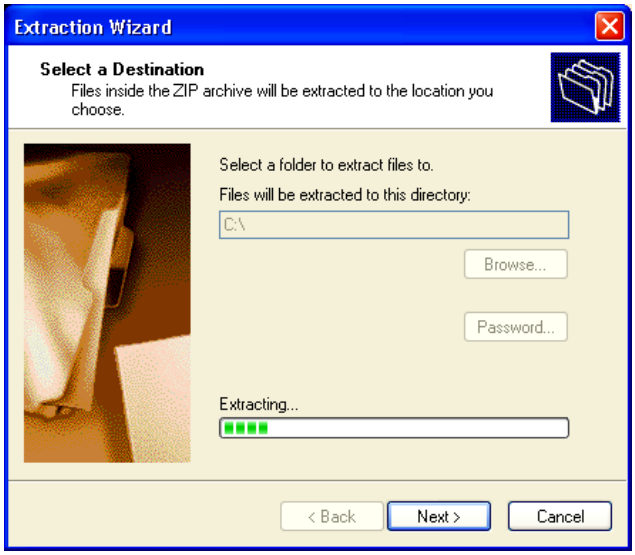
The Windows XP built-in Extraction Wizard opens, click “Next” to continue.



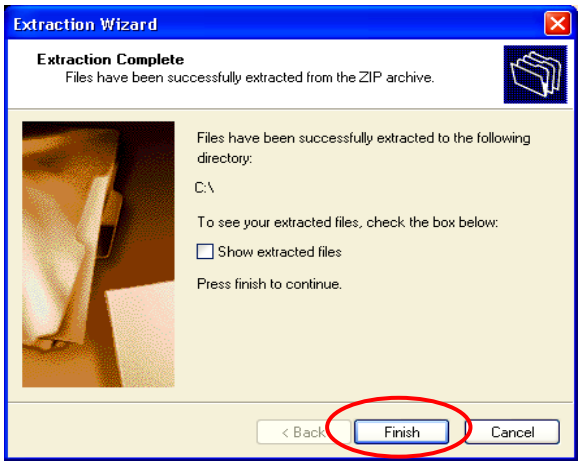
Now browse to select the root folder **c:** as the destination. Click “**Next**” to start the zip file extraction.



The zip file extraction operation will only take a couple of minutes.



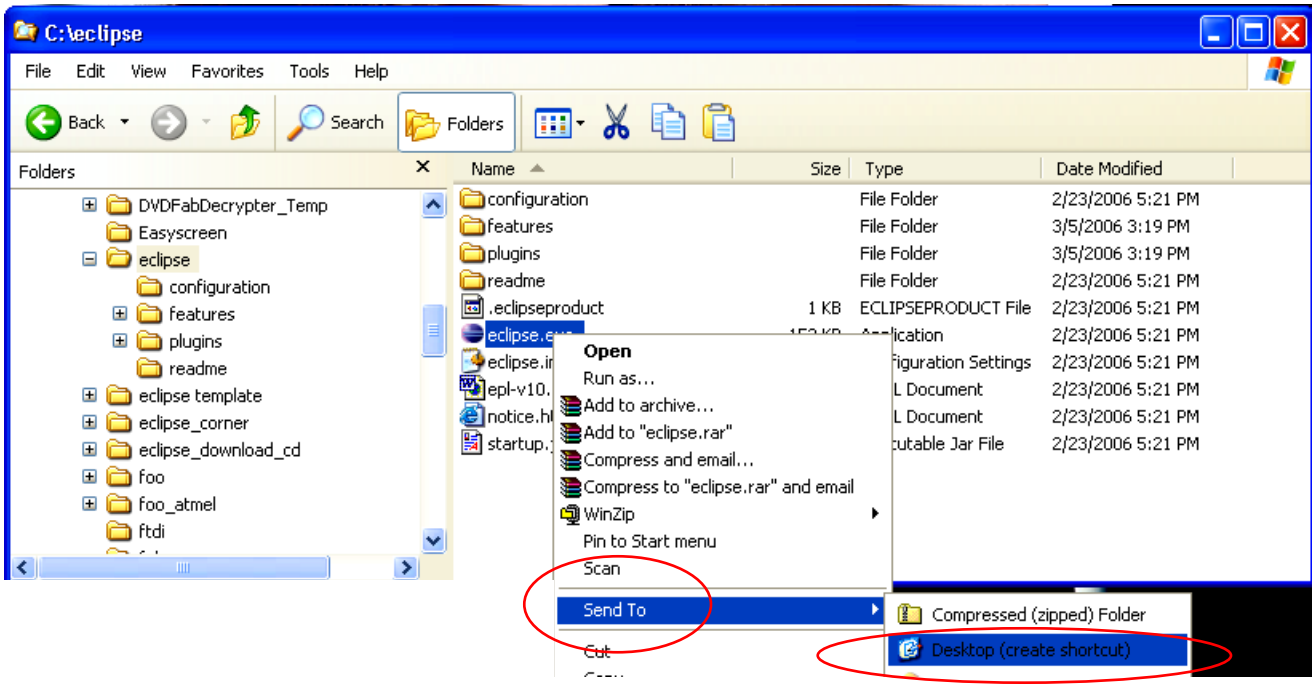
When the unzip operation completes, click “**Finish**”.



At the end of unzipping, use Windows Explorer to verify that Eclipse is installed on your c:\ drive.

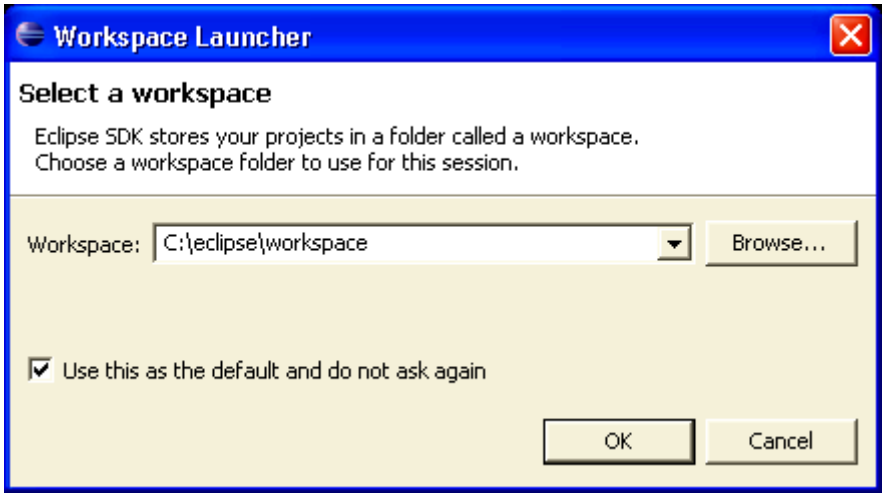
At this point, Eclipse is essentially installed. When you first execute it, some setup operations are performed. However, the beauty of Eclipse is that it is a simple executable (eclipse.exe). There are no entries made to the Windows registry.

Now is a good time to create a **desktop icon** for the Eclipse IDE. Right-click on the **eclipse.exe** executable in the **c:/eclipse** folder and select “**Send To - Desktop**” as shown below.



This will create a desktop icon; it may be prudent to rename it to indicate which version of Eclipse is being used. Here we changed the icon’s name to “Eclipse 3.1”.

Now double-click on the Eclipse icon to start Eclipse. Before Eclipse starts, it will request that you specify the location of the workspace (where your source code, etc. is stored). You can take the default but I choose to simply type the folder **c:\eclipse\workspace** directly into the text box. Click the checkbox "**Use this as a default…**" and you won't be bothered again.



Eclipse will start up and display its "splash" screen. Now we have proof of a successful Eclipse installation and you may cancel the program and continue with the downloading of CDT.

Eclipse CDT

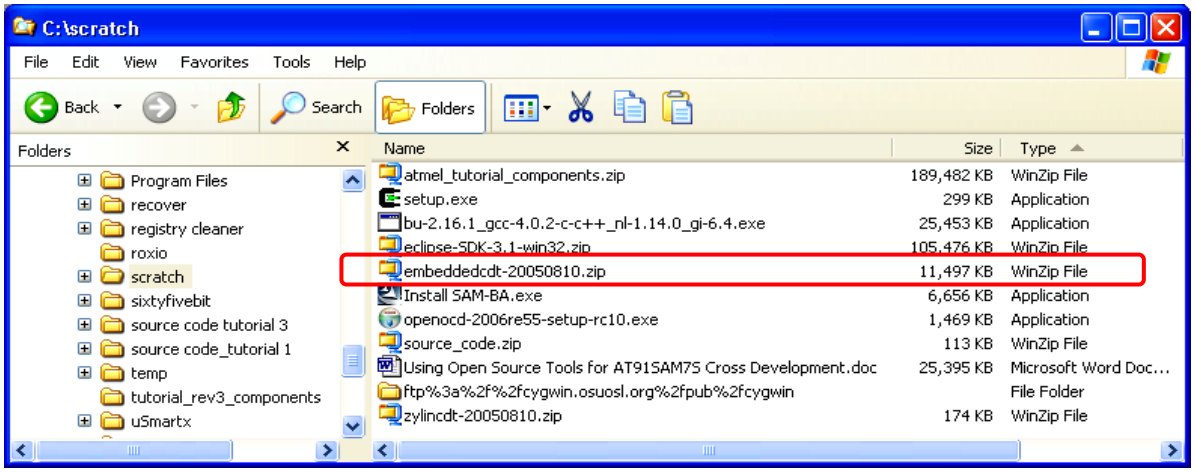
Eclipse by itself makes a wonderful Integrated Development Environment (**IDE**) for JAVA software. There are numerous books available on the Eclipse JAVA platform and many PC and Web applications are being built with it.

Our purpose is to build an IDE for embedded software development; this normally implies C/C++ programming. To do this, we need to install the **CDT** (**C D**evelopment **T**oolkit) plug-in. The problem is that **CDT** has had difficulties working with remote debuggers. The Norwegian company Zylin has developed, with the cooperation of the **CDT** team, a custom version of the **CDT** plug-in that solves these problems. The Zylin version of **CDT** properly starts the remote debugger in idle mode so you can start execution, single-step, etc.

The only proviso is that we must select a version of Eclipse compatible with the Zylin **CDT** plug-in. The Zylin **CDT** included in the "components" download was chosen for its compatibility with the Eclipse 3.1 release.

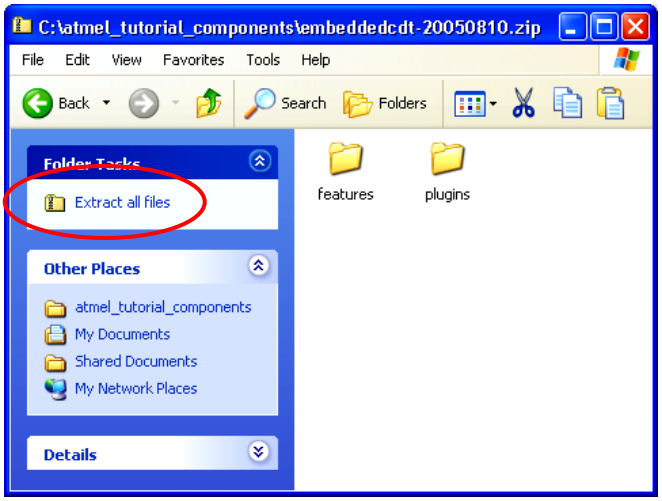
Eclipse does have an "automatic update" feature under the Help System that will conveniently download and install the **CDT** plug-in. The problem is that the **CDT** plug-in installed via this process is not the Zylin version. To install the Zylin version of the **CDT** plug-in, we will be simply extracting, one by one, the two Zylin **CDT** zip files in the "components" download into the **c:\eclipse** folder.

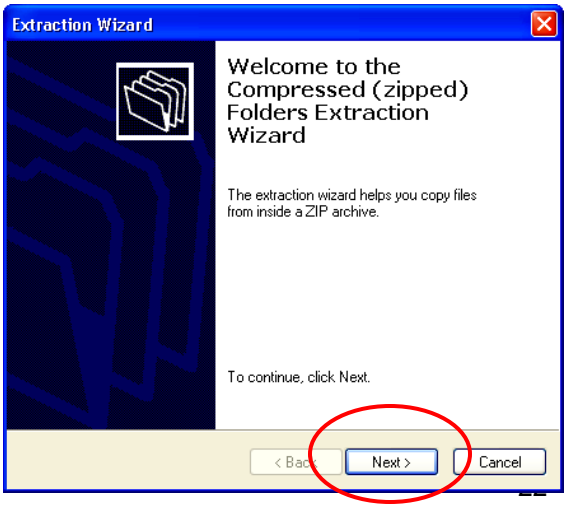
The Zylin website is at this address: www.zylin.com



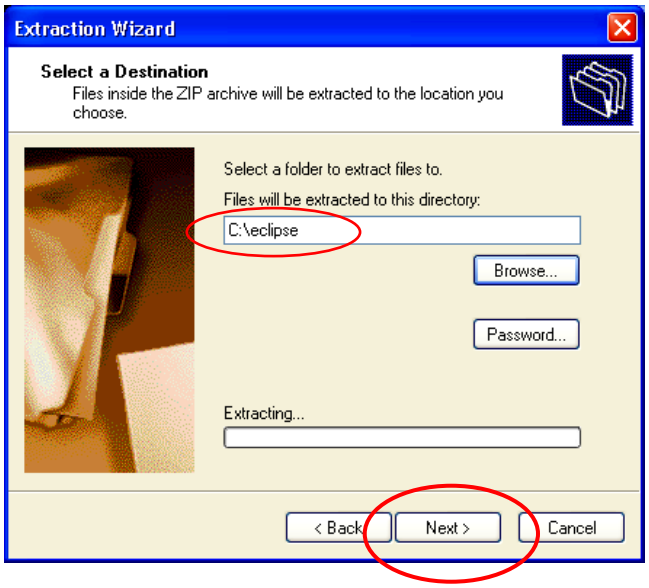
Double-click on the first of the two **CDT** zip files above: **embeddedcdt-20050810.zip**.

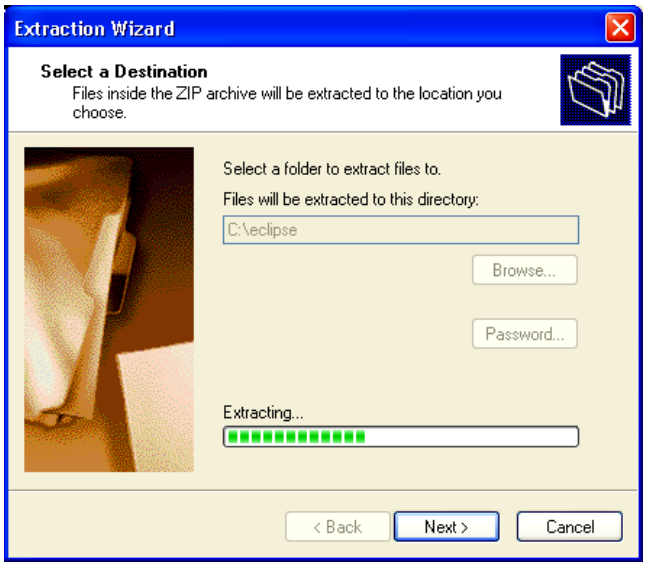
On the left-hand screen below, click on "**Extract all files**".
Click "**Next**" on the rightmost screen below to start the Windows extraction wizard.



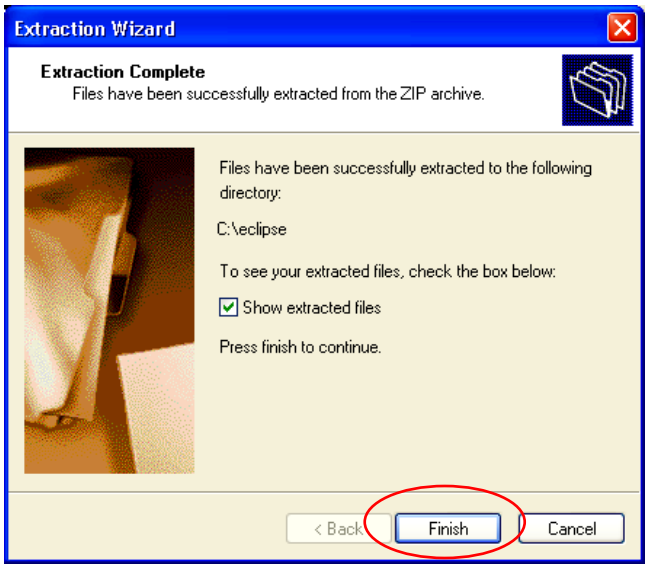


On the window on the left below, enter the folder "**c:\eclipse"** as the destination directory. Click "**Next**" to start the extraction process.

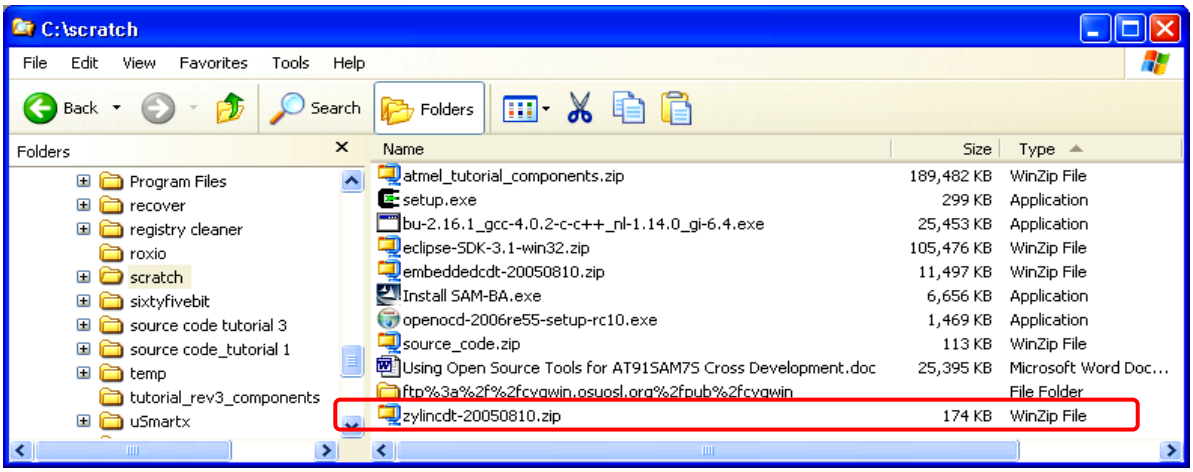




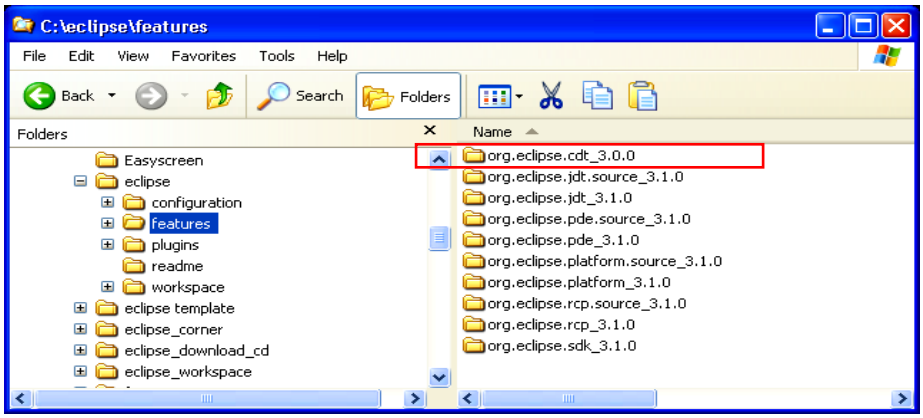
When the extraction completes, click on "**Finish**" to end the Extraction Wizard.



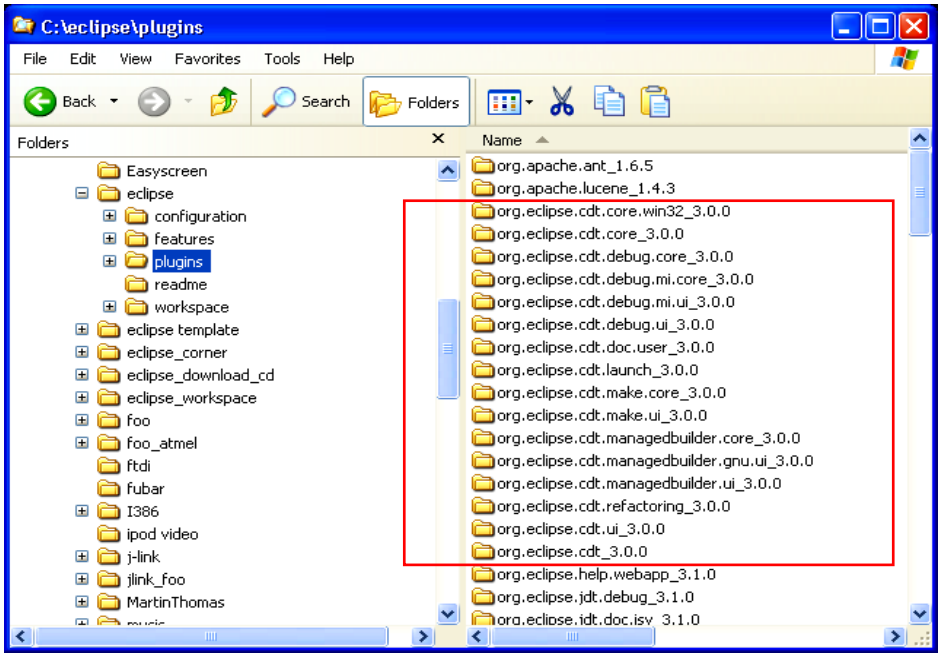
Now repeat the above steps to extract the second file "**zylincdt-20050810.zip**" to the "**c:\eclipse**" folder.



When extraction completes, check the **c:\eclipse\features** folder; there should be a CDT folder present, as shown below.

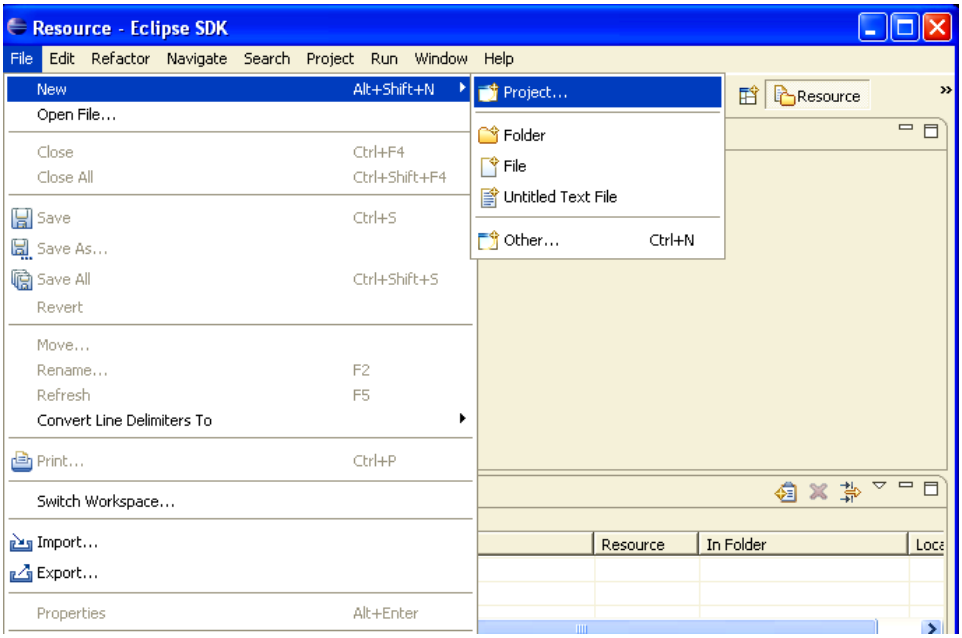


Likewise, check the **c:\eclipse\plugins** folder, there should be several CDT folders present, as shown below.

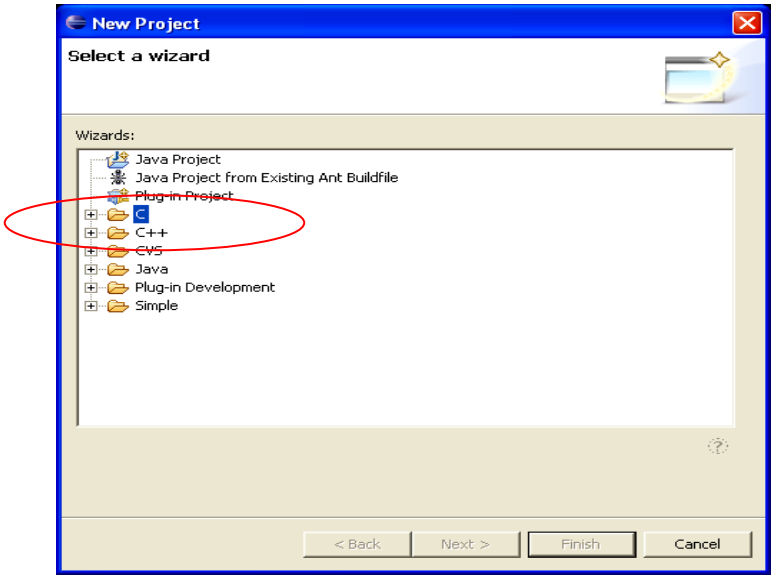


To verify that **CDT was** installed properly, start Eclipse by clicking on the desktop icon.

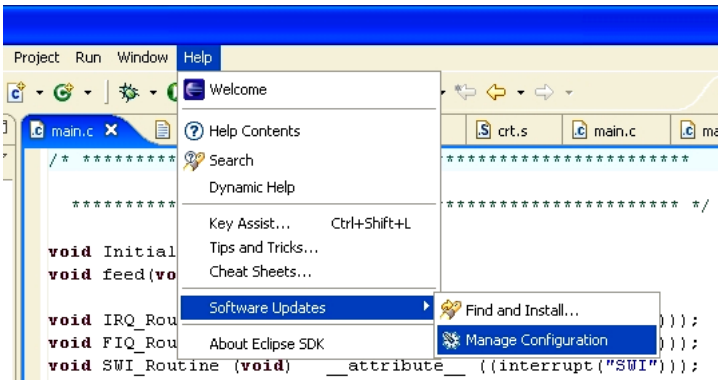
When Eclipse starts, click on "**File – New - Project…**"



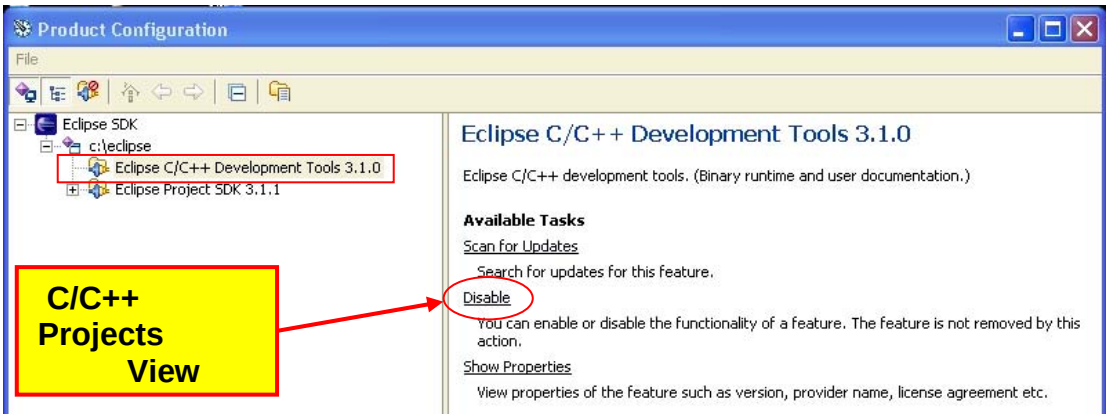
When the New Project window appears, check if C and C++ appear as potential projects. If this is true, Eclipse CDT has been installed properly.



If you don't see the C and C++ listed, here's what might have happened. It's possible to disable the CDT plug-in. To see where this may be done, click "**Help – Software Updates – Manage Configuration**".



If you click on **Eclipse C/C++ Development Tools 3.1.0**, you will see an option to **disable** the CDT plug-in. If this has been disabled, use these menus to reverse this situation.



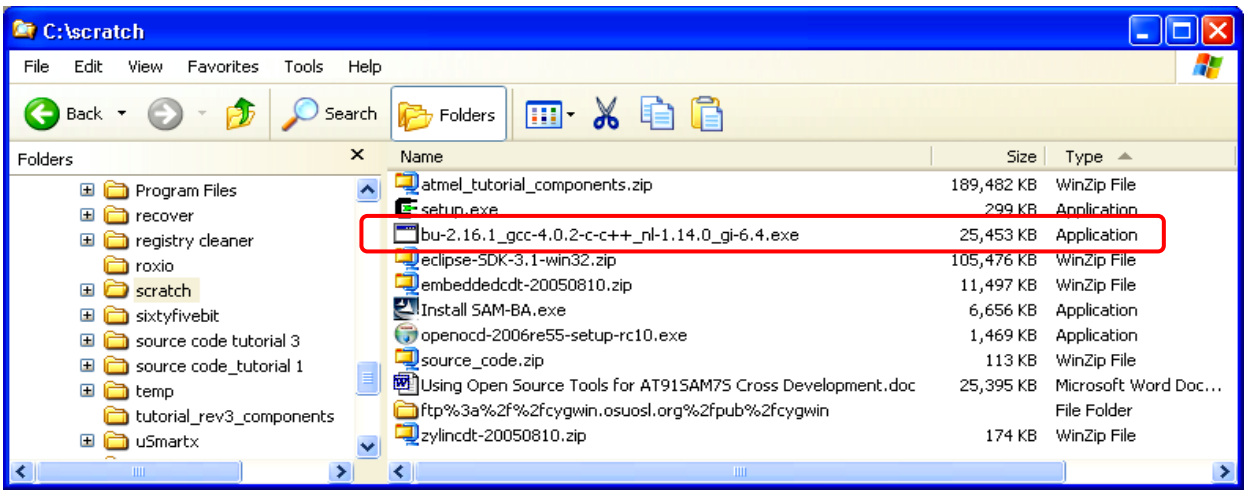
Install the GNUARM Compiler

There are a number of pre-built GNU ARM compiler toolsets available on the web. For this tutorial, we will be using the **GNUARM** pre-built ARM compiler tool suite developed by Rick Collins and Pablo Bleyer Kocik.

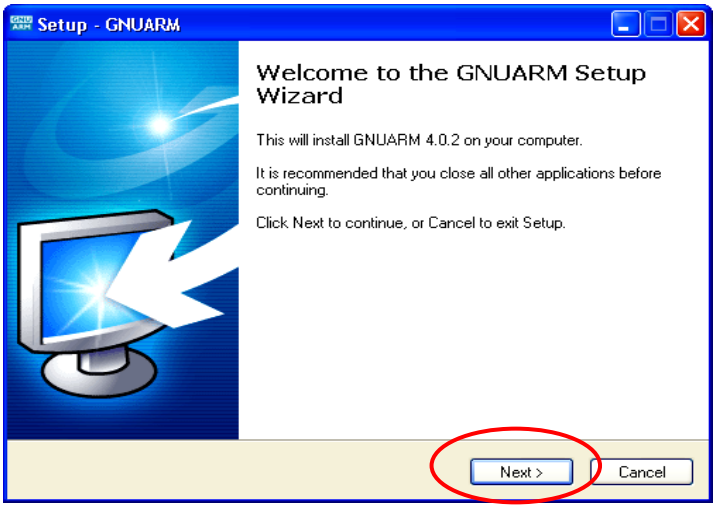
Be sure to visit the GNUARM web page: **www.gnuarm.com**

In the **c:\scratch** folder, double click on the GNUARM installer file to start installation of the ARM compiler suite.

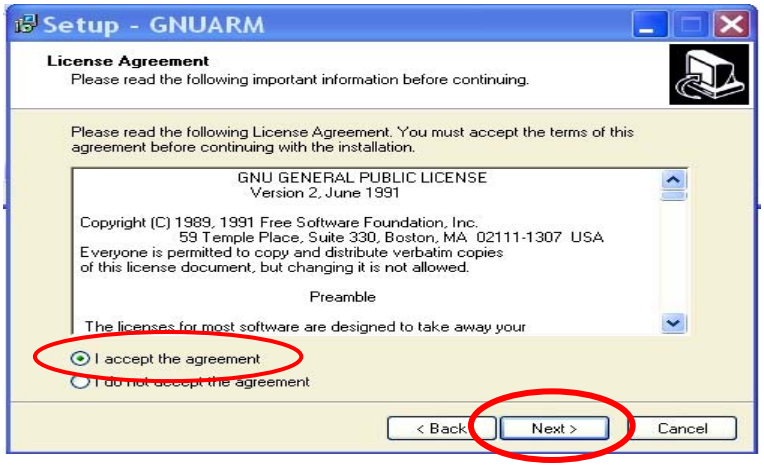
bu-2.16.1_gcc-4.0.2-c-c++_nl-1.14.0_gi-6.4.exe ← GNUARM installer



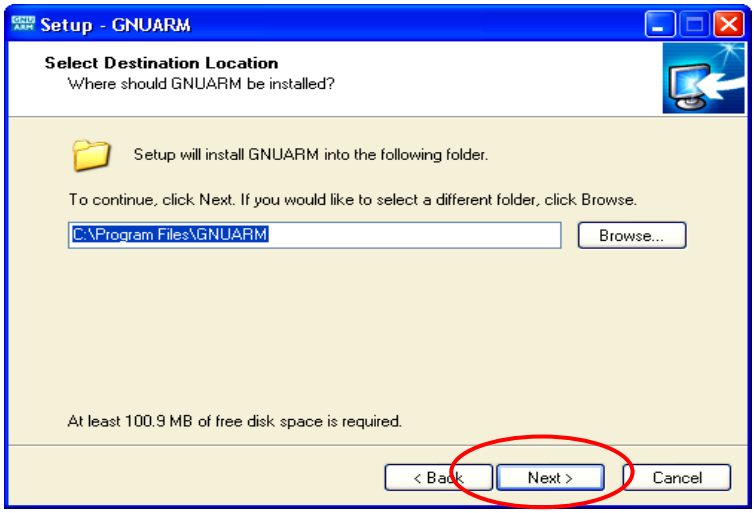
The GNUARM installer will now start. Click **"Next"** to continue.



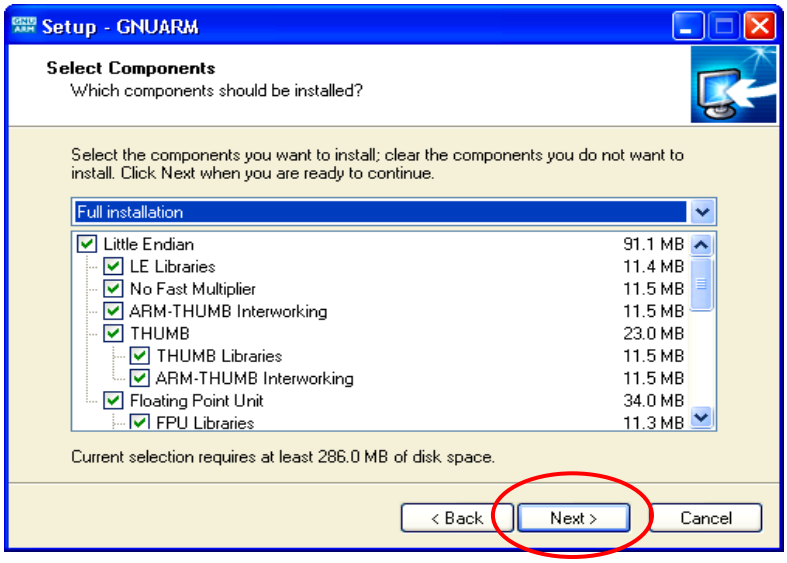
Accept the GNU license agreement – don't worry, it's still free. Click **"Next"** to continue.



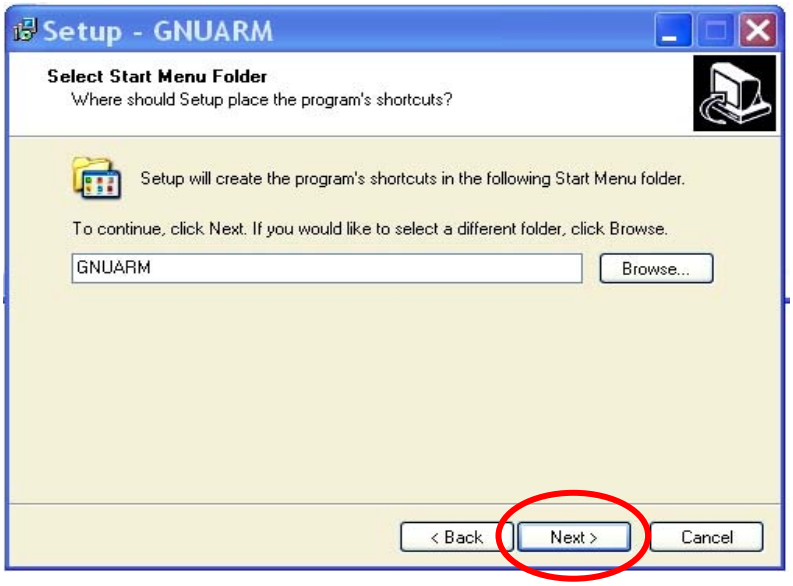
We'll take the default and let it install into the "**Program Files**" directory. Click **"Next"** to continue.



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

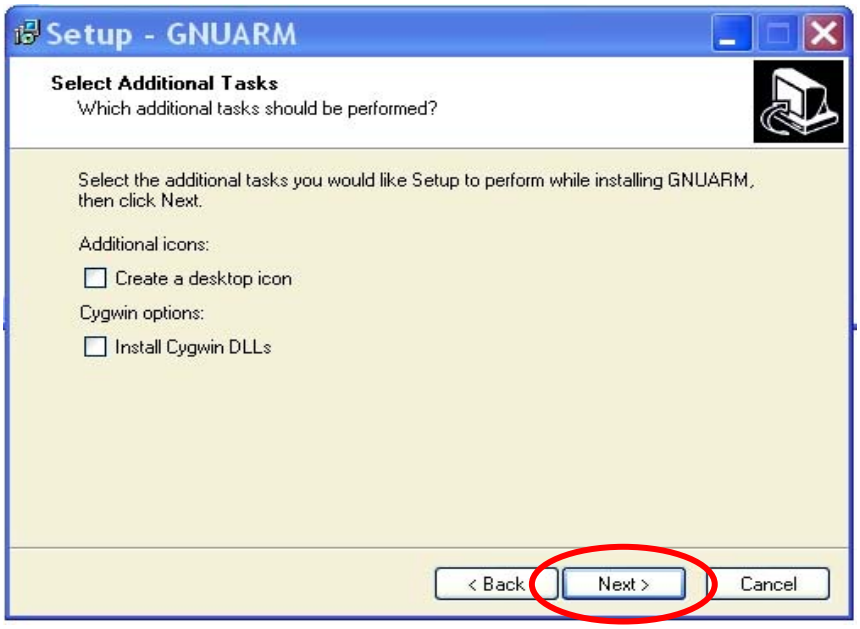


Take the default on this screen. Click **"Next"** to continue.

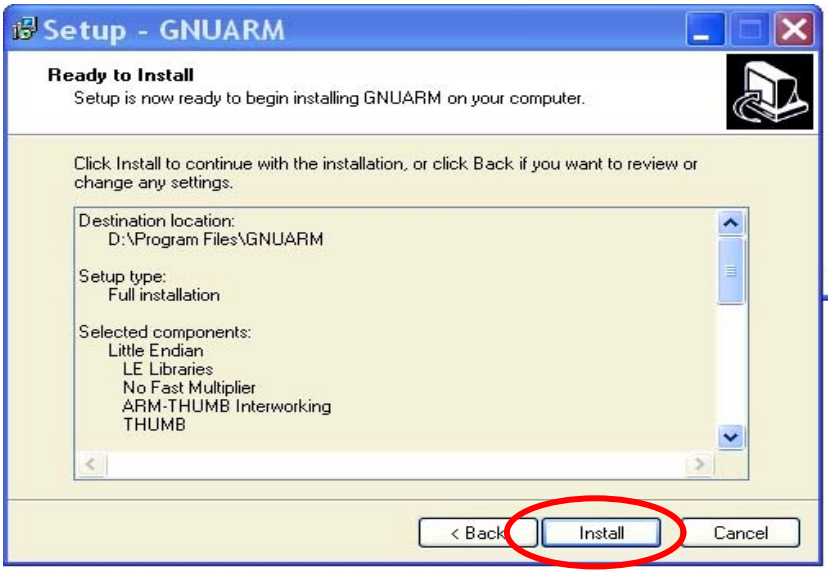


It's very important that you don't check **"Install Cygwin DLLs**" below. We already have the Cygwin DLLs installed from our Cygwin environment installation. In fact, the ARM message boards have had recent comments suggesting that the Cygwin DLL installation from within GNUARM has some problems.

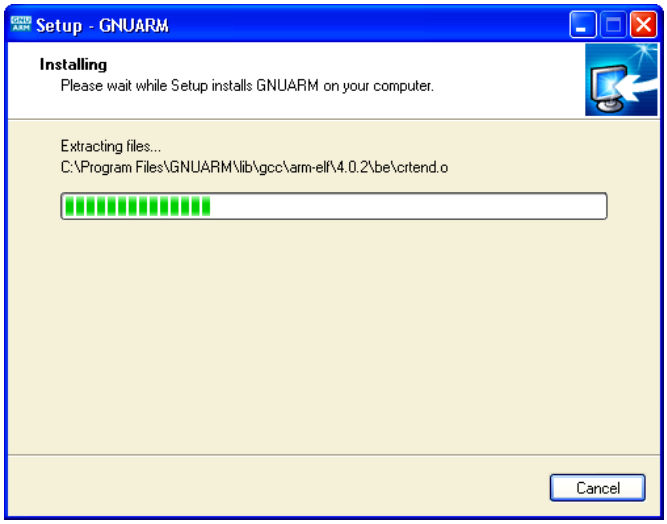
Since all operations are called from within Eclipse, we don't need a "**desktop icon**" either. Click **"Next"** to continue.



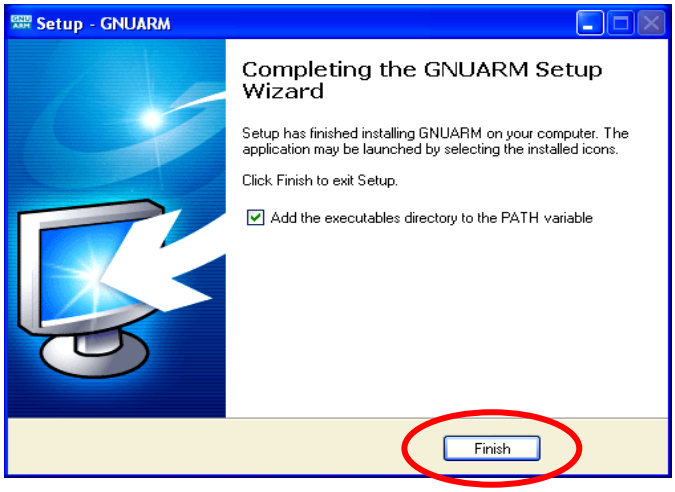
Click on “**Install**” to start the GNUARM installation.



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.



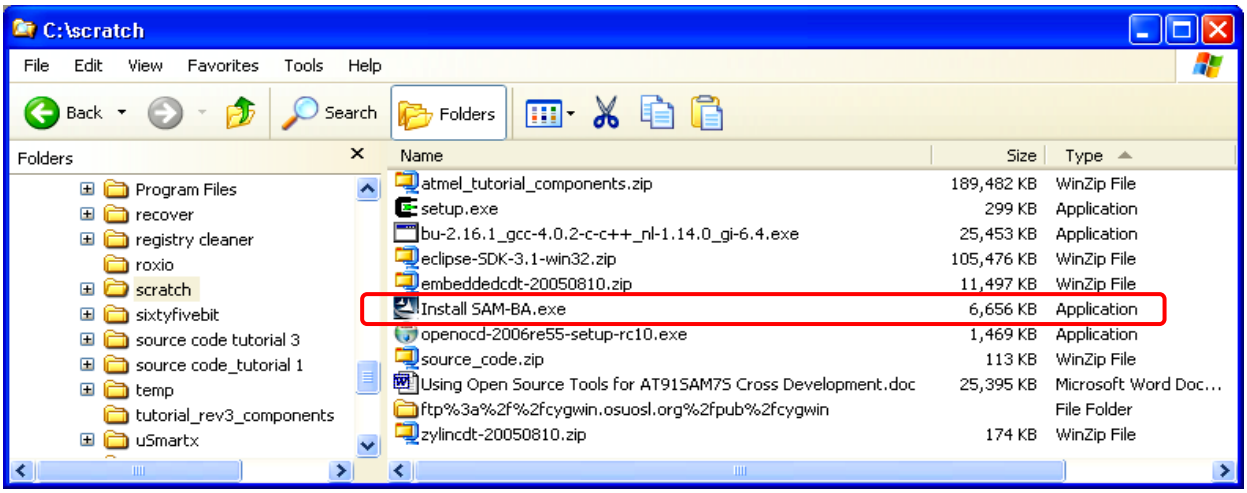
This completes the installation of the compiler suites. Since Eclipse will call these components via the make file, you won't have to think about it again.

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.

Atmel SAM-BA Flash Programmer

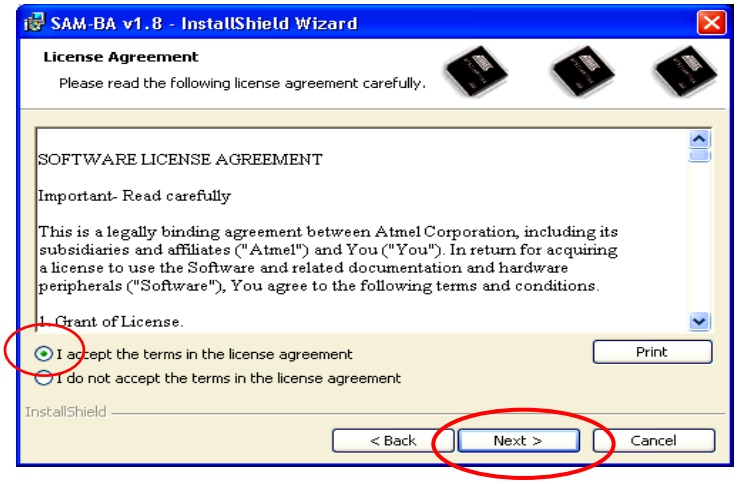
The Atmel SAM-BA flash programmer is a manufacturer-supplied utility to download the user's program into the AT91SAM7S microprocessor's flash memory via the USB interface. The Atmel SAM-BA flash programmer installer is also in the **c:\scratch** folder.

Click on the file "**Install SAM-BA.exe**" to run the installer.

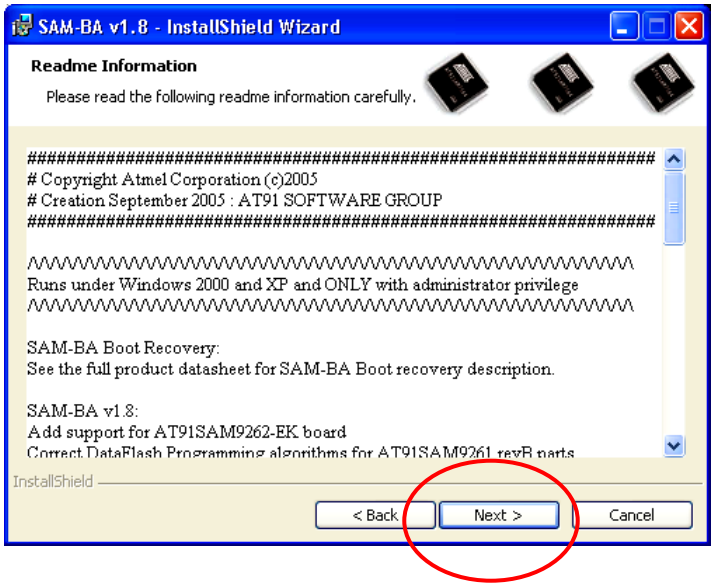


Click "**Next**" in the Wizard below to start installation.

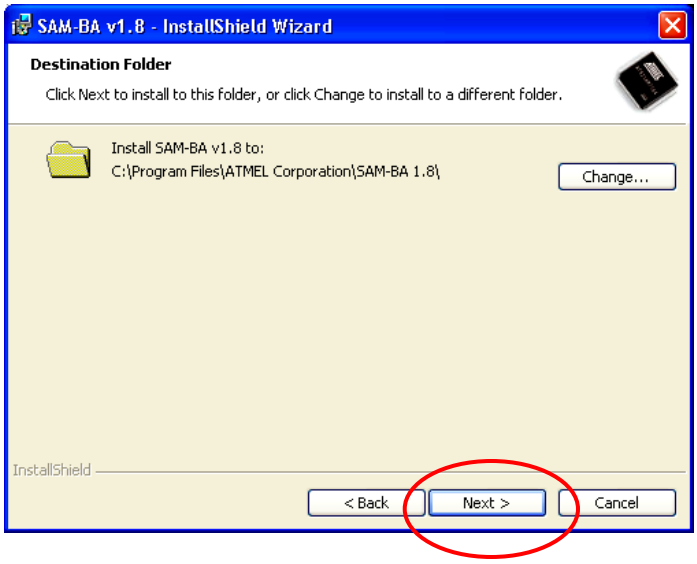
Accept the terms of the license agreement and click “**Next**”.



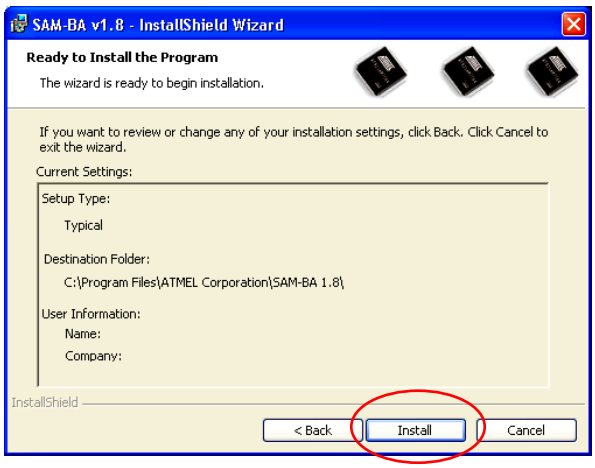
Click “**Next**” to bypass reading the Readme information.



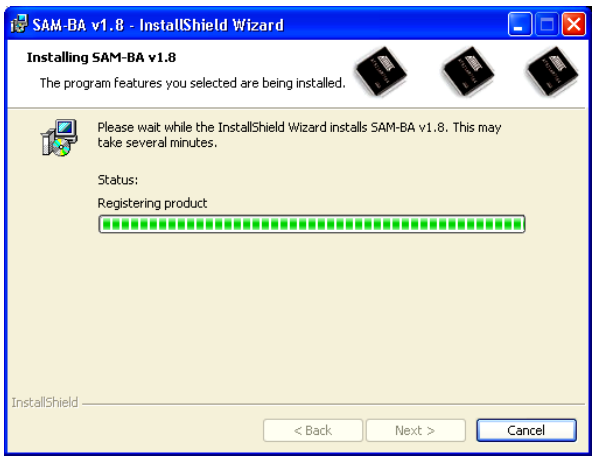
Take the default installation directory in c:\Program Files\ below; click “**Next**” to continue.



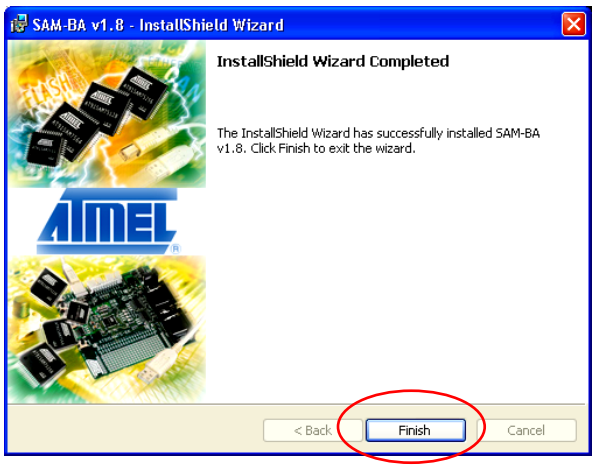
Click on “**Install**” to start the actual installation.



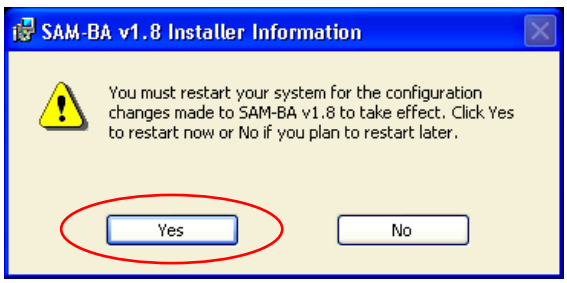
The Installer will now run without any further user intervention.



When the SAM-BA installer completes, Click “**Finish**” and we’re done.



You can either restart Windows now or wait until later.



OpenOCD Installation

Eclipse/CDT has a fabulous graphical debugger that is built on top of the venerable GNU **GDB** command line debugger. The only problem is how to connect it to a remote target such as a microprocessor circuit board. **GDB** communicates to the target via a Remote Serial Protocol that can be utilized over a serial port or an internet port.

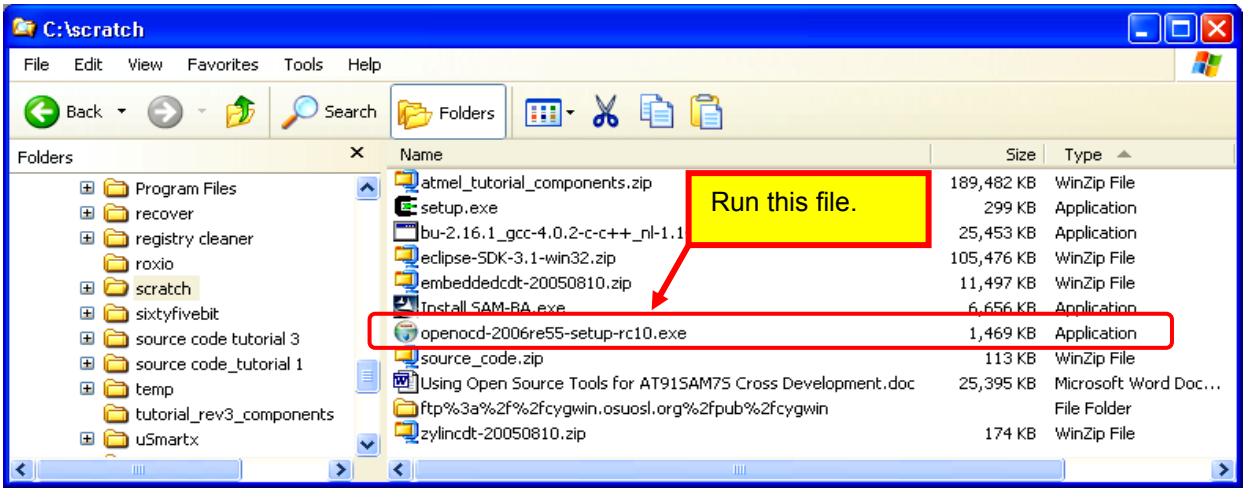
In the past, most people have used the Macraigor **OCDRemote** utility that reads **GDB** serial commands and manipulates the ARM JTAG lines using the PC's parallel port and a simple level-shifting device called a "wiggler". The Macraigor **OCDRemote** utility has always been available for free (in binary form) but it is not Open Source. Macraigor could withdraw it at any time.

To the rescue is German college student Dominic Rath who developed an open source ARM JTAG debugger as his diploma thesis at the University of Applied Sciences, FH-Augsburg in Bavaria. Dominic's thesis can be found here: http://openocd.berlios.de/thesis.pdf

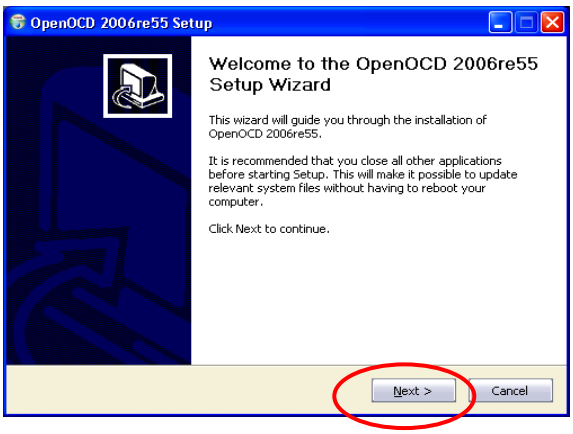
Dominic also has a website on the Berlios Open Source repository here: http://openocd.berlios.de/web/

In the true spirit of Open Source, many people are jumping on the OpenOCD bandwagon; notably Michael Fischer of Lohfelden, Germany. Michael has prepared a very nice installation package for OpenOCD. Michael also has a very interesting web site here: http://www.usbdip.de/index_en.html
Michael's web site has some excellent tutorials on this very subject (open source cross development tools and Eclipse), so please stop by his web site and read them!

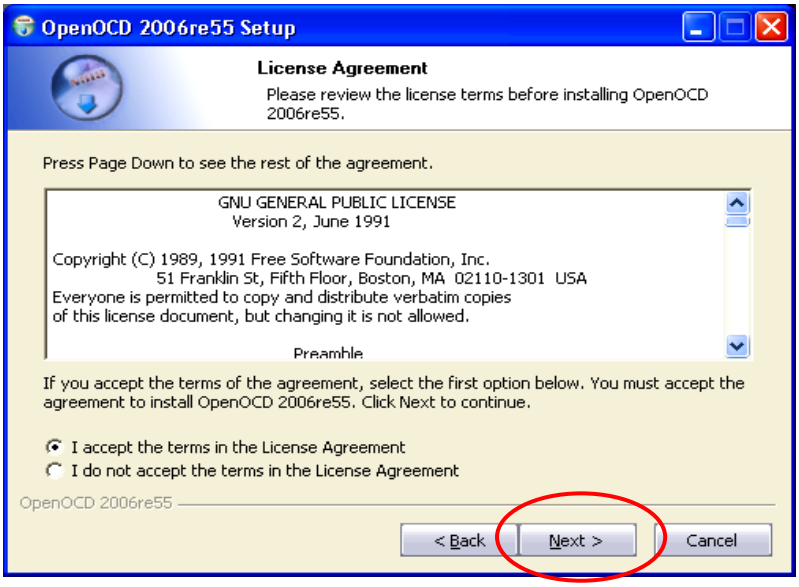
Run the file "**openocd-2006re55-setup-rc10.exe**" contained in the components folder within **C:\scratch**. You can use Windows Explorer to do this.



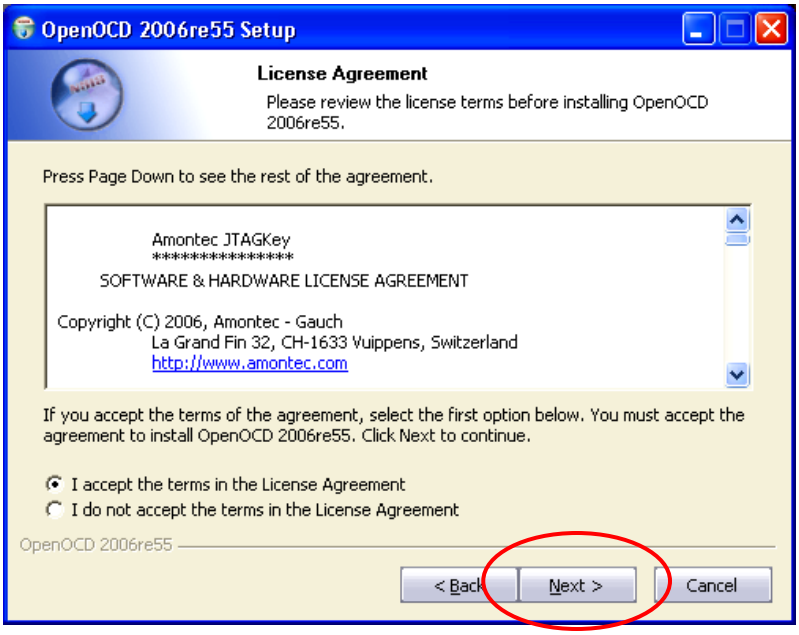
Michael Fischer's **OpenOCD Installer** will start. Click "**Next** to continue.



Accept the License Agreement by clicking the radio button as shown below and click "**Next**" to continue.

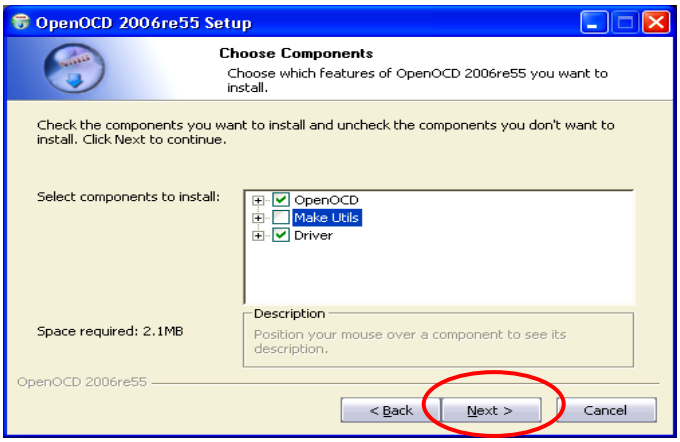


Likewise, accept the Amontec license agreement and click "**Next**" to continue. Amontec manufactures the **JTAGkey**, which is a professionally-developed version of the USB-JTAG interface described in Dominic Rath's thesis. This device retails for $177.00 (US) and is reliable and provides a significant speed boost over the parallel-port wiggler. Using the Amontec JTAGkey is covered later in this tutorial.

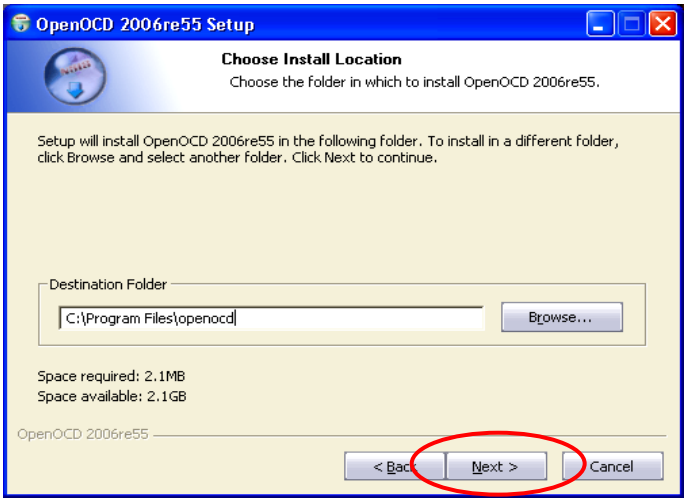


Fischer's OpenOCD installer provides everything needed to operate OpenOCD with a "wiggler" and the more advanced "JTAGkey".

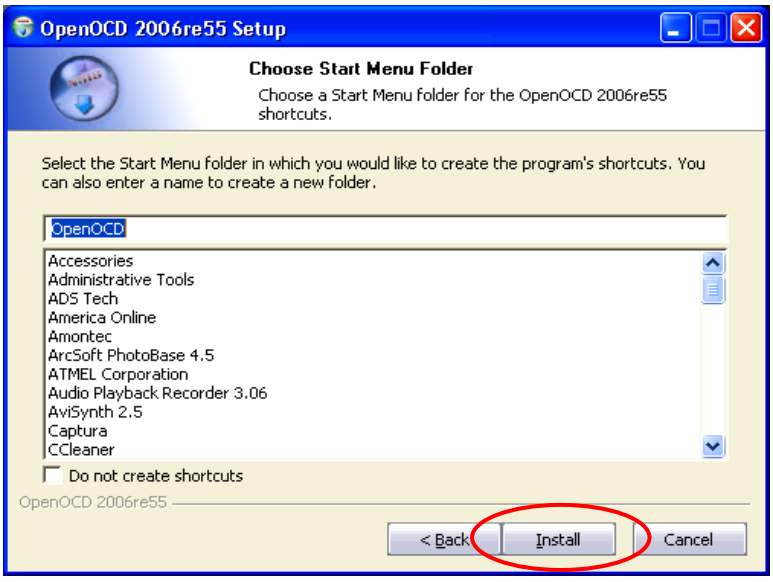
In the “**Choose Components**” dialog, select “**OpenOCD**” and “**Driver**”. We already have the Make utility installed via Cygwin. Click “**Next**” to continue.



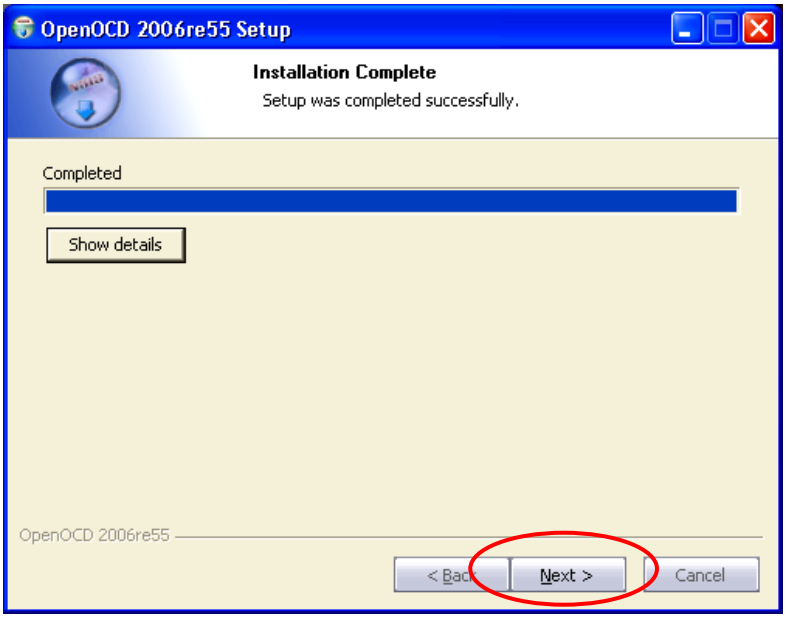
In the “**Choose Install Location**” dialog, I simplified the destination location to “**c:\Program Files\openocd**”. Click “**Next**” to continue.



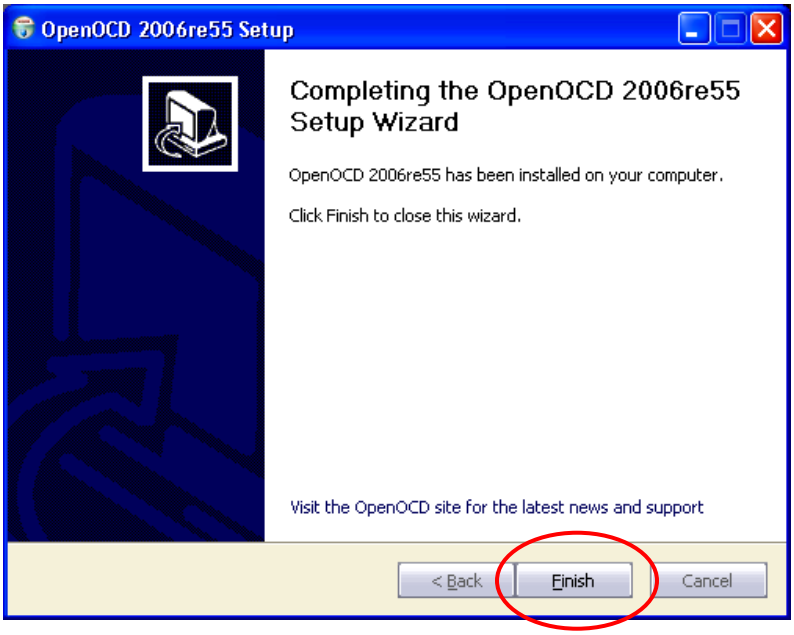
OpenOCD can be placed into the Windows Start menu by taking the default on the “**Choose Start Menu Folder**” screen below. Click on “**Install**” to start the OpenOCD installation.



The OpenOCD installation will only take a few seconds and the "**Installation Complete**" screen will advise you when it has finished. Click "**Next**" to continue.



Finally, click on "**Finish**" on the screen below to complete installation of **OpenOCD**.



One thing to note is that the OpenOCD Installer also made an entry into the Windows PATH environment variables. It added the following path: **C:\Program Files\openocd\bin**

This path includes the OpenOCD executable and suitable configuration files for the "wiggler" and the JTAGkey.

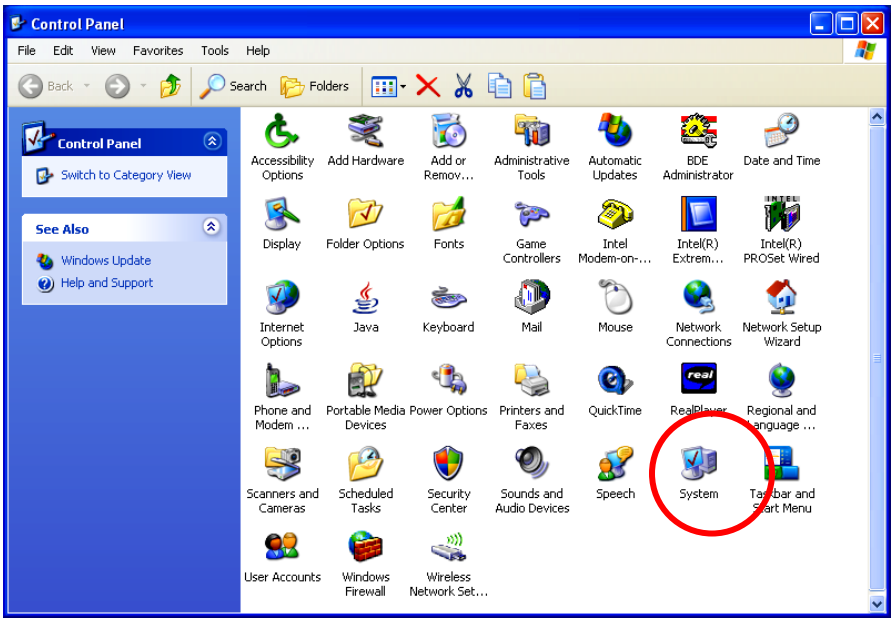
Verifying the PATH Settings

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 **OpenOCD** JTAG server.

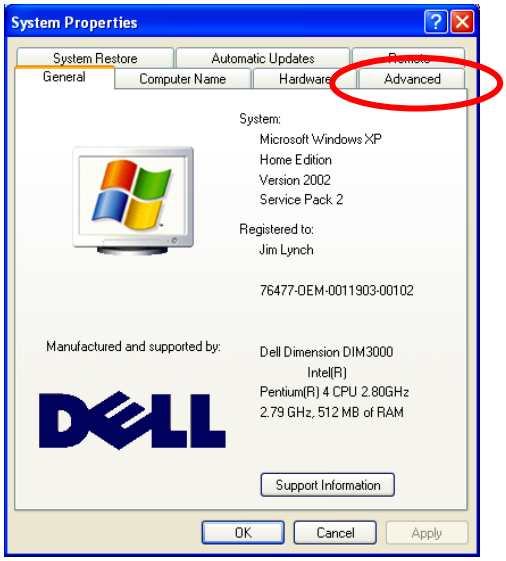
These are the three paths that **must** be present in the Windows environment:

c:\cygwin\bin
c:\Program Files\gnuarm\bin
C:\Program Files\openocd\bin

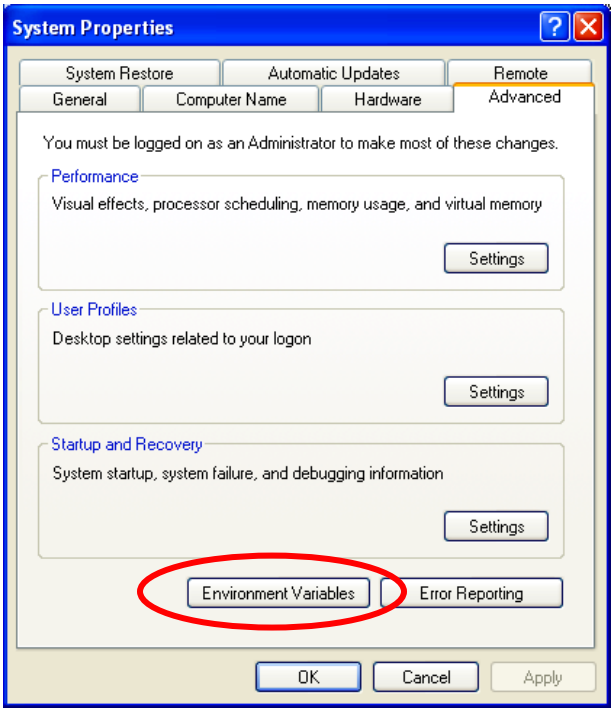
To verify that these paths are present in Windows and to make changes if required, start the Windows Control Panel by clicking "**Start – Control Panel**" and then click "**System**".



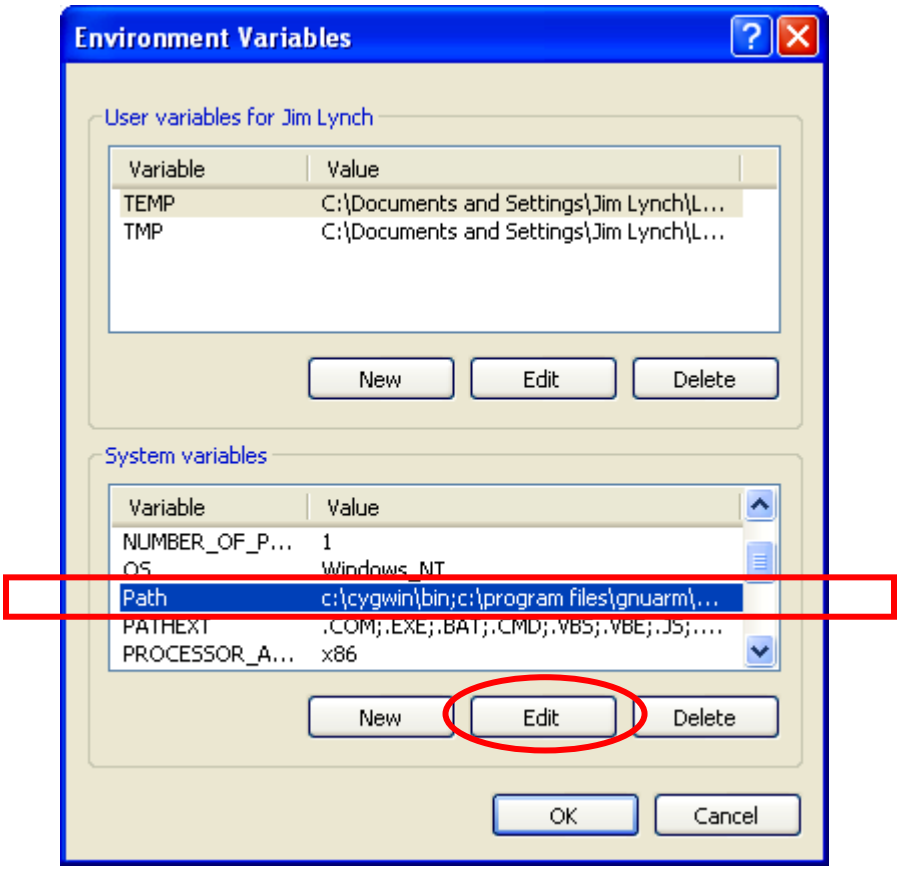
Now click on the "**Advanced**" tab below.



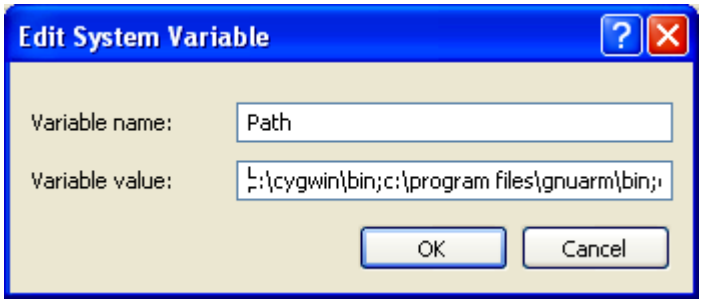
Now click on the “**Environment Variables**” button.



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 of the required paths interspersed throughout the path specification. I typically use 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:\Program Files\openocd\bin

If any of these paths are 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.

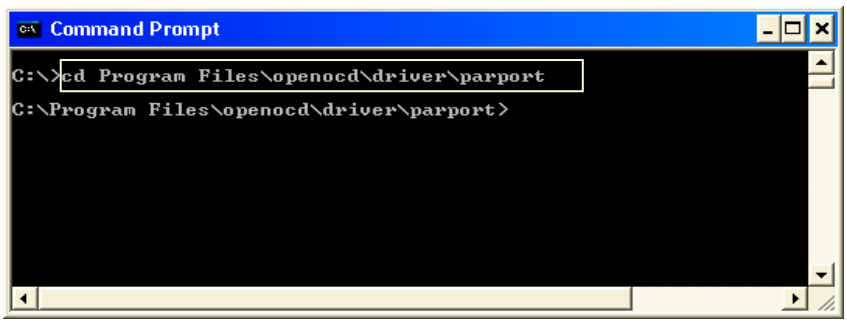
Install the GIVEIO Driver

To enable OpenOCD to use the parallel port when the “wiggler” interface is employed, a special **giveio.sys** driver has to be installed. This only needs to be done once.

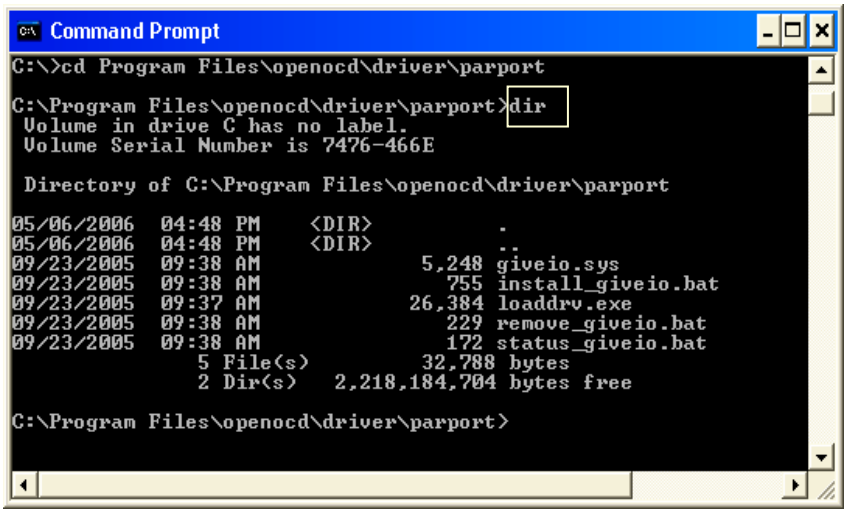
Start the installation of the giveio.sys driver by opening up a **Command Prompt** window (for really experienced readers, that’s the old DOS window). The “command prompt” can be found in your Windows “**Start – All Programs – Accessories**”.

If your Command Prompt window is not at the root folder (**c:**) shown above, you can type in the command **>CD ** to locate yourself at the root folder. Now change to the directory **c:\Program Files\openocd\driver\parport.**

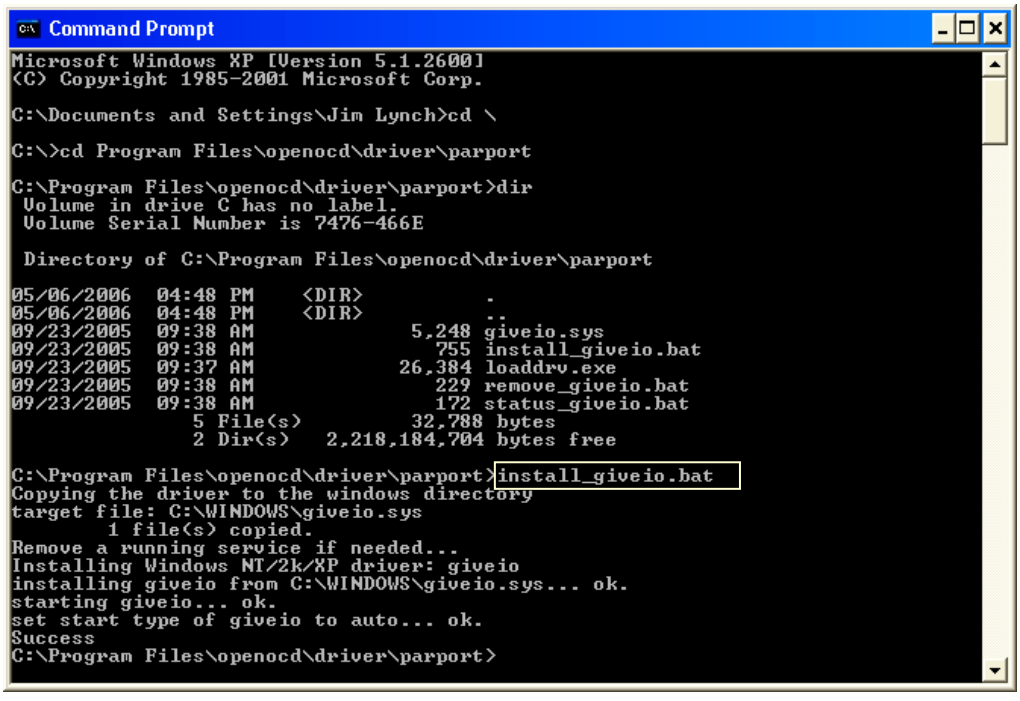
Note that we typed the command: **cd Program Files\openocd\driver\parport** to do this.



Now take a look at the contents of this folder by typing the **DIR** command.



We want to run the command batch file "**install_giveio.bat**", this will install the giveio.sys driver and load and start it. This is a permanent installation; you won't have to do this again. The batch file may be run by entering its name on the command line and hitting "**Enter**". As you can see from the command history below, **giveio** was successfully installed as a Windows driver.

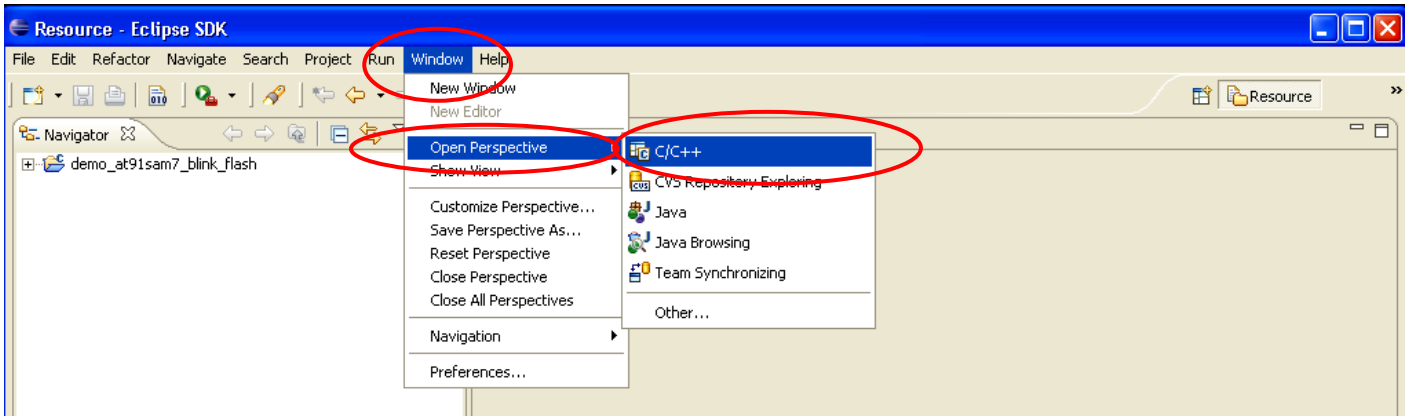


Install Atmel SAM-BA Boot Assistant as an Eclipse External Tool

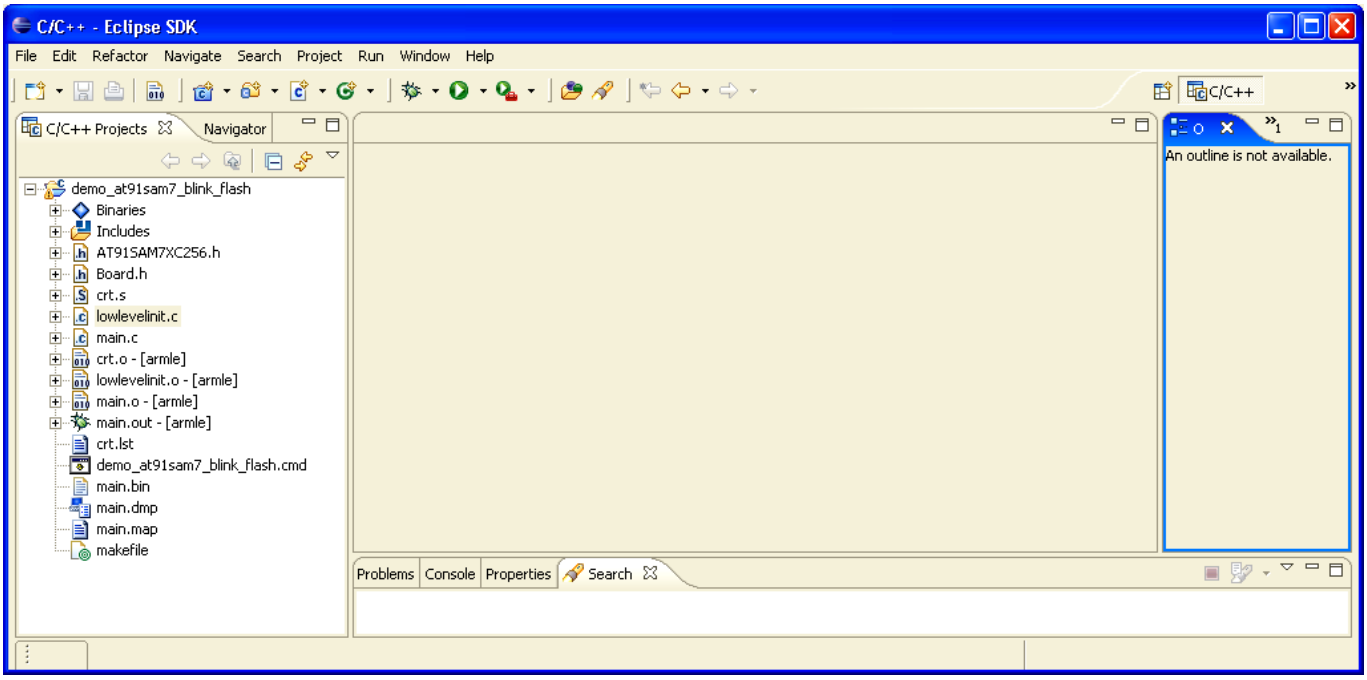
The Atmel **SAM-BA** Boot Assistant program (flash programming utility) should be installed as an Eclipse "**External Tool**". This will make it easy to program the AT91SAM7S-256 flash memory via the USB channel without having to exit Eclipse.

First, start Eclipse and switch to the C/C++ perspective. Start Eclipse by clicking on the desktop icon.

When Eclipse starts up, click on "**Window – Open Perspective – C/C++**" to switch to C-Language programming. If "**C/C++**" does not appear, locate it by clicking "**Other…**"

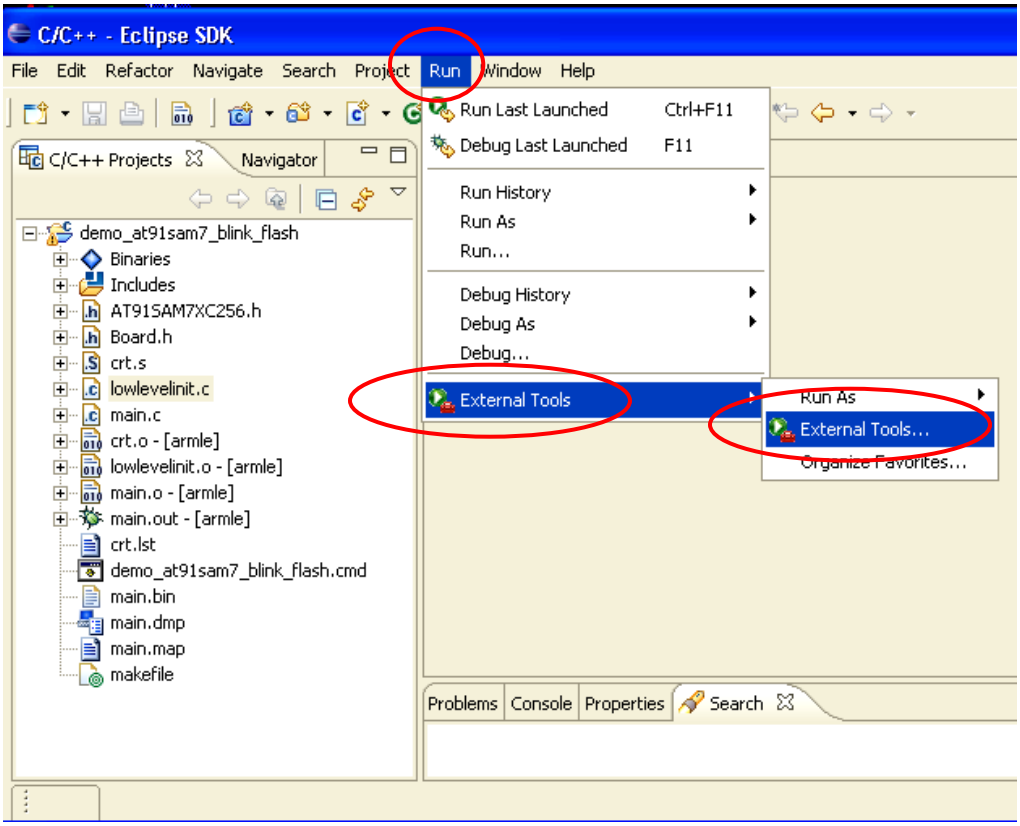


Now the C/C++ perspective should display and Eclipse will remember your choice whenever you restart.

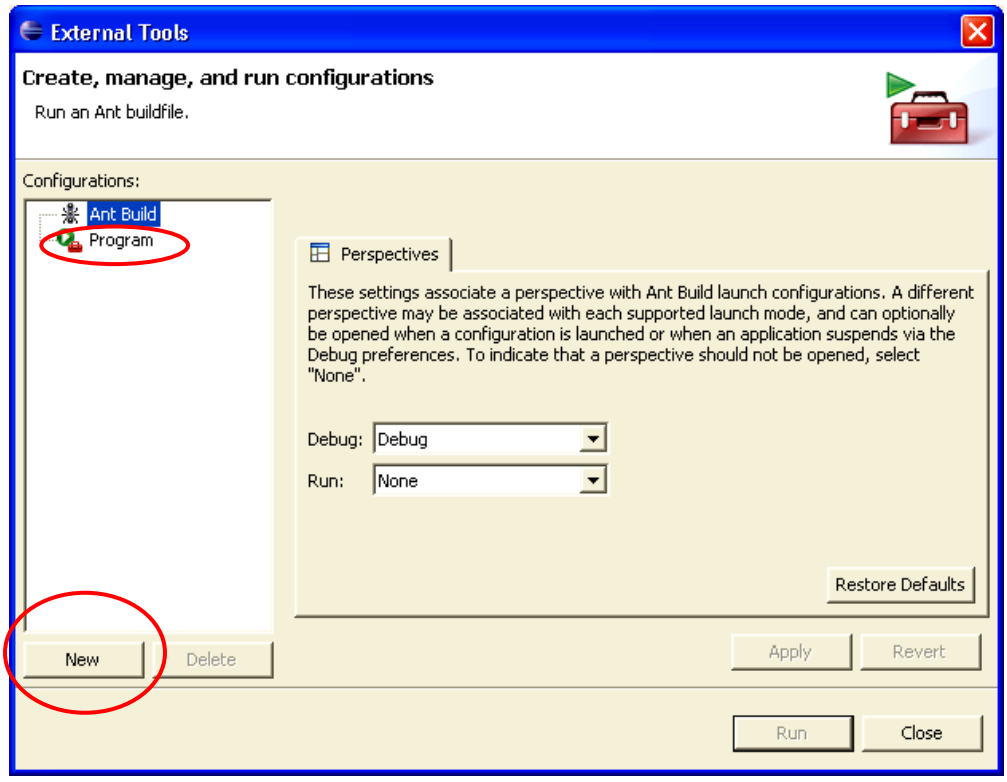


By installing **SAM-BA** as an "**External Tool**", we can start the **SAM-BA** utility conveniently from the **Run** pull-down menu.

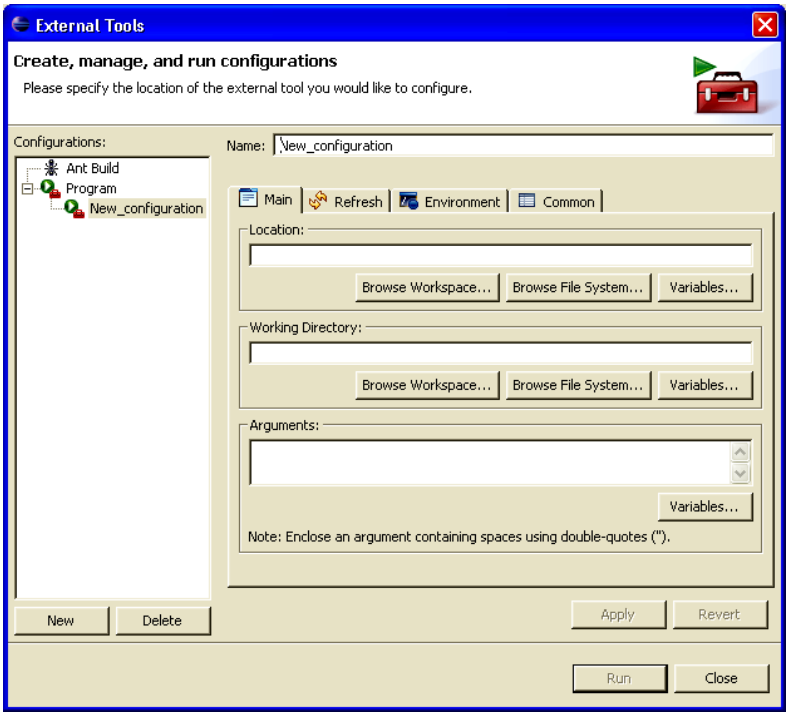
To set up **SAM-BA** as an "**External Tool**", click on "**Run – External Tools – External Tools…**".



When the "**External Tools**" window appears, click on "**Program**" followed by "**New**".

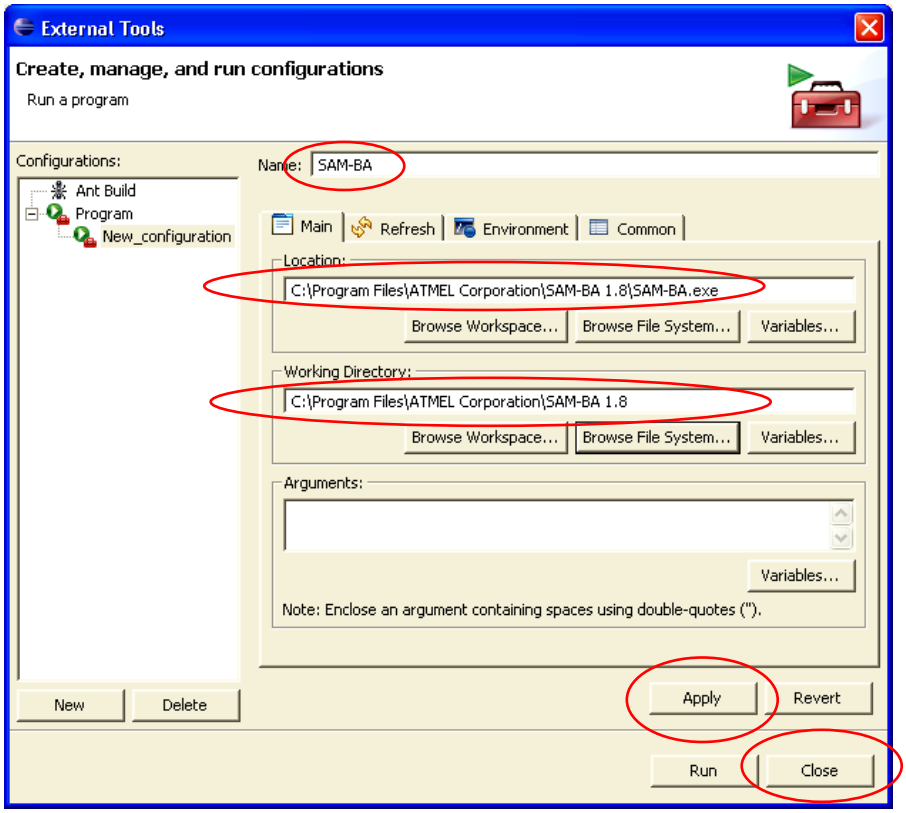


The “**External Tools**” window changes to the following form.

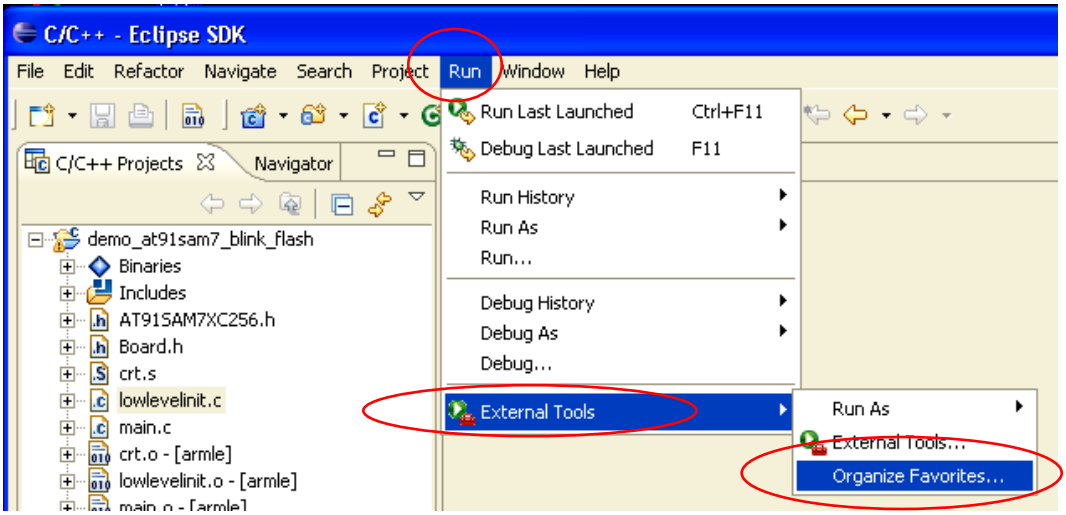


Now fill out the fields within the form as shown below. The SAM-BA executable can be found in the **c:\Program Files\ATMEL Corporation\SAM-BA** folder. Use the “**Browse**” buttons to find everything.

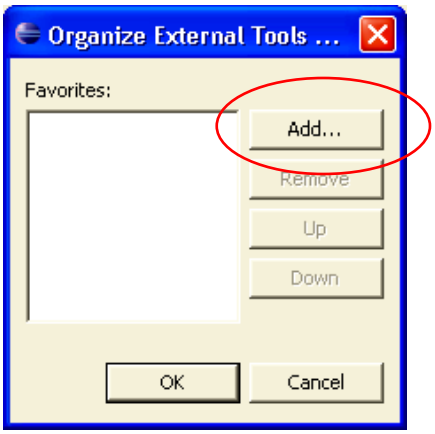
The Refresh, Environment, and Common tabs can be left at their default state. Click on “**Apply**” followed by “**Close**” to complete installation of **SAM-BA** as an external tool.

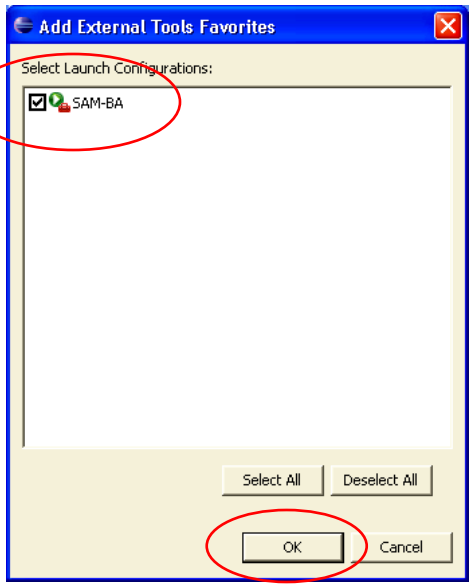


There's one more step in defining **SAM-BA** as an External Tool. We need to install it into the list of "favorites". Click on "**Run – External Tools – Organize Favorites…**".

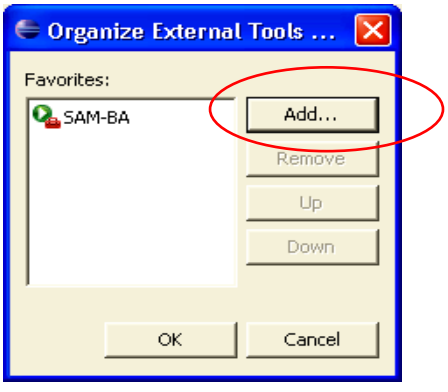


In the "**Organize External Tools…**" window, click on "**Add…**". On the subsequent window, check the box for **SAM-BA** and click "**OK**".

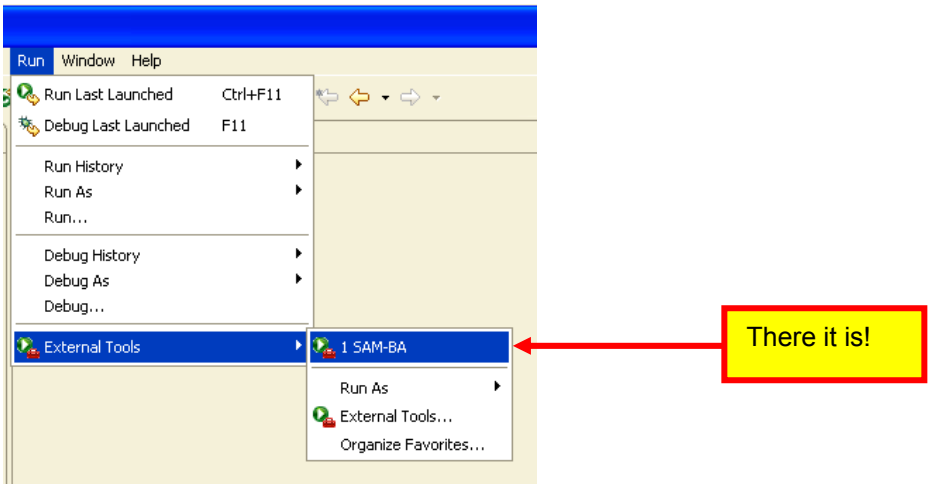




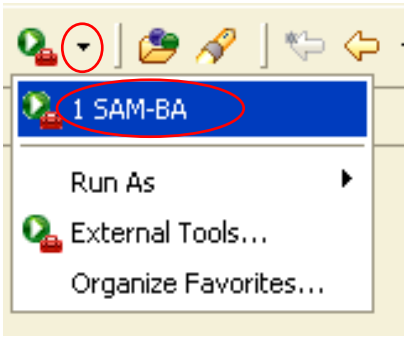
Now click "**OK**" on the "**Organize External Tools…**" and **SAM-BA** will now appear in the RUN pull-down menu.



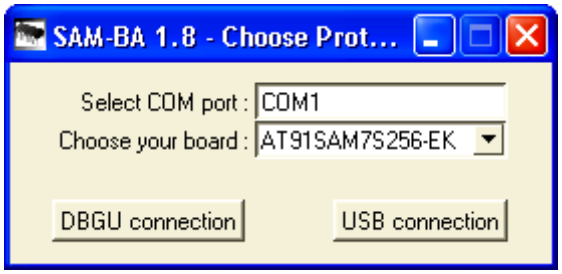
Now when we click on the **Run** pull-down menu and select "**External Tools**", we see that **SAM-BA** is present at the top of the pull-down list (a favorite).



An even more convenient way to start **SAM-BA** is to use the "E**xternal Tools**" tool button in the toolbar. Specifically click on the down arrow to reveal the external tools that have been set up. Eclipse remembers the last External Tool you selected, so clicking on just the red toolbox will start the previous tool.



By clicking on **SAM-BA**, the Atmel Flash Programming utility will start and show its Comms setup window, as shown below.



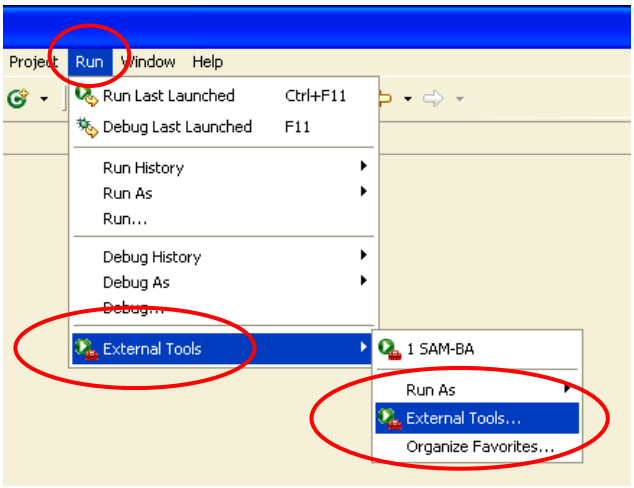
Cancel the **SAM-BA** for now; we will operate it when we have a file to download into flash.

Install OpenOCD as an Eclipse External Tool

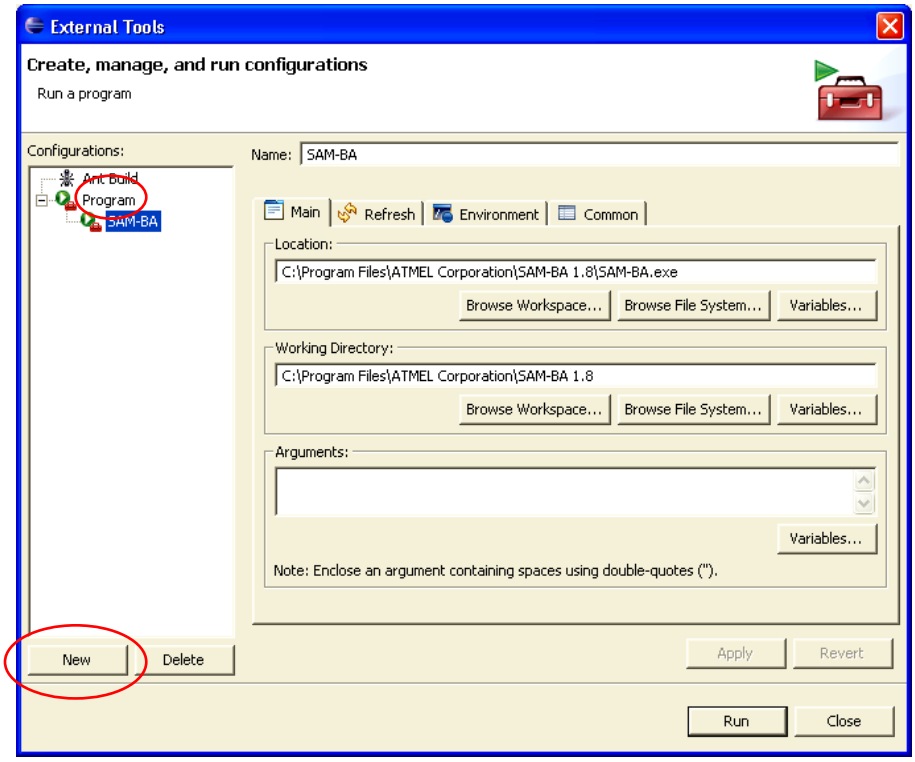
The Eclipse debugger interfaces to the GNU open-source GDB debugger. The GDB debugger can be commanded to interface to a remote target using a well-defined serial protocol called GDB Remote Serial Protocol. The **OpenOCD** utility developed by Dominic Rath reads the Remote Serial Protocol and converts each debugging command to a JTAG operation. For this tutorial, the JTAG connection is made through a simple level-shifter connected to the PC's printer port. This $20 device is called a "wiggler" and it manipulates the AT91SAM7S JTAG debugger pins to start/stop execution, set breakpoints, read and write to memory, etc.

We will now install **OpenOCD** as an Eclipse "**External Tool**" so that it will be very convenient to start it from within the Eclipse environment.

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

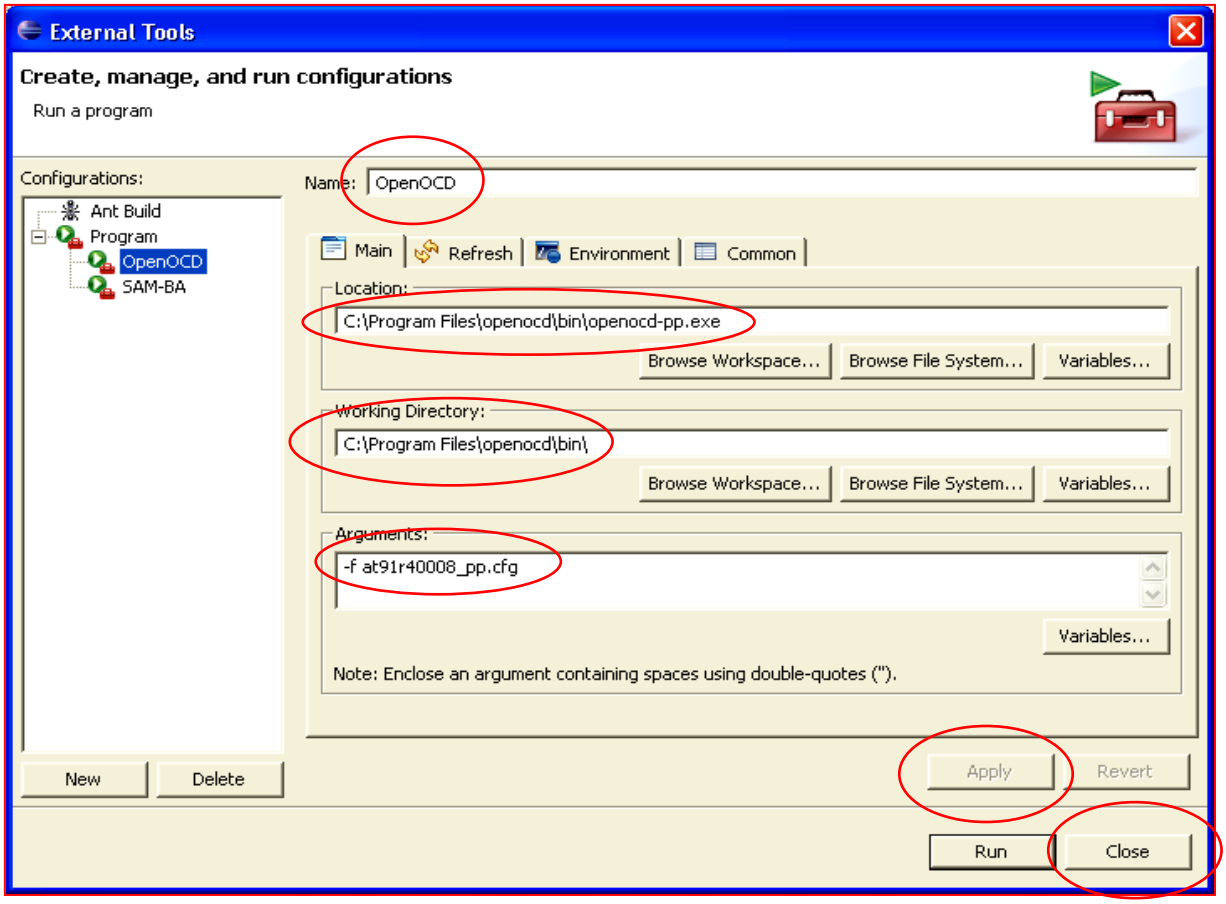


The "**External Tools**" window will appear. Click on "**Program**" and then "**New**" to establish a new External Tool.

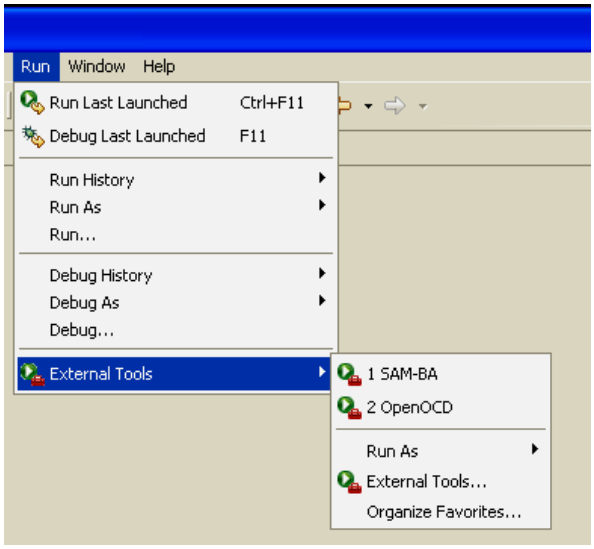


Fill out the "**External Tools**" form exactly as shown below. Click on "**Apply**" and "**Close**" to register **OpenOCD** as an external tool. Note that we included in the arguments pane a reference to the configuration file for the "wiggler".

Remember that we downloaded and installed **OpenOCD** earlier. The executable file (**openocd-pp.exe**) and the wiggler configuration file (**at91r440008_pp.cfg**) are in the "**c:\Program Files\openocd\bin**" folder. The configuration file, while designed for an AT9140008 processor, will work fine for the AT91SAM7 family also.



Using the same techniques used earlier when we installed **SAM-BA** as an external tool and added it to the list of "favorites", add **OpenOCD** to the list of favorite external tools. When you click "**Run – External Tools**", you should see that **OpenOCD** is in the favorites list also.

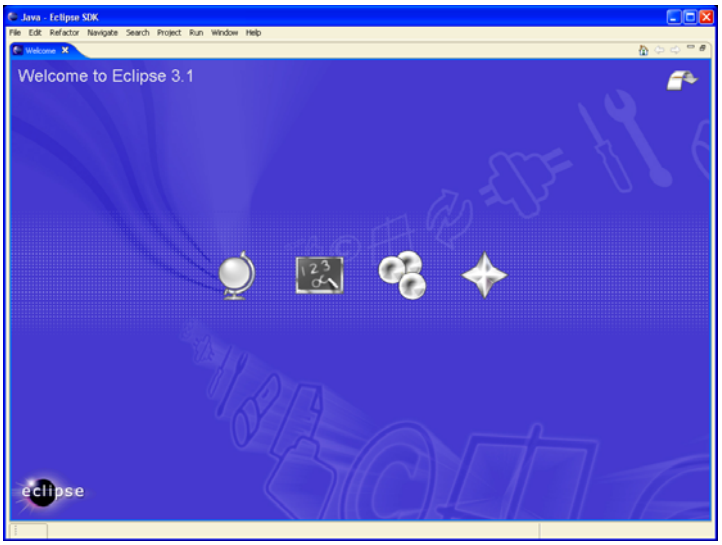


Create an Eclipse Project

Now all our hard work preparing an open source Eclipse tool set will pay off. We can now actually create a bona fide Atmel AT91SAM7 application using the Eclipse IDE and the open source compilers and debuggers.

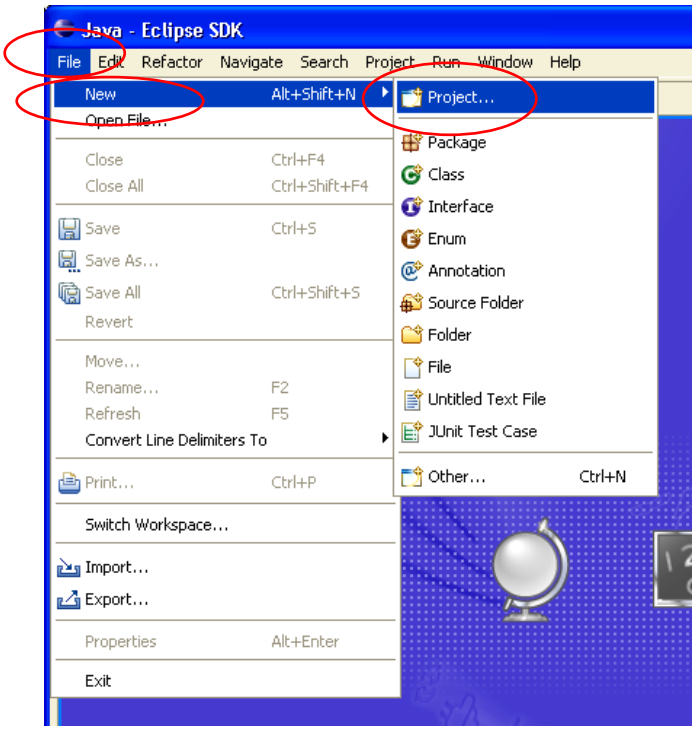
Click on the desktop Eclipse icon to start Eclipse. 

You should see the following screen.

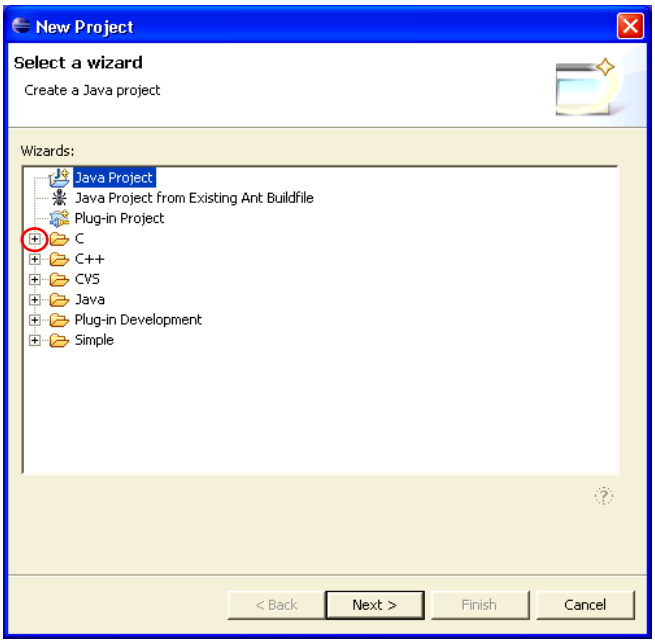


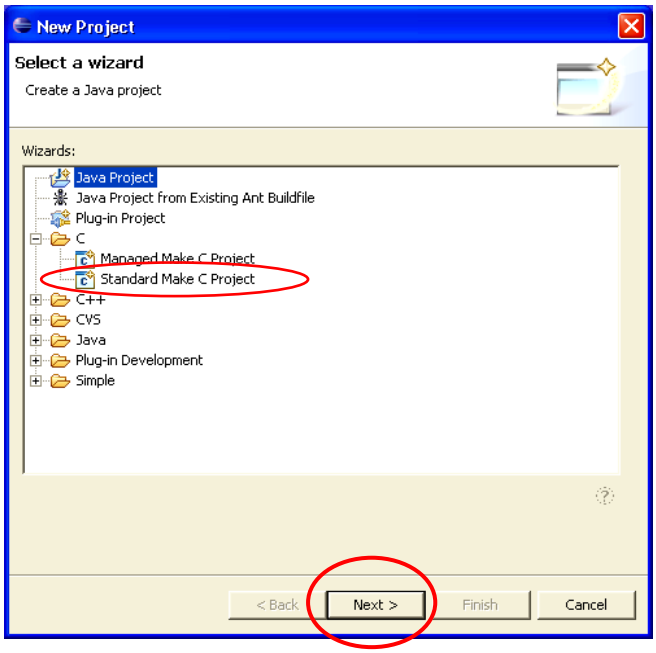
The four buttons in the center allow you to take a tour, look at tutorials, etc. You can review these at a later time by hitting the **Help** pull-down menu.

Let's jump right in and create an Eclipse C/C++ project to blink one of the LEDs. This project will run out of FLASH memory. In the **File** pull-down menu, click on "**File – New – Project…**" to get started, as shown below.

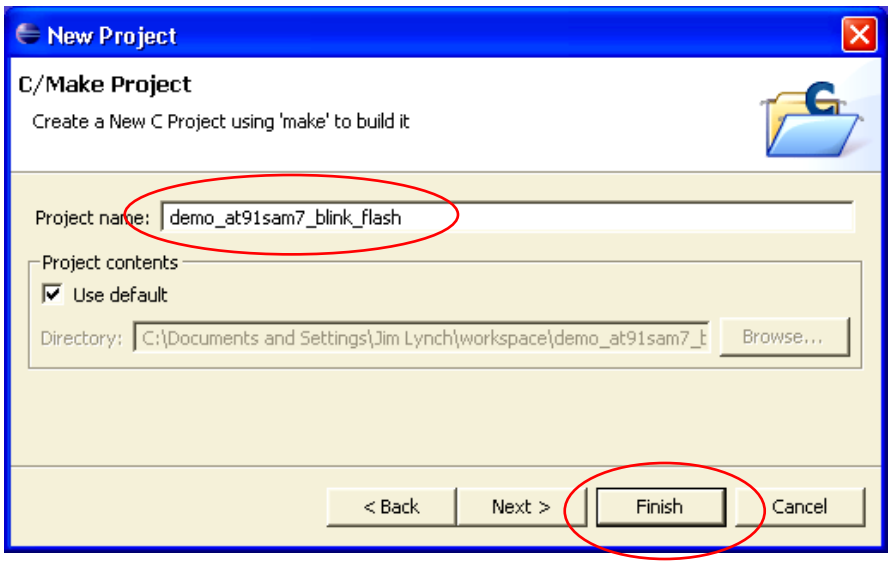


When the “**New Project**” wizard appears, expand on the “**C**” options by clicking on the “**+**” symbol and then click on “**Standard Make C Project**”. Click “**Next**” to continue.

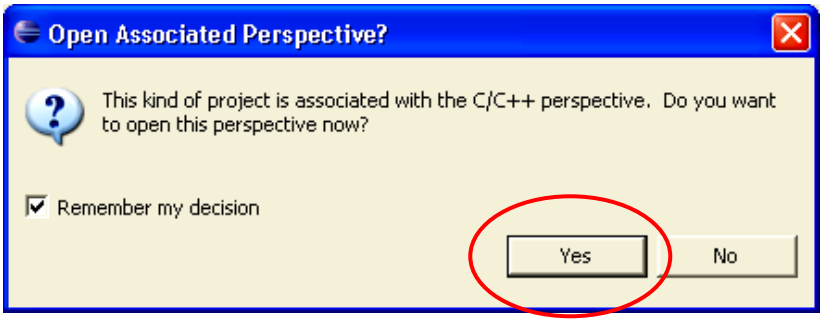




Enter the sample project name “**demo_at91sam7_blink_flash**” into the text window below. Click “**Finish**” to continue.

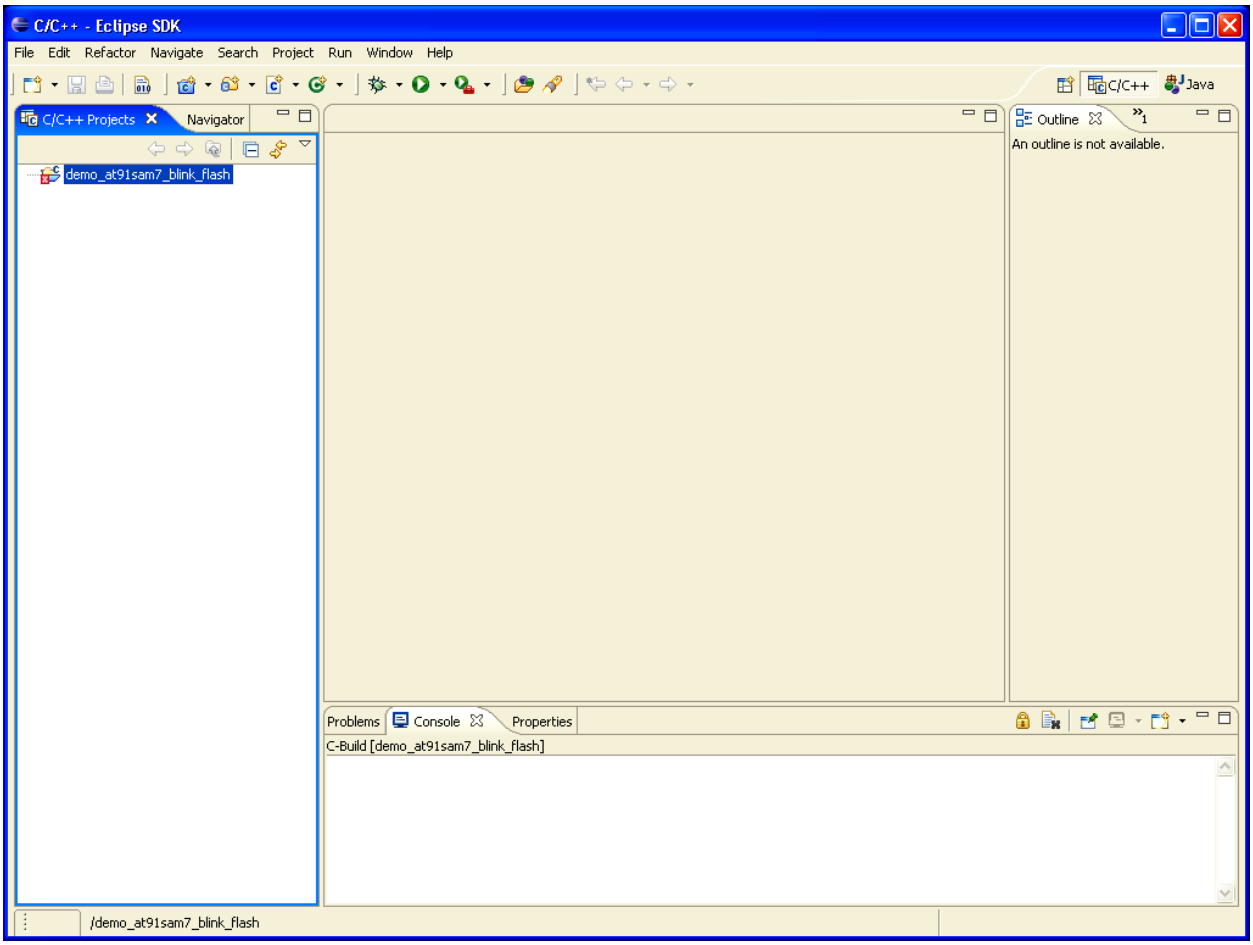


Eclipse will note that you are creating a C/C++ project and ask you if you want to switch to the C/C++ perspective (a perspective is a particular window layout). Check the “**Remember my decision**” check box and click “**Yes**” to open the C/C++ perspective.

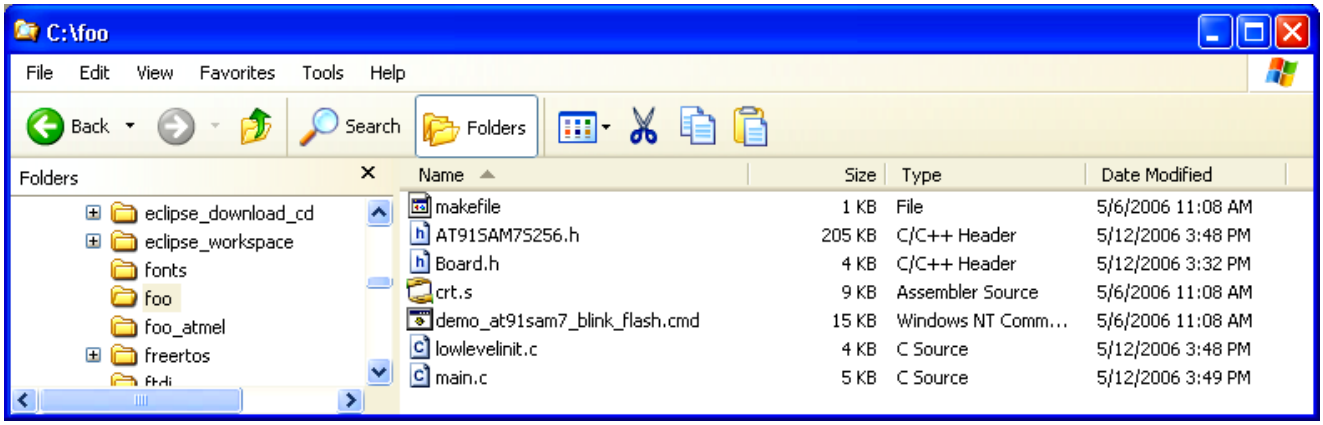


When the Eclipse C/C++ perspective appears, it will have a project defined in the project pane on the left. This project has no source files. Note that we also erased the “Eclipse Welcome” pane by clicking the delete button within that panel.

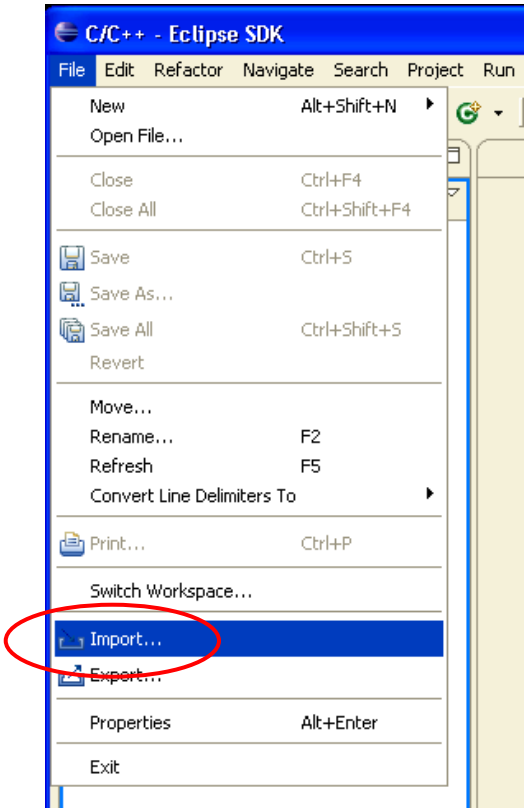
The C/C++ perspective looks like this.



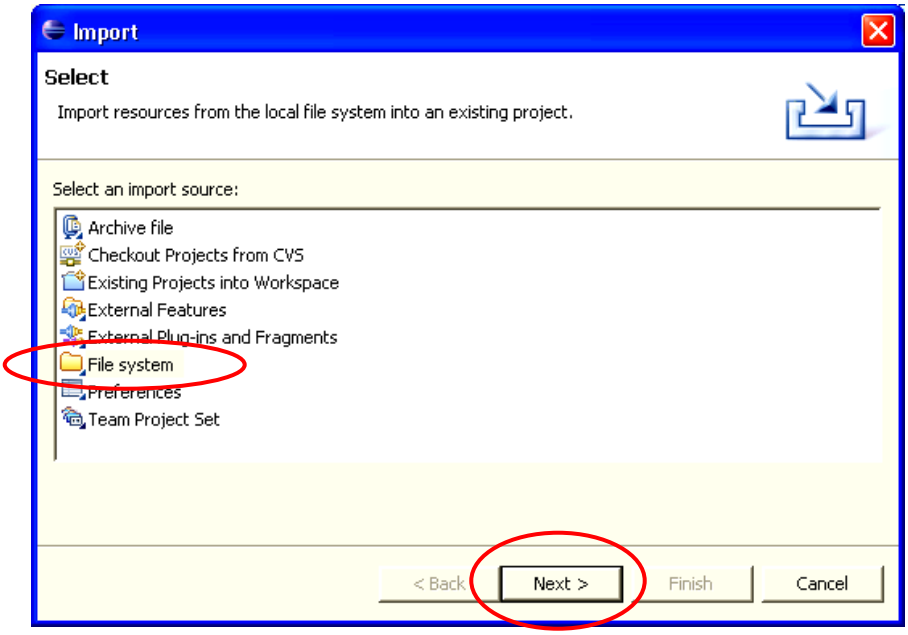
There are no source files in the project. An example project is included in the download collection you placed in the **c:\scratch** folder. Using windows Explorer, unzip the **demo_at91sam7_blink_flash** project’s source files in the file **SourceFiles.zip** to an empty directory such as **c:\foo** as shown below.



We use the Eclipse “**Import**” facility to copy the sample source files into the project. Click on “**File – Import…**”



In the “**Import**” window, click on “**File System**” and then click “**Next**” to continue.

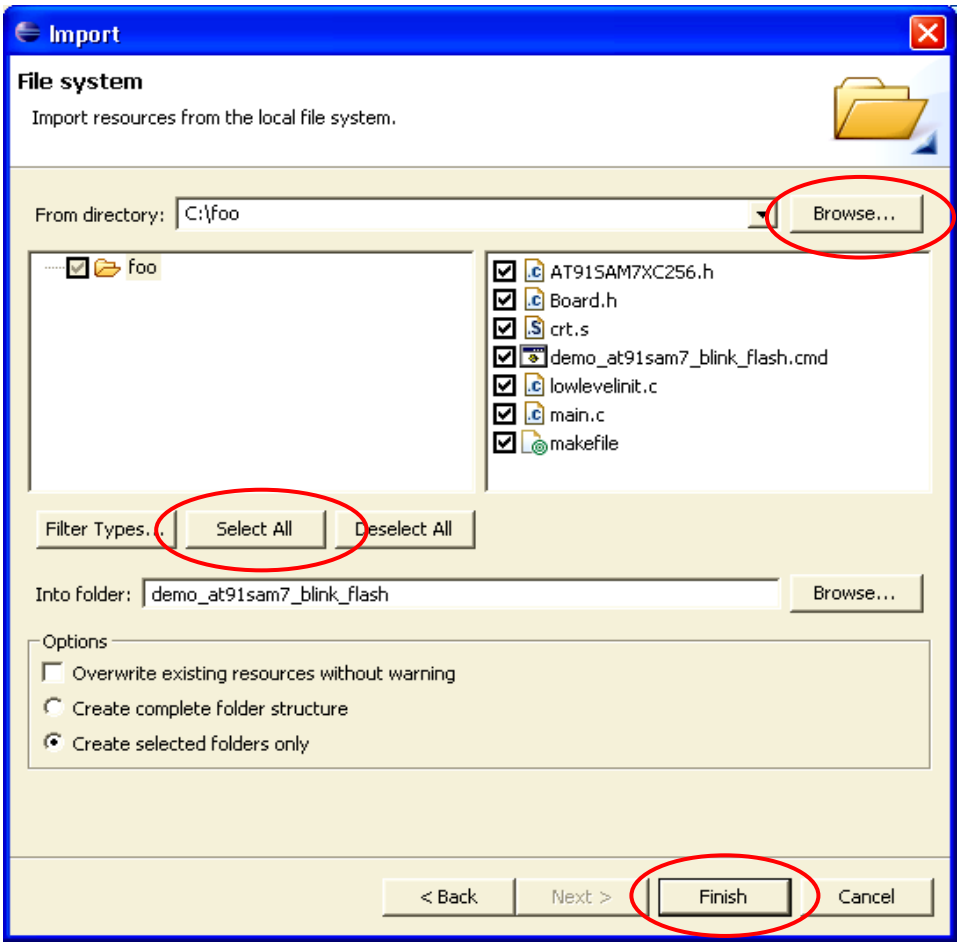


This displays the “**Import**” window as shown below.

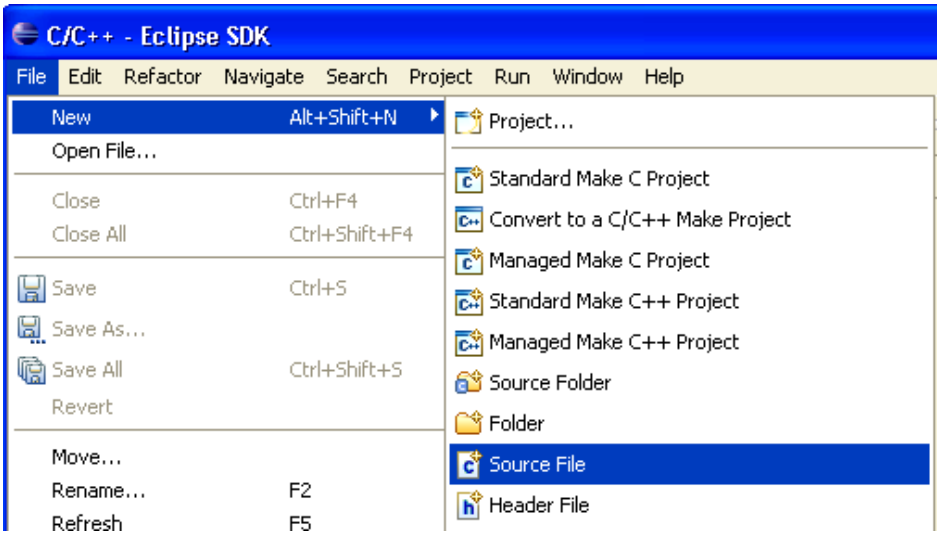
Use the “**Browse**” button in the “**From directory**” text box to navigate and find the **c:\foo** folder that holds the sample project source files unzipped earlier.

Click the “**Select All**” button to select all seven files.

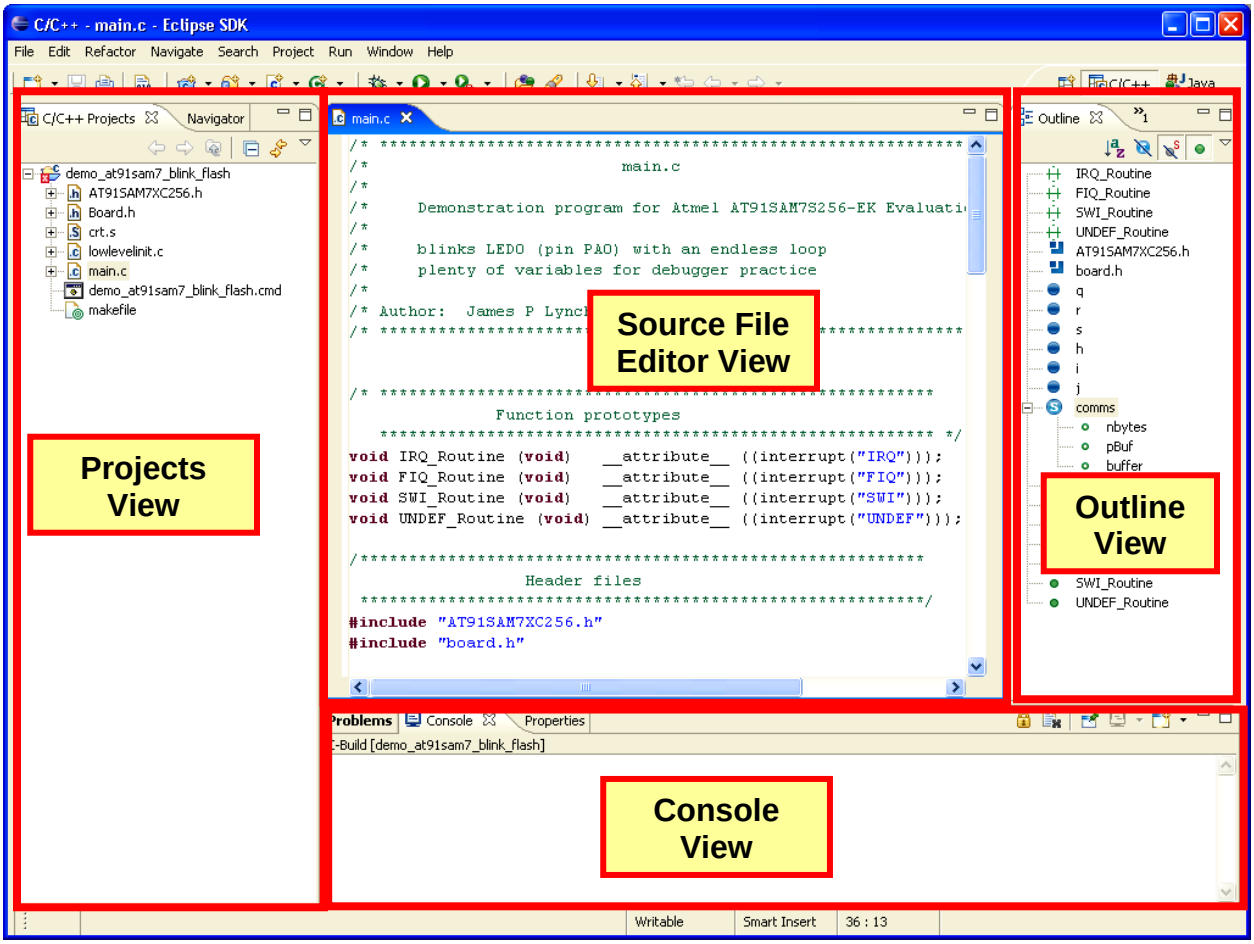
Accept the defaults on everything else and click “**Finish**” to complete the Import operation.



This will import all seven files into the demo_at91sam7_blink_flash project. Obviously, if you were creating the source files from scratch, you would use “**File – New – Source File**” and proceed from there to create an empty source file, as shown below.



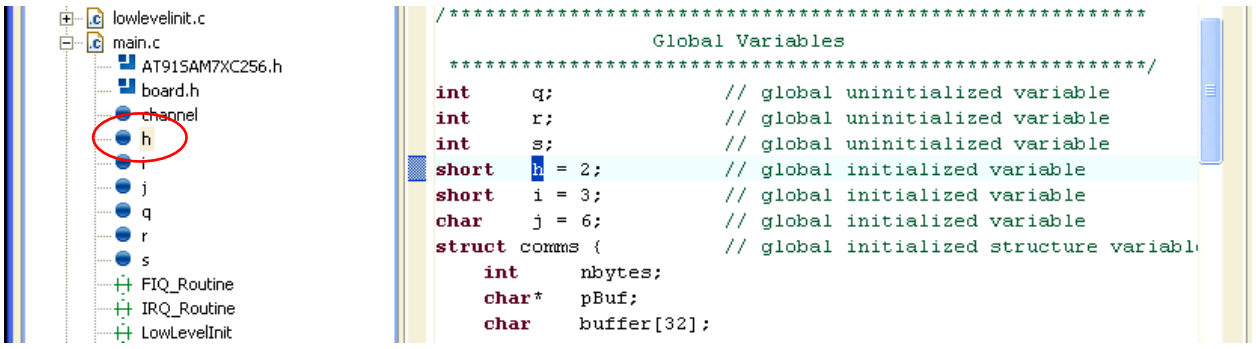
By clicking on the "**+**" sign on the project name in the C/C++ Projects panel on the left, the imported files are revealed. In the Eclipse window below, the **main.c** file has been selected by clicking on it and it thus displays in the source file editor view in the center.



In the "**C/C++ Projects"** view on the left, you can click on any source file and the Source Window will jump to that file.

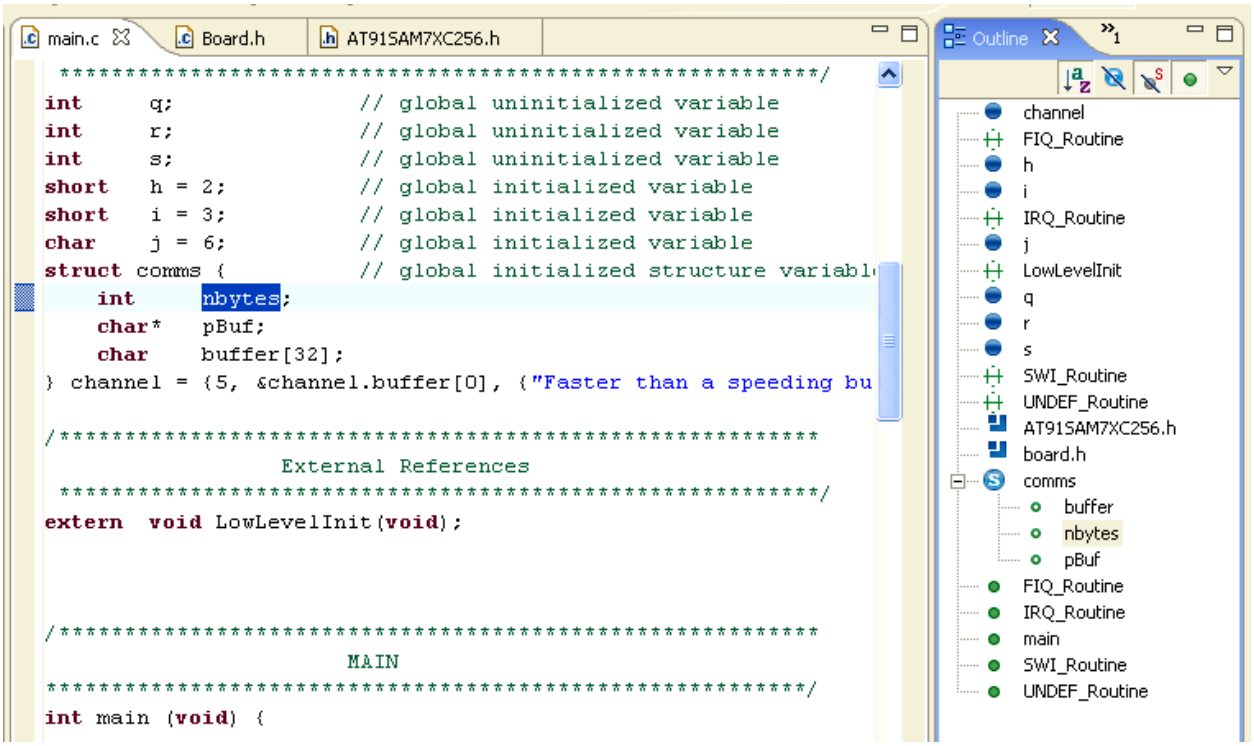
Source modules can be expanded (by clicking on the "+" expander icon) to reveal the variables and functions contained therein. This allows a very quick way to find the definition of a variable in the file.

In the sample directly below, we expanded the main.c to reveal the variables and functions. By clicking on the variable "**h**" in the C/C++ Projects view on the left, the source window jumps to the definition of that variable. This feature is more dramatic when you have a very large source file and it's tedious to scroll through all of it looking for a particular variable or function.

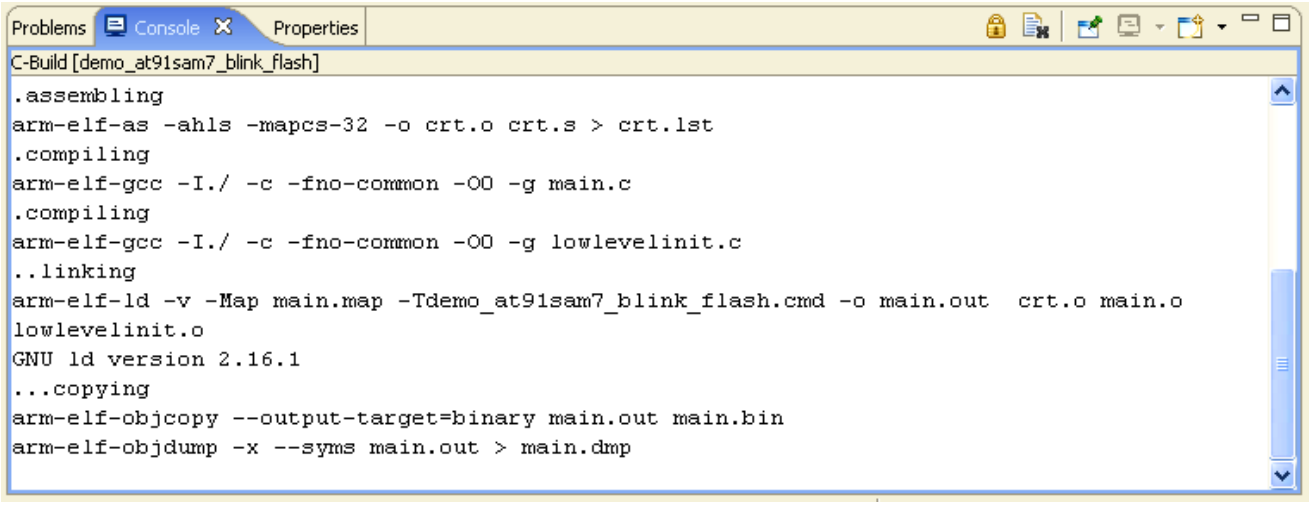


In the “**Outline**” view on the right, any C/C++ file being displayed in the source window in the center will have a tabular list of all important C/C++ elements (such as enumerations, structures, typedefs, variables, etc) to allow quick location of those elements in the source file.

In the example below, clicking on “nbytes” in the comms structural variable will cause the source file to jump to the definition of the “nbytes” element.



At the bottom of the Eclipse screen is the “**Console**” view. This shows, for example, the execution of the Make utility. In the example shown below, you can see the GNU assembler, compiler and linker steps being executed. If there are problems, you can select the “**Problems**” tab to see more information pertaining to any problems that occur.



Eclipse CDT has a fairly comprehensive User’s Guide that can be downloaded from here:

http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/cdt-home/user/C_C++_Development_Toolkit_User_Guide.pdf?cvsroot=Tools_Project

Discussion of the Source Files – FLASH Version

We will not describe every source file in detail. Most of these files are derived from other Atmel documentation and are simply modified to be compatible with the GNU tools.

AT91SAM7S256.H

This is the standard H file for the Atmel AT91SAM7S256 microprocessor.

```
//  ----------------------------------------------------------------------------
//          ATMEL Microcontroller Software Support  -  ROUSSET  -
//  ----------------------------------------------------------------------------
//  DISCLAIMER:  THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR
//  IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
//  MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
//  DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT,
//  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
//  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
//  OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
//  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
//  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
//  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//  ----------------------------------------------------------------------------
// File Name           : AT91SAM7S256.h
// Object              : AT91SAM7S256 definitions
// Generated           : AT91 SW Application Group  08/30/2005 (15:53:08)
//
// CVS Reference       : /AT91SAM7S256.pl/1.11/Tue Aug 30 12:00:29 2005//
// CVS Reference       : /SYS_SAM7S.pl/1.2/Tue Feb  1 17:01:52 2005//
// CVS Reference       : /MC_SAM7S.pl/1.3/Fri May 20 14:12:30 2005//
// CVS Reference       : /PMC_SAM7S_USB.pl/1.4/Tue Feb  8 13:58:22 2005//
// CVS Reference       : /RSTC_SAM7S.pl/1.2/Wed Jul 13 14:57:40 2005//
// CVS Reference       : /UDP_SAM7S.pl/1.1/Tue May 10 11:34:52 2005//
// CVS Reference       : /PWM_SAM7S.pl/1.1/Tue May 10 11:53:07 2005//
// CVS Reference       : /RTTC_6081A.pl/1.2/Tue Nov  9 14:43:58 2004//
// CVS Reference       : /PITC_6079A.pl/1.2/Tue Nov  9 14:43:56 2004//
// CVS Reference       : /WDTC_6080A.pl/1.3/Tue Nov  9 14:44:00 2004//
// CVS Reference       : /VREG_6085B.pl/1.1/Tue Feb  1 16:05:48 2005//
// CVS Reference       : /AIC_6075B.pl/1.3/Fri May 20 14:01:30 2005//
// CVS Reference       : /PIO_6057A.pl/1.2/Thu Feb  3 10:18:28 2005//
// CVS Reference       : /DBGU_6059D.pl/1.1/Mon Jan 31 13:15:32 2005//
// CVS Reference       : /US_6089C.pl/1.1/Mon Jul 12 18:23:26 2004//
// CVS Reference       : /SPI_6088D.pl/1.3/Fri May 20 14:08:59 2005//
// CVS Reference       : /SSC_6078A.pl/1.1/Tue Jul 13 07:45:40 2004//
// CVS Reference       : /TC_6082A.pl/1.7/Fri Mar 11 12:52:17 2005//
// CVS Reference       : /TWI_6061A.pl/1.1/Tue Jul 13 07:38:06 2004//
// CVS Reference       : /PDC_6074C.pl/1.2/Thu Feb  3 08:48:54 2005//
// CVS Reference       : /ADC_6051C.pl/1.1/Fri Oct 17 09:12:38 2003//
//  ----------------------------------------------------------------------------

#ifndef AT91SAM7S256_H
#define AT91SAM7S256_H

typedef volatile unsigned int AT91_REG;// Hardware register definition

// *****************************************************************************
//              SOFTWARE API DEFINITION  FOR System Peripherals
// *****************************************************************************
typedef struct _AT91S_SYS {
       AT91_REG         AIC_SMR[32];   // Source Mode Register
       AT91_REG         AIC_SVR[32];   // Source Vector Register
       AT91_REG         AIC_IVR;       // IRQ Vector Register
       AT91_REG         AIC_FVR;       // FIQ Vector Register
       AT91_REG         AIC_ISR;       // Interrupt Status Register
       AT91_REG         AIC_IPR;       // Interrupt Pending Register
       AT91_REG         AIC_IMR;       // Interrupt Mask Register
       AT91_REG         AIC_CISR;      // Core Interrupt Status Register
```

```
       AT91_REG         Reserved0[2];  //
       AT91_REG         AIC_IECR;      // Interrupt Enable Command Register
       AT91_REG         AIC_IDCR;      // Interrupt Disable Command Register
       AT91_REG         AIC_ICCR;      // Interrupt Clear Command Register
       AT91_REG         AIC_ISCR;      // Interrupt Set Command Register
       AT91_REG         AIC_EOICR;     // End of Interrupt Command Register
       AT91_REG         AIC_SPU;       // Spurious Vector Register
       AT91_REG         AIC_DCR;       // Debug Control Register (Protect)
       AT91_REG         Reserved1[1];  //
       AT91_REG         AIC_FFER;      // Fast Forcing Enable Register
       AT91_REG         AIC_FFDR;      // Fast Forcing Disable Register
       AT91_REG         AIC_FFSR;      // Fast Forcing Status Register
       AT91_REG         Reserved2[45]; //
       AT91_REG         DBGU_CR;       // Control Register
       AT91_REG         DBGU_MR;       // Mode Register
       AT91_REG         DBGU_IER;      // Interrupt Enable Register
       AT91_REG         DBGU_IDR;      // Interrupt Disable Register
       AT91_REG         DBGU_IMR;      // Interrupt Mask Register
       AT91_REG         DBGU_CSR;      // Channel Status Register
       AT91_REG         DBGU_RHR;      // Receiver Holding Register
       AT91_REG         DBGU_THR;      // Transmitter Holding Register
              .
              .
              .
        (Note: this is a very large file)
```

BOARD.H

This is the standard board definition file for the AT91SAM7S Evaluation Board.

```
/*-----------------------------------------------------------------------------
*         ATMEL Microcontroller Software Support  -  ROUSSET  -
*-----------------------------------------------------------------------------
* The software is delivered "AS IS" without warranty or condition of any
* kind, either express, implied or statutory. This includes without
* limitation any warranty or condition with respect to merchantability or
* fitness for any particular purpose, or against the infringements of
* intellectual property rights of others.
*-----------------------------------------------------------------------------
* File Name           : Board.h
* Object              : AT91SAM7S Evaluation Board Features Definition File.
*
* Creation            : JPP   16/Jun/2004
*-----------------------------------------------------------------------------
*/
#ifndef Board_h
#define Board_h

#include "AT91SAM7S256.h"
#define __inline inline

#define true    -1
#define false   0

/*-------------------------------*/
/* SAM7Board Memories Definition */
/*-------------------------------*/
// The AT91SAM7S64 embeds a 16-Kbyte SRAM bank, and 64 K-Byte Flash

#define  INT_SARM           0x00200000
#define  INT_SARM_REMAP     0x00000000

#define  INT_FLASH          0x00000000
#define  INT_FLASH_REMAP    0x01000000

#define  FLASH_PAGE_NB       512
#define  FLASH_PAGE_SIZE    128
```

```
/*-----------------*/
/* Leds Definition */
/*-----------------*/
/*                                PIO   Flash    PA    PB   PIN */
#define LED1            (1<<0)   /* PA0 / PGMEN0 & PWM0 TIOA0  48 */
#define LED2            (1<<1)   /* PA1 / PGMEN1 & PWM1 TIOB0  47 */
#define LED3            (1<<2)   /* PA2          & PWM2 SCK0   44 */
#define LED4            (1<<3)   /* PA3          & TWD  NPCS3  43 */
#define NB_LEB           4

#define LED_MASK        (LED1|LED2|LED3|LED4)



/*-------------------------*/
/* Push Buttons Definition */
/*-------------------------*/
/*                         PIO             Flash  PA      PB   PIN */
#define SW1_MASK        (1<<19)  /* PA19 / PGMD7  & RK   FIQ     13 */
#define SW2_MASK        (1<<20)  /* PA20 / PGMD8  & RF   IRQ0    16 */
#define SW3_MASK        (1<<15)  /* PA15 / PGM3   & TF   TIOA1   20 */
#define SW4_MASK        (1<<14)  /* PA14 / PGMD2  & SPCK PWM3    21 */
#define SW_MASK         (SW1_MASK|SW2_MASK|SW3_MASK|SW4_MASK)


#define SW1             (1<<19)     // PA19
#define SW2             (1<<20)     // PA20
#define SW3             (1<<15)     // PA15
#define SW4             (1<<14)     // PA14

/*------------------*/
/* USART Definition */
/*------------------*/
/* SUB-D 9 points J3 DBGU*/
#define DBGU_RXD         AT91C_PA9_DRXD     /* JP11 must be close */
#define DBGU_TXD         AT91C_PA10_DTXD    /* JP12 must be close */
#define AT91C_DBGU_BAUD  115200             /* Baud rate          */

#define US_RXD_PIN       AT91C_PA5_RXD0     /* JP9 must be close */
#define US_TXD_PIN       AT91C_PA6_TXD0     /* JP7 must be close */
#define US_RTS_PIN       AT91C_PA7_RTS0     /* JP8 must be close */
#define US_CTS_PIN       AT91C_PA8_CTS0     /* JP6 must be close */

/*--------------*/
/* Master Clock */
/*--------------*/

#define EXT_OC          18432000   // Exetrnal ocilator MAINCK
#define MCK             47923200   // MCK (PLLRC div by 2)
#define MCKKHz          (MCK/1000) //

#endif /* Board_h */
```

CRT.S

This assembly language startup file, crt.s, includes parts of the standard Atmel startup file with a few changes by the author to conform to the GNU assembler.

```
/* ********************************************************************************************** */
/*                                          crt.s                                                 */
/*                                                                                                */
/*                      Assembly Language Startup Code for Atmel AT91SAM7S256                      */
/*                                                                                                */
/*                                                                                                */
/* ********************************************************************************************** */

/* Stack Sizes */
.set  UND_STACK_SIZE, 0x00000010         /* stack for "undefined instruction" interrupts is 16 bytes */
.set  ABT_STACK_SIZE, 0x00000010         /* stack for "abort" interrupts is 16 bytes */
.set  FIQ_STACK_SIZE, 0x00000080         /* stack for "FIQ" interrupts  is 128 bytes */
.set  IRQ_STACK_SIZE, 0X00000080         /* stack for "IRQ" normal interrupts is 128 bytes */
.set  SVC_STACK_SIZE, 0x00000010         /* stack for "SVC" supervisor mode is 16 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 Priviledged 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) */

/* Interrupt controller offsets */
.set  AT91C_BASE_AIC, 0xFFFFF000        /* (AIC) Base Address  */
.set  AIC_IVR, 256                      /* IRQ Vector Register  */
.set  AIC_EOICR, 304                    /* End of Interrupt Command Register  */

/* identify all GLOBAL symbols  */
.global   _vec_reset
.global   _vec_undef
.global   _vec_swi
.global   _vec_pabt
.global   _vec_dabt
.global   _vec_rsv
.global   _vec_irq
.global   _vec_fiq
.global   AT91F_Irq_Handler
.global   AT91F_Default_FIQ_handler
.global   AT91F_Default_IRQ_handler
.global   AT91F_Spurious_handler

/* GNU assembler controls  */
.text             /* all assembler code that follows will go into .text section */
.arm              /* compile for 32-bit ARM instruction set */
.align            /* align section on 32-bit boundary */

/* ========================================================== */
/*                          VECTOR TABLE                      */
/*                                                            */
/*       Must be located in FLASH at address 0x00000000       */
/*                                                            */
/*       Easy to do if this file crt.s is first in the list   */
/*       for the linker step in the makefile, e.g.            */
/*                                                            */
/*           $(LD) $(LFLAGS) -o main.out  crt.o main.o        */
/*                                                            */
/* ========================================================== */

_vec_reset:       b      _init_reset         /* RESET vector - must be at 0x0000 */
_vec_undef:       b      _vec_undef          /* Undefined Instruction vector */
_vec_swi:         b      _vec_swi            /* Software Interrupt vector */
_vec_pabt:        b      _vec_pabt           /* Prefetch abort vector */
_vec_dabt:        b      _vec_dabt           /* Data abort vector */
_vec_rsv:         b      _vec_rsv            /* Reserved vector */
_vec_irq:         b      _vec_irq            /* Interrupt Request (IRQ) vector */
_vec_fiq:         b      _vec_fiq            /* Fast interrupt request (FIQ) vector */
```

```
/* Reset Handler */
_init_reset:
         /* Setup stack for each mode with interrupts initially disabled. */
         ldr   r0, =_stack_end                          /* r0 = top-of-stack  */

         msr   CPSR_c, #MODE_UND|I_BIT|F_BIT            /* switch to Undefined Instruction Mode */
         mov   sp, r0                                   /* set stack pointer for UND mode */
         sub   r0, r0, #UND_STACK_SIZE                  /* adjust r0 past UND stack */

         msr   CPSR_c, #MODE_ABT|I_BIT|F_BIT            /* switch to Abort Mode */
         mov   sp, r0                                   /* set stack pointer for ABT mode */
         sub   r0, r0, #ABT_STACK_SIZE                  /* adjust r0 past ABT stack */

         msr   CPSR_c, #MODE_FIQ|I_BIT|F_BIT            /* switch to FIQ Mode */
         mov   sp, r0                                   /* set stack pointer for FIQ mode */
         sub   r0, r0, #FIQ_STACK_SIZE                  /* adjust r0 past FIQ stack */

         msr   CPSR_c, #MODE_IRQ|I_BIT|F_BIT            /* switch to IRQ Mode */
         mov   sp, r0                                   /* set stack pointer for IRQ mode */
         sub   r0, r0, #IRQ_STACK_SIZE                  /* adjust r0 past IRQ stack */

         msr   CPSR_c, #MODE_SVC|I_BIT|F_BIT            /* switch to Supervisor Mode */
         mov   sp, r0                                   /* set stack pointer for SVC mode */
         sub   r0, r0, #SVC_STACK_SIZE                  /* adjust r0 past SVC stack */

         msr   CPSR_c, #MODE_SYS|I_BIT|F_BIT            /* switch to System Mode */
         mov   sp, r0                                   /* set stack pointer for SYS mode */
                                                        /* we now start execution in SYSTEM mode */
                                                        /* This is exactly like USER mode (same stack) */
                                                        /* but SYSTEM mode has more privileges */


/* copy initialized variables .data section  (Copy from ROM to RAM) */
          ldr     R1, =_etext
          ldr     R2, =_data
          ldr     R3, =_edata
1:        cmp     R2, R3
          ldrlo   R0, [R1], #4
          strlo   R0, [R2], #4
          blo     1b


/* Clear uninitialized variables .bss section (Zero init)  */
          mov     R0, #0
         ldr     R1, =_bss_start
         ldr     R2, =_bss_end
2:       cmp     R1, R2
         strlo   R0, [R1], #4
         blo     2b

/* Enter the C code  */
         b       main



/* ------------------------------------------------------------ */
/* - Function             : AT91F_Irq_Handler                   */
/* - Treatments           : IRQ Controller Interrupt Handler.   */
/* - Called Functions     : AIC_IVR[interrupt]                  */
/* ------------------------------------------------------------ */
AT91F_Irq_Handler:

/* Adjust and save LR_irq in IRQ stack */
         sub      lr, lr, #4
         stmfd    sp!, {lr}

/* Save and r0 in IRQ stack */
         stmfd    sp!, {r0}

/* Write in the IVR to support Protect Mode */
/* No effect in Normal Mode */
/* De-assert the NIRQ and clear the source in Protect Mode */
         ldr      r14, =AT91C_BASE_AIC
         ldr      r0 , [r14, #AIC_IVR]
         str      r14, [r14, #AIC_IVR]

/* Enable Interrupt and Switch in Supervisor Mode */
         msr      CPSR_c, #MODE_SVC

/* Save scratch/used registers and LR in User Stack */
         stmfd    sp!, { r1-r3, r12, r14}
```

```
/* ================================================ */
/* Branch to the routine pointed by the AIC_IVR     */
/* ================================================ */

/* Branch to the routine pointed by the AIC_IVR */
         mov      r14, pc
         bx       r0

/* ================================================ */
/* Manage Exception Exit                           */
/* ================================================ */

/* Restore scratch/used registers and LR from User Stack */
         ldmia    sp!, { r1-r3, r12, r14}

/* Disable Interrupt and switch back in IRQ mode */
         msr      CPSR_c, #MODE_IRQ|I_BIT

/* Mark the End of Interrupt on the AIC */
         ldr      r14, =AT91C_BASE_AIC
         str      r14, [r14, #AIC_EOICR]

/* Restore SPSR_irq and r0 from IRQ stack */
         ldmia    sp!, {r0}






/* Restore adjusted  LR_irq from IRQ stack directly in the PC */
         ldmia    sp!, {pc}^



AT91F_Default_FIQ_handler:  b     AT91F_Default_FIQ_handler

AT91F_Default_IRQ_handler:  b     AT91F_Default_IRQ_handler

AT91F_Spurious_handler:     b     AT91F_Spurious_handler

.end
```

LOWLEVELINIT.C

This C language routine is a copy of the Atmel version in the IAR sample projects.

```
//*----------------------------------------------------------------------------------------------------------
//*         ATMEL Microcontroller Software Support  -  ROUSSET  -
//*----------------------------------------------------------------------------------------------------------
//* The software is delivered "AS IS" without warranty or condition of any
//* kind, either express, implied or statutory. This includes without
//* limitation any warranty or condition with respect to merchantability or
//* fitness for any particular purpose, or against the infringements of
//* intellectual property rights of others.
//*----------------------------------------------------------------------------------------------------------
//* File Name              : Cstartup_SAM7.c
//* Object                 : Low level initializations written in C for IAR tools
//* 1.0   08/Sep/04 JPP    : Creation
//* 1.10  10/Sep/04 JPP    : Update AT91C_CKGR_PLLCOUNT filed
//*----------------------------------------------------------------------------------------------------------


// Include the board file description
#include "AT91SAM7XC256.h"
#include "Board.h"

// The following functions must be write in ARM mode this function called directly
// by exception vector
extern void AT91F_Spurious_handler(void);
extern void AT91F_Default_IRQ_handler(void);
extern void AT91F_Default_FIQ_handler(void);
```

```
//*----------------------------------------------------------------------------------------------
//* \fn    AT91F_LowLevelInit
//* \brief This function performs very low level HW initialization
//*        this function can be use a Stack, depending the compilation
//*        optimization mode
//*----------------------------------------------------------------------------------------------
void LowLevelInit(void)
{
         int            i;
         AT91PS_PMC     pPMC = AT91C_BASE_PMC;

          //* Set Flash Wait sate
         //  Single Cycle Access at Up to 30 MHz, or 40
         //  if MCK = 47923200 I have 50 Cycle for 1 usecond ( flied MC_FMR->FMCN
         AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN)&(50 <<16)) | AT91C_MC_FWS_1FWS;

         //* Watchdog Disable
         AT91C_BASE_WDTC->WDTC_WDMR= AT91C_WDTC_WDDIS;

         //* Set MCK at 47 923 200
         // 1 Enabling the Main Oscillator:
         // SCK = 1/32768 = 30.51 uSecond
         // Start up time = 8 * 6 / SCK = 56 * 30.51 = 1,46484375 ms
         pPMC->PMC_MOR = (( AT91C_CKGR_OSCOUNT & (0x06 <<8) | AT91C_CKGR_MOSCEN ));

         // Wait the startup time
         while(!(pPMC->PMC_SR & AT91C_PMC_MOSCS));

         // 2 Checking the Main Oscillator Frequency (Optional)
         // 3 Setting PLL and divider:
         // - div by 5 Fin = 3,6864 =(18,432 / 5)
         // - Mul 25+1: Fout =       95,8464 =(3,6864 *26)

         // for 96 MHz the erroe is 0.16%
         // Field out NOT USED = 0
         pPMC->PMC_PLLR = ((AT91C_CKGR_DIV & 14) |
                           (AT91C_CKGR_PLLCOUNT & (10<<8)) |
                           (AT91C_CKGR_MUL & (72<<16)));

         // Wait the startup time
         while(!(pPMC->PMC_SR & AT91C_PMC_LOCK));

         // 4. Selection of Master Clock and Processor Clock
         // select the PLL clock divided by 2
         pPMC->PMC_MCKR = AT91C_PMC_CSS_PLL_CLK | AT91C_PMC_PRES_CLK_2;

         // Set up the default interrupts handler vectors
         AT91C_BASE_AIC->AIC_SVR[0] = (int) AT91F_Default_FIQ_handler;
         for (i=1;i < 31; i++)
         {
             AT91C_BASE_AIC->AIC_SVR[i] = (int) AT91F_Default_IRQ_handler;
         }
         AT91C_BASE_AIC->AIC_SPU  = (int) AT91F_Spurious_handler;

}
```

MAIN.C

This simple C language routine creates some variables and blinks LED1 in an infinite loop.

```
/* ******************************************************************************************* */
/*                          main.c                                                               */
/*                                                                                              */
/*     Demonstration program for Atmel AT91SAM7S256-EK Evaluation Board                         */
/*                                                                                              */
/*     blinks LED0 (pin PA0) with an endless loop                                               */
/*     plenty of variables for debugger practice                                                */
/*                                                                                              */
/* ******************************************************************************************* */


/* ******************************************************
              Function prototypes
   ****************************************************** */
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 "AT91SAM7XC256.h"
#include "board.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 {                       // global initialized structure variable
         int      nbytes;
         char*    pBuf;
         char     buffer[32];
} channel = {5, &channel.buffer[0], {"Faster than a speeding bullet"}};

/*********************************************************
                 External References
 *********************************************************/
extern   void LowLevelInit(void);

/*********************************************************
                      MAIN
*********************************************************/
int   main (void) {

     int          j;                                 // loop counter (stack variable)
     int          a,b,c;                             // uninitialized variables
     char         d;                                 // uninitialized variables
     int          w = 1;                             // initialized variable
     int          k = 1;                             // 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";      // initialized string pointer
     struct EntryLock {                              // initialized structure variable
          long    key;
          int     nAccesses;
          char    name[17];
     } Access = {14705, 0, "Sophie Marceau"};

     // Initialize the Atmel AT91SAM7S256 (watchdog, PLL clock, default interrupts, etc.)
     LowLevelInit();

     // Set up the LEDs (PA0 - PA3)
     // at boot, all peripherials are disabled and all pins are inputs
     AT91PS_PIO     pPIO = AT91C_BASE_PIOA;     // pointer to PIO data structure
     pPIO->PIO_PER = LED_MASK;                  // PIO Enable Register - allow PIO to control pins P0 - P3
     pPIO->PIO_OER = LED_MASK;                  // PIO Output Enable Register - sets pins P0 - P3 to puts
     pPIO->PIO_SODR = LED_MASK;                 // PIO Set Output Data Register - turns off the four LEDs

     // endless loop to toggle the green  LED DS1
     while (1) {
          for (j = 0; j < 300000; j++ );      // wait 500 msec
              pPIO->PIO_CODR = LED1;          // turn LED1 (DS1) on
          for (j = 0; j < 300000; j++ );      // wait 500 msec
              pPIO->PIO_SODR = LED1;          // turn LED1 (DS1) off

          k += 1;                             // count the number of blinks
     }
}


/*  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) ;
}
```

DEMO_AT91SAM7_BLINK_FLASH.CMD

This linker script file was prepared by the author and is compatible with the GNU Linker.

```
// ***************************************************************************************************************
//   demo_at91sam7_blink_flash.cmd                      LINKER  SCRIPT
//
//
//   The Linker Script defines how the code and data emitted by the GNU C compiler and assembler are
//   to be loaded into memory (code goes into FLASH, variables go into RAM).
//
//   Any symbols defined in the Linker Script are automatically global and available to the rest of the
//   program.
//
//   To force the linker to use this LINKER SCRIPT, just add the -T demo_at91sam7_blink_flash.cmd
//   directive to the linker flags in the makefile. For example,
//
//               LFLAGS  =  -Map main.map -nostartfiles -T demo_at91sam7_blink_flash.cmd
//
//
//   The order that the object files are listed in the makefile determines what .text section is placed first.
//
//   For example:  $(LD) $(LFLAGS) -o main.out  crt.o main.o lowlevelinit.o
//
//                         crt.o is first in the list of objects, so it will be placed at address 0x0000
//
//
//   The top of the stack (_stack_end) is (last_byte_of_ram +1) - 4
//
//   Therefore:   _stack_end = (0x0020FFFF + 1) - 4  =  0x00210000 - 4  =  0x0020FFFC
//
//   Note that this symbol (_stack_end) is automatically GLOBAL and will be used by the crt.s
//   startup assembler routine to specify all stacks for the various ARM modes
//
//                            MEMORY MAP
//                     |                                             |
//           .-------->|---------------------------------------------|0x00210000
//           .         |                                             |0x0020FFFC  <---------- _stack_end
//           .         |    UDF Stack  16 bytes                      |
//           .         |                                             |
//           .         |---------------------------------------------|0x0020FFF0
//           .         |                                             |
//           .         |    ABT Stack  16 bytes                      |
//           .         |                                             |
//           .         |---------------------------------------------|0x0020FFE0
//           .         |                                             |
//           .         |                                             |
//           .         |    FIQ Stack  128 bytes                     |
//           .         |                                             |
//           .         |                                             |
//           .         |---------------------------------------------|0x0020FF60
//           .         |                                             |
//           .         |                                             |
//           .         |    IRQ Stack  128 bytes                     |
//           .         |                                             |
//           .         |                                             |
//           .         |---------------------------------------------|0x0020FEE0
//           .         |                                             |
//           .         |    SVC Stack  16 bytes                      |
//           .         |                                             |
//           .         |---------------------------------------------|0x0020FED0
//           .         |                                             |
//           .         |     stack area for user program             |
//           .         |                                             |
//           .         |                                             |
//           .         |                                             |
//           .         |                                             |
//           .         |                 free ram                    |
//          ram        |                                             |
//           .         |                                             |
//           .         |                                             |
//           .         |.............................................|0x00200048 <---------- _bss_end
//           .         |                                             |
//           .         |  .bss   uninitialized variables             |
//           .         |.............................................|0x0020003C <---------- _bss_start, _edata
//           .         |                                             |
//           .         |  .data  initialized variables               |
//           .         |                                             |
//           .-------->|---------------------------------------------|0x00200000 <---------- _data
//                     |                                             |
//
//
//
//
//                     |                                             |
```

```
//           .--------> |-----------------------------------------------|
//           .          |                                               |0x000FFFFF
//           .          |                                               |
//           .          |                                               |
//           .          |                                               |
//           .          |                                               |
//           .          |                                               |
//           .          |     unused flash eprom                        |
//           .          |                                               |
//           .          |-----------------------------------------------|0x000003E4
//           .          |                                               |
//           .          |       copy of .data area                      |
//         flash        |                                               |
//           .          |-----------------------------------------------|0x000003A8 <---------- _etext
//           .          |                                               |
//           .          |                                               |
//           .          |                                               |0x00000198  lowlevelinit()
//           .          |            C code                             |
//           .          |                                               |
//           .          |                                               |
//           .          |                                               |
//           .          |-----------------------------------------------|0x00000108  main()
//           .          |                                               |
//           .          |    Startup Code  (crt.s)                      |
//           .          |         (assembler)                           |
//           .          |                                               |
//           .          |-----------------------------------------------|0x00000020
//           .          |                                               |
//           .          |   Interrupt Vector Table                      |
//           .          |            32 bytes                           |
//           .--------->|-----------------------------------------------|0x00000000 _startup
//
//
// ***********************************************************************************************************


/* identify the Entry Point  (_vec_reset is defined in file crt.s)  */
ENTRY(_vec_reset)


/* specify the AT91SAM7S memory areas  */
MEMORY
{
        flash : ORIGIN = 0, LENGTH = 256K               /* FLASH EPROM         */
        ram   : ORIGIN = 0x00200000, LENGTH = 64K       /* static RAM area     */
}


/* define a global symbol _stack_end  (see analysis in annotation above) */
_stack_end = 0x2FFFFC;

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

         .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 */
         } >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 */
```

MAKEFILE.MAK

This makefile was prepared by the author and is compatible with the GNU Make utility.

```
NAME   = demo_at91sam7_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 -O0 -g
AFLAGS  = -ahls -mapcs-32 -o crt.o
LFLAGS  = -Map main.map -Tdemo_at91sam7_blink_flash.cmd
CPFLAGS = --output-target=binary
ODFLAGS = -x --syms

all: test

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

test: main.out
         @ echo "...copying"
         $(CP) $(CPFLAGS) main.out main.bin
         $(OD) $(ODFLAGS) main.out > main.dmp



main.out: crt.o main.o lowlevelinit.o demo_at91sam7_blink_flash.cmd
         @ echo "..linking"
         $(LD) $(LFLAGS) -o main.out  crt.o main.o lowlevelinit.o

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

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

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

Adjusting the Optimization Level

It's a fact of life in embedded programming that debuggers hate optimized code. When you attempt to single-step optimized code, the debuggers will do strange things and appear not to work. To get around this problem, change the compiler optimization level to ZERO. This is done in the makefile above; modify the CPFLAGS macro substitution as follows:

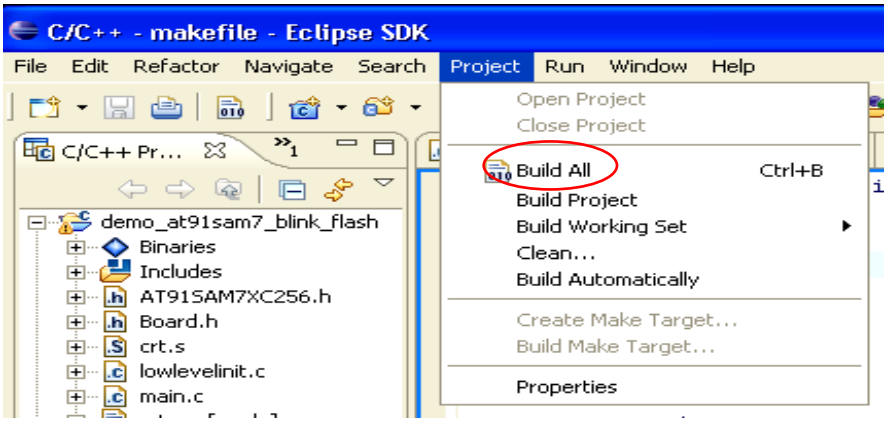
CFLAGS = -I./ -c -fno-common **-O0** –g Where the switch: -O0 means no optimization.

When debugging is completed, you can increase the optimization level to –O3 which will result in more compact and efficient code.

Building the FLASH Application

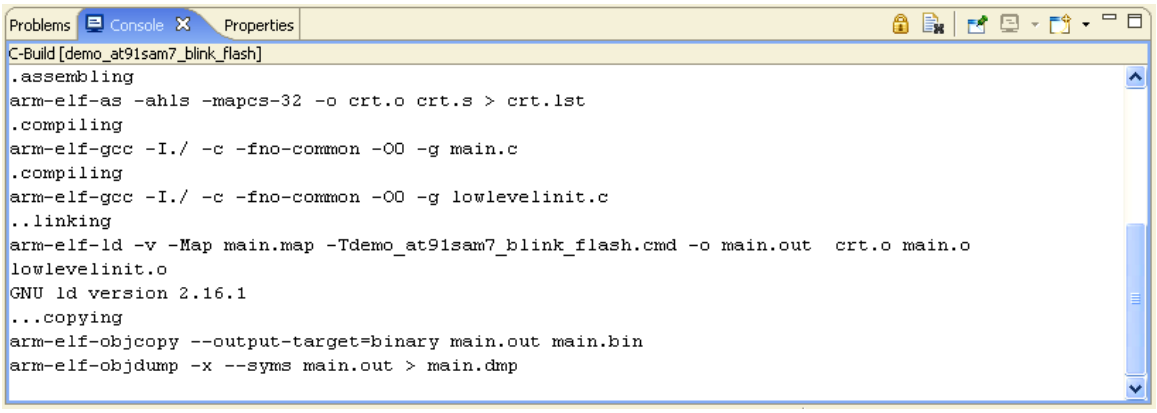
The “**Project**” pull-down menu has several options to build the application. “**Clean**” will erase all object, list, map, and output and dump files, thus forcing Eclipse to compile, assemble and link every file. This may be time-consuming in a large project with many files. “**Build All**” will only compile and link those files that are “out-of-date”.

The usual procedure is to “**Build All**” and this may be selected from the “**Projects**” pull-down menu, as shown below.



Even more convenient is the “**Build All**” button in the Eclipse toolbar. 

The Console view at the bottom of the Eclipse screen will show the progress of the build operation.



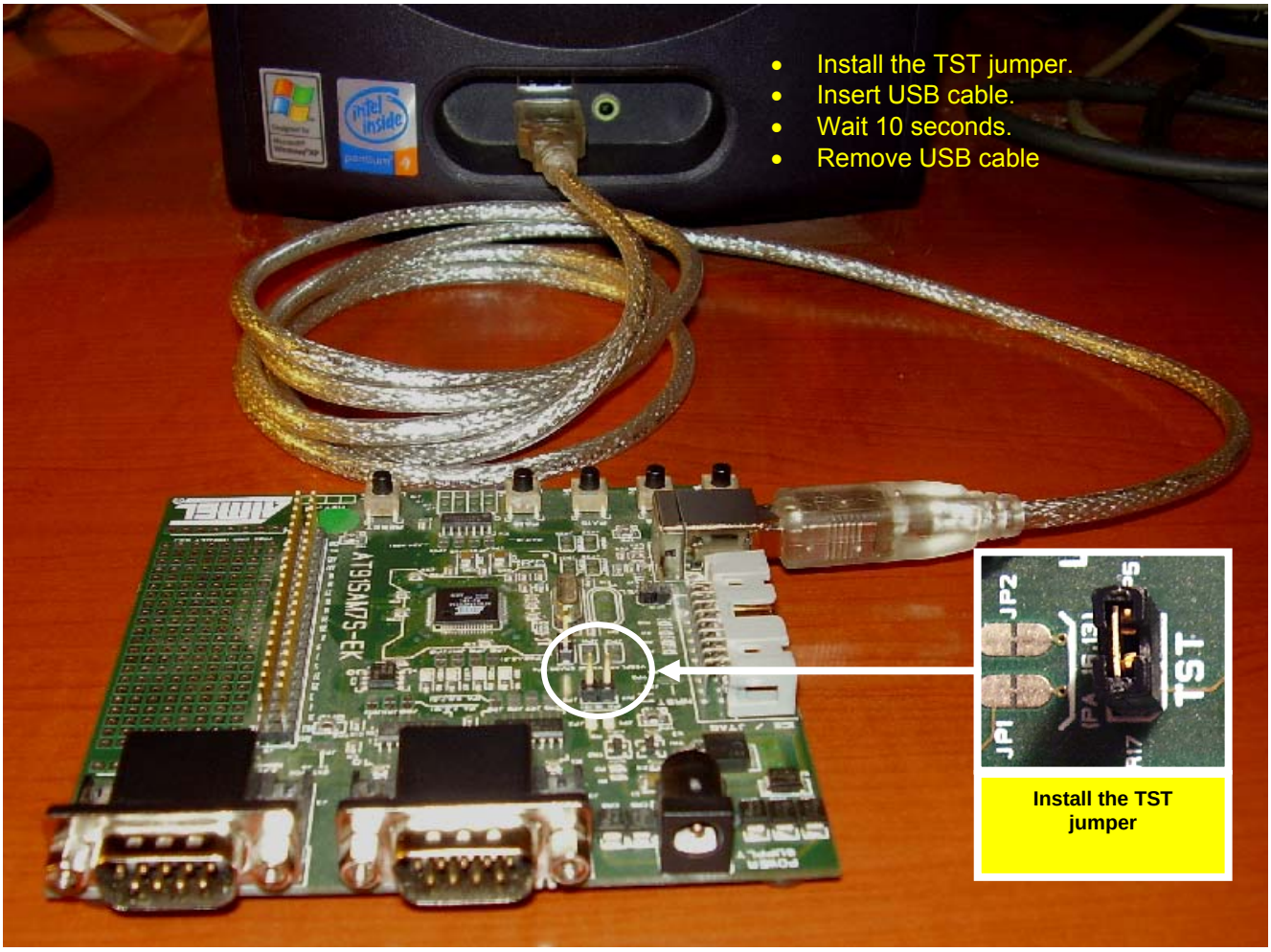
Notice that the “objcopy” utility has created a “**main.bin**” file; this is required by the SAM-BA flash programming Utility.

If there are compile or link errors, they will be visible in the Console view and the “**Problems**” tab will show more detail about any problems. You can click on the “problems” and jump directly to the offending source line.

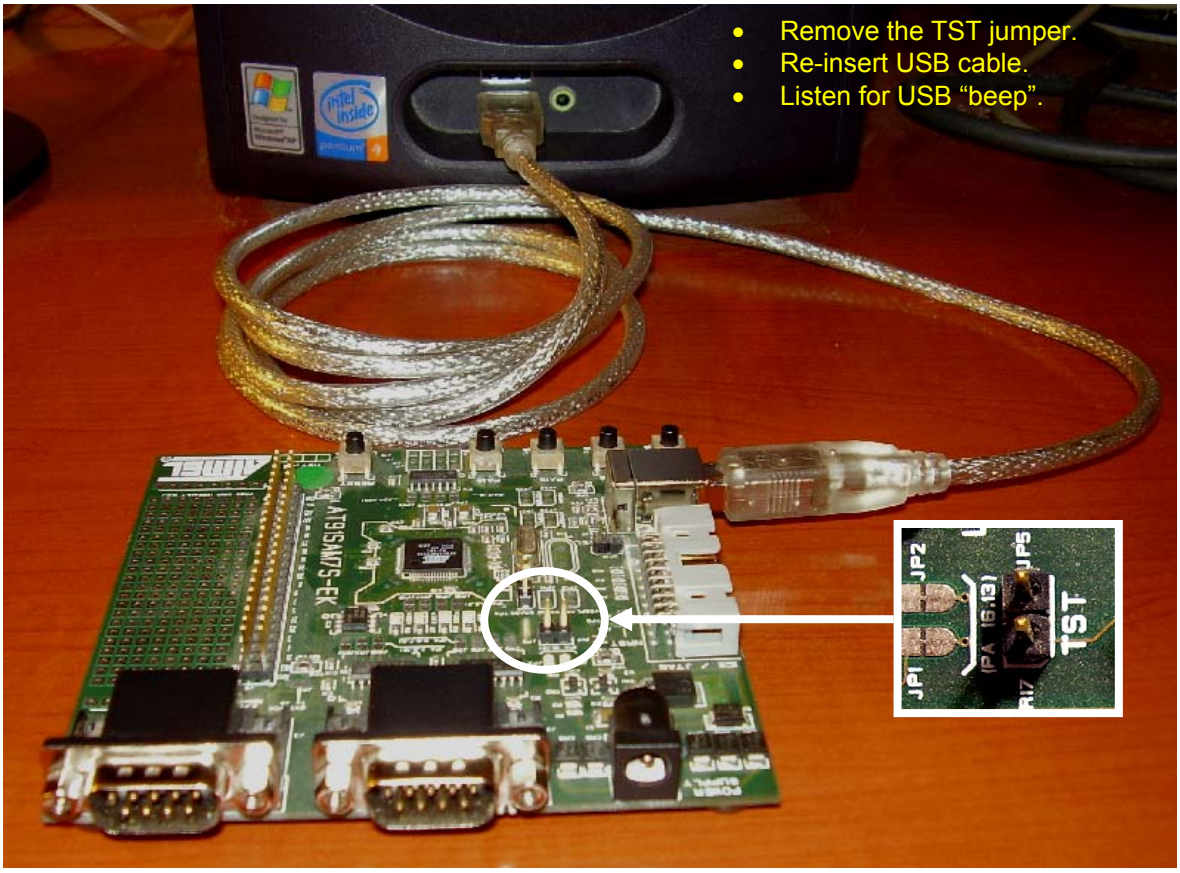
Programming the Application into Flash

Now let's use the **SAM-BA** Boot Assistant to program our blinker application into FLASH memory.

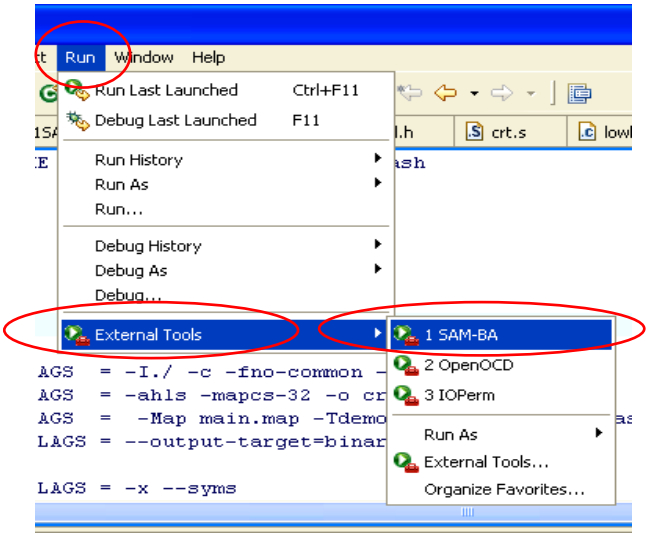
On the Atmel **AT91SAM7S-EK** evaluation board, locate and install the TST jumper. Plug in the USB cable as shown below. This powers the board. Wait 10 seconds. In this mode, the **AT91SAM7S-EK** board is erasing flash and loading in a USB driver.



Now remove the USB cable and remove the TST jumper. Reconnect the USB cable as shown below to apply power and boot the USB driver. You should hear the familiar USB "beep" indicating that Windows has detected the USB device.

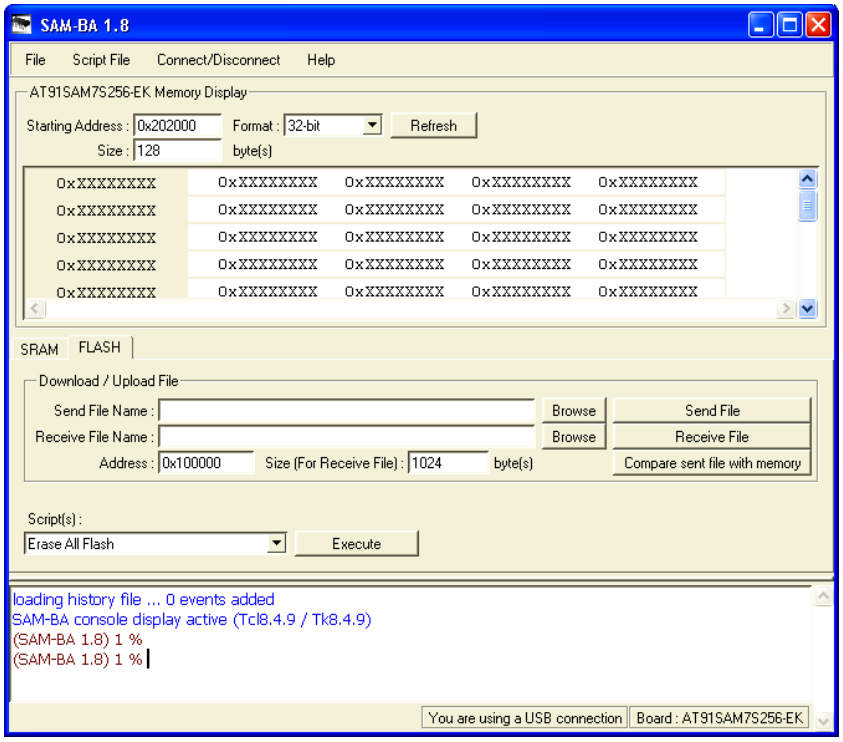


Using the Eclipse “**Run**” pull-down menu, click on “**Run – External Tools – SAM-BA**”.



When the small SAM-BA communications dialog window appears, click on “**USB connection**”.

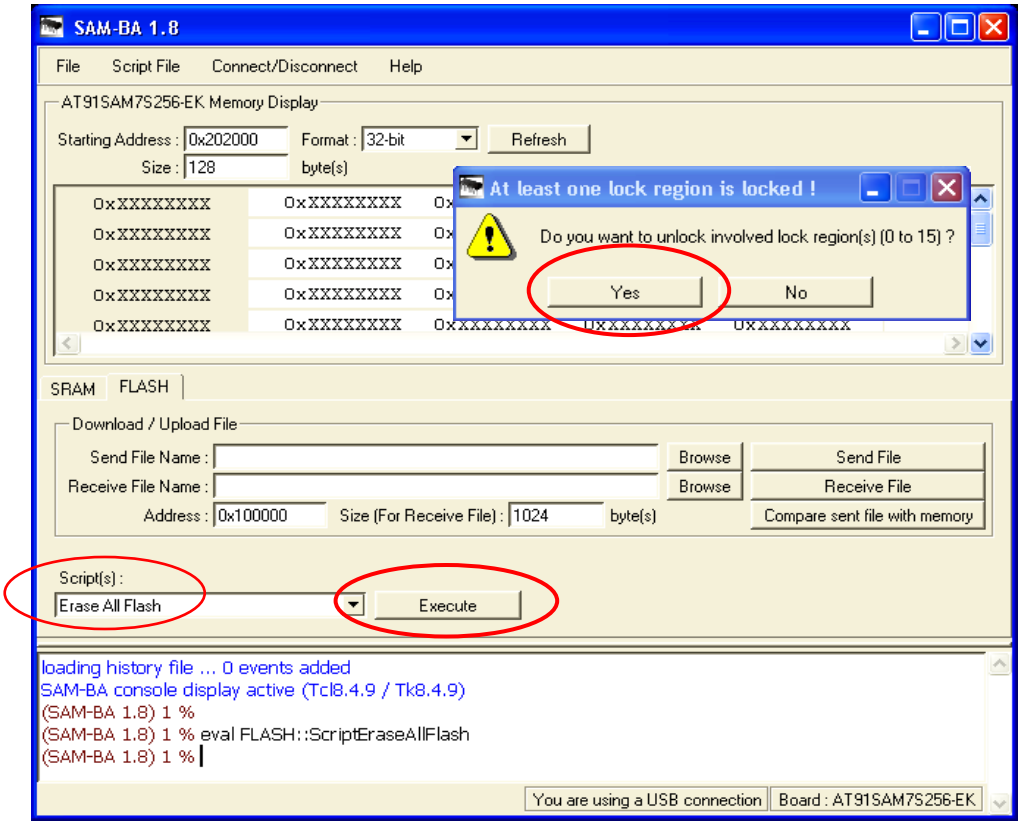
Now the SAM-BA main display will appear. Note in the bottom status line it reports that “You are using a USB Connection” and correctly identifies the board as “AT91SAM7S256-EK”.



As a simple test, let’s erase the onboard FLASH memory. Under the “**Scripts**”, select “**Erase all Flash**” and click “**Execute**”.

A reminder that one FLASH region is “locked” will appear. Click “**Yes**” to unlock that region.

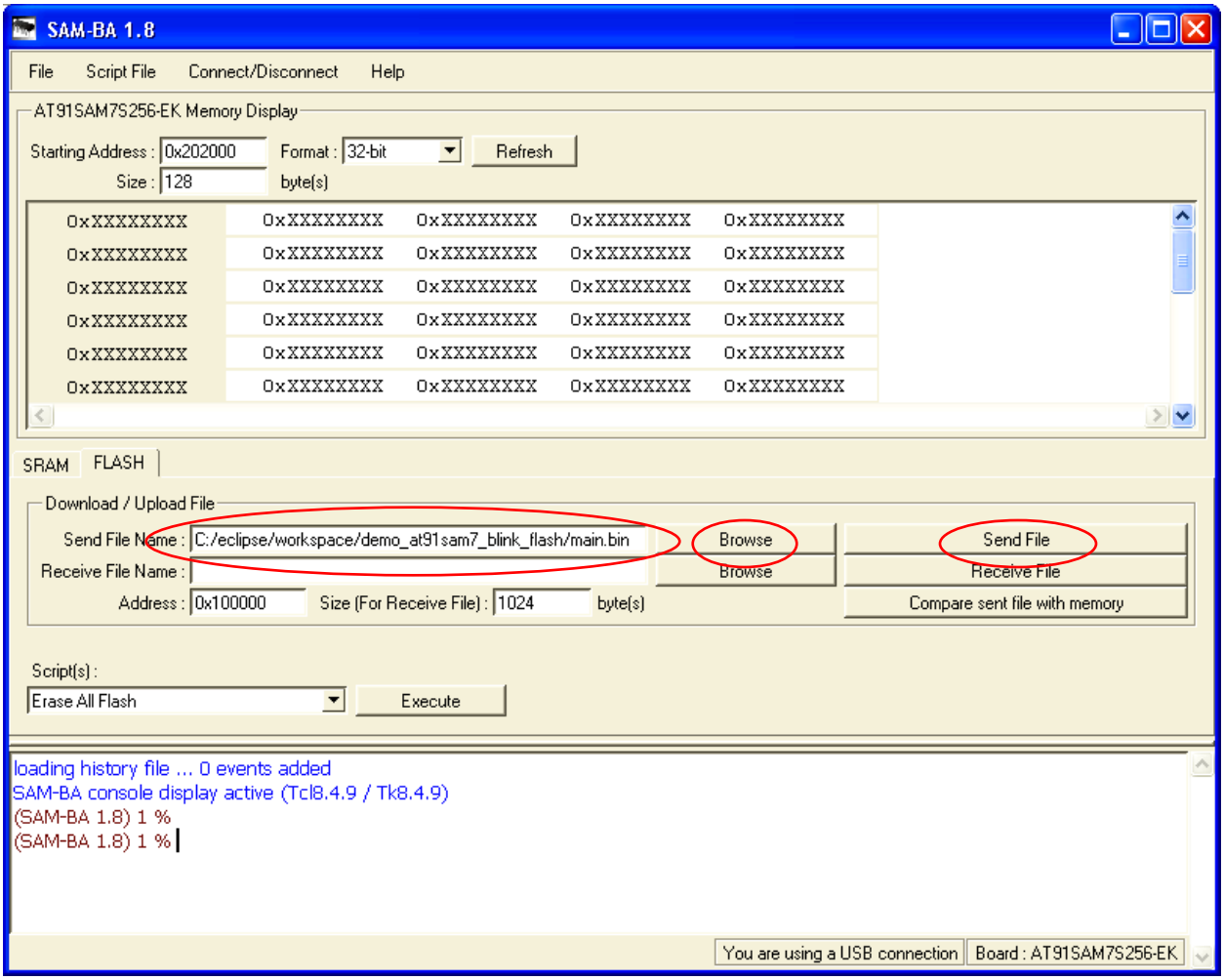
The SAM-BA utility will erase the FLASH as indicated in the command history at the bottom.



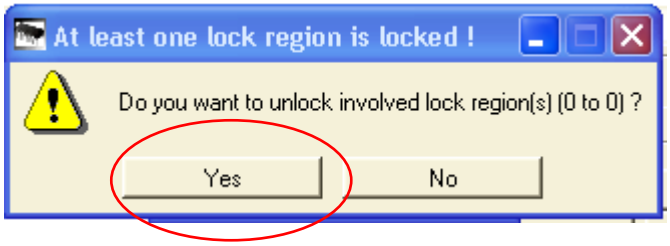
Be aware that erasing the AT91SAM7S256 onboard flash also wiped out the USB driver. Therefore, re-inserting the USB cable will not give the familiar USB beep signal indicating detection of the USB peripheral. In other words, you cannot restart SAM-BA again without reloading the USB driver.

Now we can program our blink application into FLASH memory. Note that the SAM-BA utility is expecting a "binary" file; this is the file "**main.bin**".

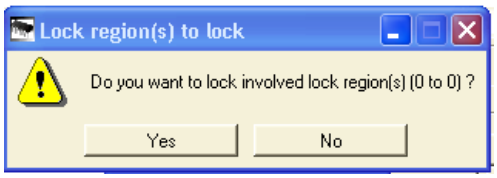
Within the Flash Download/Upload File tab, use the "**Browse**" button to find and select the binary file. In this example, it is the file: **c:/eclipse/workspace/demo_at91sam7_blink_flash.bin**. Then click "**Send File**" to actually send the file and program the flash.



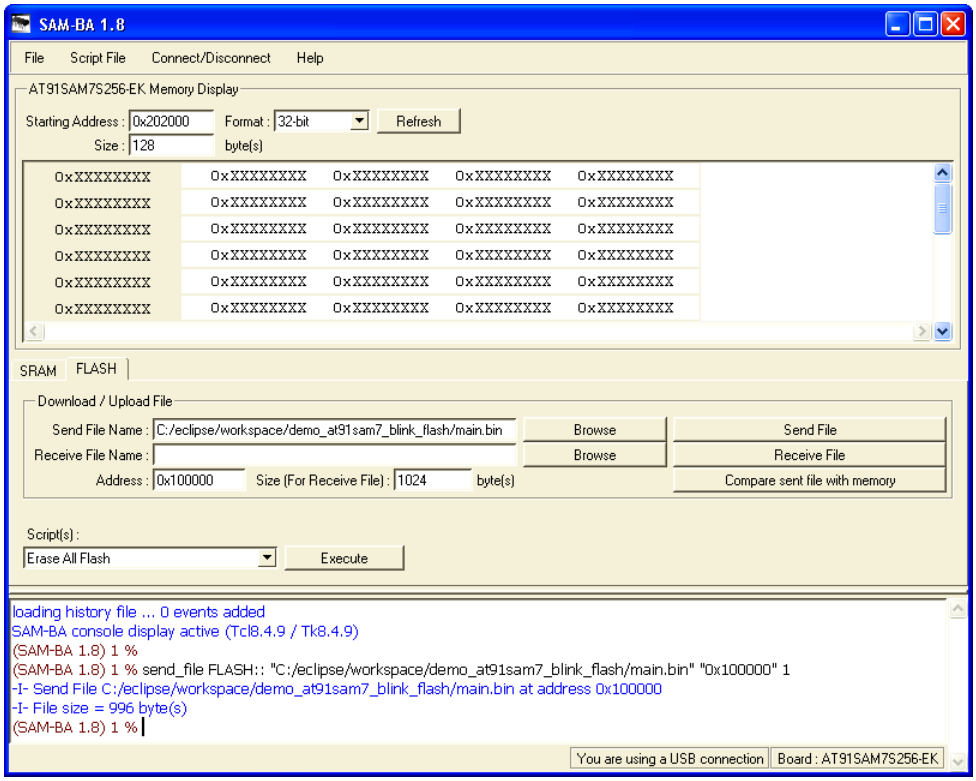
A message box will appear asking if you want to unlock a "locked" region. Click "Yes".



The binary file will now load into the FLASH memory. Another message box will pop up asking if you want to lock the region you just unlocked; click “**Yes**”.



Note that the console region at the bottom of the SAM-BA shows that 996 bytes were loaded into FLASH.

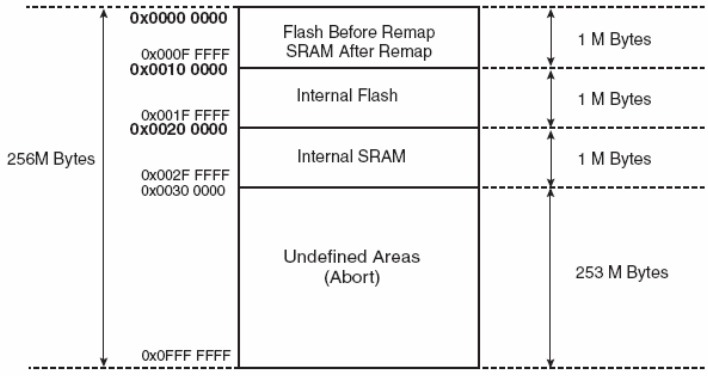


Alert readers might notice that the summary indicates that the 996 bytes were loaded at address 0x100000. This is true. However, page 19 of AT91SAM7S256 data package shows that FLASH is actually at address 0x100000 and is subsequently “mapped” into the 1 mb region at address 0x000000 at boot when the remap control register MC_RCR bit 0 is cleared (the default).

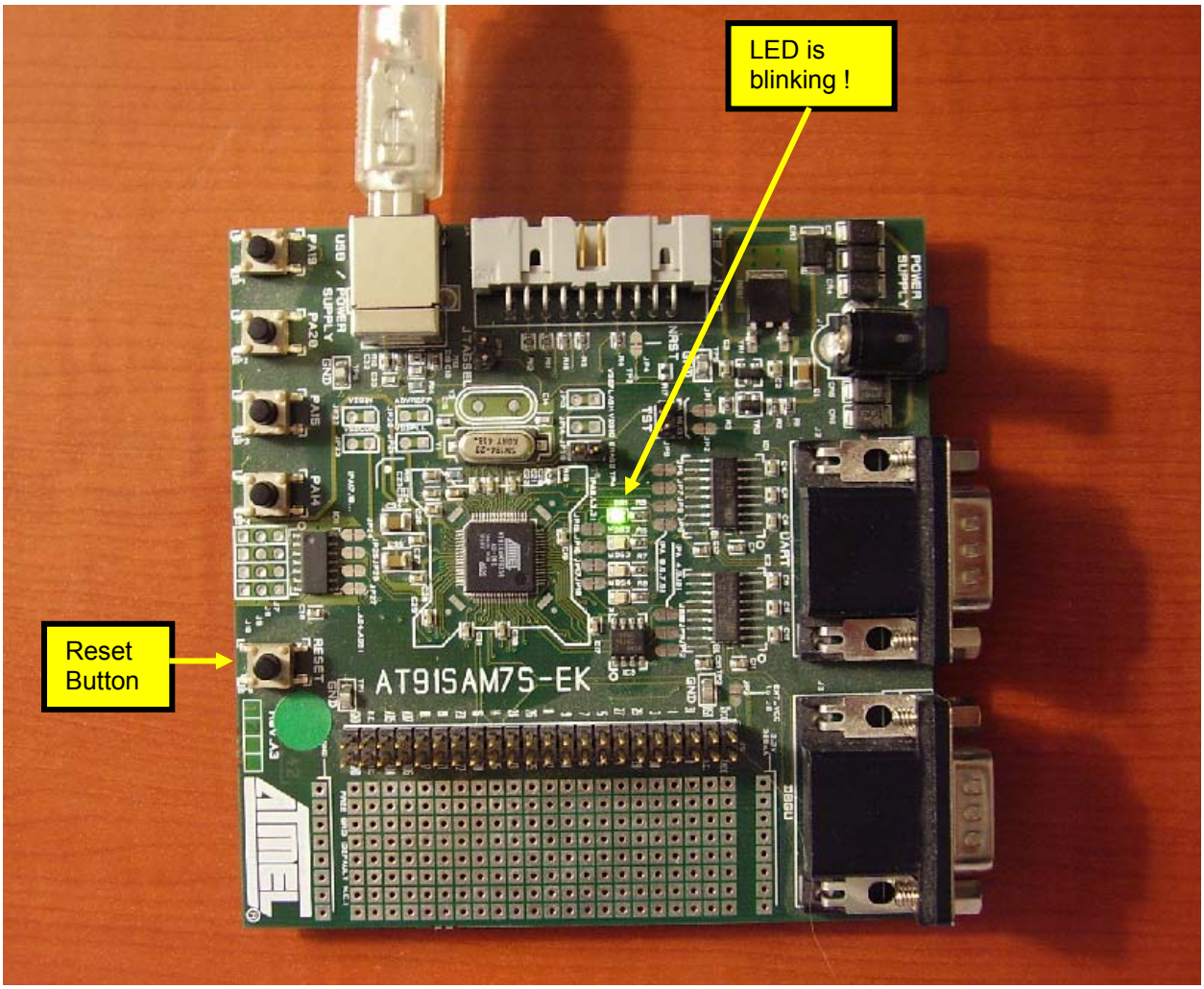
9.5.3 Internal Flash

The AT91SAM7S256/128/64/321/32 features one bank of 256/128/64/32/32 Kbytes of Flash. At any time, the Flash is mapped to address 0x0010 0000. It is also accessible at address 0x0 after the reset and before the Remap Command.

Figure 9-1. Internal Memory Mapping



To test the application, hit the “Reset” button on the Atmel AT91SAM7S-EK. The board is still powered from the USB cable. The LED1 should start blinking.



Congratulations! You now have a full-fledged ARM cross development system operational and it didn’t cost a thing!

Debugging the FLASH Application

Hardware Setup

Unless you are a perfect programmer, you will occasionally require the services of a debugger to trap and identify software bugs. Eclipse has a wonderful visual source code debugger that interfaces to the GNU GDB debugger.

The connection from your PC to the AT91SAM7S-EK target board's JTAG connector is accomplished using an inexpensive device called a "wiggler". This can be purchased from Olimex for $19.00 US. It's just a simple voltage level-shifter.

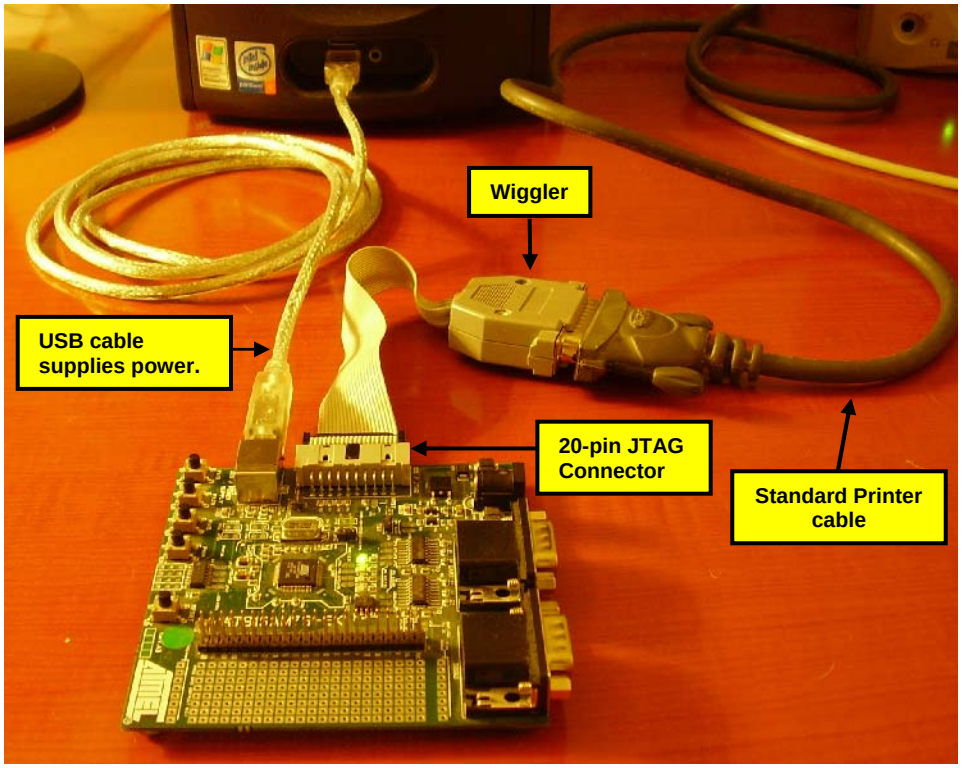
The ARM-JTAG device is available from:

www.olimex.com

www.sparkfun.com

www.microcontrollershop.com

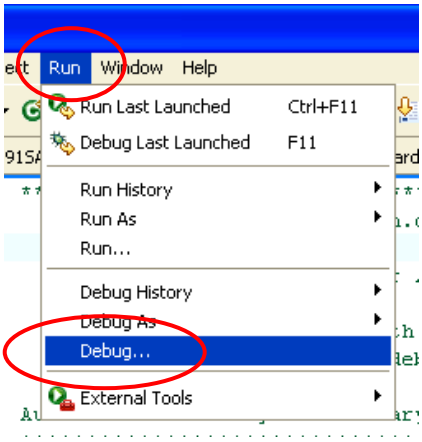
The following is the hardware setup to debug the Atmel AT91SAM7S-EK evaluation board using the inexpensive "wiggler" device. The USB cable is left connected to supply board power. The ARM-JTAG interface is attached to the PC's printer port; in the author's setup, a stock printer cable from the local computer store was employed. The JTAG 20-pin connector is keyed so it can't be inserted improperly.



Create a Debug Launch Configuration

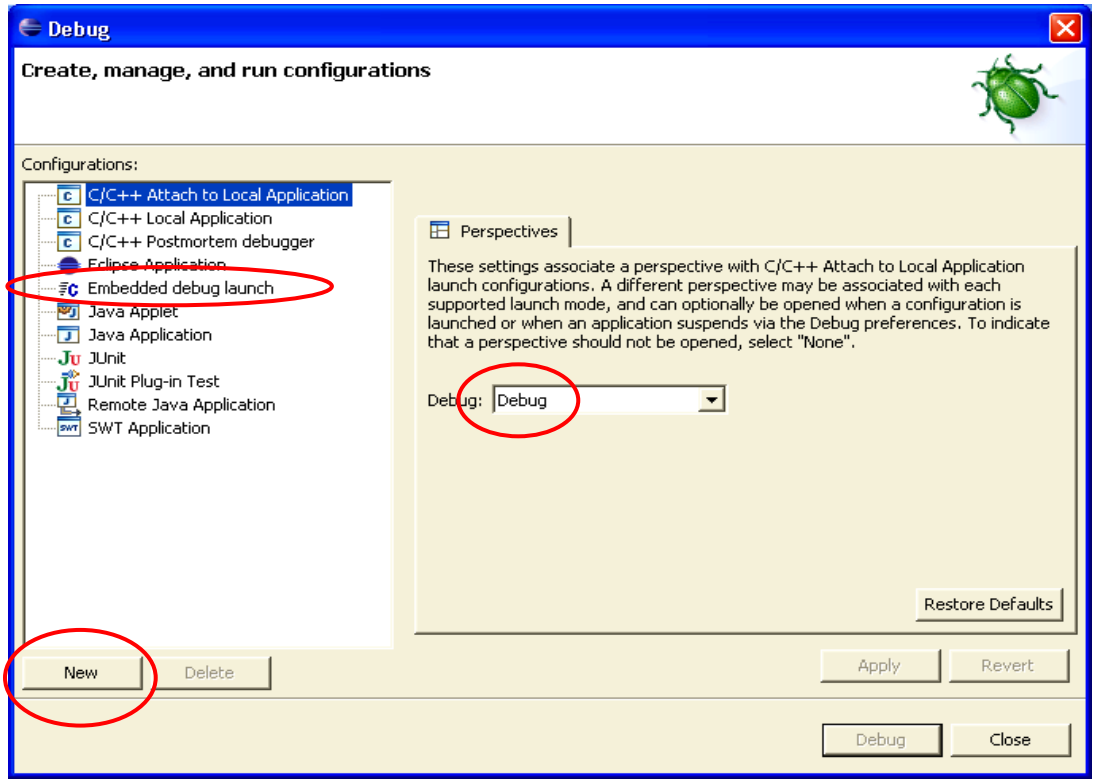
Before we can debug the application, we have to create a Debug Launch Configuration for this project. The Debug Launch Configuration locates the GDB debugger for Eclipse, locates the executable file (in this case it's only used to look up symbol information), and provides a startup script of GDB commands that are to be run as the debugger starts up.

Click on "**Run – Debug…**" to bring up the Debug Configuration Window.



In the "Debug – Create, manage, and run configurations" window, click on "**Embedded debug launch**" followed by "**New**". This is the special debug launch configuration created by Zylin.

Also ensure that under the "perspectives" tab "Debug" is chosen (it's the default).

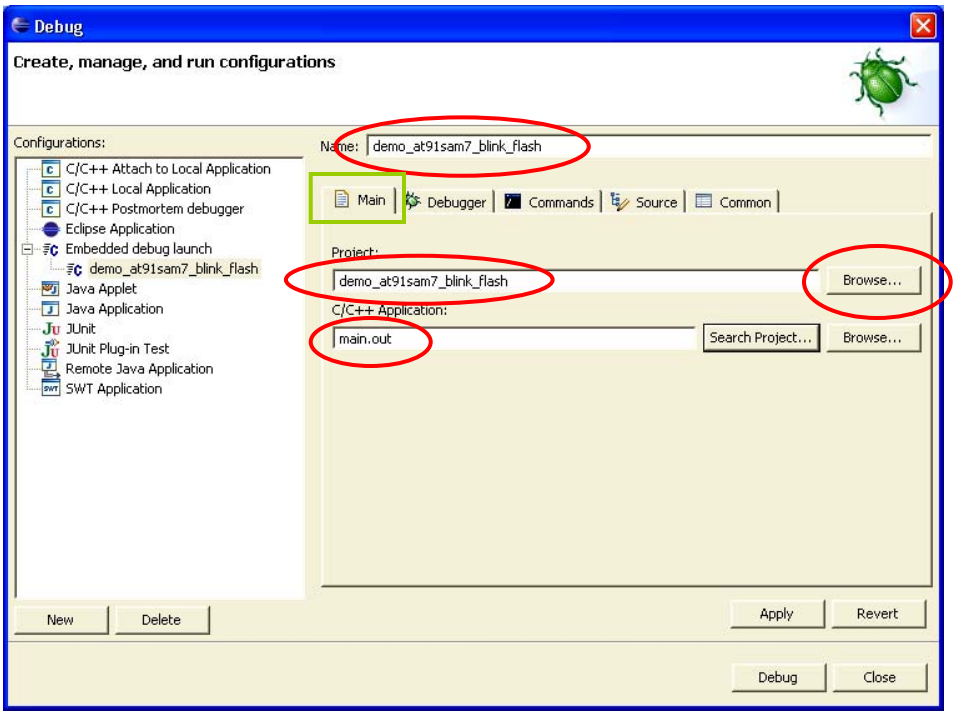


The Debug configuration window now changes to a multiple-tab debug configuration form, as shown below.

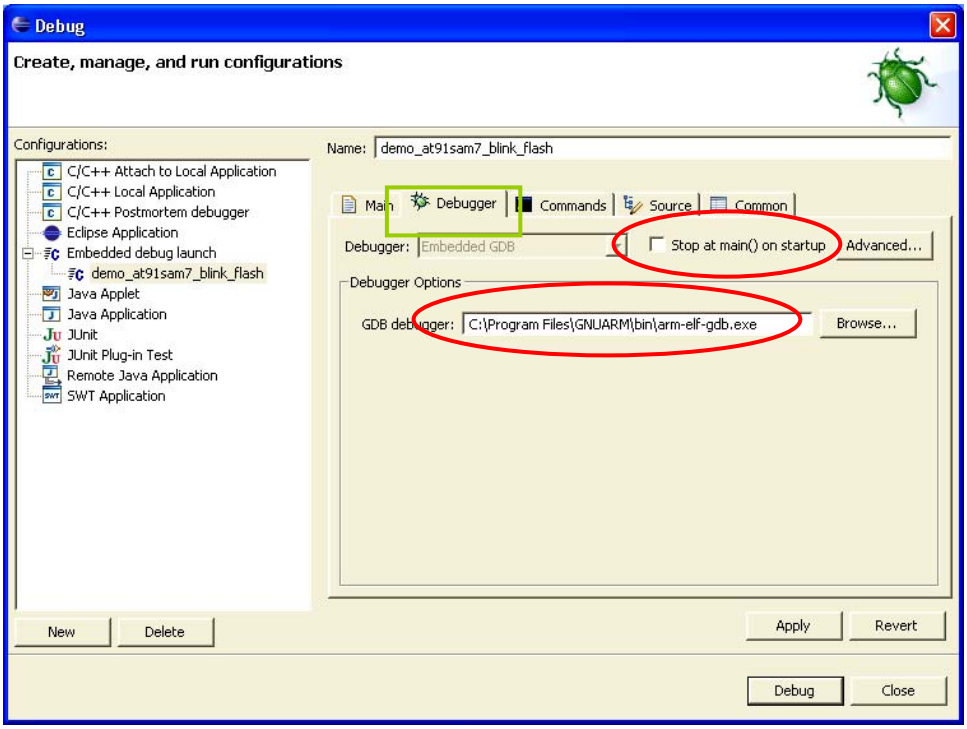
Now fill in the Debug Configuration **Main Tab** window as shown below. The **Name** can be anything you choose, but since there is usually going to be a debug configuration for each project you set up, the name of the project itself is a wise choice.

You can use the "**Browse**" button to find the matching project; in the example below it is the "**demo_at91sam7_blink_flash**" project.

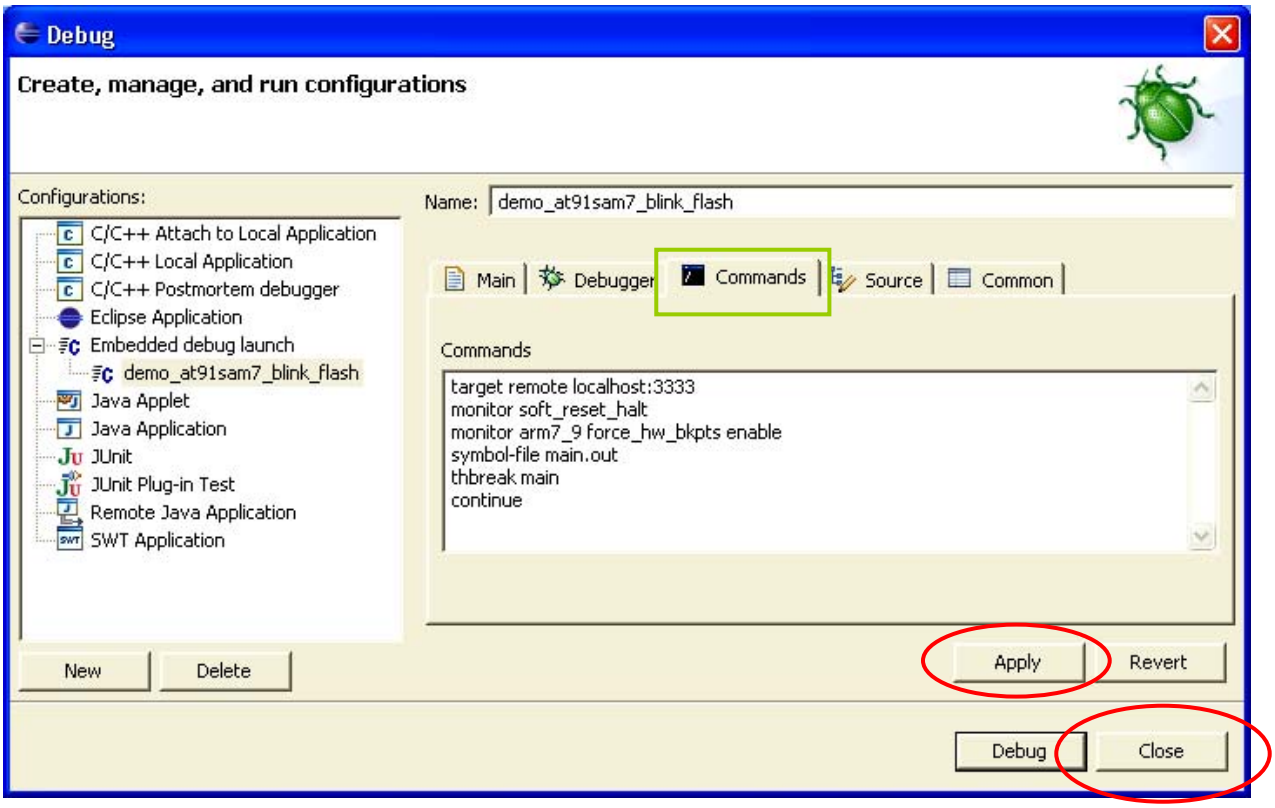
In the "**C/C++ Application**" text box, the ELF file "**main.out**" is the correct choice. Note here that this file is only used by the debugger to resolve symbol addresses. We have already loaded the executable code into AT91SAM7S256 flash memory using the "**main.bin**" file and the **SAM-BA** Flash Loader.



Now click on the "**Debugger**" tab. Use the "**Browse**" button to search for the GDB debugger. In this case, it is the file "**arm-elf-gdb.exe**". This file is in the folder "**c:\Program Files\GNUARM\bin**".
Note that the box "**Stop at Main( ) on startup**" is left un-checked. We will be emitting a debugger startup script command to set a breakpoint at main( ) instead.



Under the “**Commands**” tab, enter the six GDB startup commands as shown below.



The six GDB startup commands require some explanation. If the command line starts with the word “monitor”, then that command is an OpenOCD command. Otherwise, the command is a GDB command.

OpenOCD commands are described in the OpenOCD documentation which can be downloaded from:
http://developer.berlios.de/docman/display_doc.php?docid=1367&group_id=4148

GDB commands are described in several books and in the official document that can be downloaded from:
http://dsl.ee.unsw.edu.au/dsl-cdrom/gnutools/doc/gnu-debugger.pdf

target remote localhost:3333

This is a **GDB** command. 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:3333** (the default specification for the **OpenOCD** GDB Server).

monitor soft_reset_halt

This is an **OpenOCD** command This is a special reset command developed by Dominic Rath for ARM microprocessors.

monitor arm7_9 force_hw_bkpts enable

This is an **OpenOCD** command. It converts all breakpoint commands emitted by Eclipse/GDB into hardware breakpoint commands. The ARM7 architecture supports two hardware breakpoints. This allows you to debug a program in FLASH.

symbol-file main.out

This is a **GDB** command. It instructs the debugger to utilize the symbolic information in the **main.out** file for debugging.

thbreak main

This is a **GDB** command. It sets a temporary hardware breakpoint at the entry point main(). Once the debugger breaks at main(), this breakpoint is automatically removed.

continue

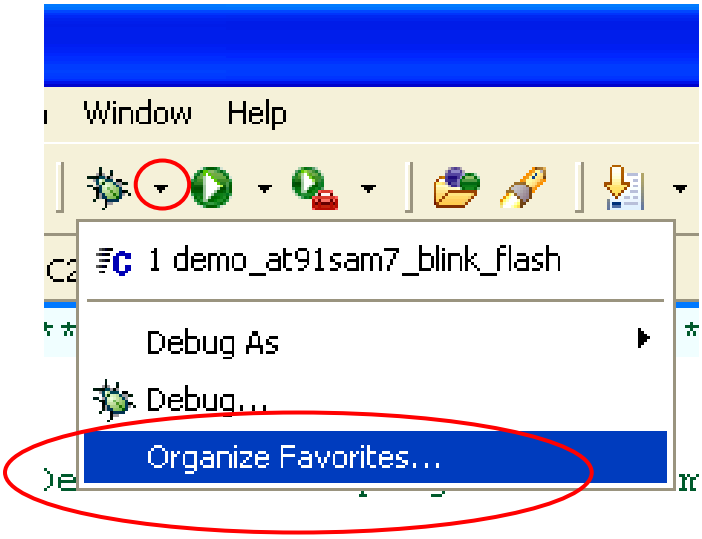
This is a **GDB** command. It forces the ARM processor out of breakpoint/halt state and resumes execution from the reset vector till it breaks at main().

The "Source" and "Common" tabs can be left at their default values. Click on "**Apply**" and then "**Close**" above to finish specification of this Debug Configuration.

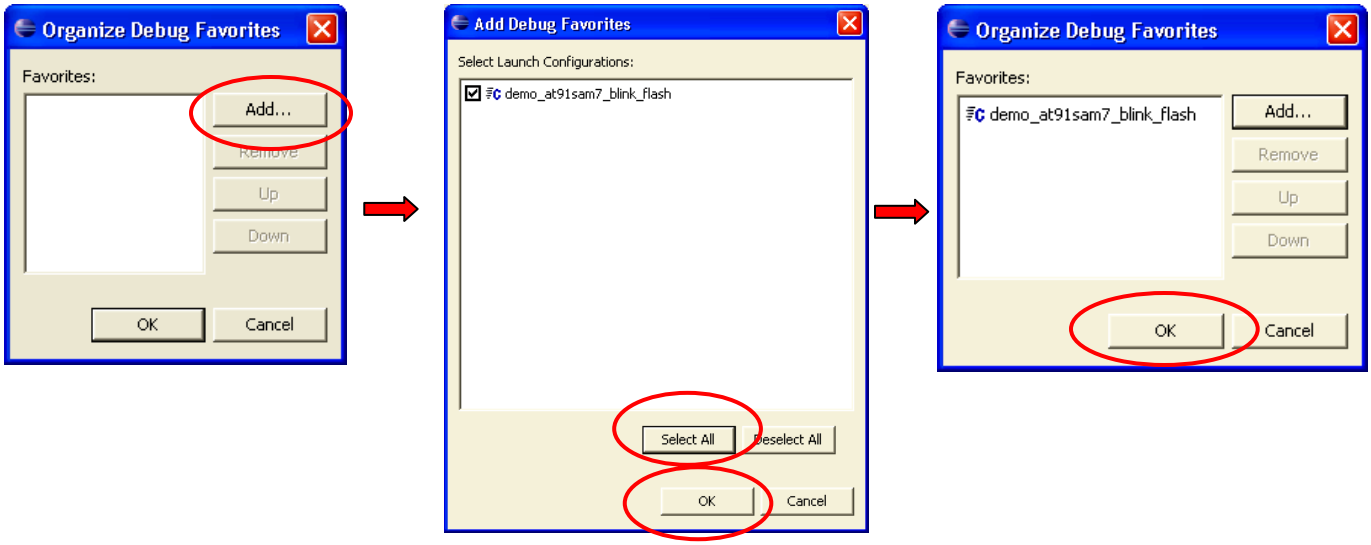
Eclipse may ask you if you want to save this configuration, answer "**Yes**".

One final maneuver is to add the **demo_at91sam7_blink_flash** embedded debug launch configuration into the Debug pull-down menu's list of favorites.

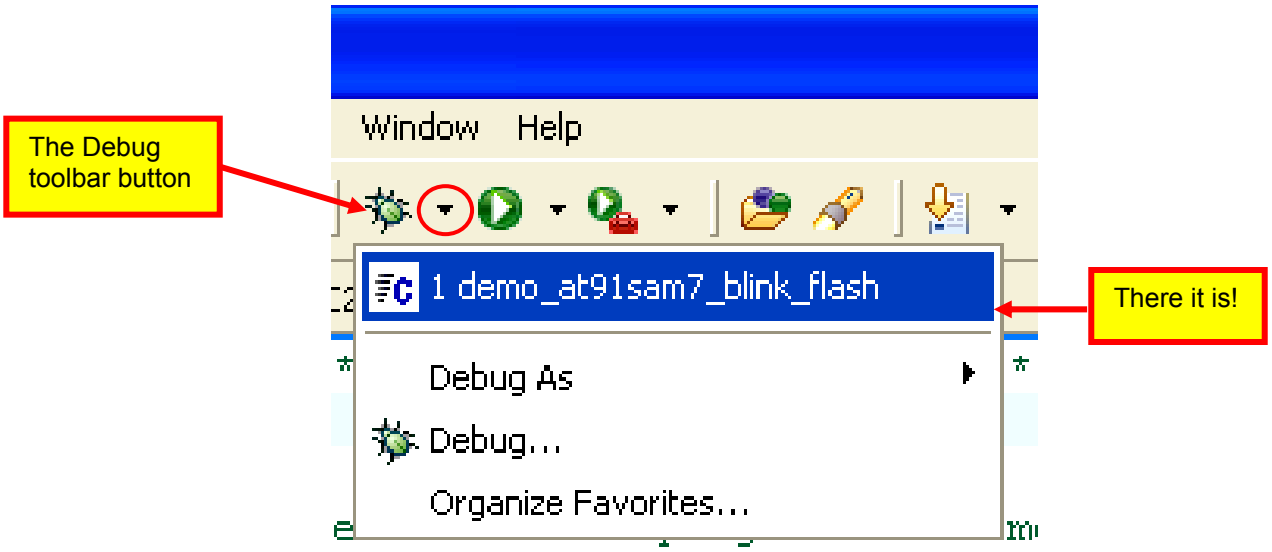
In the toolbar, click on the **down arrowhead** next to the debug symbol and then click "**Organize Favorites…**".



In a sequence similar to other “Organize Favorites” operations that we have already performed, Click on “**Add…**” and either checkmark the “**demo_at91sam7_blink_flash**” or click the “**Select All**” button. Finally, click “**OK**” to enter this debug launch configuration into the debugger list of favorites, as shown below.



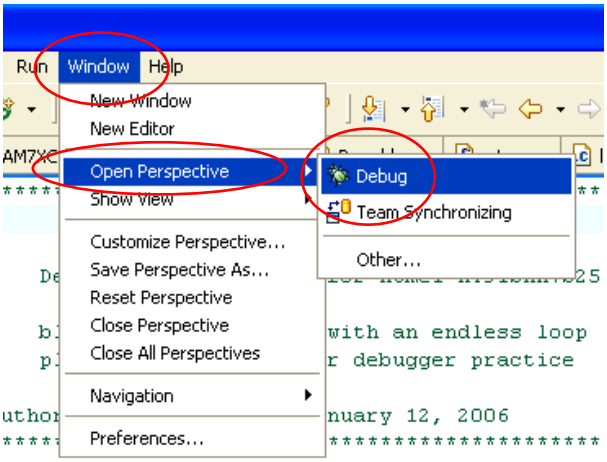
Now when you click on the Debug Toolbar button’s down arrowhead, you will see the “**demo_at91SAM7_blink_flash**” debug launch configuration installed as a favorite, as shown below.



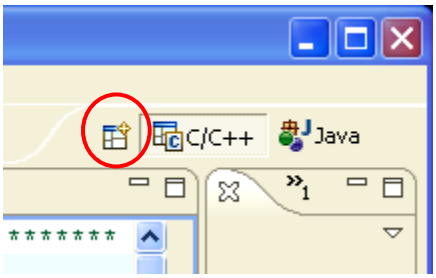
Now everything is in place to debug the project that we loaded into FLASH memory with the SAM-BA utility.

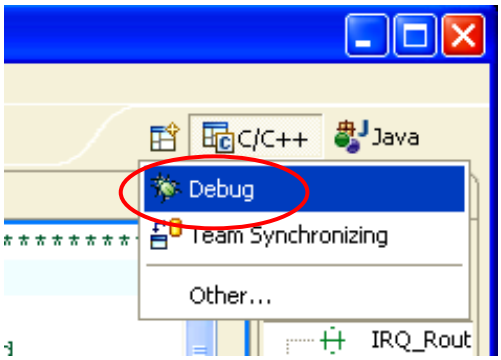
Open the Eclipse Debug Prespective

To debug, we need to switch from the C/C++ perspective to the Debug perspective. The standard way is to click on "**Window – Open Perspective – Debug**" as shown below.

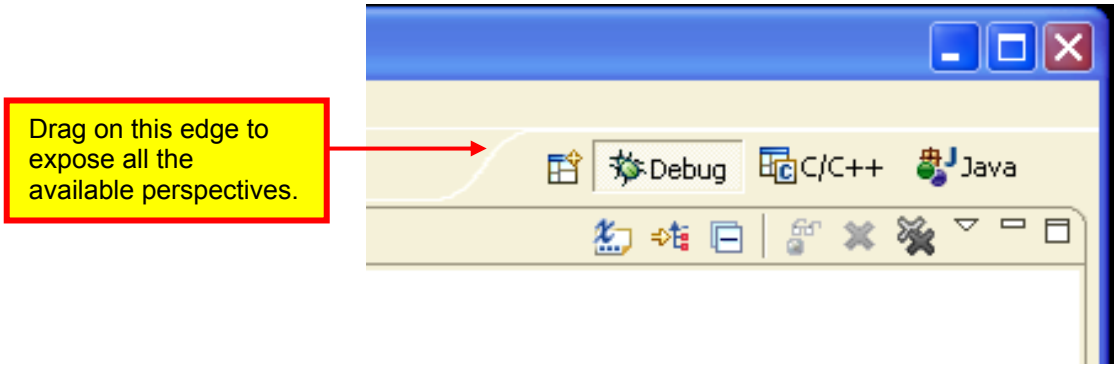


A more convenient way to switch perspectives is to click on the "perspective" buttons at the Eclipse upper-right window location. Click on the "**OpenPerspective**" toolbar button below on the left and then choose "**Debug**" when the other perspectives are displayed.

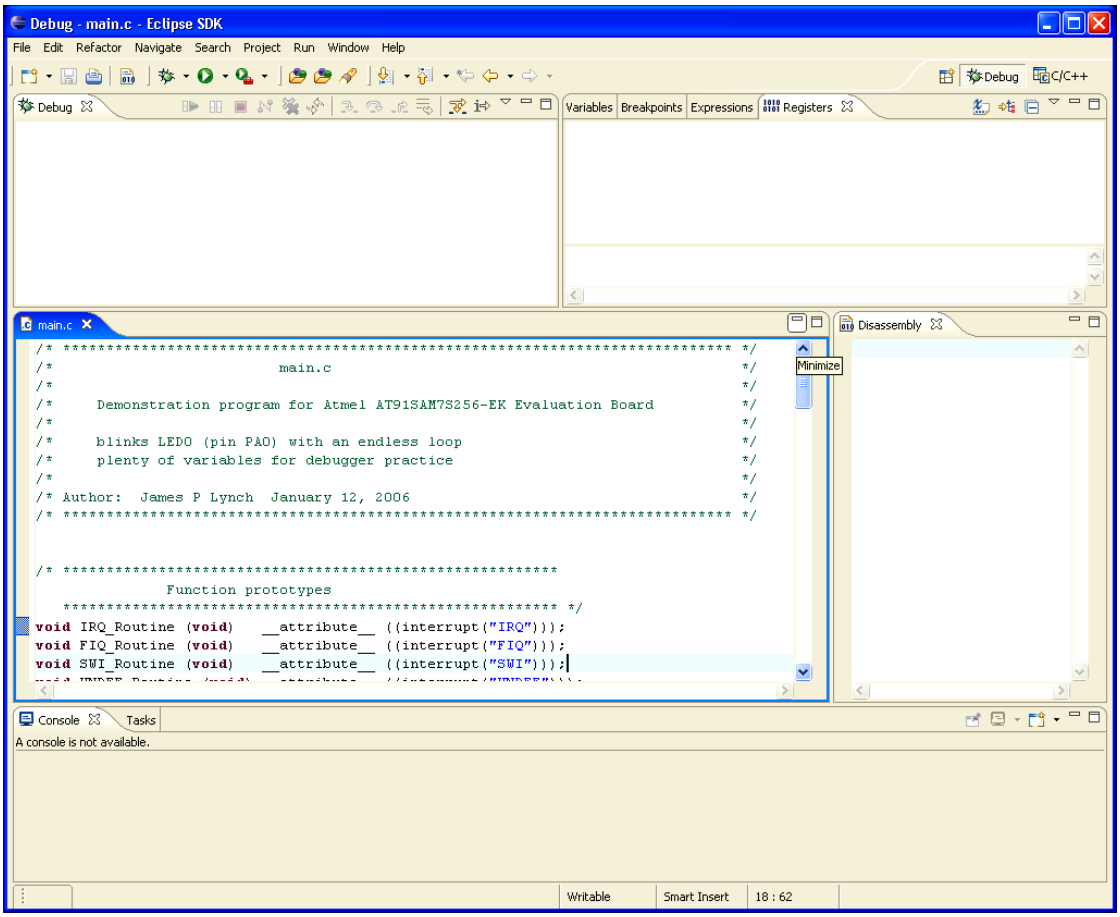




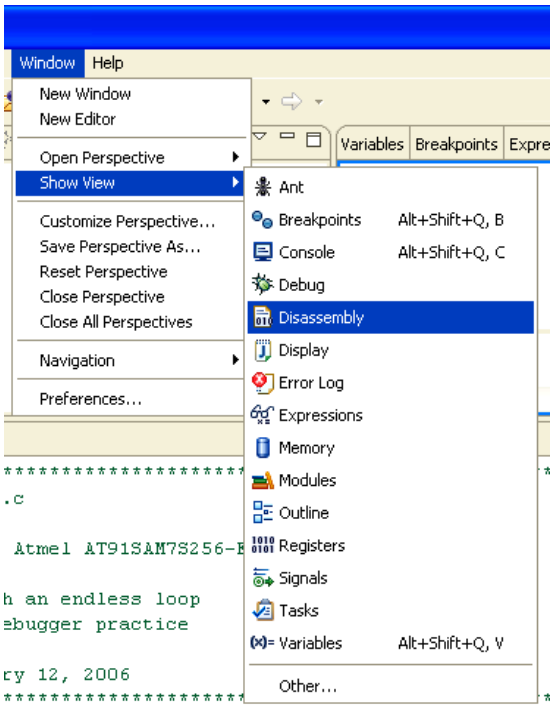
Now we have a "Debug" button as shown below. You may have to drag on the edge to expose all the perspective buttons. You can also right-click on any of the buttons and "Close" them to narrow the display to only the perspectives you are interested in.



Click on the “Debug” perspective button at the upper-right to open the Debug Perspective display. 



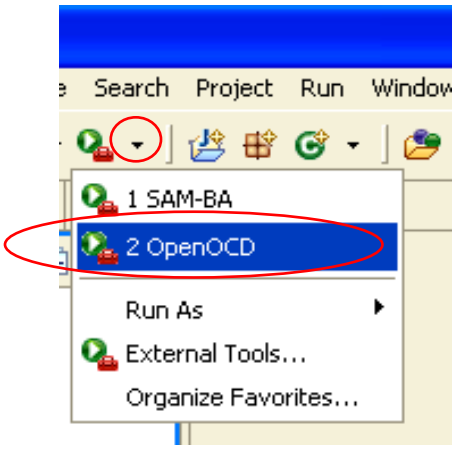
If your display doesn’t look exactly like the debug display above, click on “**Window – Show View**” and select any of the missing elements.



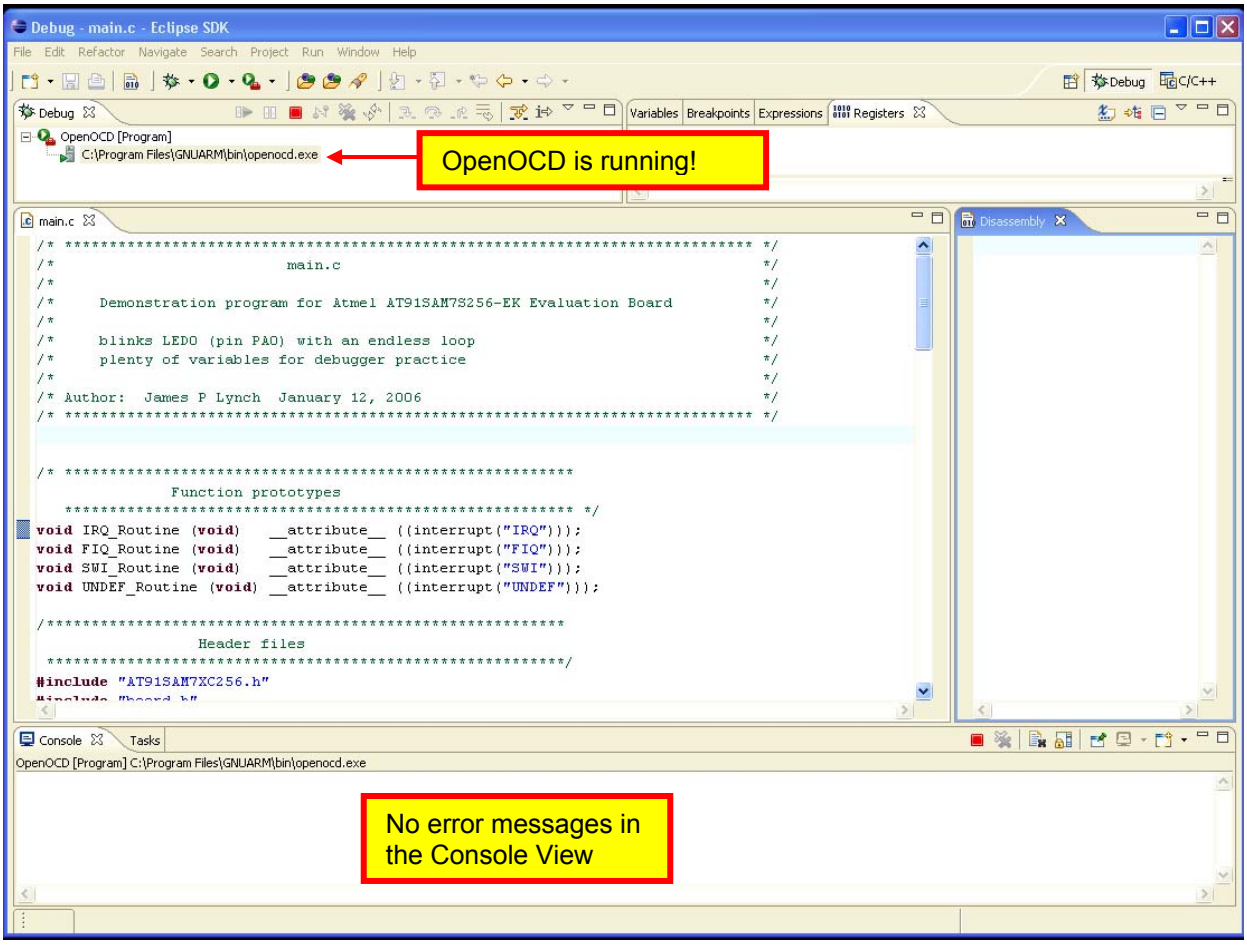
Starting OpenOCD

Before starting **OpenOCD**, get into the habit of cycling the power of the Atmel **AT91SAM7-EK** board. Pulling and then re-inserting the USB cable will accomplish this. If you are using a 9 volt power supply, remove and re-insert the power plug to do this.

To start **OpenOCD**, click on the "**External Tools**" toolbar button's down arrowhead and then select "**OpenOCD**". Alternatively, you can click on the "**Run**" pull-down menu and select "**External Tools**" followed by "**OpenOCD**".



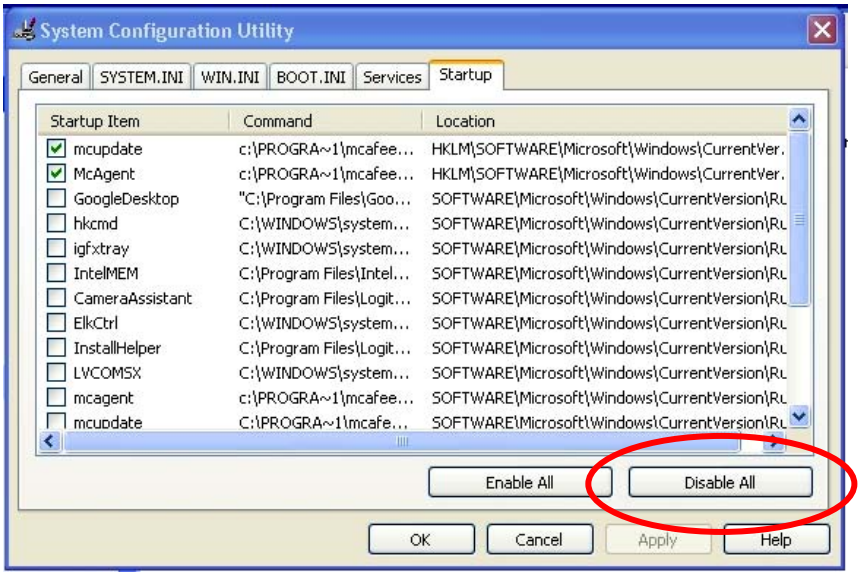
The debug view will show that **OpenOCD** is running and the console view shows no errors.



If for some reason, **OpenOCD** will not properly start in your system, you can try the following things.

- Cycle power on the target board before starting **OpenOCD**
- Make sure your computer is not running cpu-intensive applications in the background, such as internet telephone applications (**SKYPE** for example). The **OpenOCD/wiggler** system does "bit-banging" on the **LPT1** printer port which is fairly low in the Windows priority order.

  For Windows XP users, here is a simple way to get rid of all those background programs. Click "**Start – Help and Support – Use Tools… - System Configuration Utility – Open System Configuration Utility – Startup Tab**"

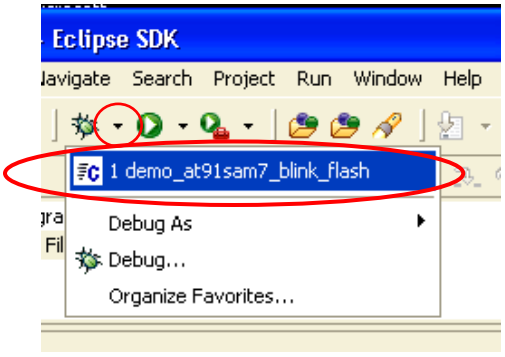


Click on "**Disable All**". Windows will ask you to re-boot and the PC will restart with none of the start-up programs running. Use the same procedure to reverse this action.

Start the Eclipse Debugger

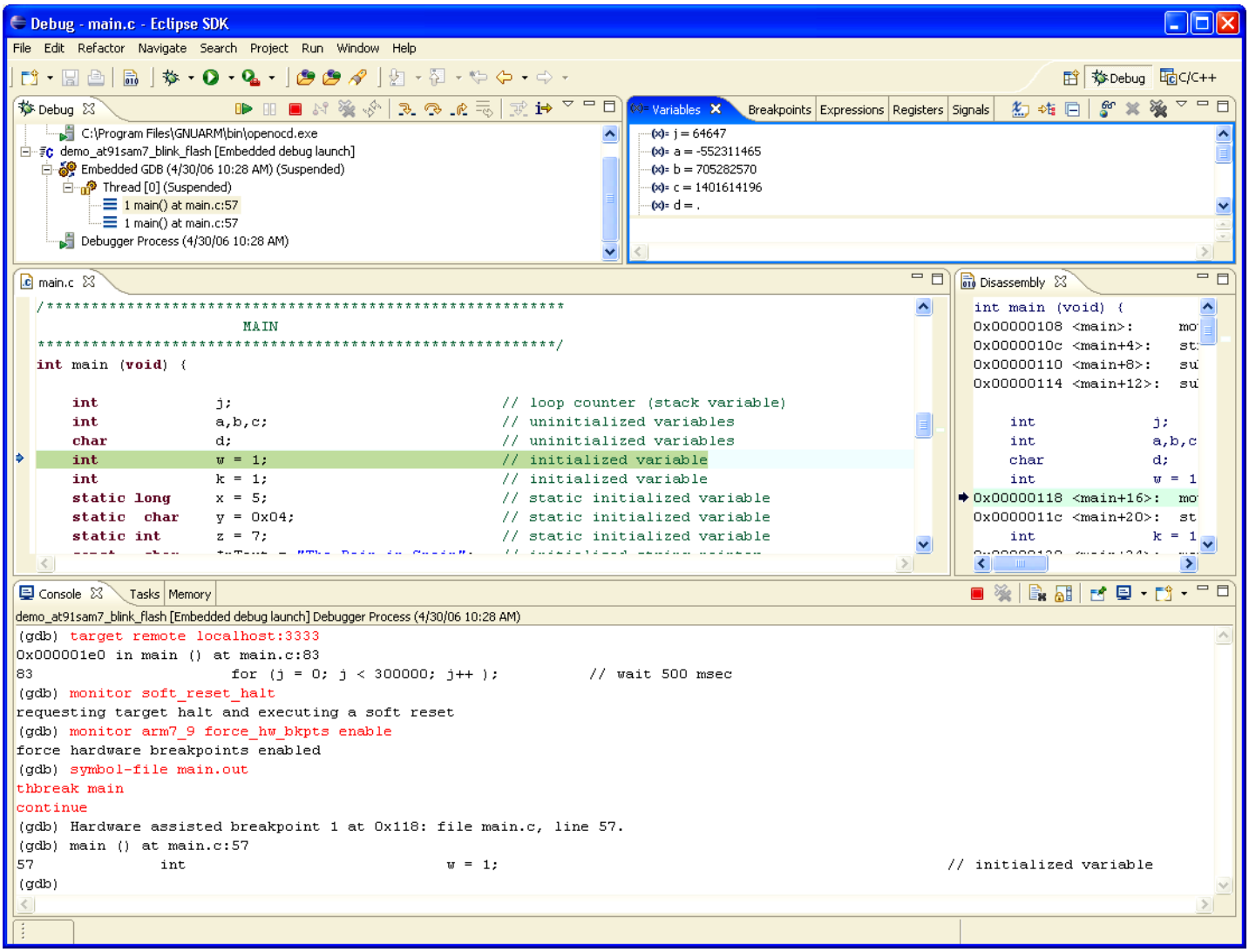
To start the Eclipse debugger, click on the "**Debug**" toolbar button's down arrowhead and select the debug launch configuration "**demo_at91sam7_blink_flash**" as shown below.

Alternatively, you can start the debugger by clicking on "**Run – Debug…**" and then select the "**demo_at91sam7_blink_flash**" embedded launch configuration and then click "**debug**". Obviously, the debug toolbar button is more convenient.



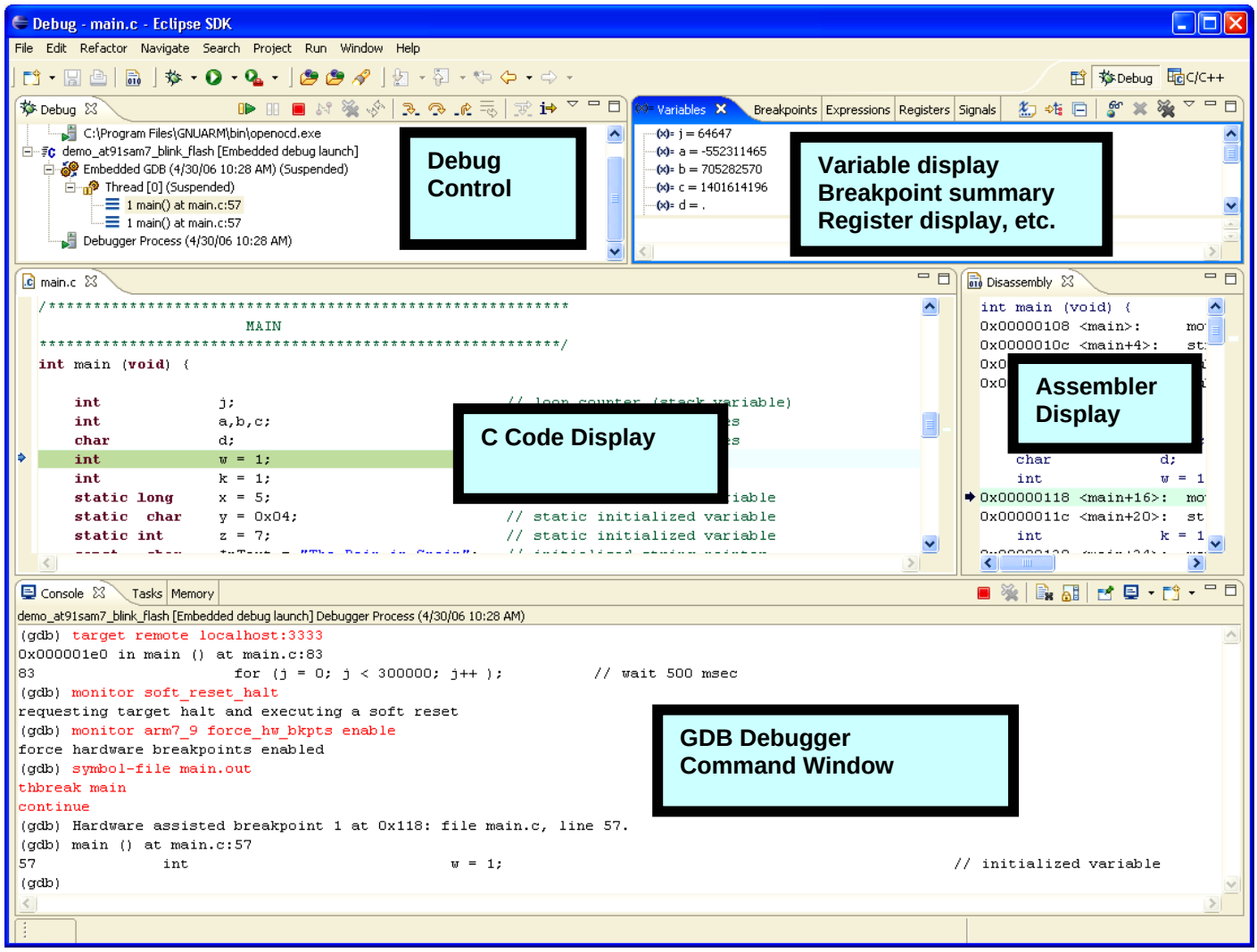
If the Eclipse debugger starts properly, the debug view (upper left) shows that the debugger has stopped at line 57 in main(). The console view (bottom) shows each GDB command that was executed in red letters and some **OpenOCD** messages after some of the commands.

If the Eclipse debugger doesn't connect properly, then there will be a progress bar at the bottom left status line that runs forever. In this case, terminate everything and power cycle the target board again.

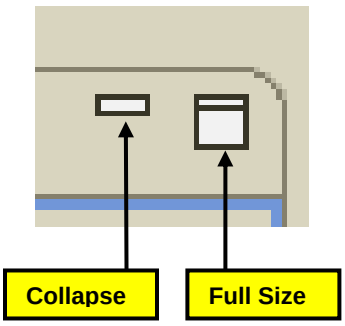


Components of the DEBUG Perspective

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

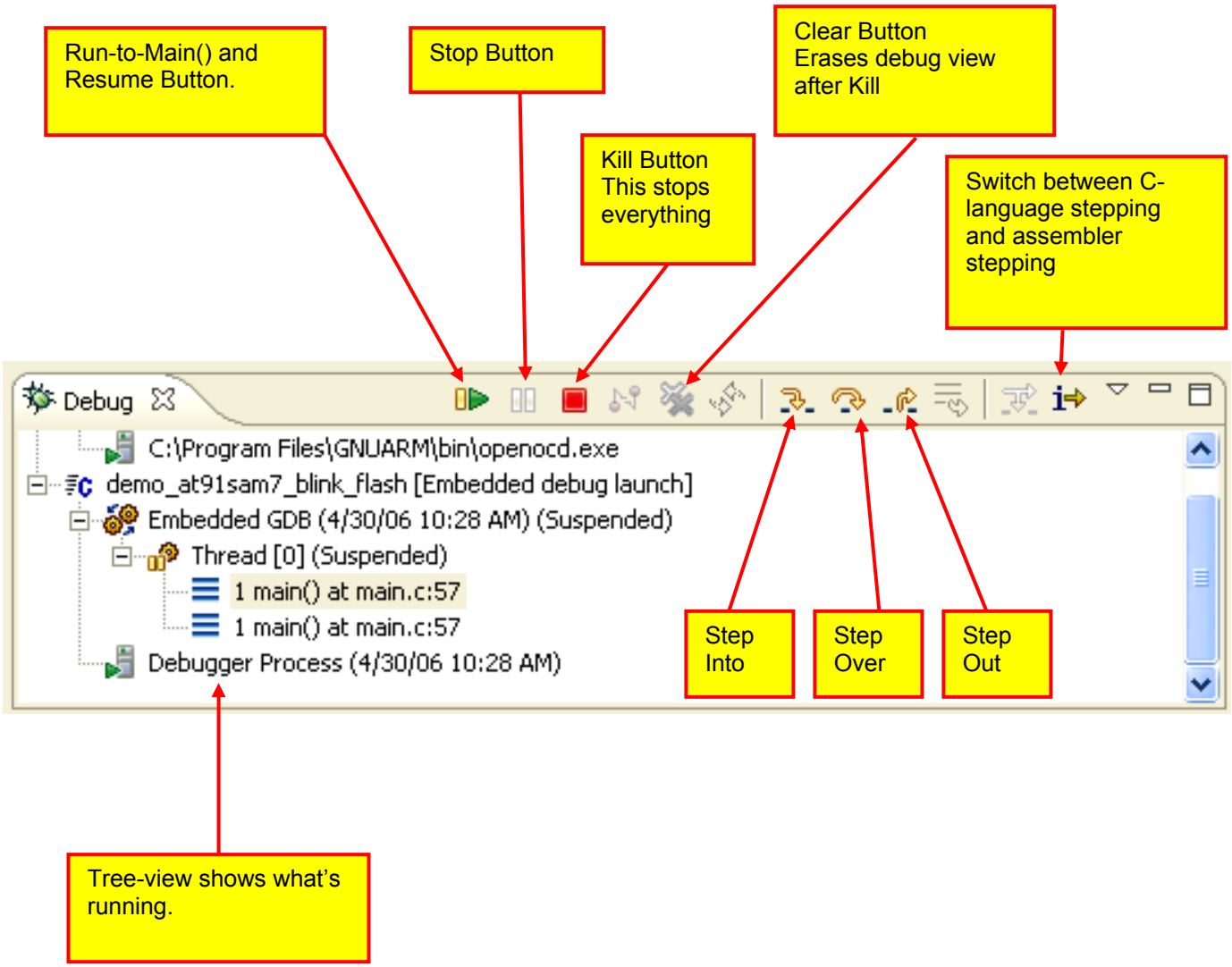


While this may be obvious to most, you can minimize and restore any of the windows in the Debug perspective by clicking on the "maximize" and "minimize" buttons at the top right corner of each window.



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 **Resume/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.

- You can only set two breakpoints at a time when debugging FLASH. If you are stepping, you should have no breakpoints set since Eclipse needs the hardware breakpoints for single-stepping.

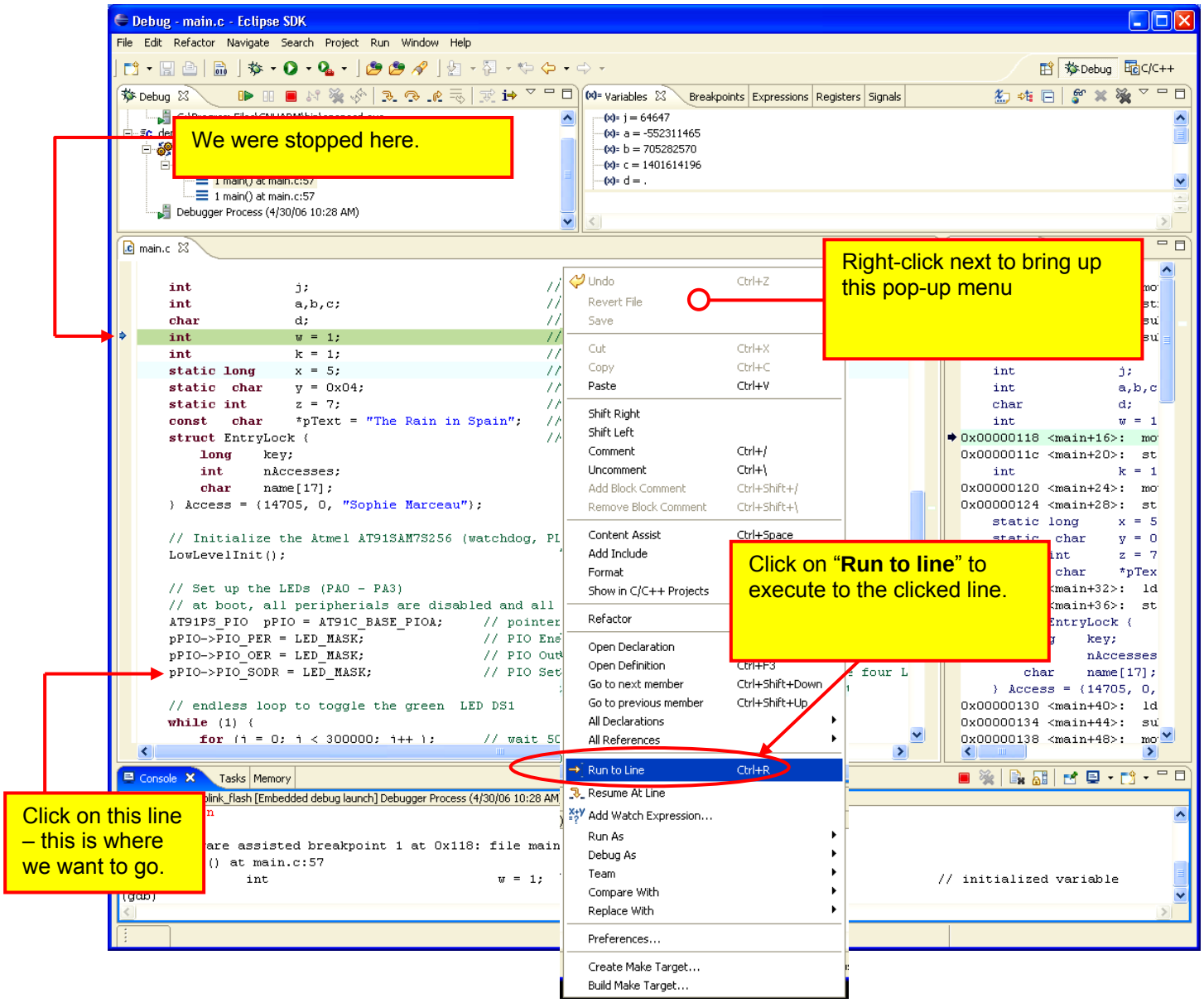
- If you re-compile your application, you must stop the debugger and OpenOCD, re-build and burn the main.bin file into FLASH using the Atmel SAM-BA Flash Utility. The Eclipse/GDB debugger cannot program FLASH memory. Well, to be fair, OpenOCD can indeed program flash memory; but this is not part of this tutorial.

Run and Stop with the Right-Click Menu

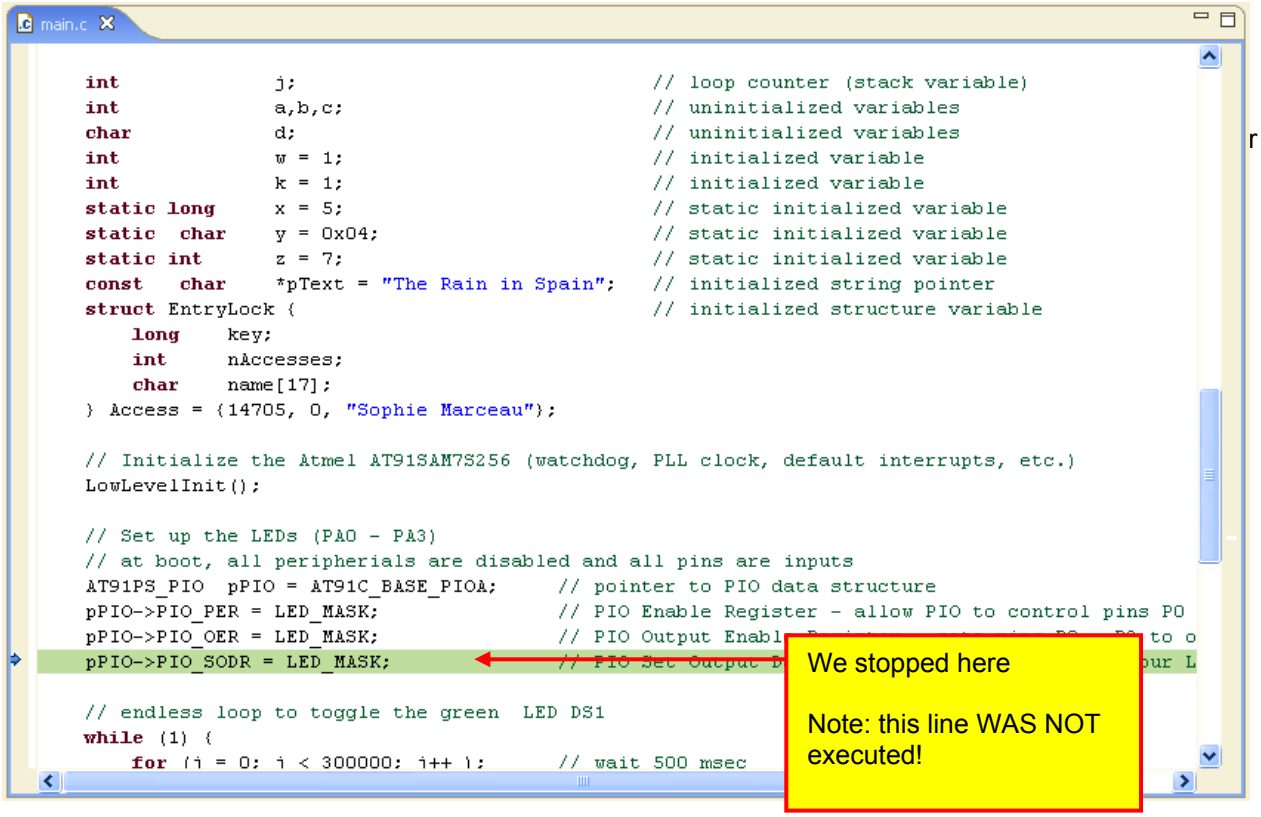
The easiest method of running is to employ the right-click menu. In the example below, the blue arrowhead cursor indicates where the program is currently stopped - just after main( ).

To go to the **pPIO->PIO_SODR = LED_MASK;** 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.

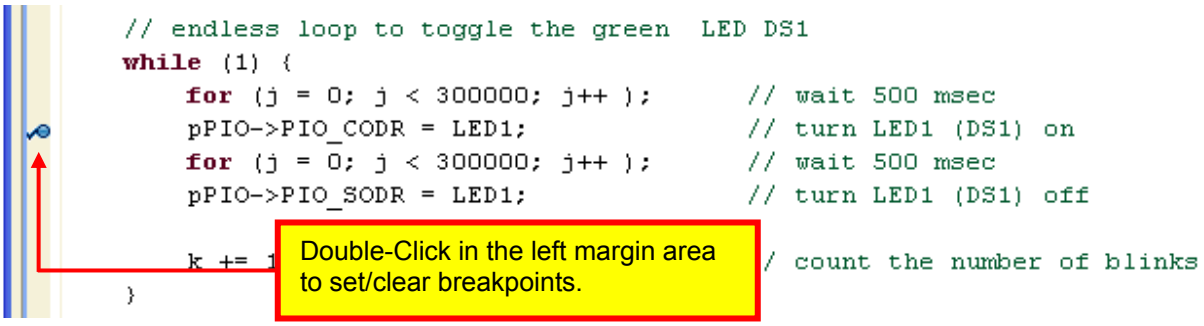


When you click on the "**Run to line**" choice, the program will execute to the line the cursor resides on and then stop (N.B. it will not execute the line).

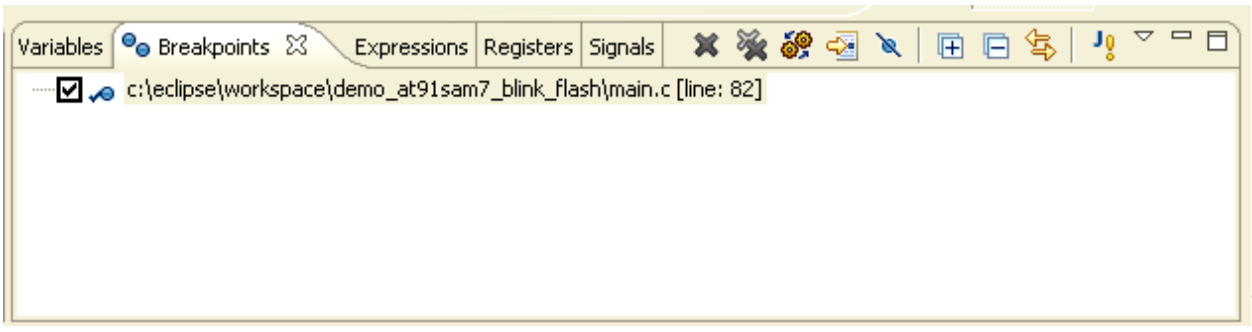


Setting a Breakpoint

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.

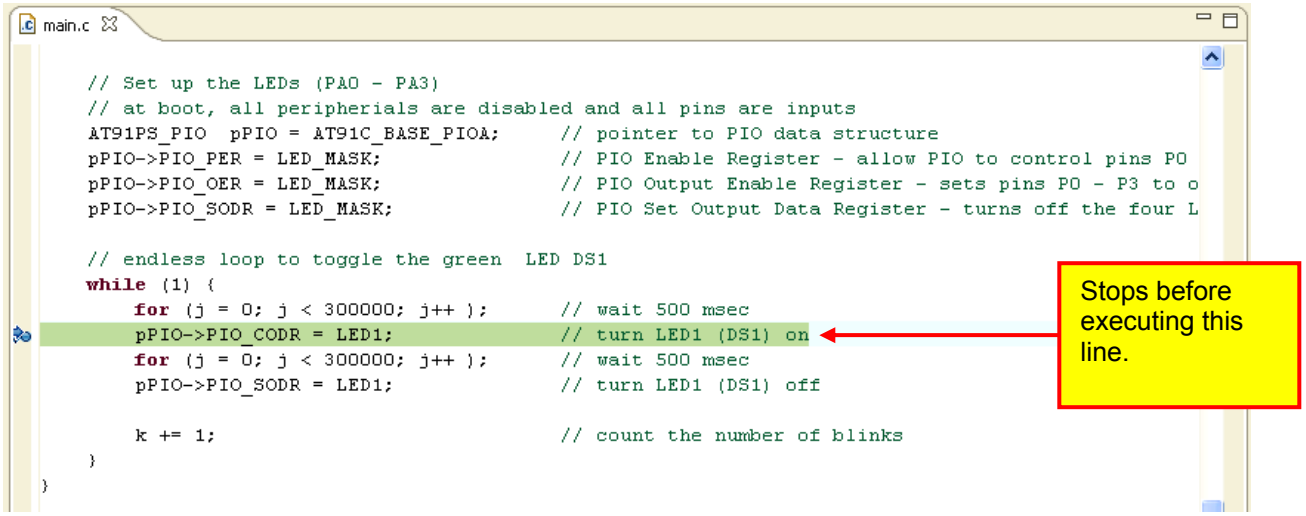


Note in the upper right “Breakpoint Summary” pane, the new breakpoint at line 82 has been indicated, as shown below.



Now click on the "**Run/Continue**" button in the Debug view.

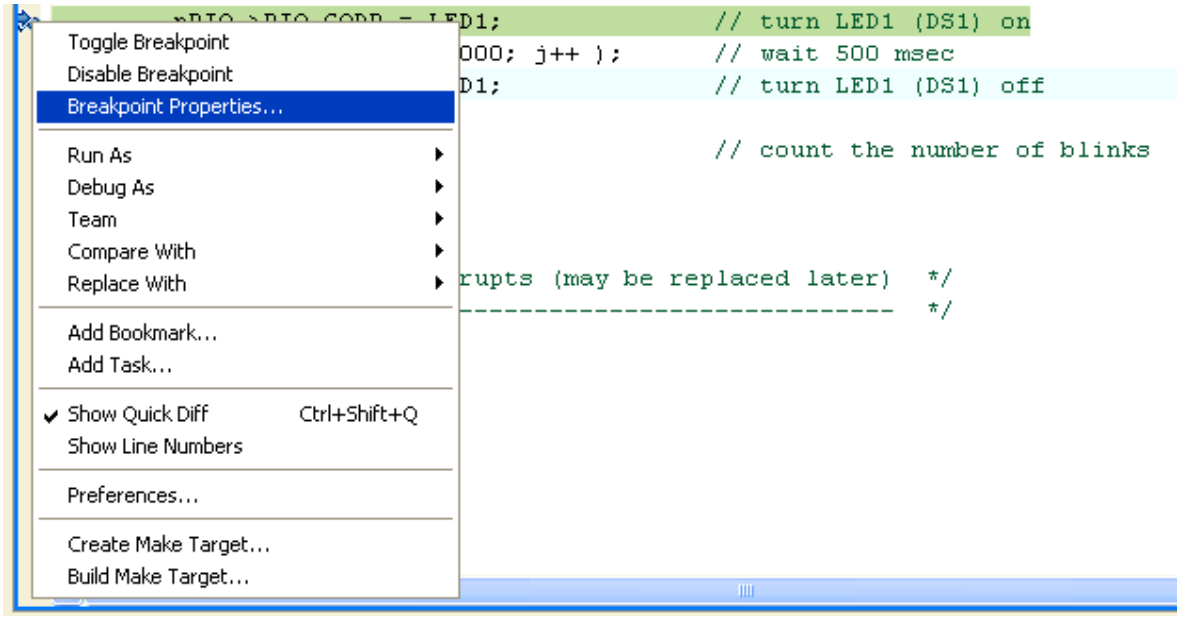
Assuming that this is the only breakpoint set, the program will execute to the breakpoint line and stop.



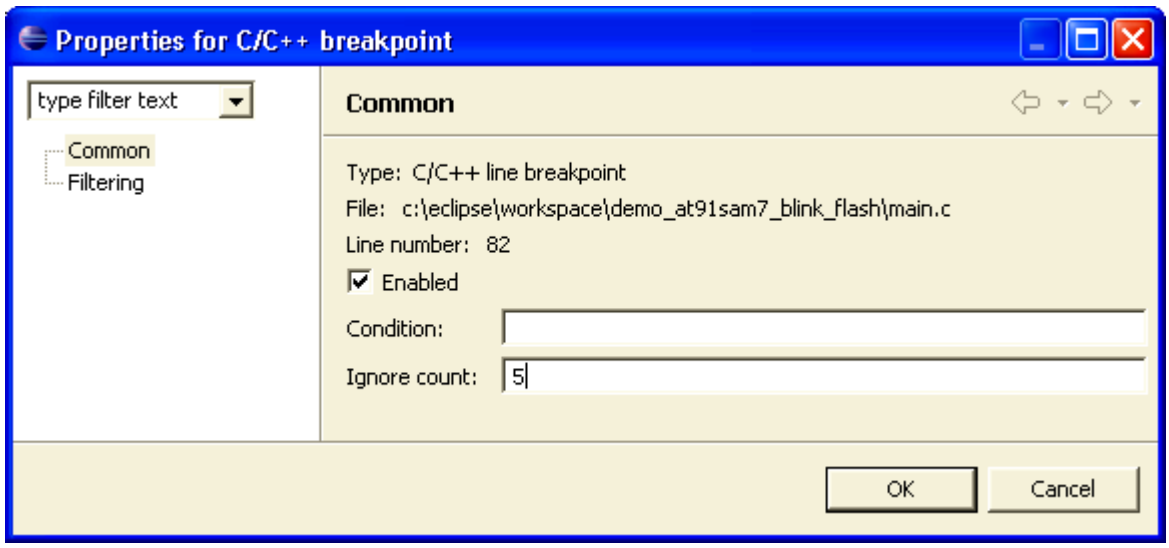
Since this is a FLASH application and breakpoints are "hardware" breakpoints, you are limited to **only two breakpoints specified at a time**. Setting more than two breakpoints will cause the debugger to malfunction!

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.

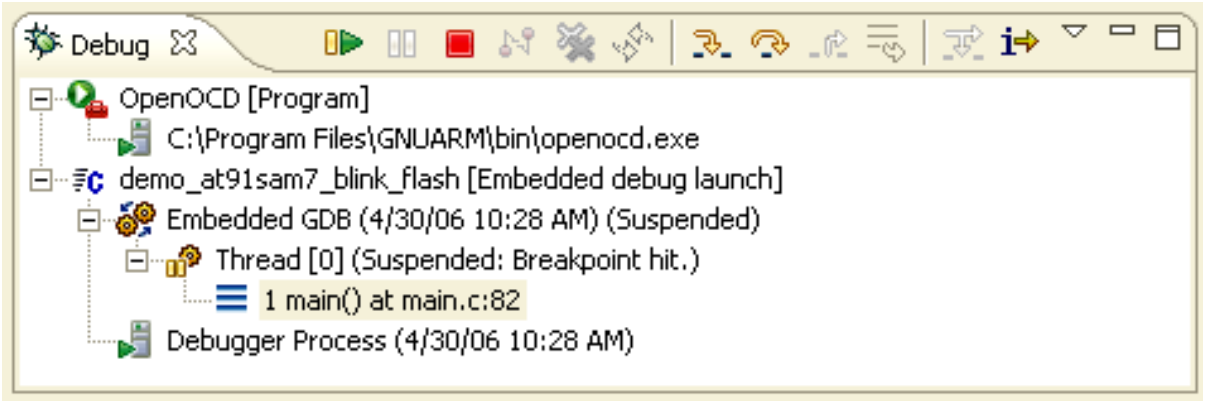
This brings up the pop-up menu below; click on "**Breakpoint Properties …**".



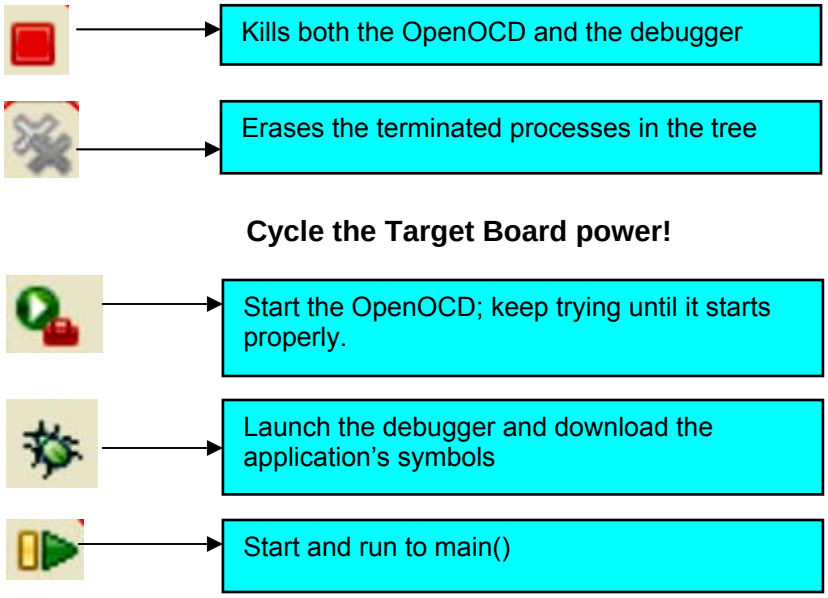
In the "**Properties for C/C++ breakpoint**" window, set the **Ignore Count** to 5. This means that the debugger will ignore the first five times it encounters the breakpoint and then stop.



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



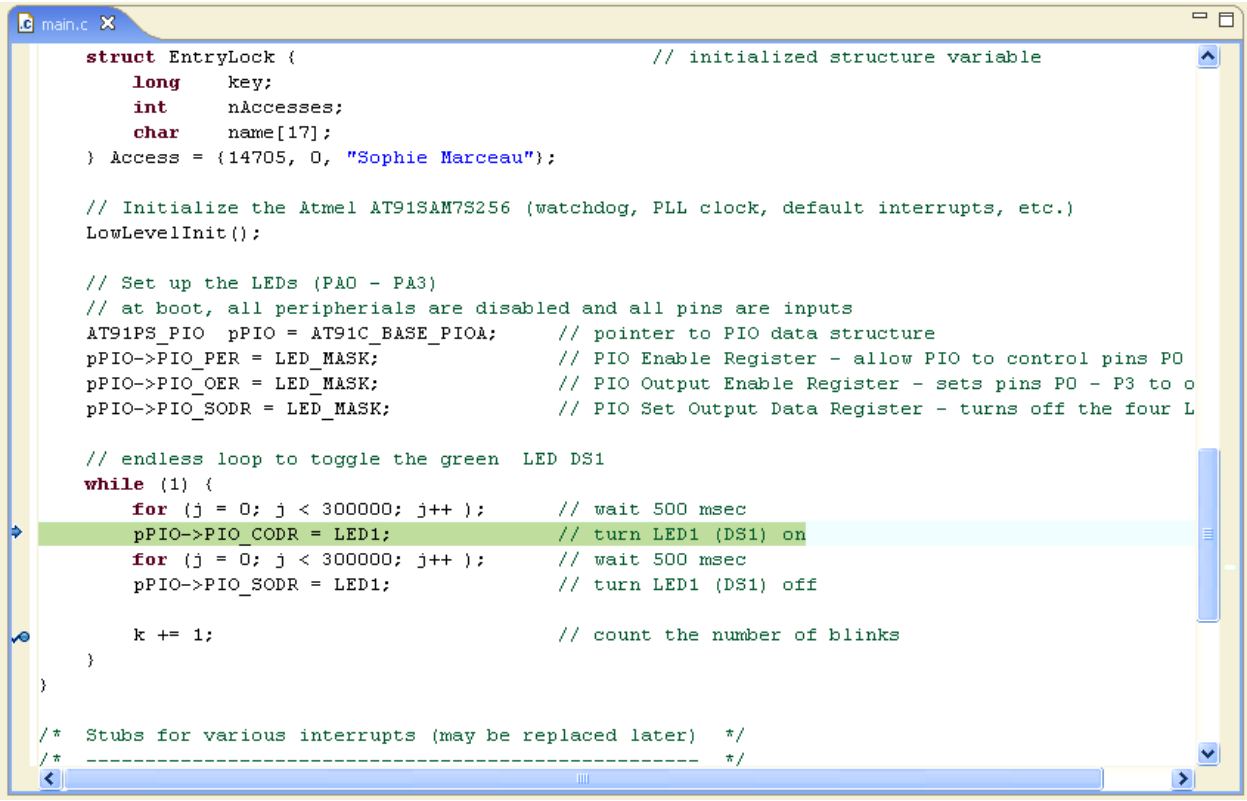
Get used to this sequence:



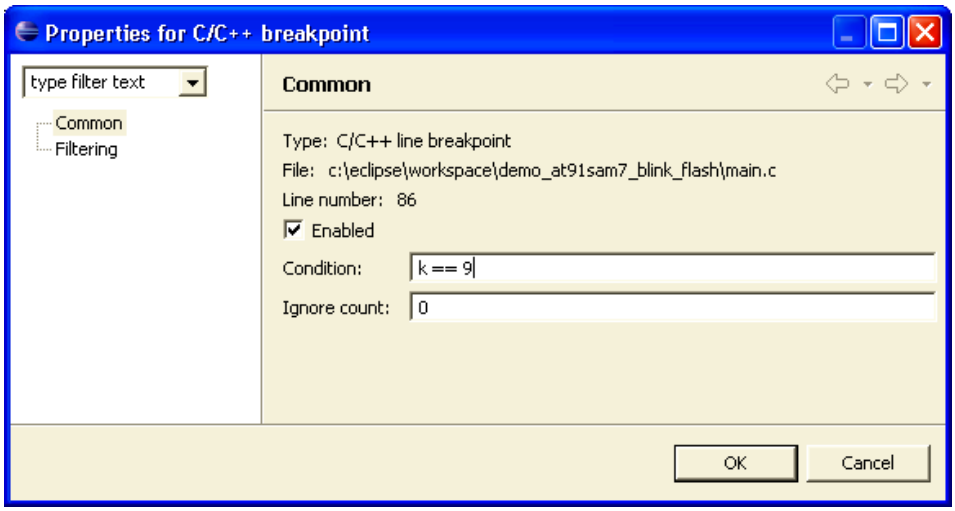
Now when you hit the **Run/Continue** button again, the program will blink 5 times and stop. Don't expect this feature to run in real-time. Each time the breakpoint is encountered the debugger will automatically continue until the "ignore" count is reached. This involves quite a bit of debugger communication at a very slow baud rate. The "wiggler" works by bit-banging the PC's parallel LPT1 port; this limits the JTAG speed to less than 500 kHz.

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, there's a line in the blink loop that increments a variable "k". Double-click on that line to set a breakpoint.



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



If you need to restart the debugger, you need to **kill the OpenOCD and the Debugger and then restart both**; 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.

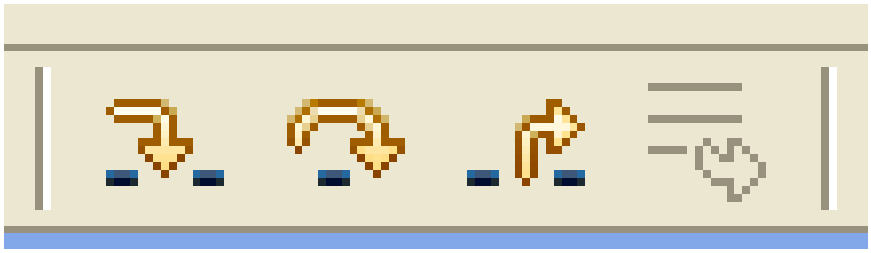
```
    // endless loop to toggle the green  LED DS1
    while (1) {
        for (j = 0; j < 300000; j++ );      // wait 500 msec
        pPIO->PIO_CODR = LED1;              // turn LED1 (DS1) on
        for (j = 0; j < 300000; j++ );      // wait 500 msec
        pPIO->PIO_SODR = LED1;              // turn LED1 (DS1) off

        k += 1;                             // count the number of blinks
    }     k=9
}
```

If you specify that it should break when k == 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 k==50000, the debugger will stop. Obviously, that requires a lot of serial communication at a very slow baud rate. 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 86 only if the variable k is equal to 9.

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 Step Over Step Out Of

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 the line that calls the **LowLevelInit( )** function. Hit the **Start** button to go to that breakpoint.

```
    struct EntryLock {                              // initialized structure variable
        long    key;
        int     nAccesses;
        char    name[17];
    } Access = {14705, 0, "Sophie Marceau"};

    // Initialize the Atmel AT91SAM7S256 (watchdog, PLL clock, default interrupts, etc.)
    LowLevelInit();

    // Set up the LEDs (PA0 - PA3)
    // at boot, all peripherials are disabled and all pins are inputs
    AT91PS_PIO  pPIO = AT91C_BASE_PIOA;     // pointer to PIO data structure
    pPIO->PIO_PER = LED_MASK;               // PIO Enable Register - allow PIO to control pins P
    pPIO->PIO_OER = LED_MASK;               // PIO Output Enable Register - sets pins P0 - P3 to
    pPIO->PIO_SODR = LED_MASK;              // PIO Set Output Data Register - turns off the four
```

Click the “**Step Into**” button  The debugger will enter the LowLevelInit( ) function.

```
void LowLevelInit(void)
{
    int             i;
    AT91PS_PMC      pPMC = AT91C_BASE_PMC;

    //* Set Flash Wait sate
    //  Single Cycle Access at Up to 30 MHz, or 40
    //  if MCK = 47923200 I have 50 Cycle for 1 usecond ( flied MC_FMR->FMCN
    AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN)&(50 <<16)) | AT91C_MC_FWS_1FWS;

    //* Watchdog Disable
    AT91C_BASE_WDTC->WDTC_WDMR= AT91C_WDTC_WDDIS;
```

Click the “**Step Over**” button  The debugger will execute one instruction.

```
void LowLevelInit(void)
{
    int             i;
    AT91PS_PMC      pPMC = AT91C_BASE_PMC;

    //* Set Flash Wait sate
    //  Single Cycle Access at Up to 30 MHz, or 40
    //  if MCK = 47923200 I have 50 Cycle for 1 usecond ( flied MC_FMR->FMCN
    AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN)&(50 <<16)) | AT91C_MC_FWS_1FWS;

    //* Watchdog Disable
    AT91C_BASE_WDTC->WDTC_WDMR= AT91C_WDTC_WDDIS;
```

Notice that the “**Step Out Of**” button is illuminated. Click the “**Step Out Of**” button 

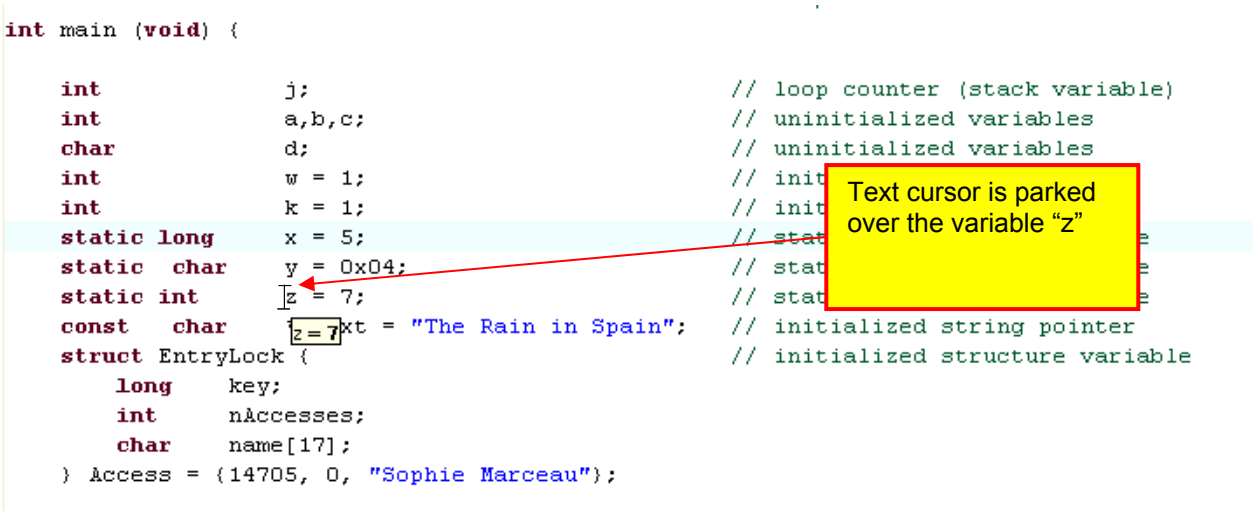
The debugger will execute the remaining instructions in LowLevelInit( ) and return to just after the function call.

```
    struct EntryLock {                              // initialized structure variable
        long    key;
        int     nAccesses;
        char    name[17];
    } Access = {14705, 0, "Sophie Marceau"};

    // Initialize the Atmel AT91SAM7S256 (watchdog, PLL clock, default interrupts, etc.)
    LowLevelInit();

    // Set up the LEDs (PA0 - PA3)
    // at boot, all peripherials are disabled and all pins are inputs
    AT91PS_PIO  pPIO = AT91C_BASE_PIOA;     // pointer to PIO data structure
    pPIO->PIO_PER = LED_MASK;               // PIO Enable Register - allow PIO to control pi
    pPIO->PIO_OER = LED_MASK;               // PIO Output Enable Register - sets pins P0 - P
    pPIO->PIO_SODR = LED_MASK;              // PIO Set Output Data Register - turns off the
```

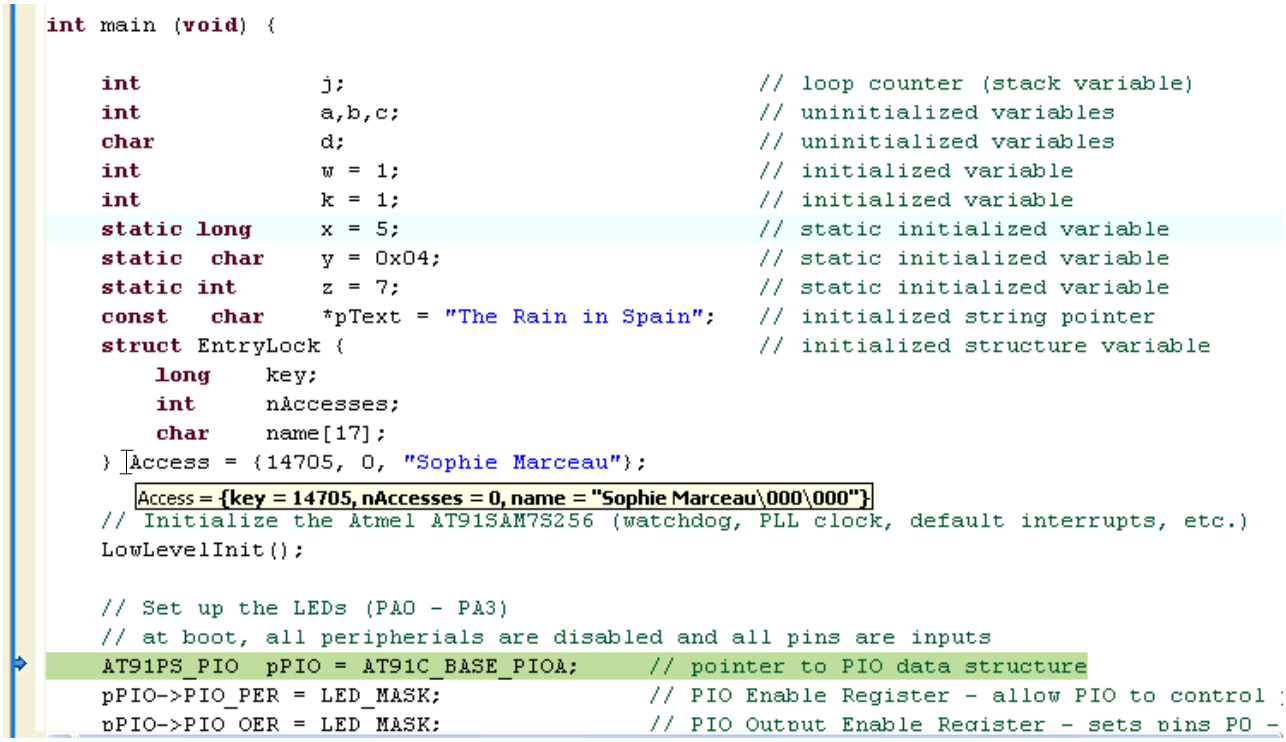
Inspecting and Modifying Variables

The simple way to inspect variables is to just park the cursor over the variable name in the source window; the current value will pop up in a tiny text box. Execution must be stopped for this to work; either by breakpoint or pause. In this operation, try to position the text cursor just to the left of the variable name.

Of course, the debugger must be stopped for variable inspection to work!

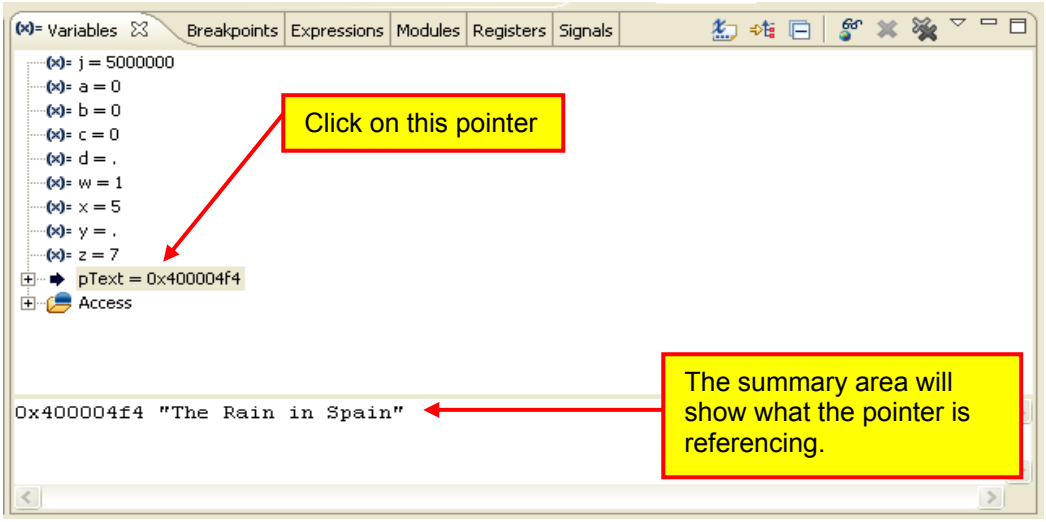


For a structured variable, parking the cursor over the variable name will show the values of all the internal component parts.

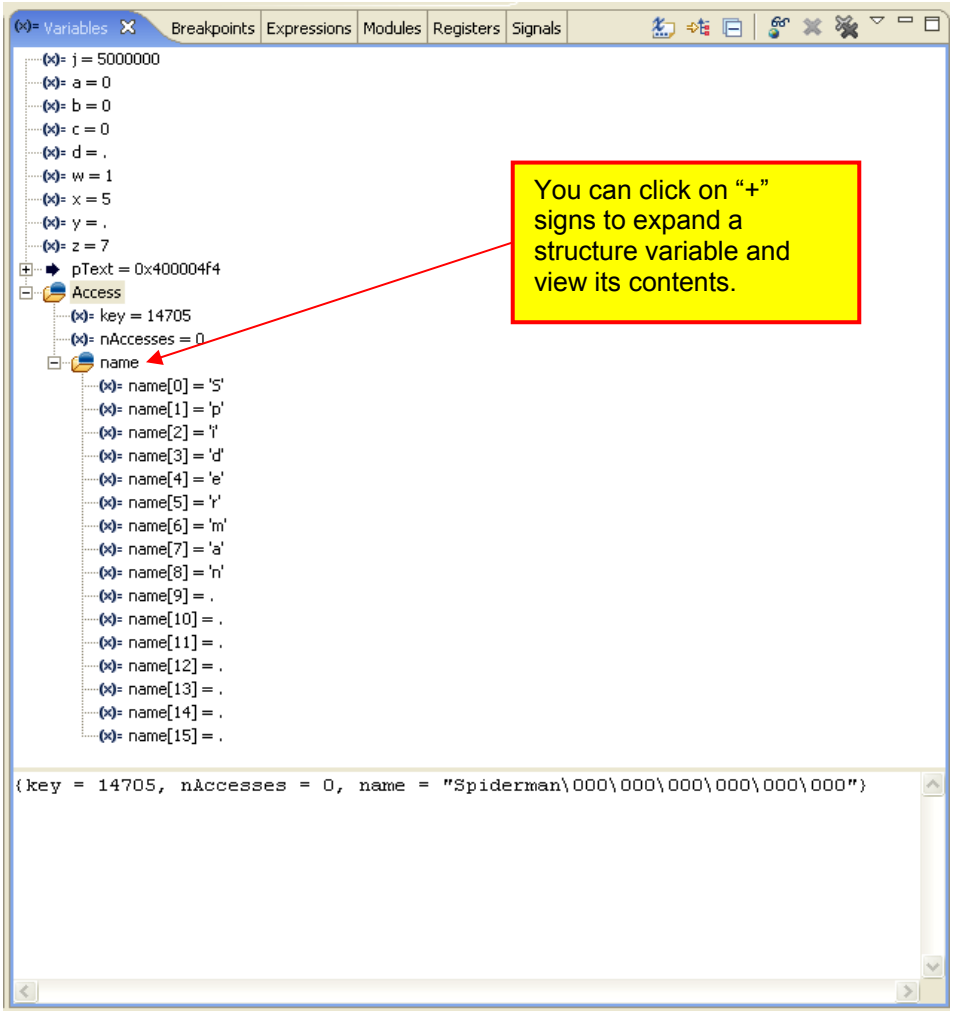


Another way to look at the local variables is to inspect the "**Variables**" view. This will automatically display all automatic variables in the current stack frame. It can also display any global variables that you choose. For simple scalar variables, the value is printed next to the variable name.

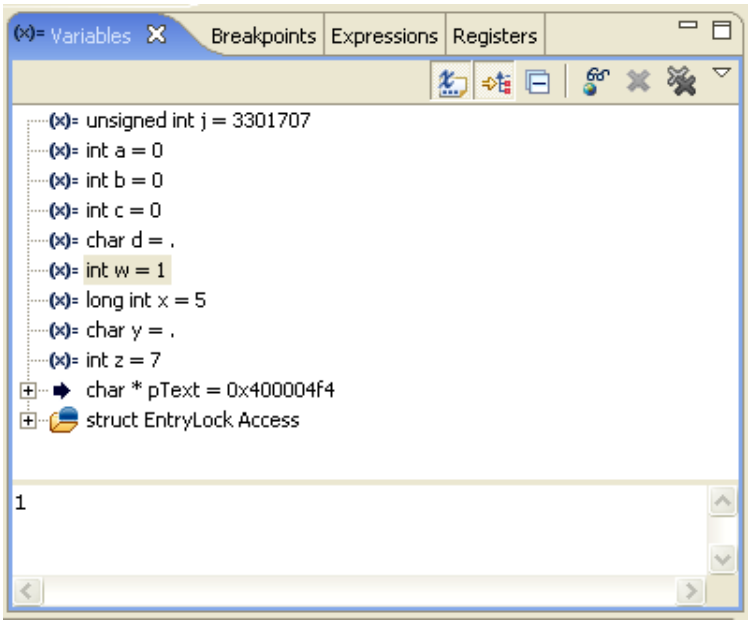
If you click on a variable, its value appears in the summary area at the bottom. This is handy for a structured variable or a pointer; wherein the debugger will expand the variable in the summary area.



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



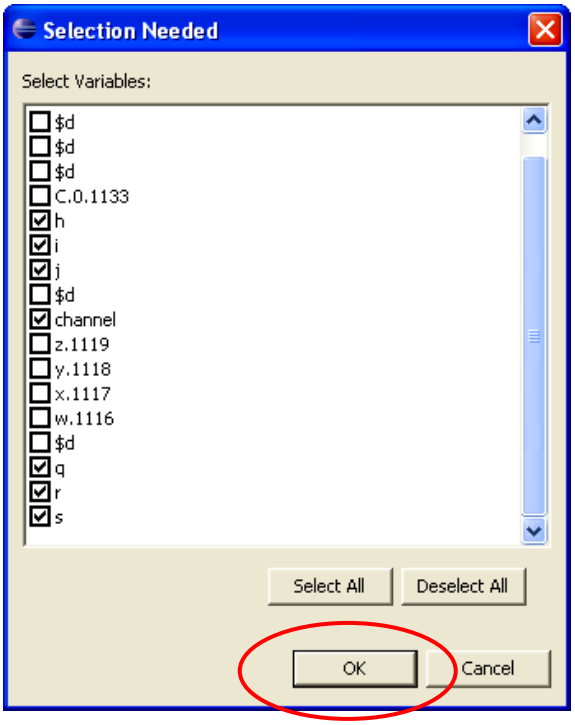
If you click on the “Show Type Names” button,  each variable name will be displayed with its type, as shown below.



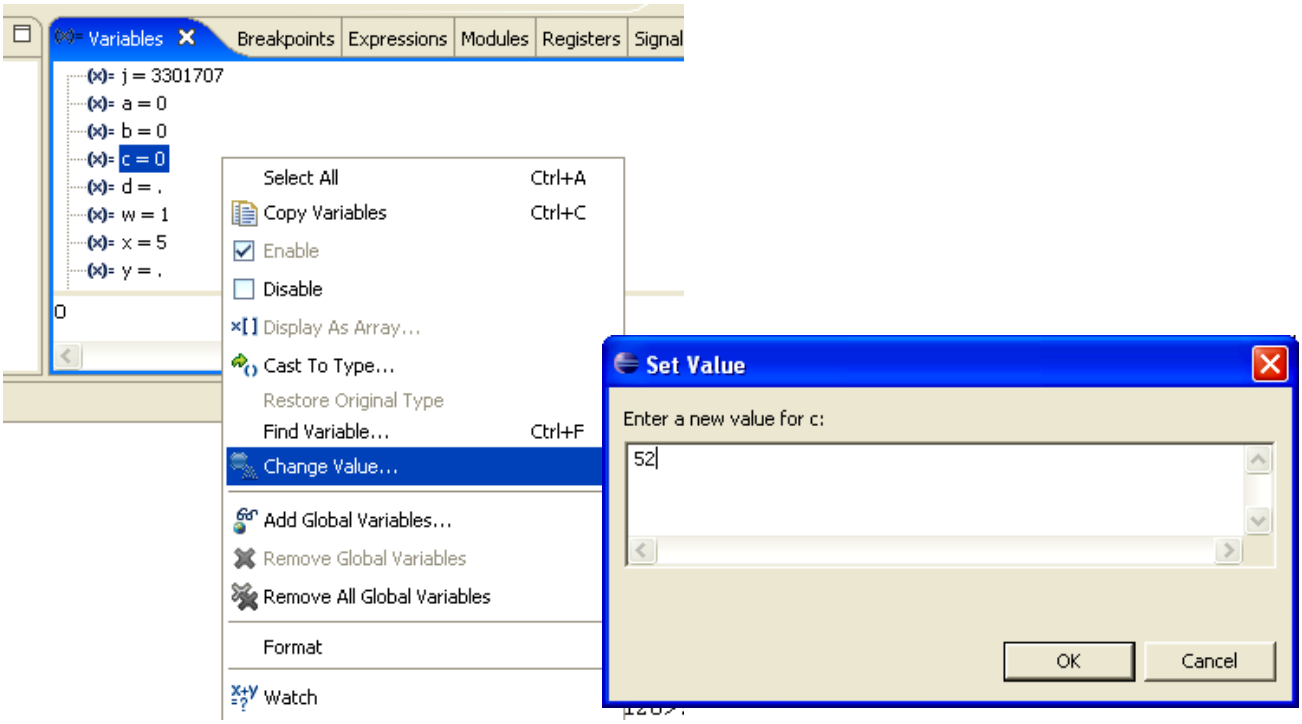
Global variables have to be individually selected for display within the “Variables” view.

Use the “Add Global Variables” button  to open the selection dialog.

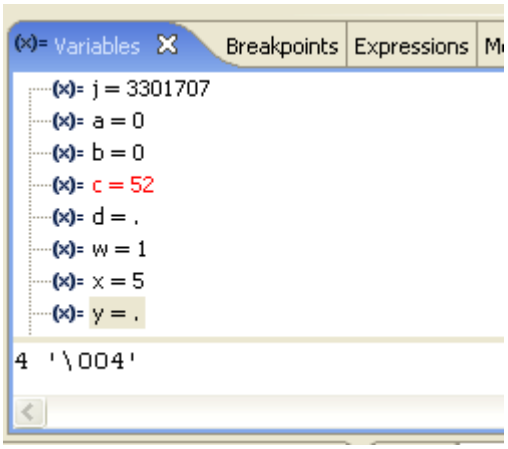
Check the variables you want to display and then click “**OK**” add them to the **Variables** view,



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.



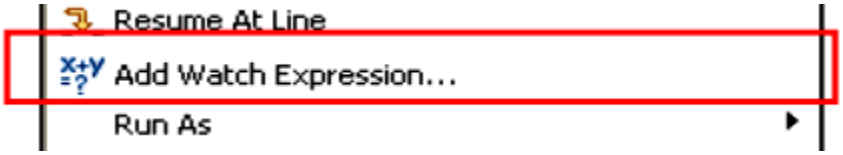
Now the "**Variables**" view should show the new value for the variable "c". Note that it has been colored red to indicate that it has been changed.



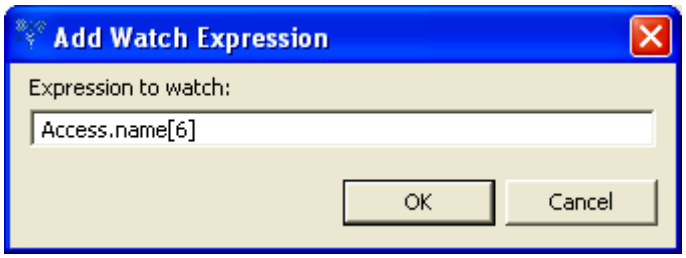
Watch Expressions

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.

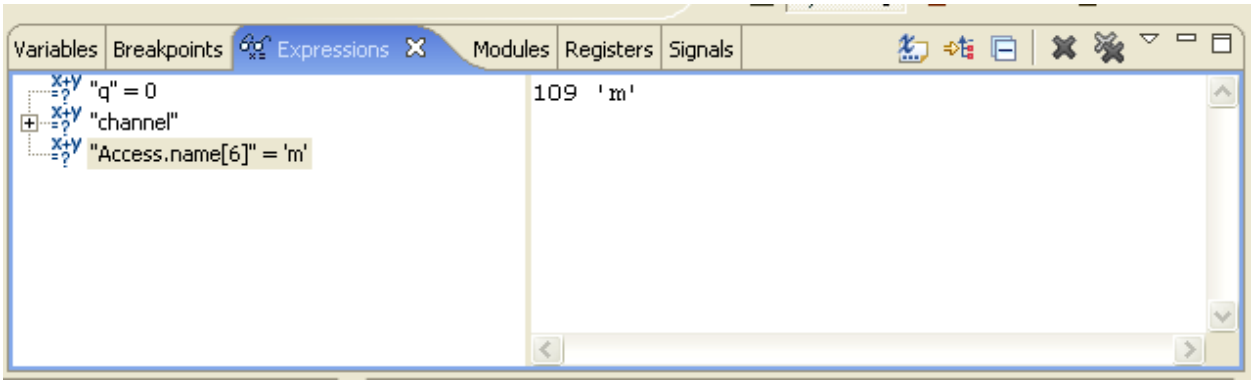
For example, to display the 6th character of the name in the structured variable “**Access**”, bring up the right-click menu and select “**Add Watch Expression…**”.



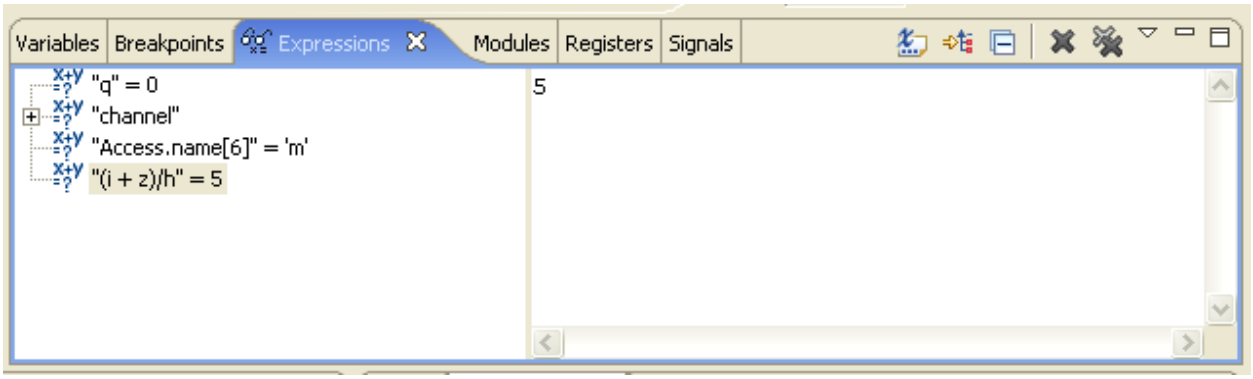
Enter the fully qualified name of the 6th character of the name[] array.



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



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

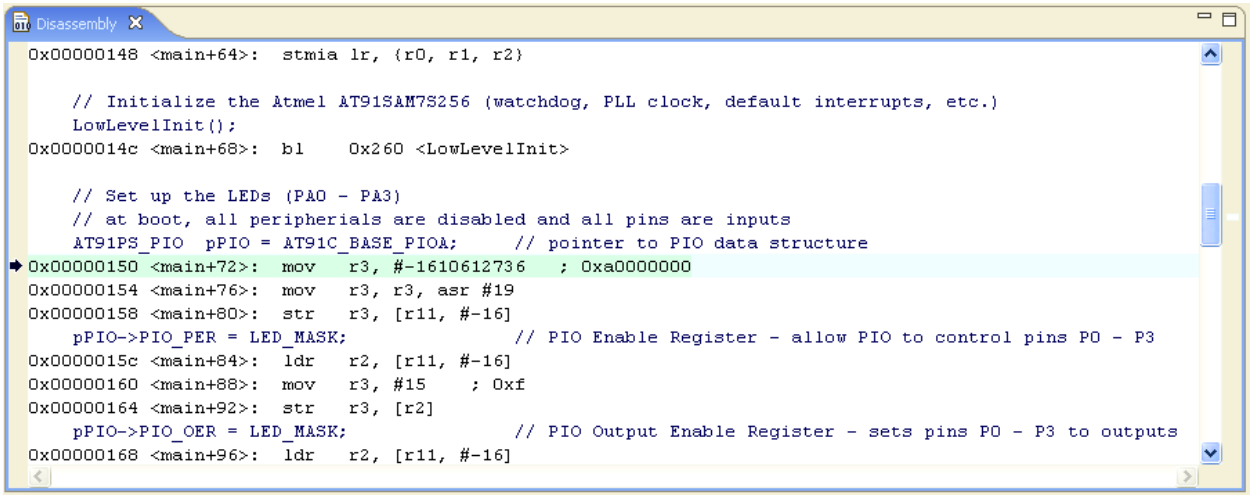


Assembly Language Debugging

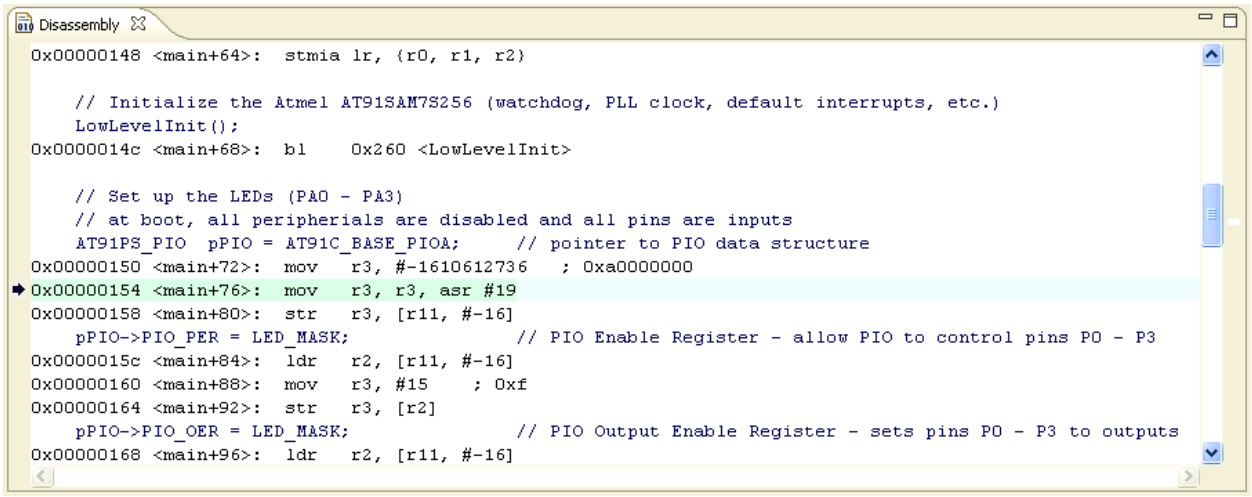
The Debug perspective includes an Assembly Language view.

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 0x0000150.



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

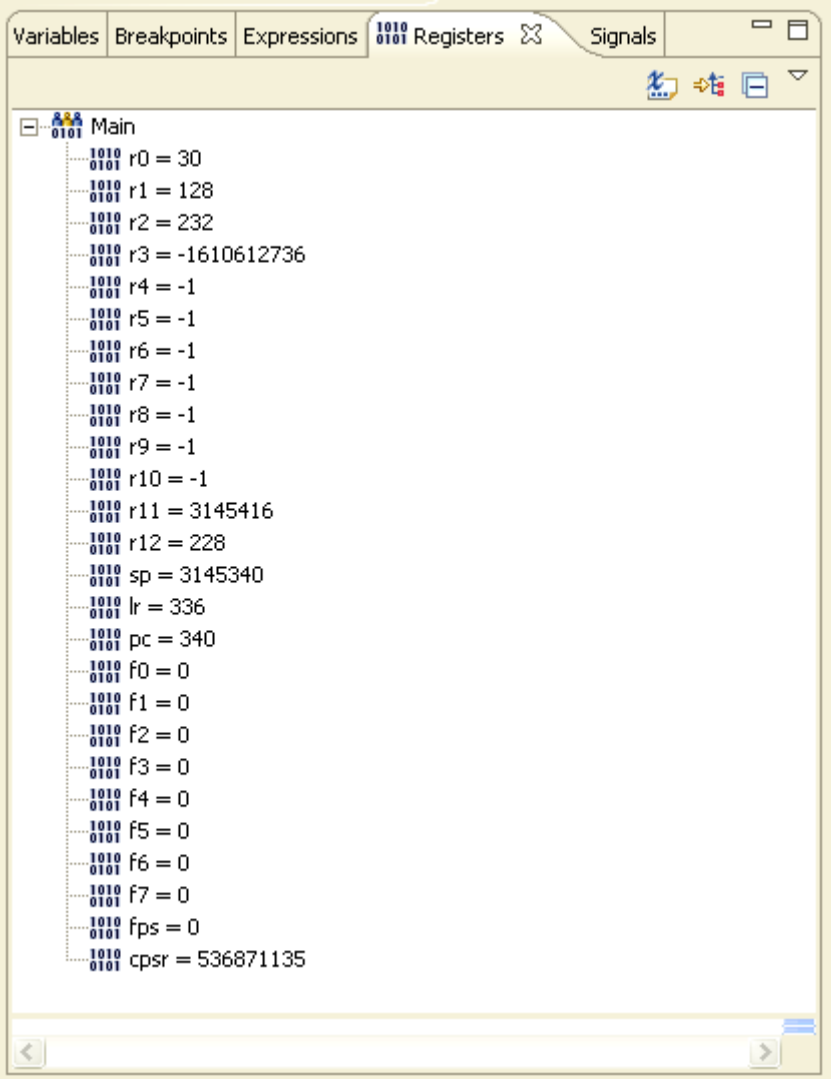


The "**Step Into**" and "**Step Out Of**" buttons work in the same was as for C code.

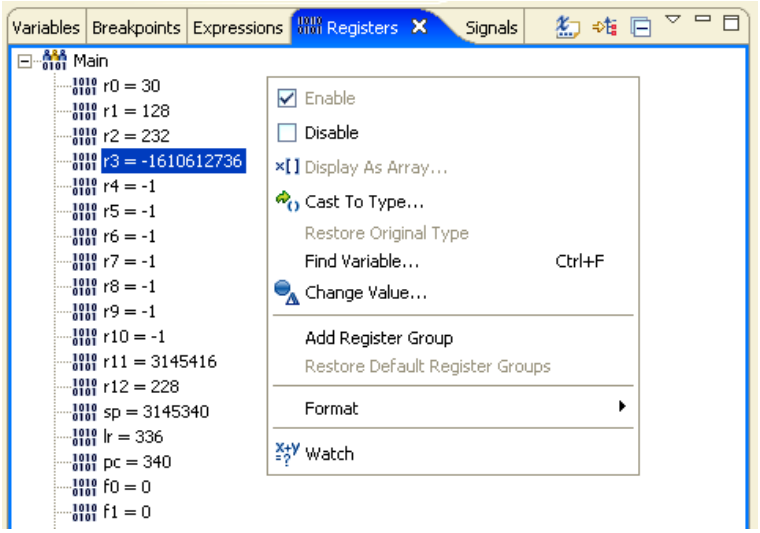
Inspecting Registers

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.

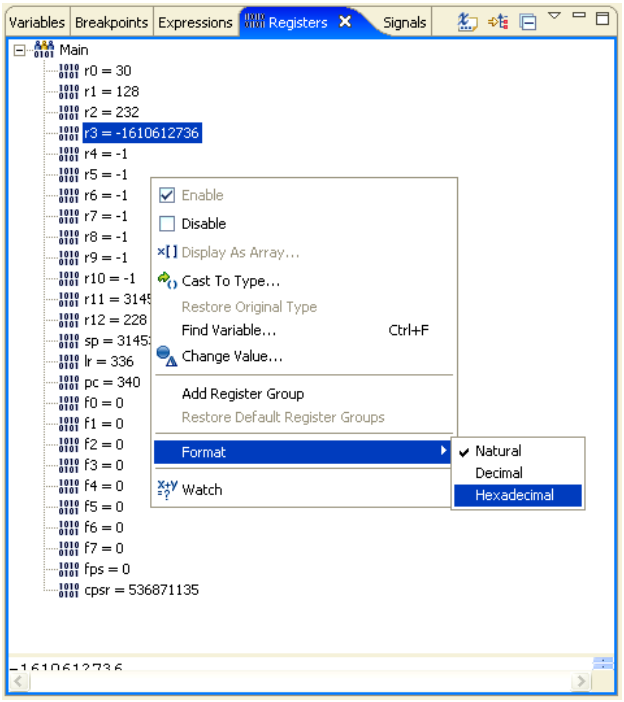
Click on the “**+**” symbol next to Main and the registers will appear. The Atmel AT91SAM7S256 doesn’t have any floating point registers so registers F0 through FPS are not applicable.



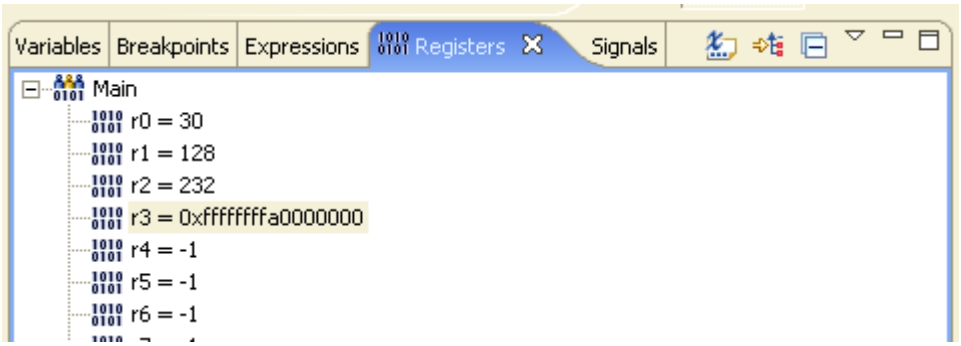
If you don’t like a particular register’s numeric format, you can click to highlight it and then bring up the right-click menu.



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

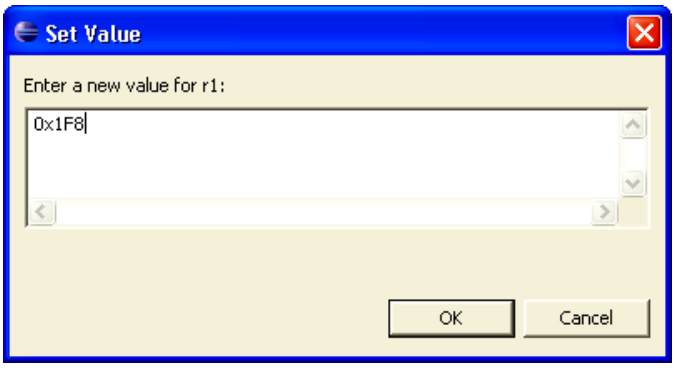


Now R3 is displayed in hexadecimal.

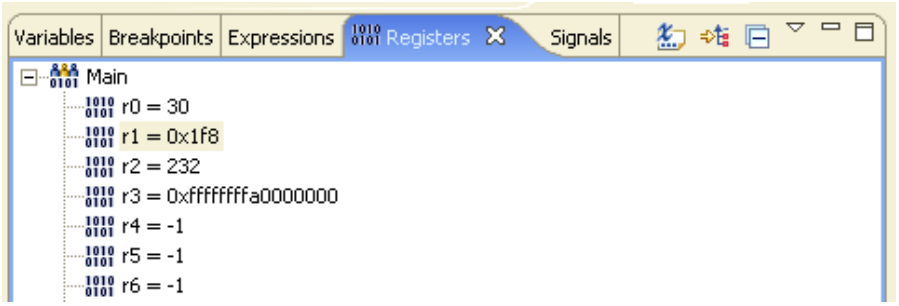


Of course, the right click menu lets you change the value of any register. For example, to change **r1** from **128** to **0x1F8**, just select the register, right-click and select "**Change Value…**"

In the "Set Value" dialog box, enter the hexadecimal value **0x1F8** and click "**OK**" to accept.



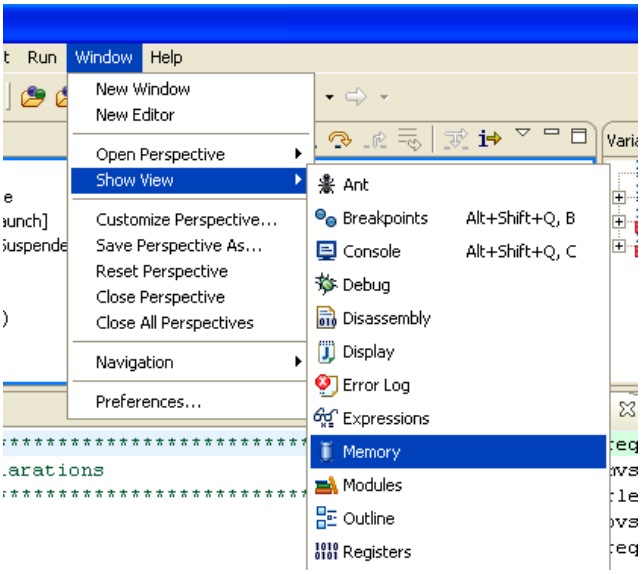
The register display now shows the new value for R1 (we also changed the display format to hexadecimal using the right-click menu).



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.

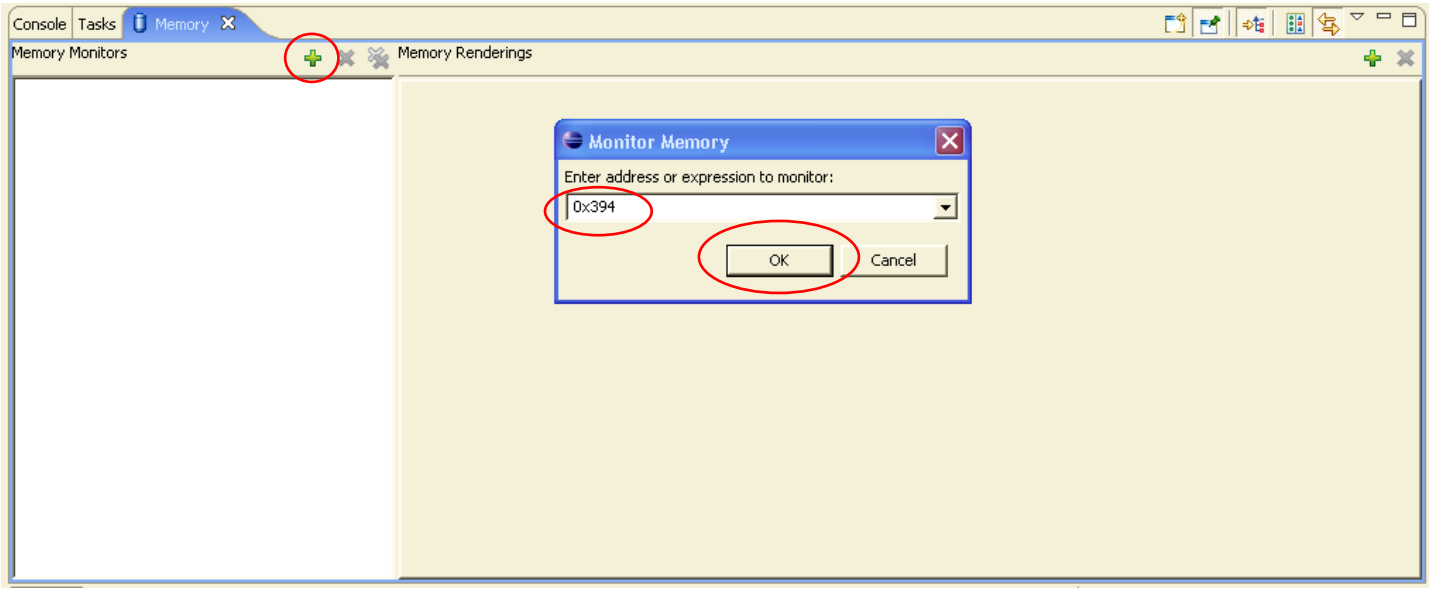
Inspecting Memory

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.



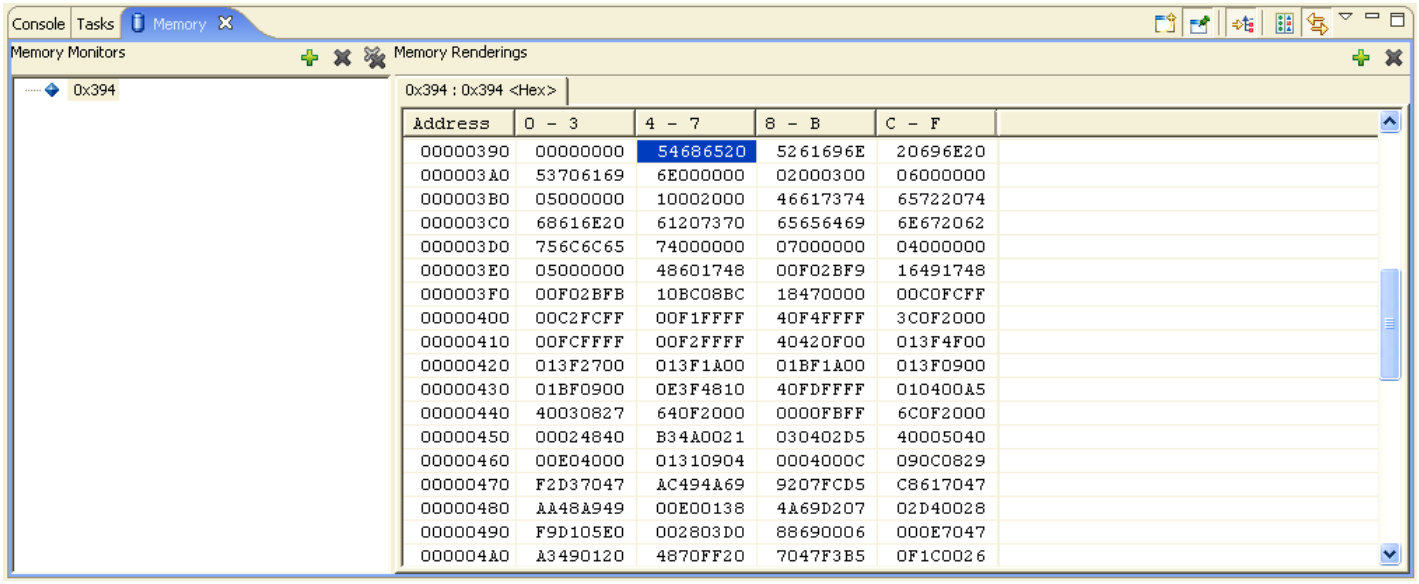
The memory view appears in the "**Console**" view at the bottom of the Debug perspective. At this point, nothing has been defined. Memory is displayed as one or more "**memory monitors**". You can create a memory monitor by clicking on the "**+**" symbol.

Enter the address **0x394** (address of the string "The Rain in Spain") in the dialog box and click "**OK**".



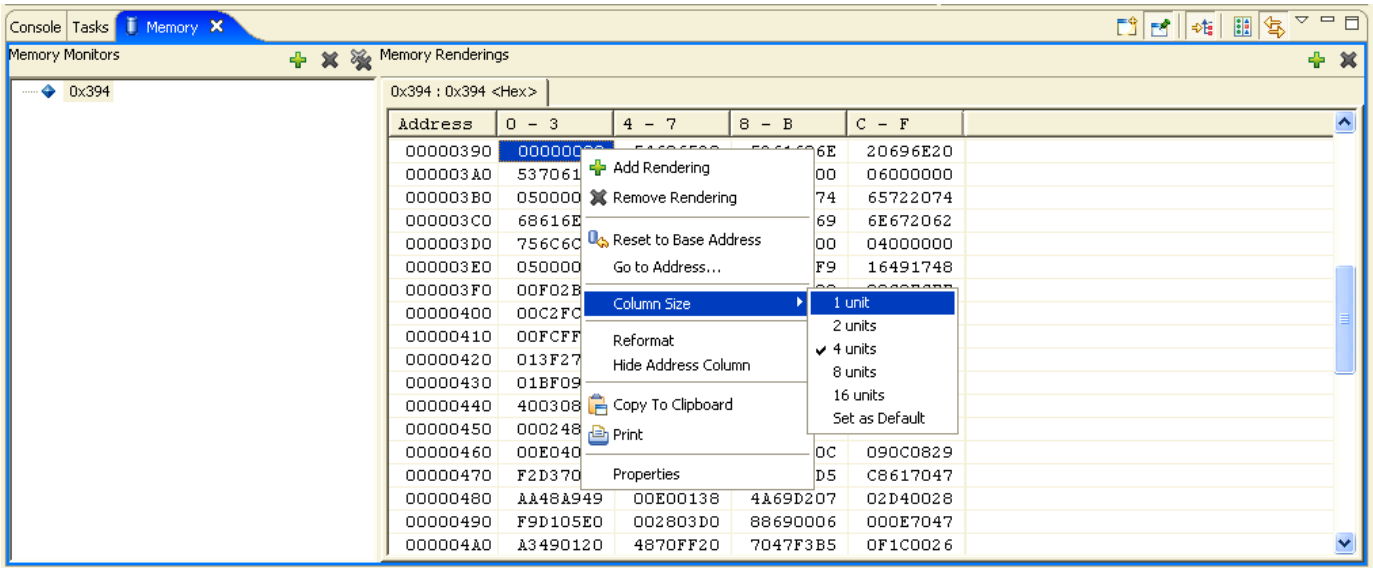
The memory monitor is created, although it defaults to 4-byte display mode. The display of the address columns and the associated memory contents is called a "**Rendering**".

The address **0x394** 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.

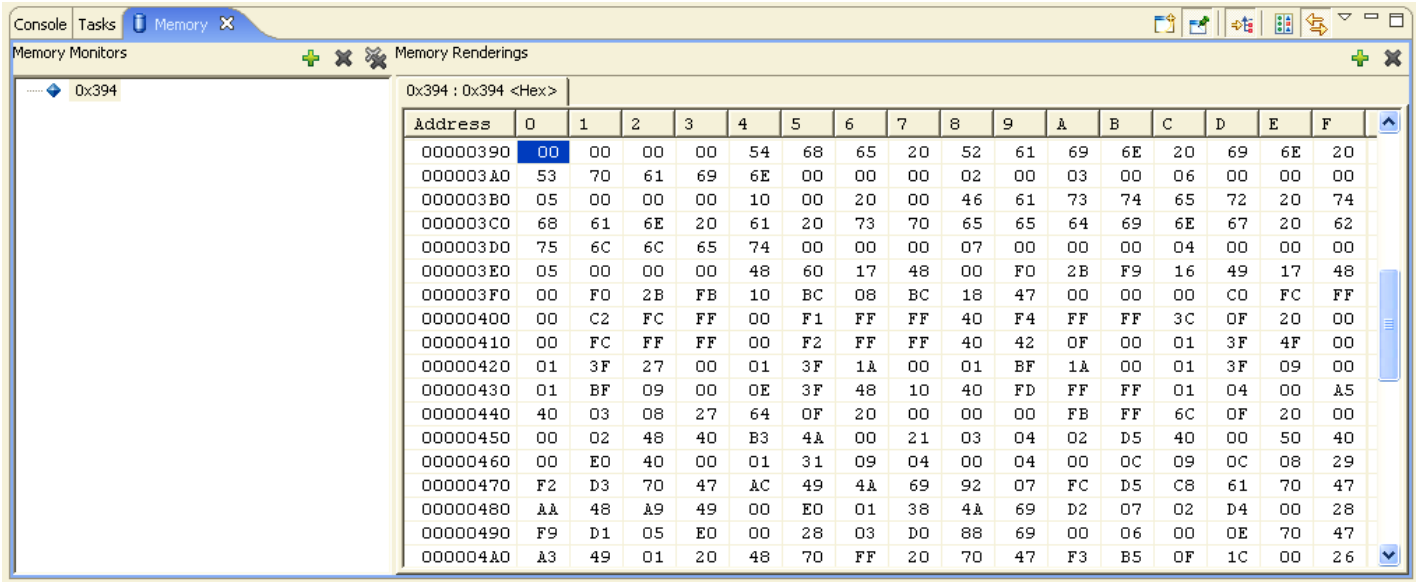


There's also a "**Go to Address…**" right-click menu option that will jump all over memory for you.

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 as shown below.

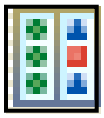


Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000390	00	00	00	00	54	68	65	20	52	61	69	6E	20	69	6E	20
000003A0	53	70	61	69	6E	00	00	00	02	00	03	00	06	00	00	00
000003B0	05	00	00	00	10	00	20	00	46	61	73	74	65	72	20	74
000003C0	68	61	6E	20	61	20	73	70	65	65	64	69	6E	67	20	62
000003D0	75	6C	6C	65	74	00	00	00	07	00	00	00	04	00	00	00
000003E0	05	00	00	00	48	60	17	48	00	F0	2B	F9	16	49	17	48
000003F0	00	F0	2B	FB	10	BC	08	BC	18	47	00	00	00	C0	FC	FF
00000400	00	C2	FC	FF	00	F1	FF	FF	40	F4	FF	FF	3C	0F	20	00
00000410	00	FC	FF	FF	00	F2	FF	FF	40	42	0F	00	01	3F	4F	00
00000420	01	3F	27	00	01	3F	1A	00	01	BF	1A	00	01	3F	09	00
00000430	01	BF	09	00	0E	3F	48	10	40	FD	FF	FF	01	04	00	A5
00000440	40	03	08	27	64	0F	20	00	00	00	FB	FF	6C	0F	20	00
00000450	00	02	48	40	B3	4A	00	21	03	04	02	D5	40	00	50	40
00000460	00	E0	40	00	01	31	09	04	00	04	00	0C	09	0C	08	29
00000470	F2	D3	70	47	AC	49	4A	69	92	07	FC	D5	C8	61	70	47
00000480	AA	48	A9	49	00	E0	01	38	4A	69	D2	07	02	D4	00	28
00000490	F9	D1	05	E0	00	28	03	D0	88	69	00	06	00	0E	70	47
000004A0	A3	49	01	20	48	70	FF	20	70	47	F3	B5	0F	1C	00	26

The Eclipse memory display allows you to simply type new values into the displayed cells. Of course, this example is in FLASH and that wouldn't work. Memory displays in the RAM area can be edited.

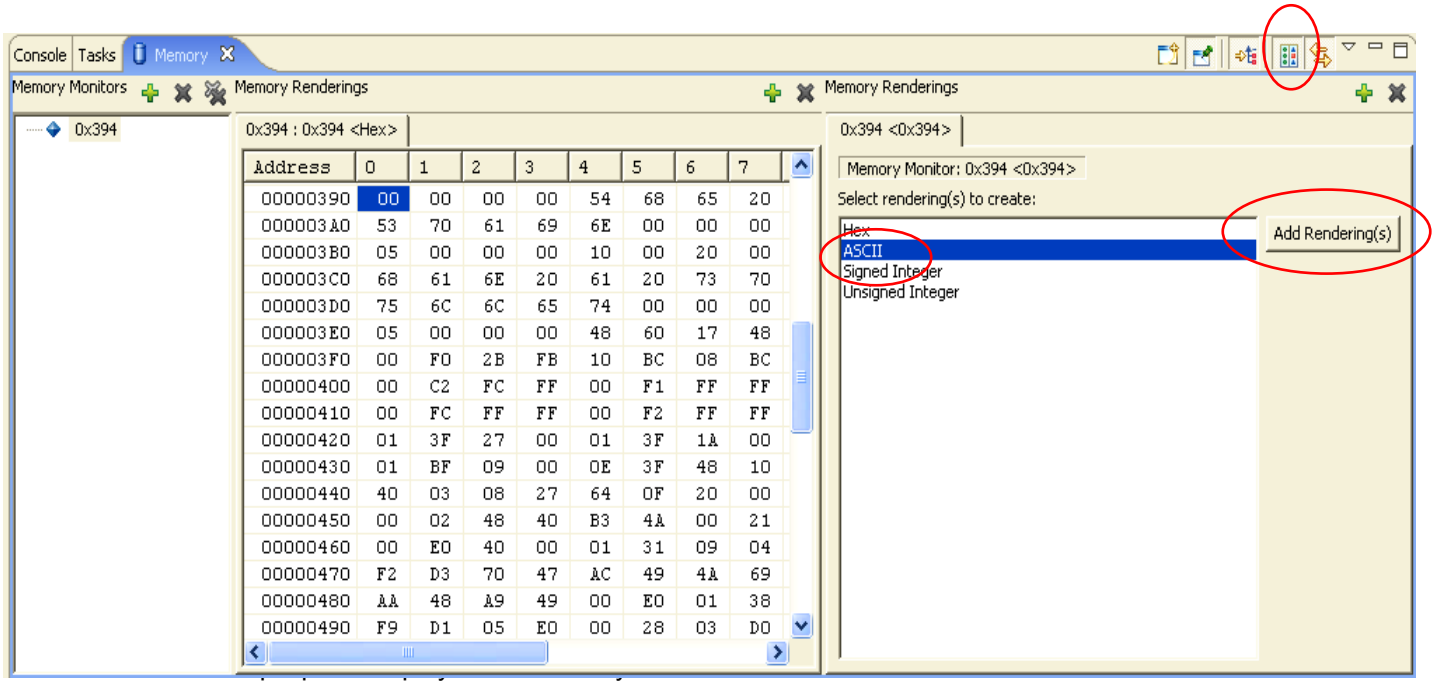
Now we will add a second rendering that will display the memory monitor in ASCII.

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.



Address	0	1	2	3	4	5	6	7
00000390	00	00	00	00	54	68	65	20
000003A0	53	70	61	69	6E	00	00	00
000003B0	05	00	00	00	10	00	20	00
000003C0	68	61	6E	20	61	20	73	70
000003D0	75	6C	6C	65	74	00	00	00
000003E0	05	00	00	00	48	60	17	48
000003F0	00	F0	2B	FB	10	BC	08	BC
00000400	00	C2	FC	FF	00	F1	FF	FF
00000410	00	FC	FF	FF	00	F2	FF	FF
00000420	01	3F	27	00	01	3F	1A	00
00000430	01	BF	09	00	0E	3F	48	10
00000440	40	03	08	27	64	0F	20	00
00000450	00	02	48	40	B3	4A	00	21
00000460	00	E0	40	00	01	31	09	04
00000470	F2	D3	70	47	AC	49	4A	69
00000480	AA	48	A9	49	00	E0	01	38
00000490	F9	D1	05	E0	00	28	03	D0

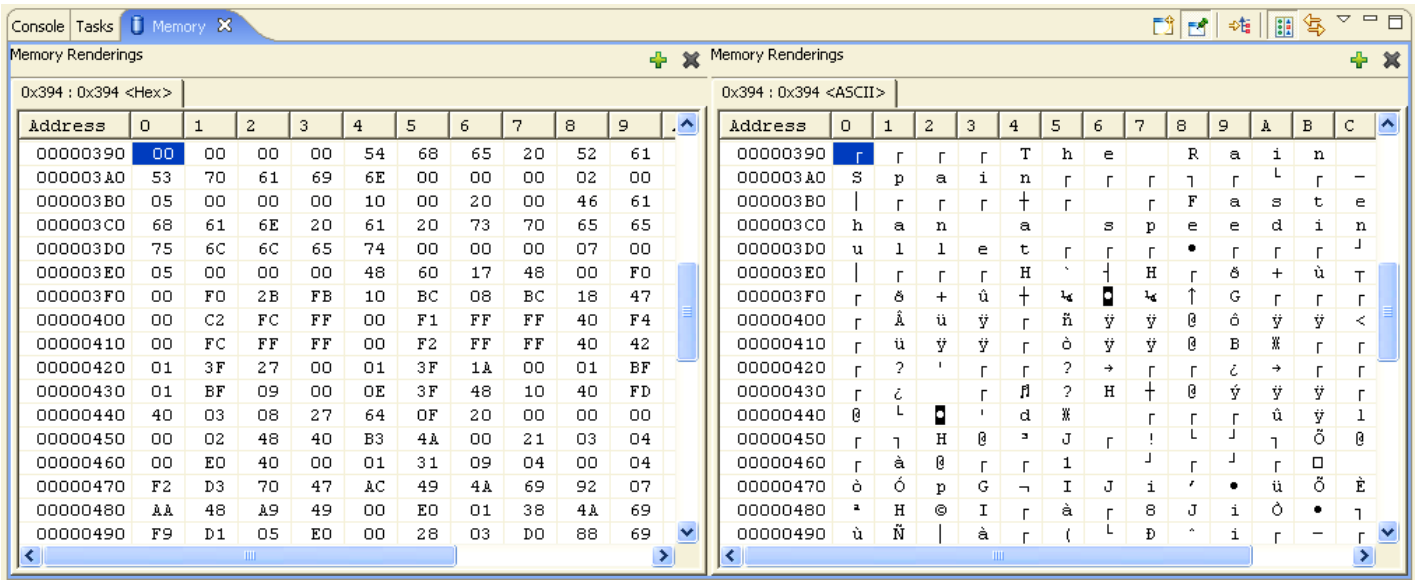
Now we have a display of the hex values and the corresponding ASCII characters.

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.



Admittedly, 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!

Create an Eclipse Project to Run in RAM

There are two reasons to run an application entirely within onboard RAM memory; to gain a speed advantage and to be able to set an unlimited number of software breakpoints.

Execution within RAM is about two times faster than execution within FLASH memory. Many programmers will just copy the routines that need the increased execution speed from FLASH to RAM at run-time and thenceforth call the routines resident in RAM. This is not the subject of this tutorial so we will address this idea any further.

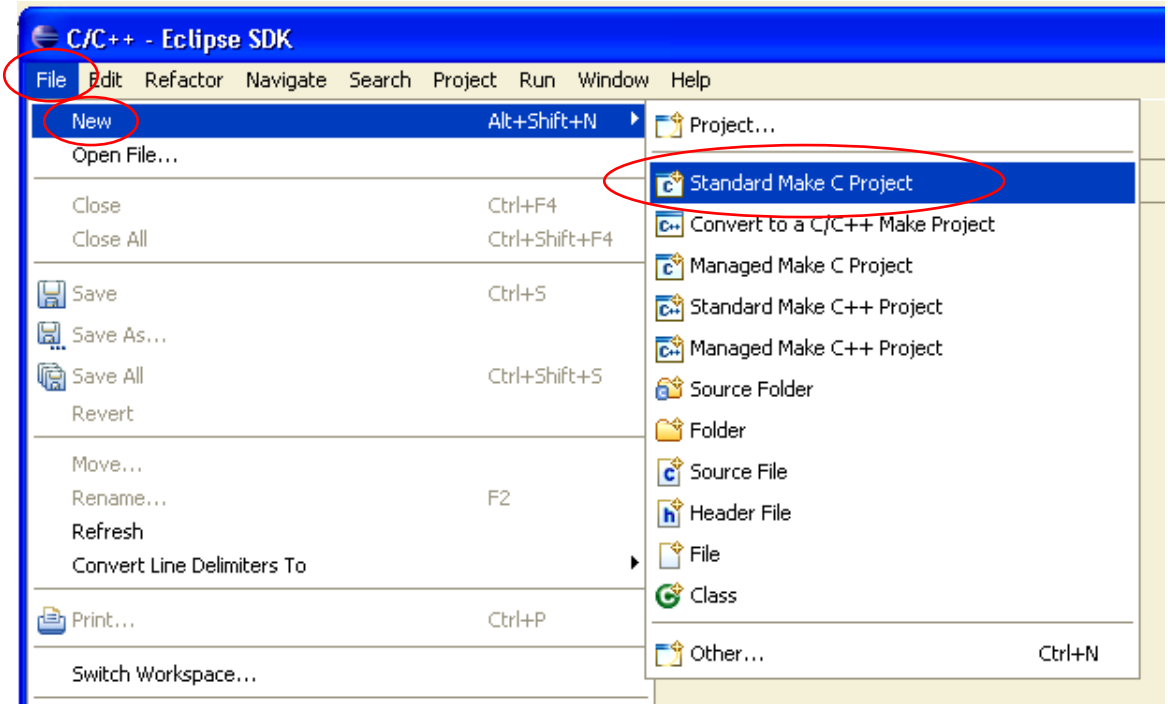
In the FLASH example shown previously, the OpenOCD utility permitted the Eclipse debugger to use the two on chip breakpoint units; allowing a breakpoint to be set in FLASH. This limits us to just two breakpoints. Note also that the OpenOCD setup converted every Eclipse breakpoint specification into a hardware-assisted breakpoint. This works great but there may be occasions where the two-breakpoint limit is not satisfactory.

Creating an Eclipse project that runs entirely out of on chip RAM is simple if a bit counter-intuitive. We use the Linker command script to place the code (.text), initialized variables (.data) and uninitialized variables (.bss) all into FLASH at address 0x00000000. When the debugger starts up, we toggle the MC Memory Remap Control Register to place the RAM memory at address 0x000000. We then let the OpenOCD/JTAG unit load the main.out file (containing the executable code) into RAM now at address 0x00000000 and away we go! It's almost as if Flash memory has become read/write.

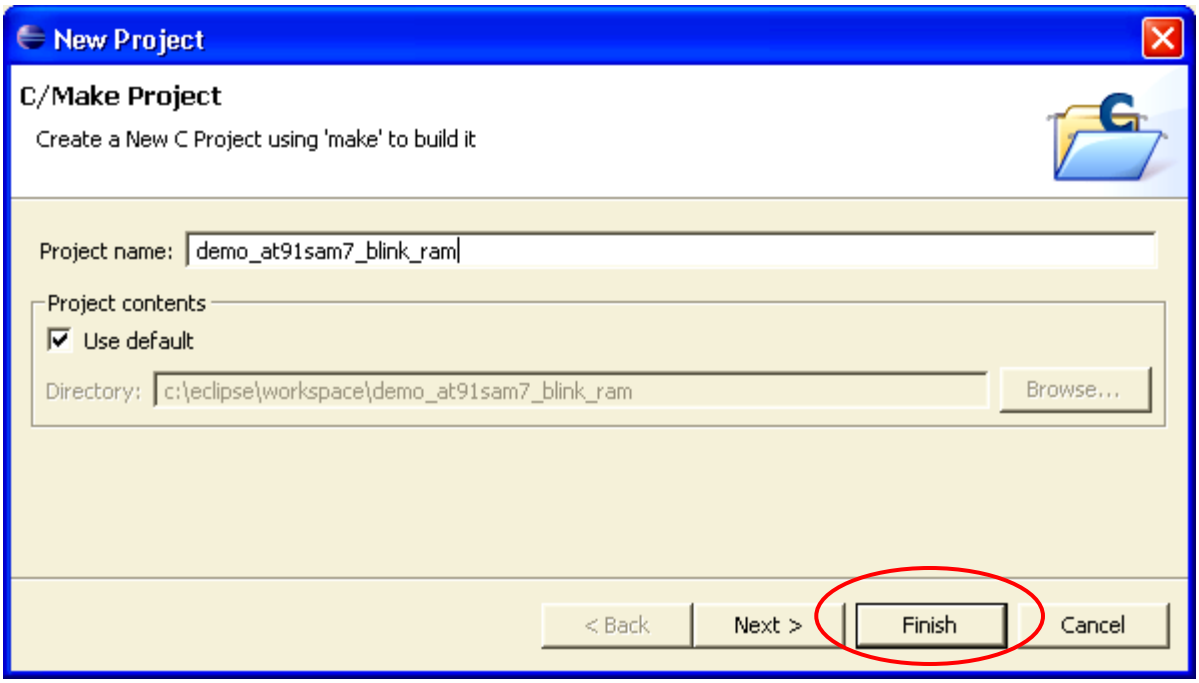
With this approach, we get an unlimited number of software breakpoints and can use the JTAG device to download the code (we don't have to use the SAM-BA utility). The disadvantage, of course, is that the application is limited to 64 Kbytes.

Close the current Eclipse project using the "**Project**" pull-down menu and selecting "**Close Project**".

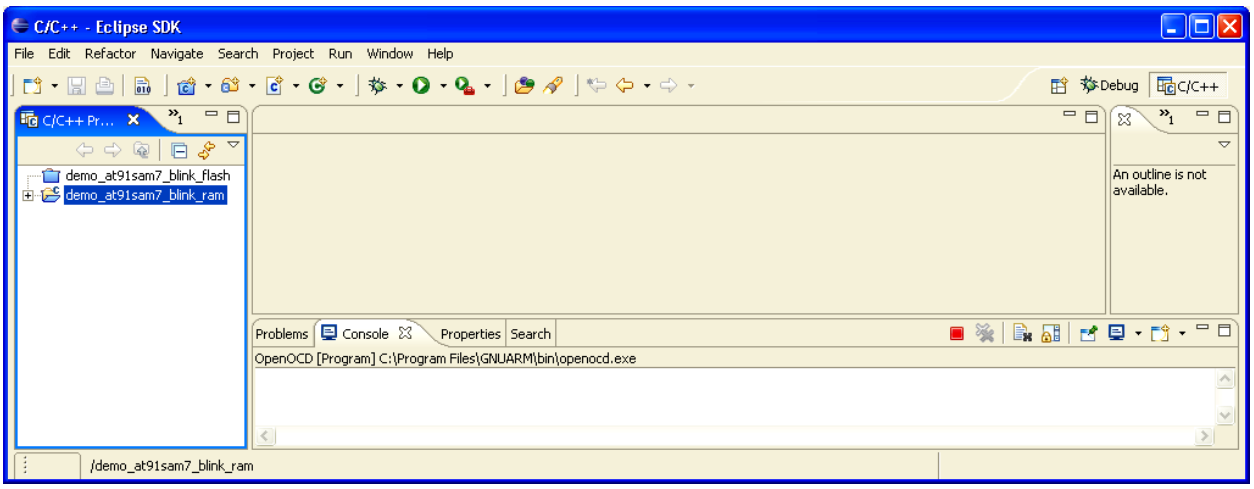
In the File menu, click on "**File – New – Standard Make C Project**" as shown below.



Give the new project the name “**demo_at91sam7_blink_ram**” and click “**Finish**”.



Now we have a project that has no files.



Now “**Import**” the source files from the download area for the project “**demo_at91sam7_blink_ram**” using the techniques learned earlier.

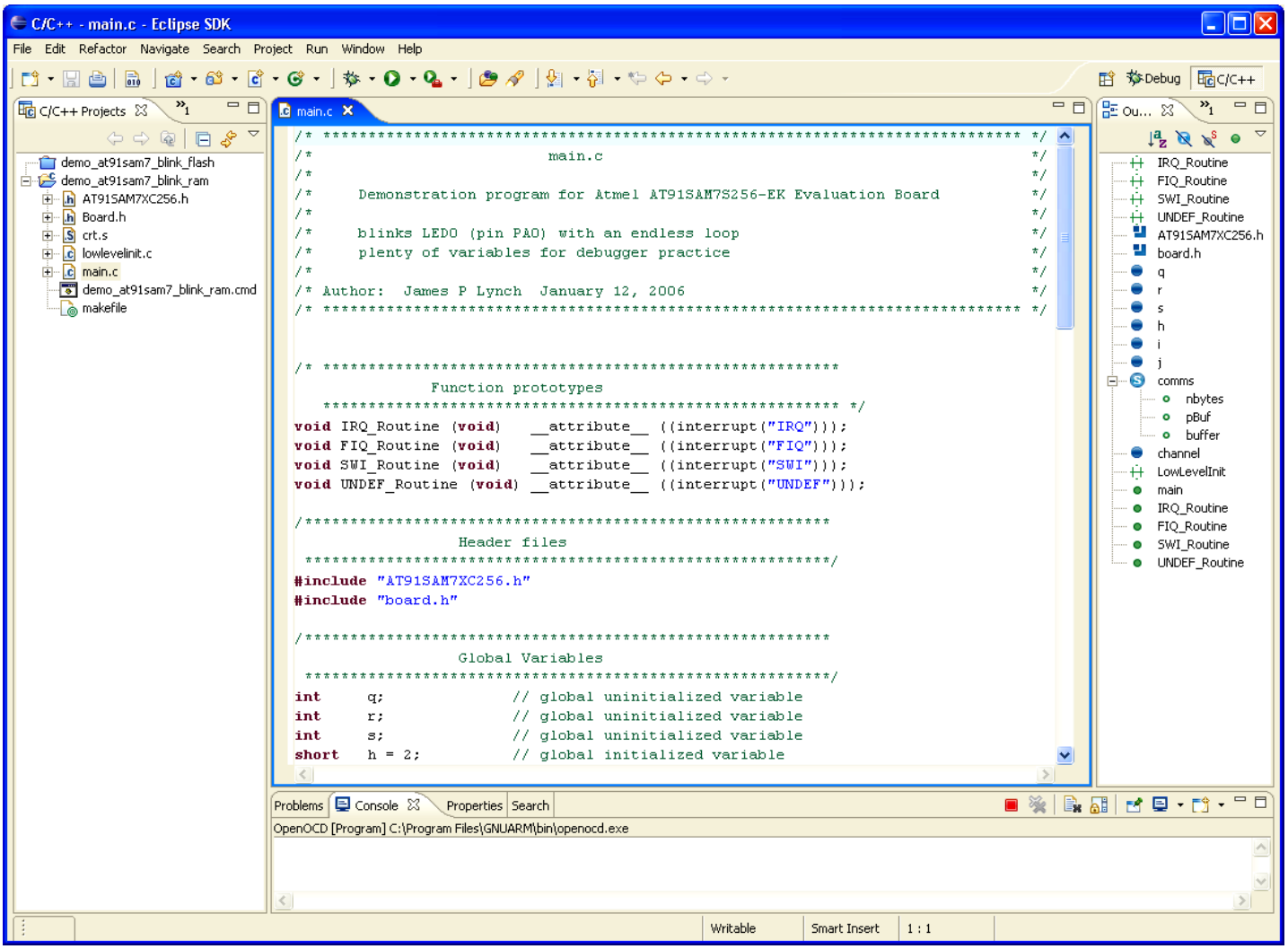
Only two files are different from the FLASH version:

demo_at91sam7_blink_ram.cmd - This file is different in that all code and variables are linked and loaded into address 0x00000000.

makefile.mak - this file references the file above (demo_at91sam7_blink_ram.cmd) so there are some minor edits therein.

All other files are exactly the same as the FLASH example.

Now we have a project with the proper files imported.



Only two files have changes and they are shown below.

DEMO_AT91SAM7_BLINK_RAM.CMD

```
/* ********************************************************************************************************** */
/*   demo_at91sam7_blink_ram.cmd              LINKER  SCRIPT                          AT91SAM7S256   */
/*                                                                                                            */
/*   The Linker Script normally defines how the code and data emitted by the GNU C compiler and               */
/*   assembler are to be loaded into memory (code goes into FLASH, variables go into RAM).                   */
/*                                                                                                            */
/*   However, this linker script is for execution entirely within RAM-space.                                  */
/*                                                                                                            */
/*                                                                                                            */
/*   This is fairly easy to accomplish:                                                                       */
/*                                                                                                            */
/*            1.  put everything (.text, .data, .bss) into FLASH                                              */
/*                                                                                                            */
/*            2.  cycle power on your board before starting the debugger                                      */
/*                                                                                                            */
/*            3.  When you start the debugger, set address 0xFFFFFF00 to 0x01                                 */
/*                This is the MC Remap Control Register                                                       */
/*                It re-maps RAM to address 0x00000000  (where the flash normally resides)                    */
```

```
/*                This is a "toggle" operation (why the power-cycle ensures that you don't get confused) */
/*                Now your code and variables will be at 0x00000000 and above.                           */
/*                                                                                                       */
/*            4.  Using your debugger, "Load" the entire application into RAM (now at 0x00000000)       */
/*                                                                                                       */
/*            5.  The application now runs entirely in RAM, is VERY fast, and you can                    */
/*                set unlimited breakpoints!                                                             */
/*                                                                                                       */
/*                                                                                                       */
/*   Any symbols defined in the Linker Script are automatically global and available to the rest of the  */
/*   program.                                                                                            */
/*                                                                                                       */
/*   To force the linker to use this LINKER SCRIPT, just add the -T demo_at91sam7_blink_ram.cmd          */
/*   directive to the linker flags in the makefile. For example,                                         */
/*                                                                                                       */
/*                         LFLAGS  =  -Map main.map -Tdemo_at91sam7_blink_ram.cmd                        */
/*                                                                                                       */
/*   The order that the object files are listed in the makefile determines what .text section is         */
/*   placed first.                                                                                       */
/*                                                                                                       */
/*   For example:  $(LD) $(LFLAGS) -o main.out  crt.o main.o lowlevelinit.o                              */
/*                                                                                                       */
/*                  crt.o is first in the list of objects, so it will be placed at address 0x00000000    */
/*                                                                                                       */
/*                                                                                                       */
/*   The top of the stack (_stack_end) is (last_byte_of_ram +1) - 4                                      */
/*                                                                                                       */
/*   Therefore:   _stack_end = (0x0000FFFF + 1) - 4  =  0x00010000 - 4  =  0x0000FFFC                    */
/*                                                                                                       */
/*   Note that this symbol (_stack_end) is automatically GLOBAL and will be used by the crt.s            */
/*   startup assembler routine to specify all stacks for the various ARM modes                           */
/*                                                                                                       */
/*                             MEMORY MAP                                                                */
/*                    |                                  |                                               */
/*            .-------->|----------------------------------|0x00010000                                     */
/*            .         |                                  |0x0000FFFC  <---------- _stack_end             */
/*            .         |    UDF Stack  16 bytes           |                                               */
/*            .         |----------------------------------|0x0000FFEC                                     */
/*            .         |                                  |                                               */
/*            .         |    ABT Stack  16 bytes           |                                               */
/*            .         |----------------------------------|0x0000FFDC                                     */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |    FIQ Stack  128 bytes          |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |----------------------------------|0x0000FF5C                                     */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |    IRQ Stack  128 bytes          |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |----------------------------------|0x0000FEDC                                     */
/*            .         |                                  |                                               */
/*            .         |    SVC Stack  16 bytes           |                                               */
/*            .         |                                  |                                               */
/*            .         |----------------------------------|0x0000FECC                                     */
/*            .         |                                  |                                              */
/*            .         |     stack area for user program  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |          free ram                |                                               */
/*          ram         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |..................................|0x000003F0 <---------- _bss_end                */
/*            .         |                                  |                                               */
/*            .         |  .bss   uninitialized variables  |                                               */
/*            .         |..................................|0x000003E4 <---------- _bss_start, _edata      */
/*            .         |                                  |                                               */
/*            .         |  .data  initialized variables    |                                               */
/*            .         |                                  |                                               */
/*            .         |----------------------------------|0x000003A8 <----------- _etext, _data          */


/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |0x00000260  lowlevelinit()                     */
/*            .         |            C code                |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |                                  |                                               */
/*            .         |----------------------------------|0x00000108  main()                             */
/*            .         |                                  |                                               */
/*            .         |    Startup Code  (crt.s)         |                                               */
```

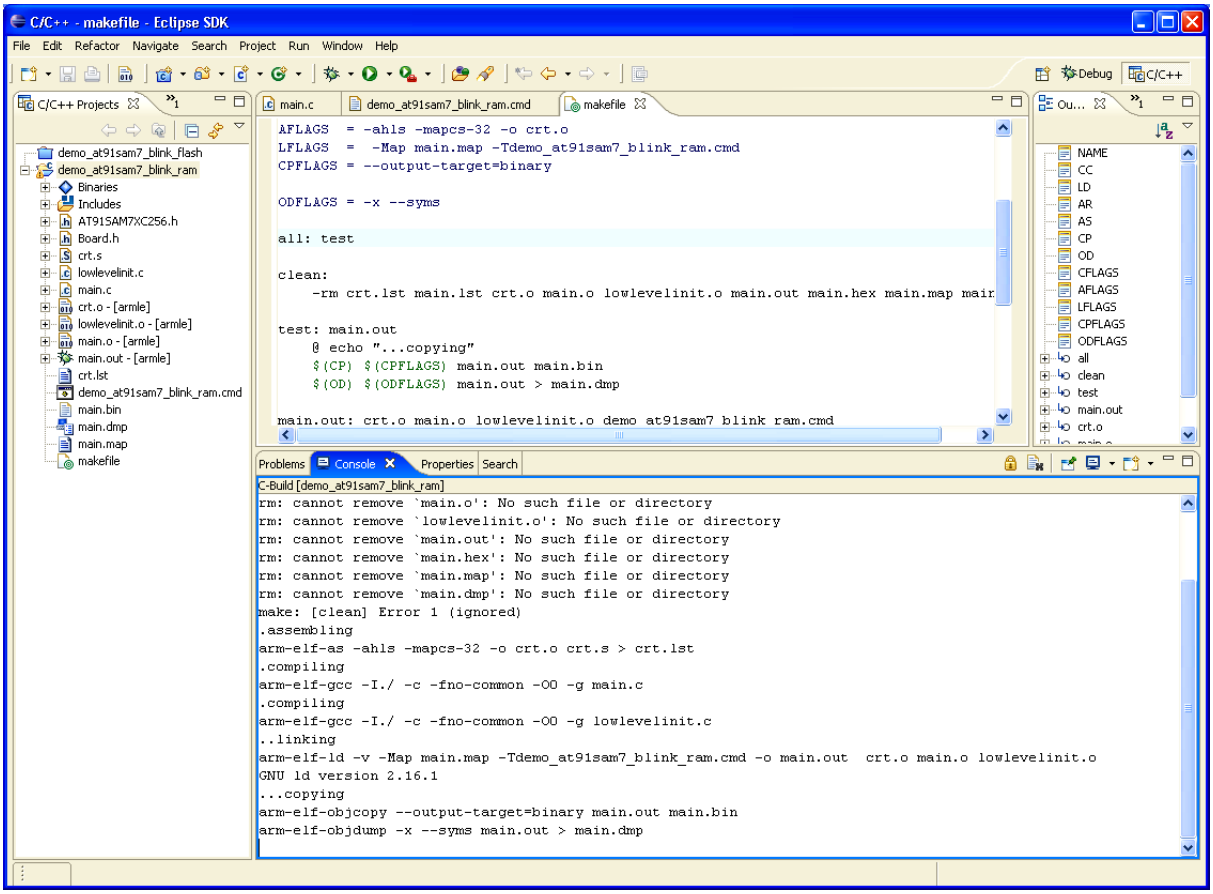
```
/*            .         |         (assembler)             |                                          */
/*            .         |                                 |                                          */
/*            .         |---------------------------------|0x00000020                                */
/*            .         |                                 |                                          */
/*            .         | Interrupt Vector Table          |                                          */
/*            .         |          32 bytes               |                                          */
/*            .-------->|---------------------------------|0x00000000 _startup                       */
/*                                                                                                   */
/*                                                                                                   */
/*  Author:  James P. Lynch                                                                          */
/*                                                                                                   */
/* ************************************************************************************************* */


/* identify the Entry Point  (_vec_reset is defined in file crt.s)  */
ENTRY(_vec_reset)

/* specify the AT91SAM7S256 memory areas  */
MEMORY
{
         flash     : ORIGIN = 0,          LENGTH = 256K  /* FLASH EPROM             */
         ram       : ORIGIN = 0x00200000, LENGTH = 64K   /* static RAM area */
}


/* define a global symbol _stack_end  (see analysis in annotation above) */
_stack_end = 0xFFFC;

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

    .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 */
    } >flash                         /* put all executable code (.text) into FLASH    */

    .data :                          /* collect all initialized .data sections that go into FLASH  */
    {
         _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  */
    } >flash                         /* put all initialized variables (.data) into FLASH  */

    .bss :                           /* collect all uninitialized .bss sections that go into FLASH  */
    {
         _bss_start = .;             /* define a global symbol marking the start of the .bss section */
         *(.bss)                     /* all .bss sections  */
    } >flash                        /* put all uninitialized variables (.bss) into FLASH  */

    . = 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 */
```

MAKEFILE.MAK

```
NAME   = demo_at91sam7_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 -Tdemo_at91sam7_blink_ram.cmd
CPFLAGS = --output-target=binary

ODFLAGS = -x --syms

all: test

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

test: main.out
        @ echo "...copying"
        $(CP) $(CPFLAGS) main.out main.bin
        $(OD) $(ODFLAGS) main.out > main.dmp

main.out: crt.o main.o lowlevelinit.o demo_at91sam7_blink_ram.cmd
        @ echo "..linking"
        $(LD) $(LFLAGS) -o main.out  crt.o main.o lowlevelinit.o

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

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

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

Build the RAM Project

Using the “**Build All**” button, build the new RAM Project.

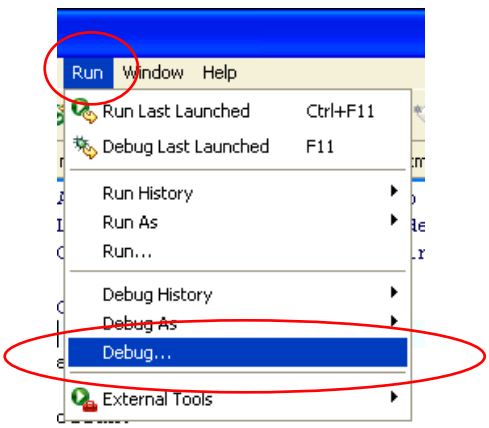


In this version, we will be using the “**main.out**” file to download the executable into RAM via the JTAG.

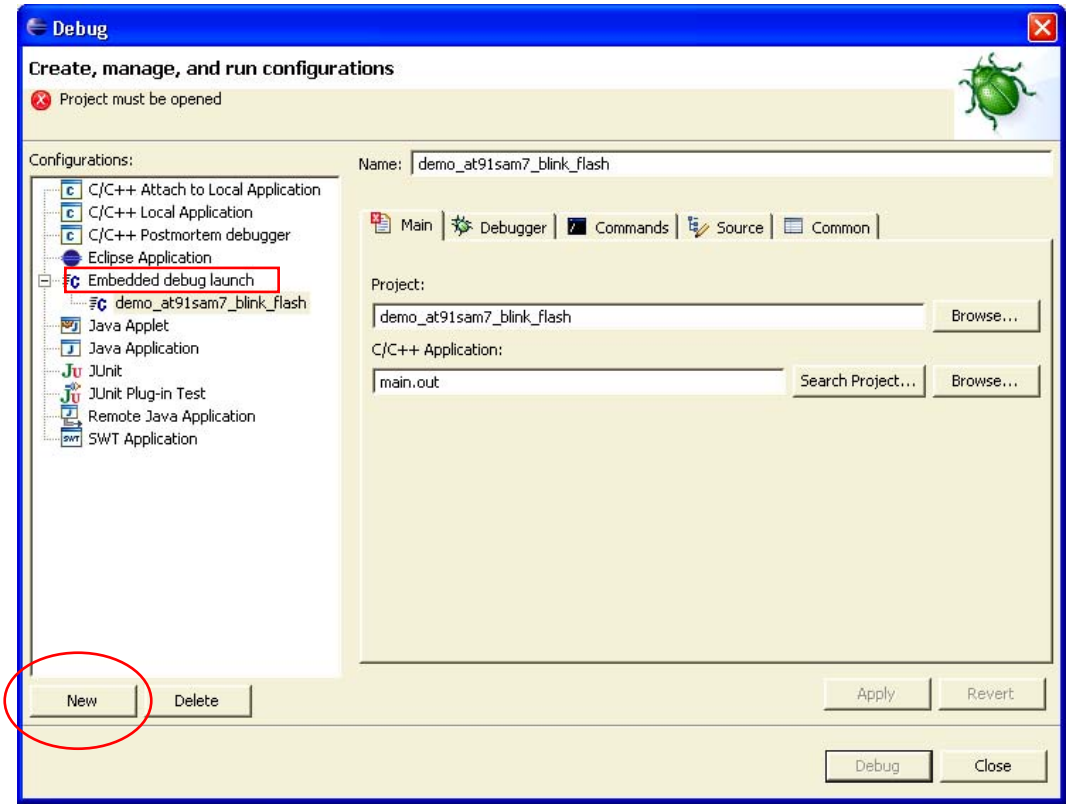
Create an Embedded Debug Launch Configuration

A separate Debug Launch Configuration is appropriate since the debugger startup script will be different and the downloading into RAM will be performed by the JTAG interface.

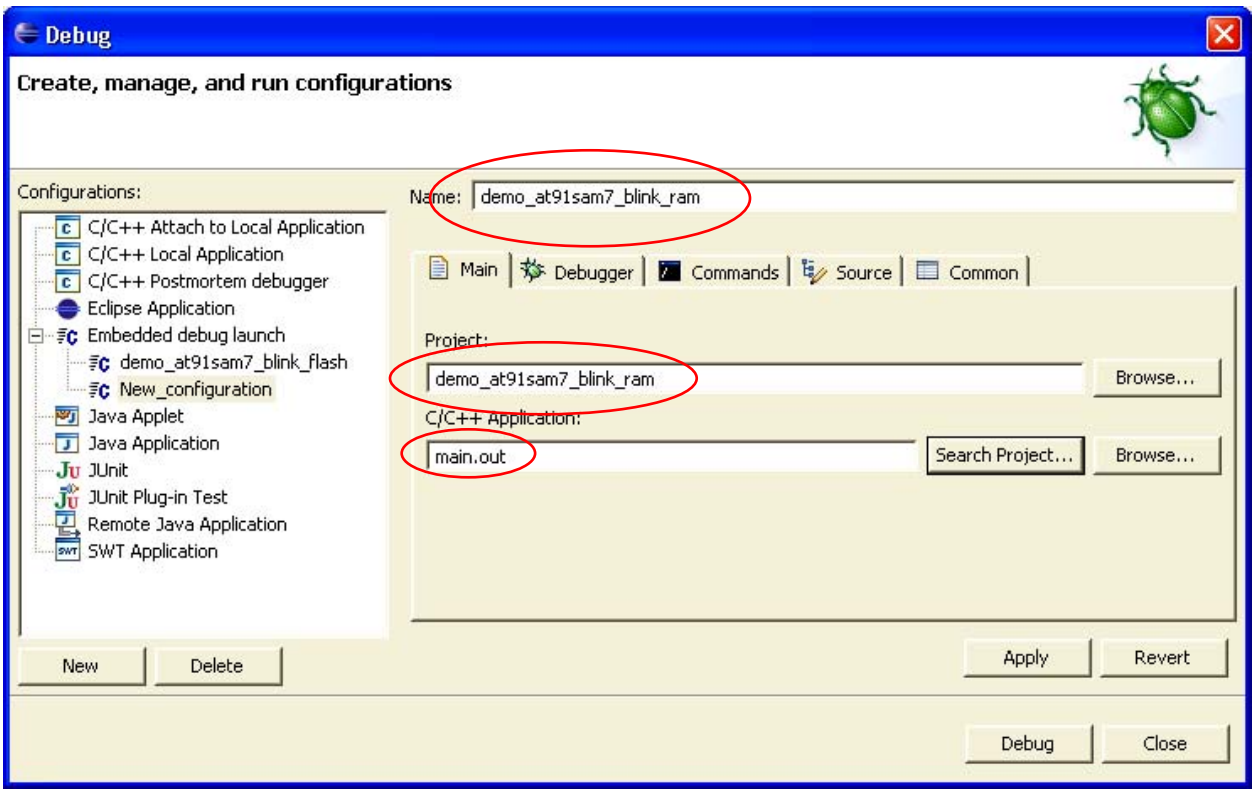
Click on “**Run**” followed by “**Debug…**”.



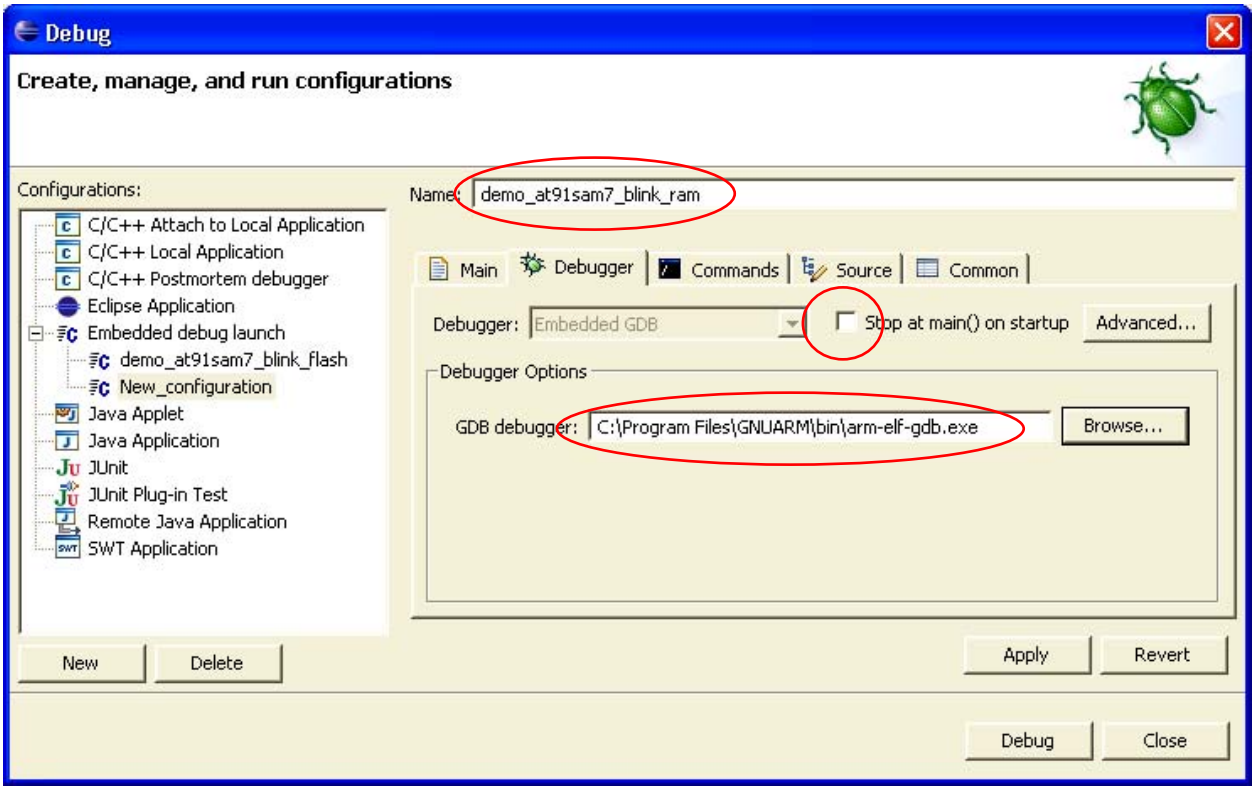
When the Debug “Create, manage, and run configurations” screen appears, click on “**Embedded debug launch**” followed by “**New**”.



A new and empty “Embedded debug launch” configuration screen will appear. Under the “**Main**” tab, fill out the new configuration screen as shown below.

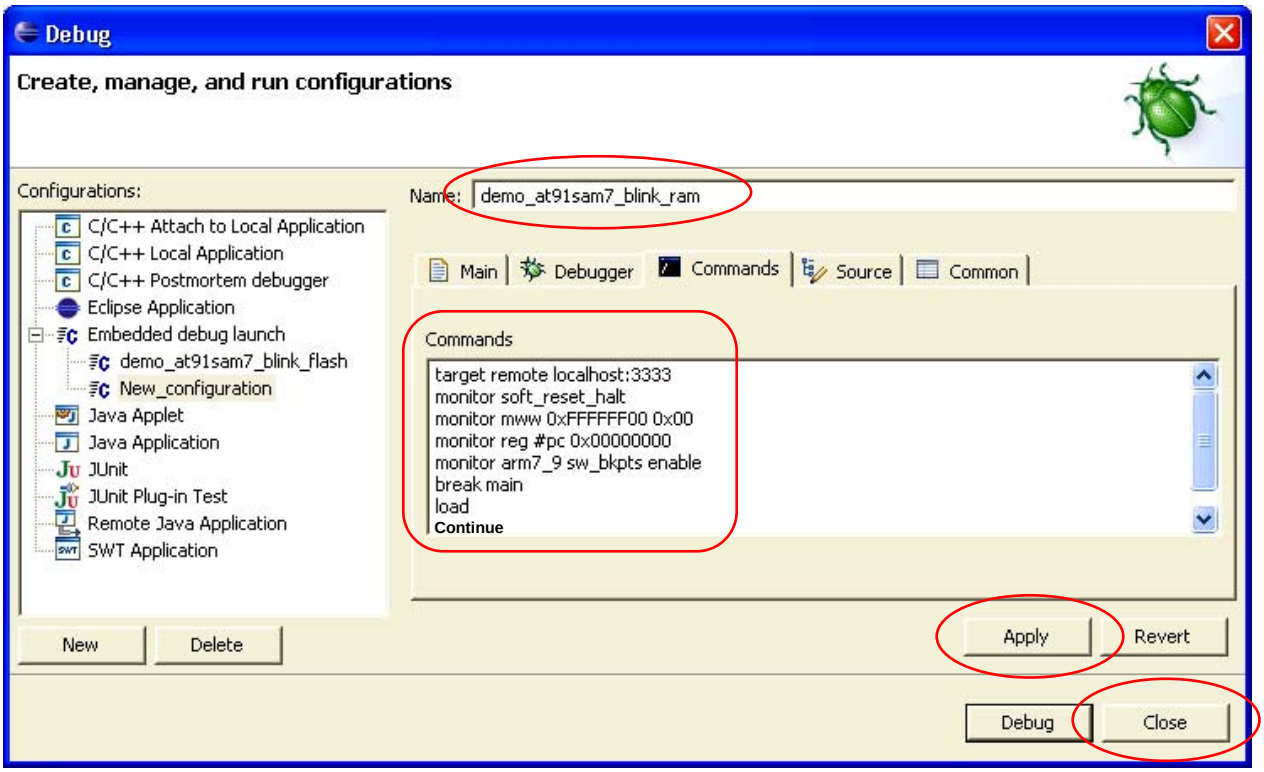


Under the “**Debugger**” tab, fill out the screen as shown below.



Finally under the “**Commands**” tab, fill out the screen as shown below.

The other two tabs can be left at their default values. Click on “**Apply**” followed by “**Close**” to complete specification of the “embedded launch configuration” for the RAM-targeted application.



The eight GDB startup commands require some explanation. If the command line starts with the word “monitor”, then that command is an OpenOCD command. Otherwise, the command is a GDB command.

target remote localhost:3333

This is a GDB command. 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:3333 (the default specification for the OpenOCD GDB Server).

monitor soft_reset_halt

This is an OpenOCD command. This is a special reset command developed by Dominic Rath for ARM microprocessors.

monitor mww 0xFFFFFF00 0x00

This is an OpenOCD command. It sets the Atmel AT91SAM7S256 MC Remap Control Register to 1 which toggles the remap state. This effectively overlays RAM on top of low memory starting at address 0x00000000. Be forewarned that this register has a “toggle” action and it behooves you to power-cycle the AT91SAM7S board before running the debugger to ensure that you are starting from a RESET state.

monitor reg #pc 0x00

This is an OpenOCD command. It sets the PC to address 0x00000000. This ensures that we will start at the reset vector when the “continue” command is given.

monitor arm7_9 arm7_9 sw_bkpts enable

This is an OpenOCD command. It enables software breakpoint commands emitted by Eclipse/GDB.

break main

This is a GDB command. It sets a breakpoint at the entry point main().

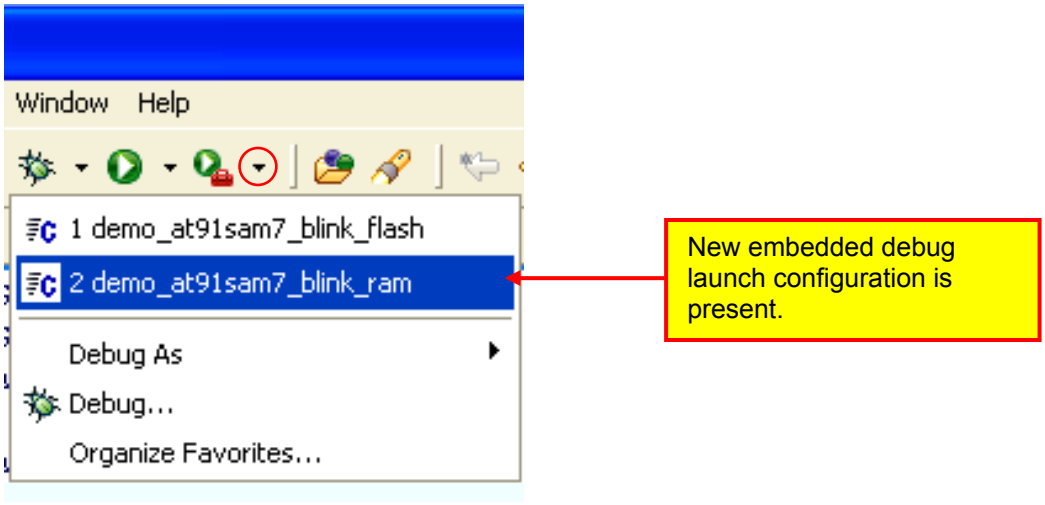
load main

This is a GDB command. It instructs the debugger to load the executable code into RAM and utilize the symbolic information in the main.out file for debugging.

continue

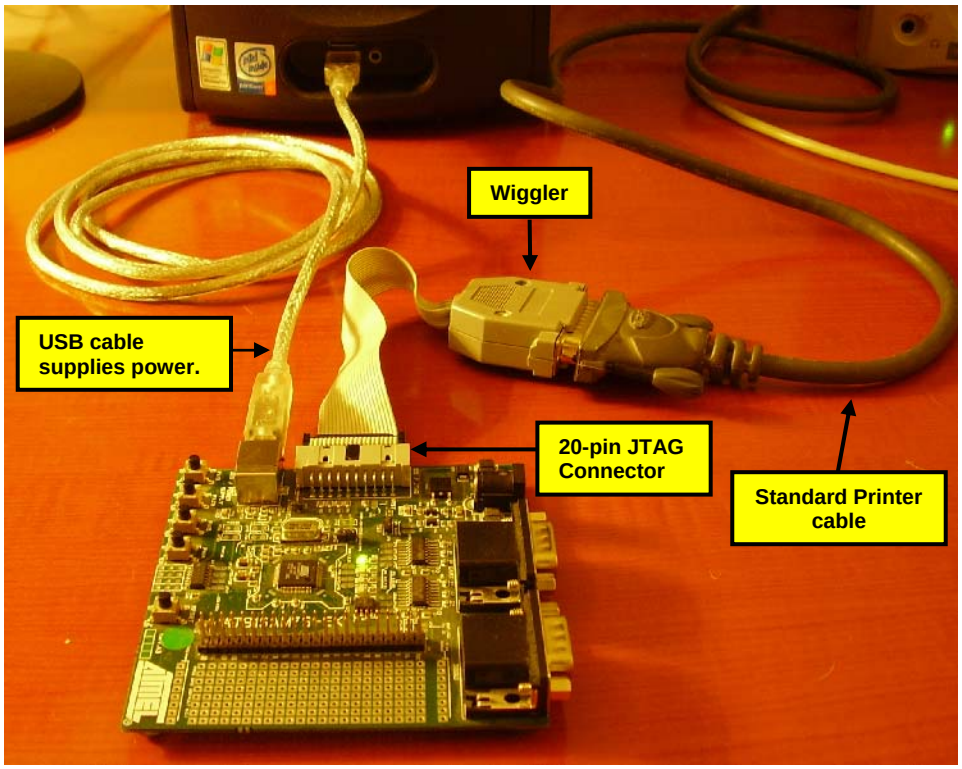
This is a GDB command. It forces the ARM processor out of breakpoint/halt state and resumes execution from main().

Using the procedures shown earlier, add the "**demo_at91sam7_blink_ram**" embedded launch configuration to the list of favorites. Thus, when we click on the "**Debug**" toolbar button's down arrowhead, we see both launch configurations available.



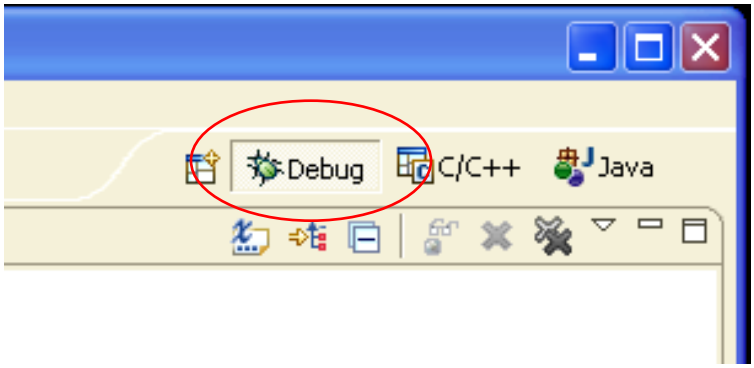
Set up the hardware

In this RAM setup, we use the USB cable to supply power. The "wiggler" is attached to the PC's printer port (LPT1) and the 20-pin JTAG socket on the AT91SAM7S-EK evaluation board. This is exactly the same hardware setup at that used for FLASH debugging.

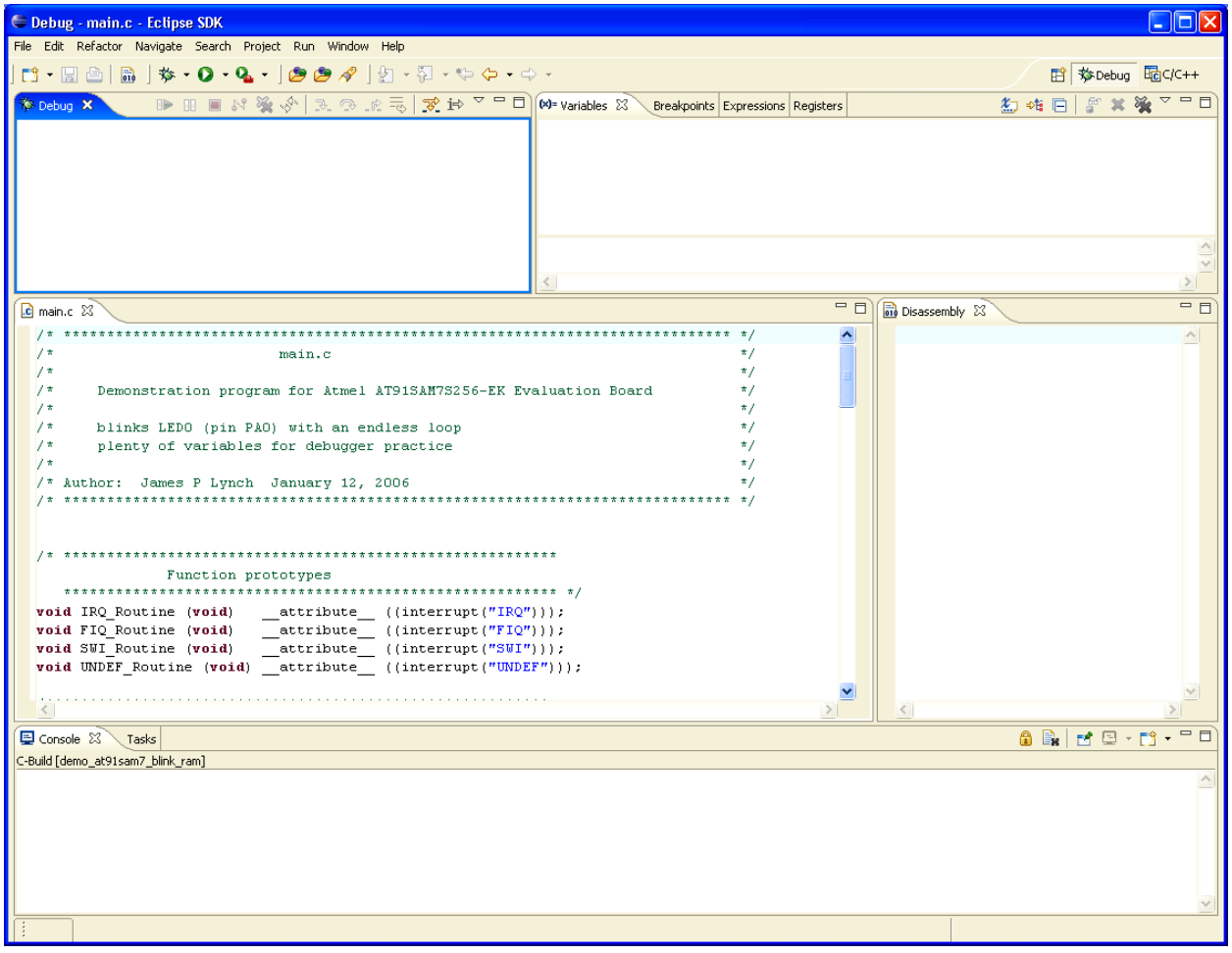


Open the Eclipse “Debug” Perspective

As shown earlier, click on the “**Debug**” perspective button located at the upper right part of the Eclipse screen.



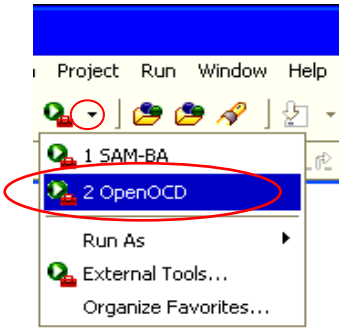
Now the **Debug** perspective will appear, as shown below.



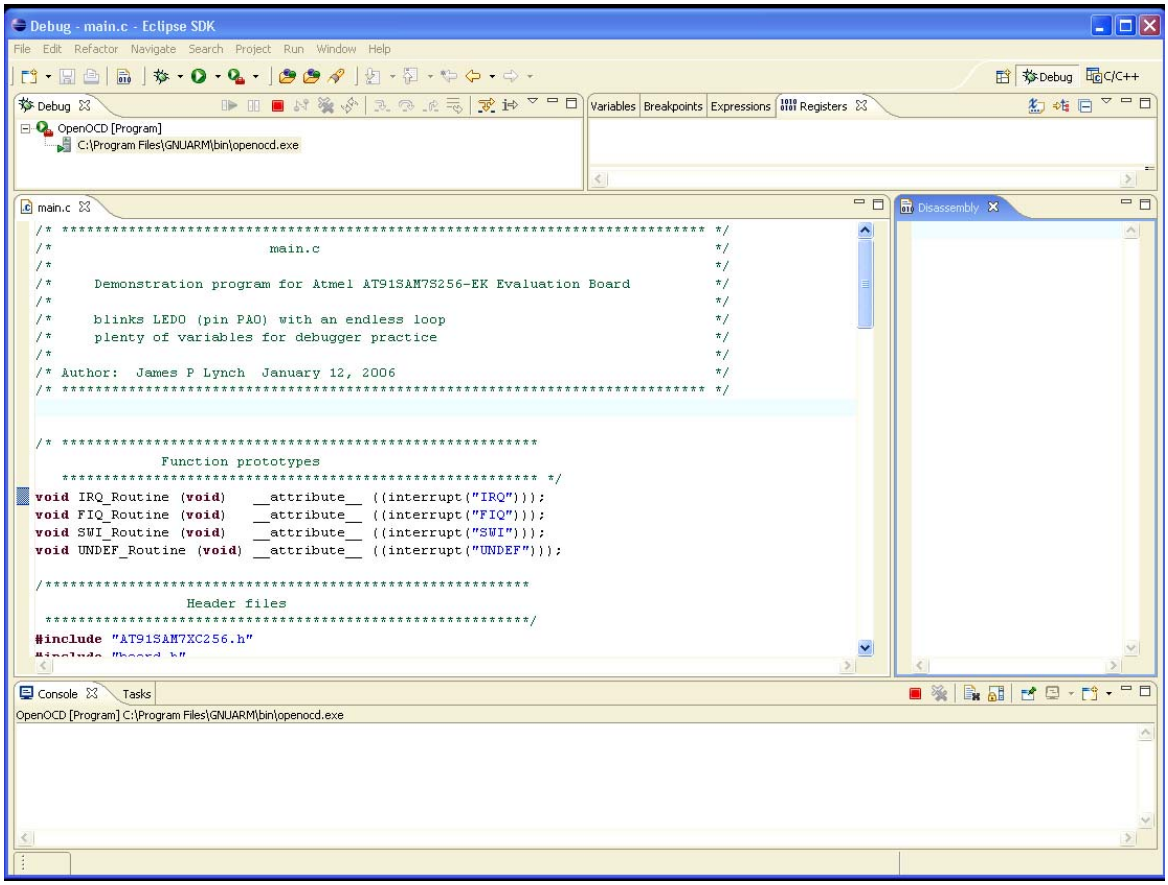
Start OpenOCD

Before starting **OpenOCD**, get into the habit of cycling the power of the Atmel **AT91SAM7-EK** board. Pulling and then re-inserting the USB cable will accomplish this. If you are using a 9 volt power supply, remove and re-insert the power plug. For the RAM application, cycling the power is CRUCIAL for success (due to the toggling nature of the MC Remap Control Register).

To start **OpenOCD**, click on the "**External Tools**" toolbar button's down arrowhead and then select "**OpenOCD**". Alternatively, you can click on the "**Run**" pull-down menu and select "**External Tools**" followed by "**OpenOCD**".



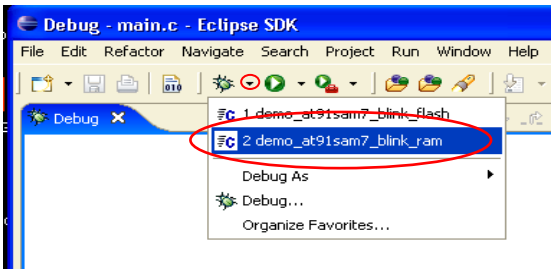
The debug view will show that **OpenOCD** is running and the console view shows no errors (warnings are OK).



Start the Eclipse Debugger

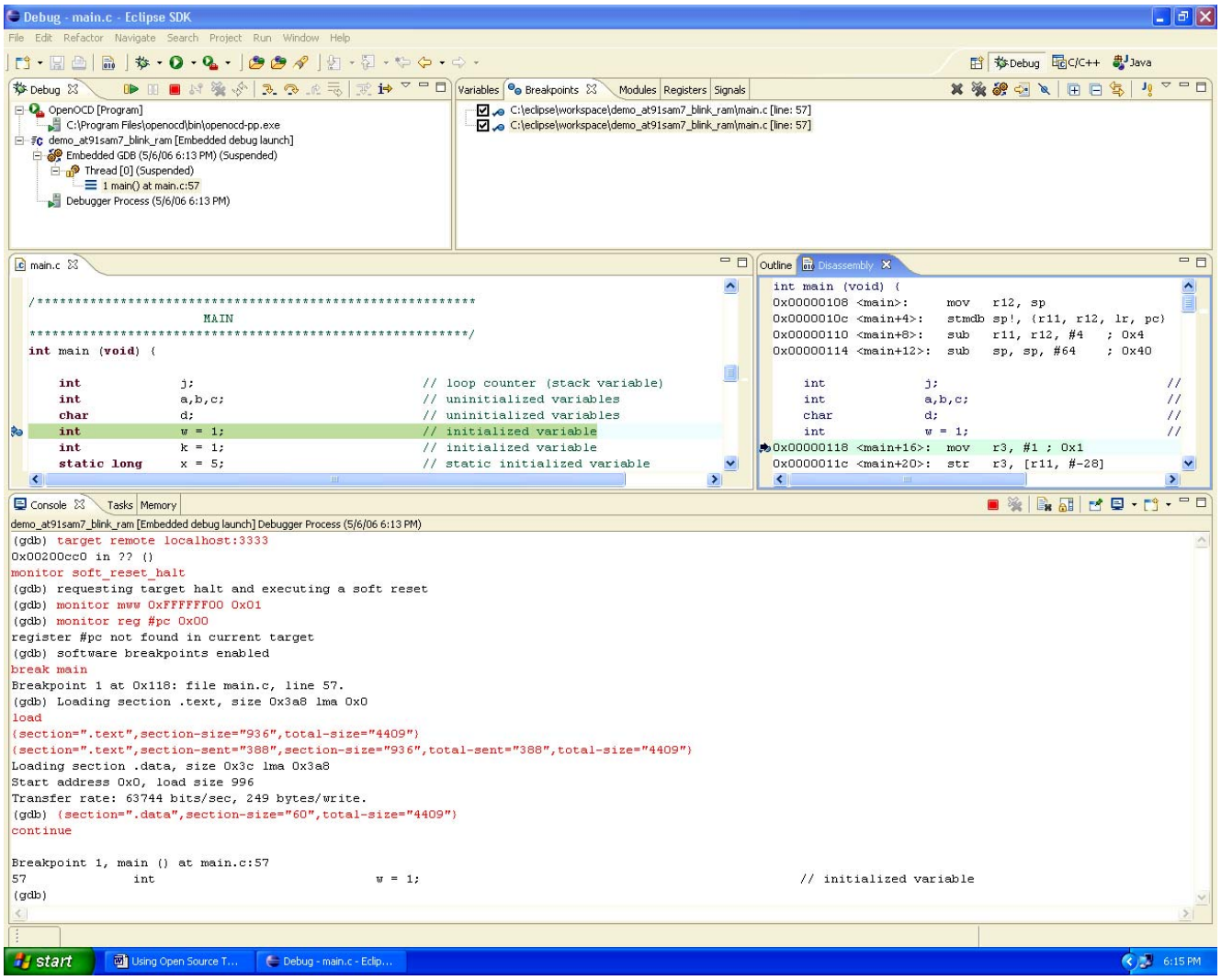
To start the Eclipse debugger, click on the "**Debug**" toolbar button's down arrowhead and select the debug launch configuration "**demo_at91sam7_blink_ram**" as shown below.

Alternatively, you can start the debugger by clicking on "**Run – Debug…**" and then select the "**demo_at91sam7_blink_ram**" embedded launch configuration and then click "**debug**". Obviously, the debug toolbar button is more convenient.



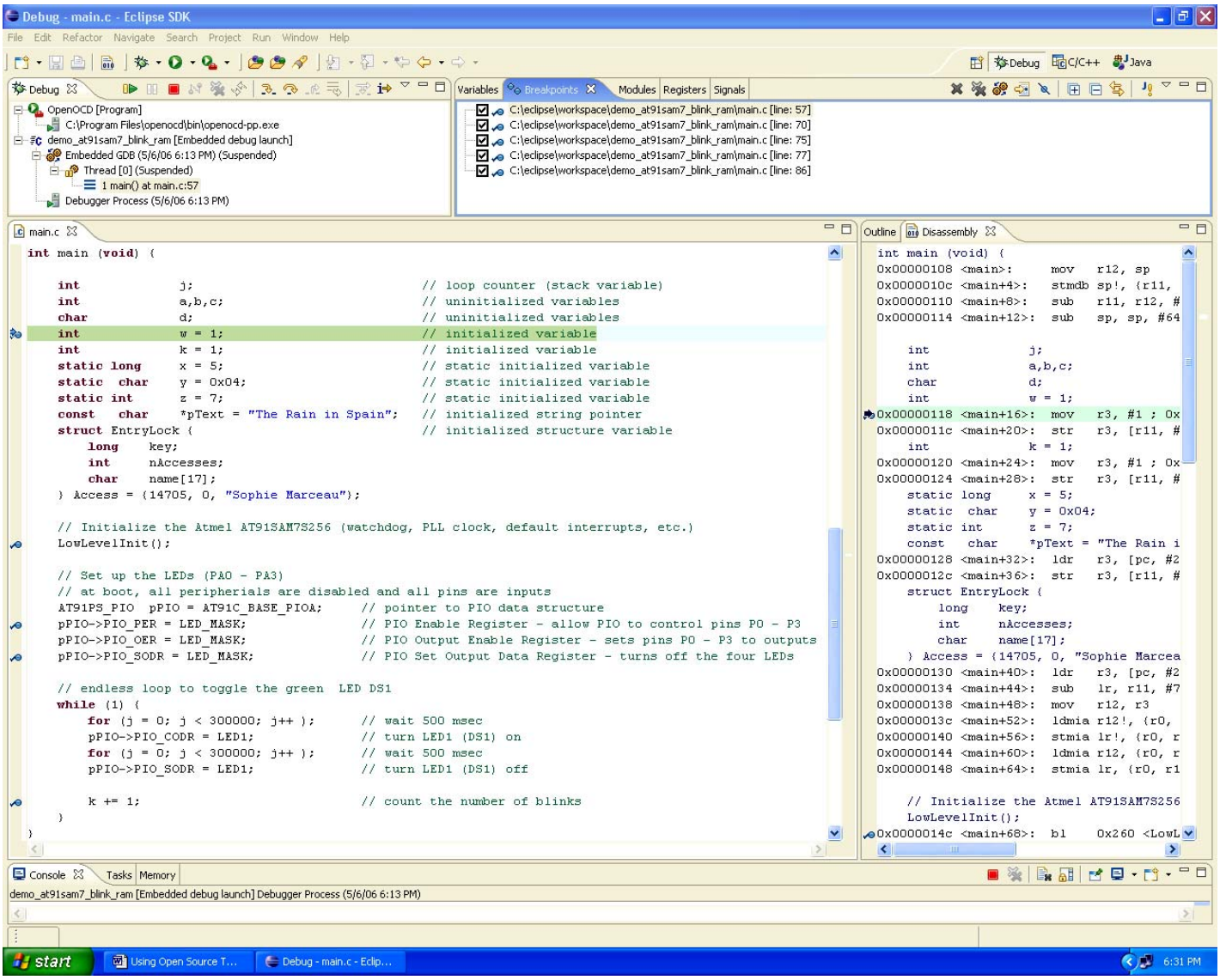
If the Eclipse debugger starts properly, the debug view (upper left) shows that the debugger has stopped at line 57 in main(). The console view (bottom) shows each GDB command that was executed in red letters and some **OpenOCD** messages after some of the commands.

If the Eclipse debugger doesn't connect properly, then there will be a progress bar at the bottom left status line that runs forever. In this case, terminate everything and power cycle the target board again.

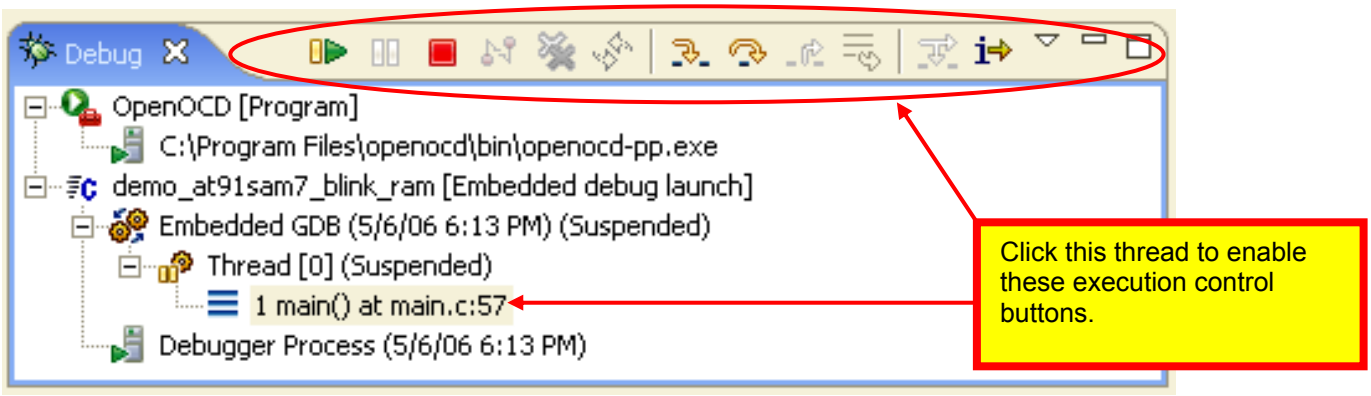


Setting Software Breakpoints

The big advantage of running entirely from on chip RAM is that you can set an unlimited number of software breakpoints. In the example below, we have set four breakpoints plus the breakpoint set at main( ).



Let's remind ourselves that the Eclipse debugger can handle multiple threads of execution. Since our ARM system only has one thread, you must click on it (highlight it) to enable the execution control commands to work. As shown below, the thread "1 main() at main.c:57" has been clicked and thus highlighted.



Click on the “Resume”  button and the debugger executes to our first breakpoint.

```
    // Initialize the Atmel AT91SAM7S256 (watchdog, PLL clock,
    LowLevelInit();
```

Click on the “Resume”  button again and the debugger executes to our second breakpoint.

```
    AT91PS_PIO  pPIO = AT91C_BASE_PIOA;
    pPIO->PIO_PER = LED_MASK;
    pPIO->PIO_OER = LED_MASK;
```

Click on the “Resume”  button again and the debugger executes to our third breakpoint.

```
    pPIO->PIO_OER = LED_MASK;
    pPIO->PIO_SODR = LED_MASK;
```

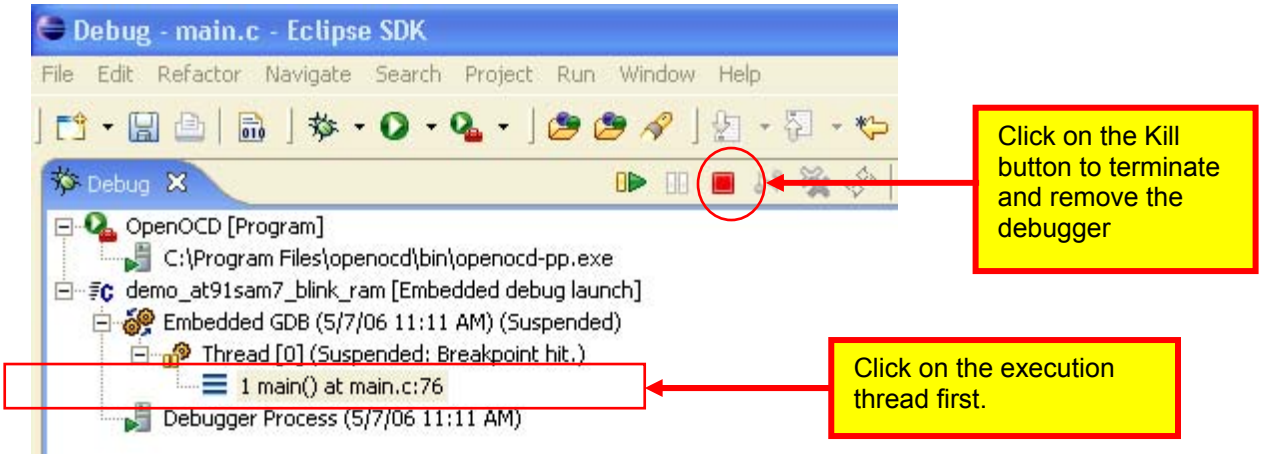
And so on, now we have an unlimited number of breakpoints available. All other features of the debugger still apply to the RAM-targeted application.

Compiling from the Debug Perspective

You can conveniently stop the debugger and the OpenOCD, modify your source file and re-compile your application all within the Debug perspective. The following procedure is a safe way to do this.

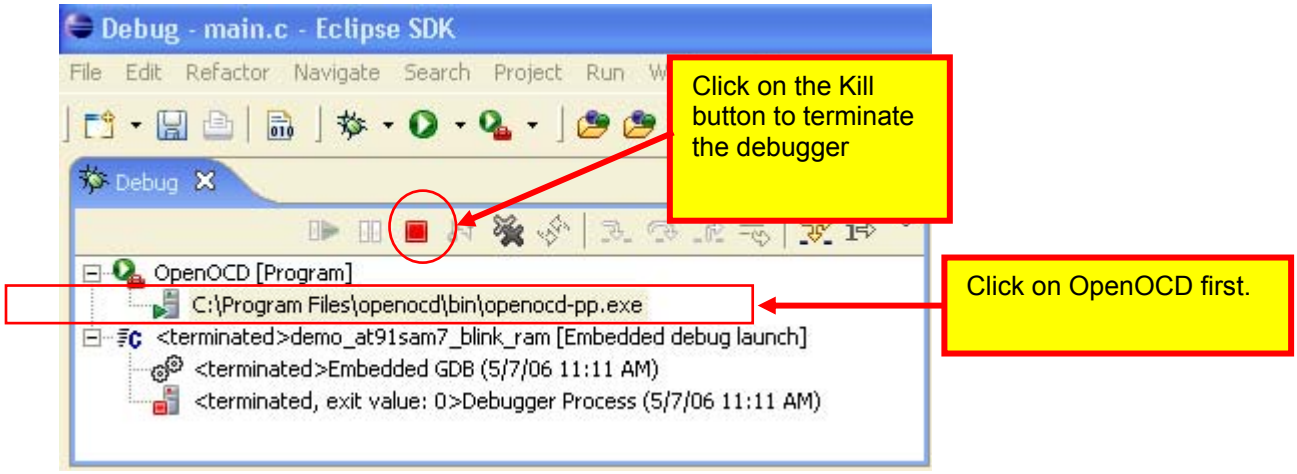
- **Stop the Debugger**

  Click on the execution thread to highlight it and then click on the KILL button to terminate it.



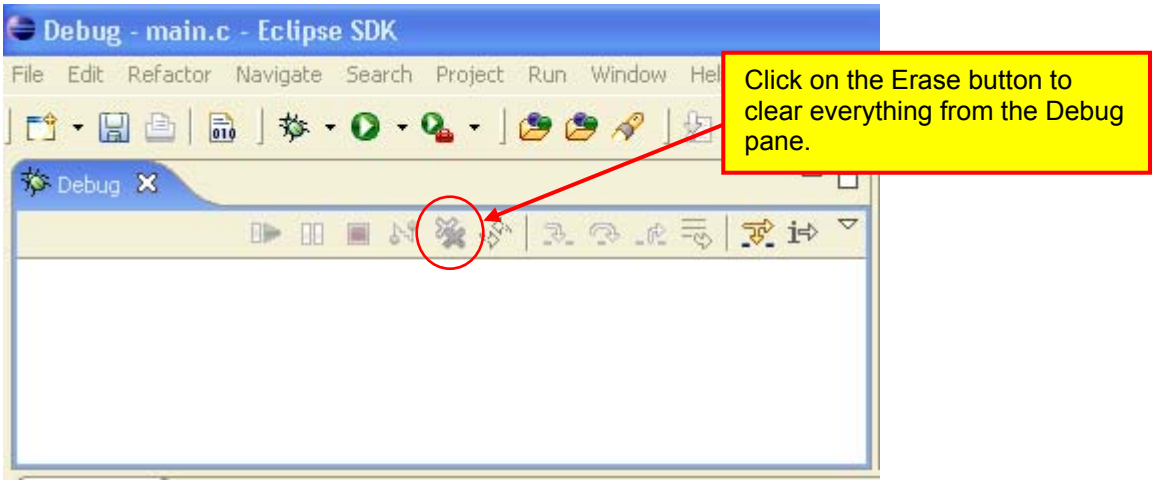
- **Stop OpenOCD**

Click on OpenOCD followed by clicking on the KILL button to terminate the OpenOCD JTAG utility.



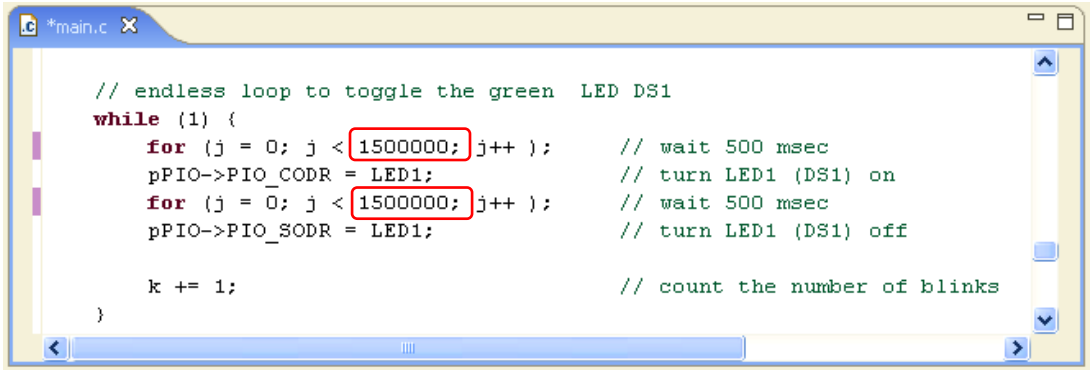
- **Erase the Debug Pane**

Click on the Erase button to clear everything from the Debug pane.



- **Modify the Source File**

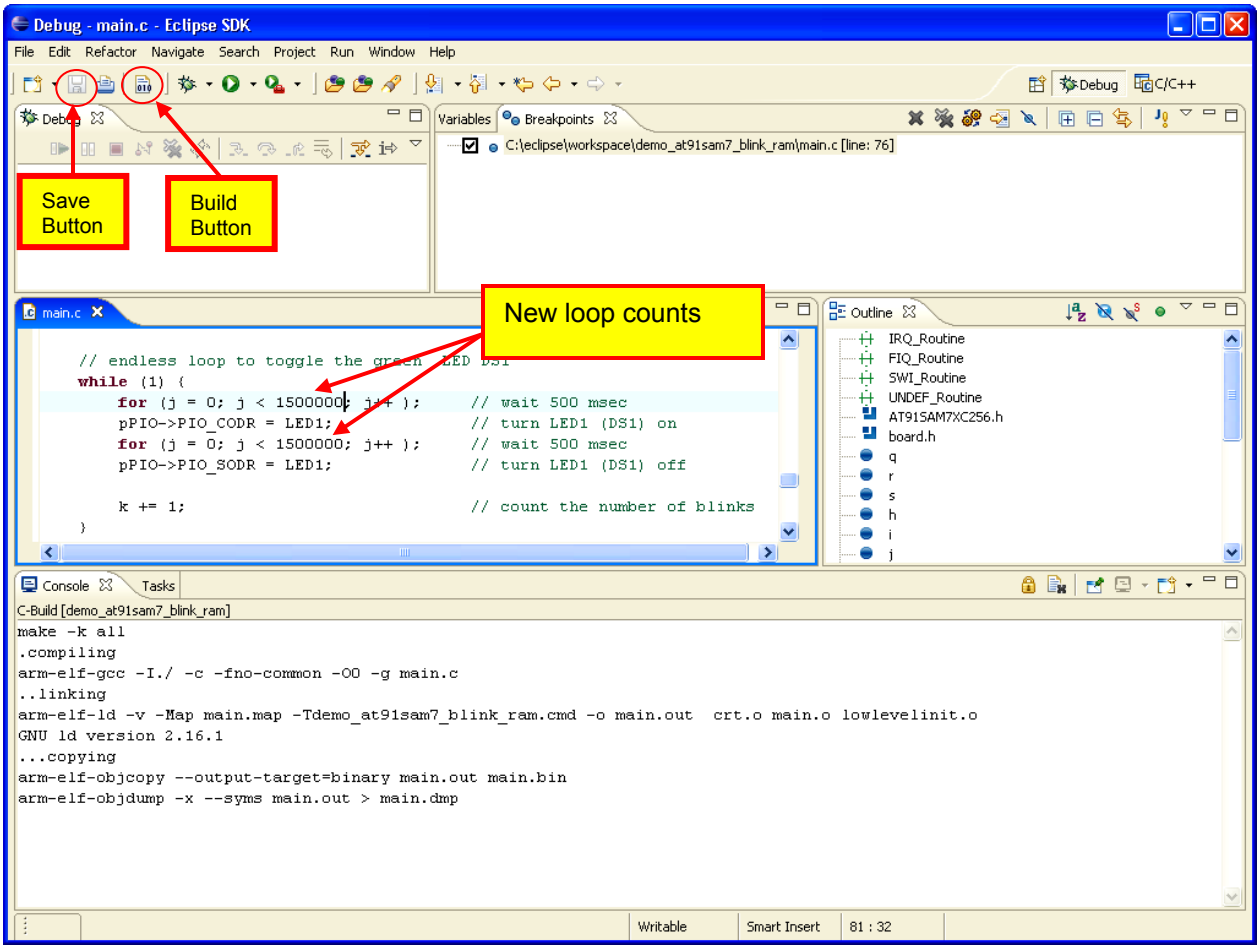
Here we have changed the wait time by modifying the loop counts.



- **Re-Compile and Link the Application**

  To change the blink rate, we modified the loop counts. We then saved the source file using the "**Save**" button.

  Next we re-built the application by clicking on the "**Build All**" button, as shown below. The Console view shows that the compile and link steps ran successfully. Note that it only compiled the source file main.c.



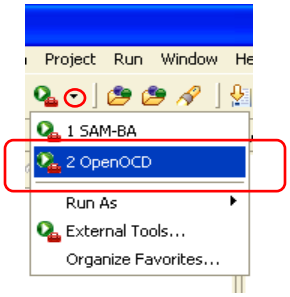
- **Cycle the Target Board Power**

  This can be done by detaching and re-inserting the USB cable (which supplies board power).

  This step may not always be necessary but it insures that we start from the same place every time. The Atmel AT91SAM7S256 has a memory map register that "toggles" and it's possible to get out-of-sync.

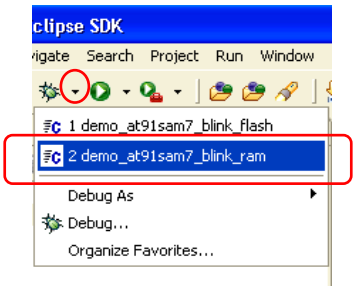
- **Start OpenOCD**

  Using the External Tools toolbar button, find and start the OpenOCD JTAG utility.



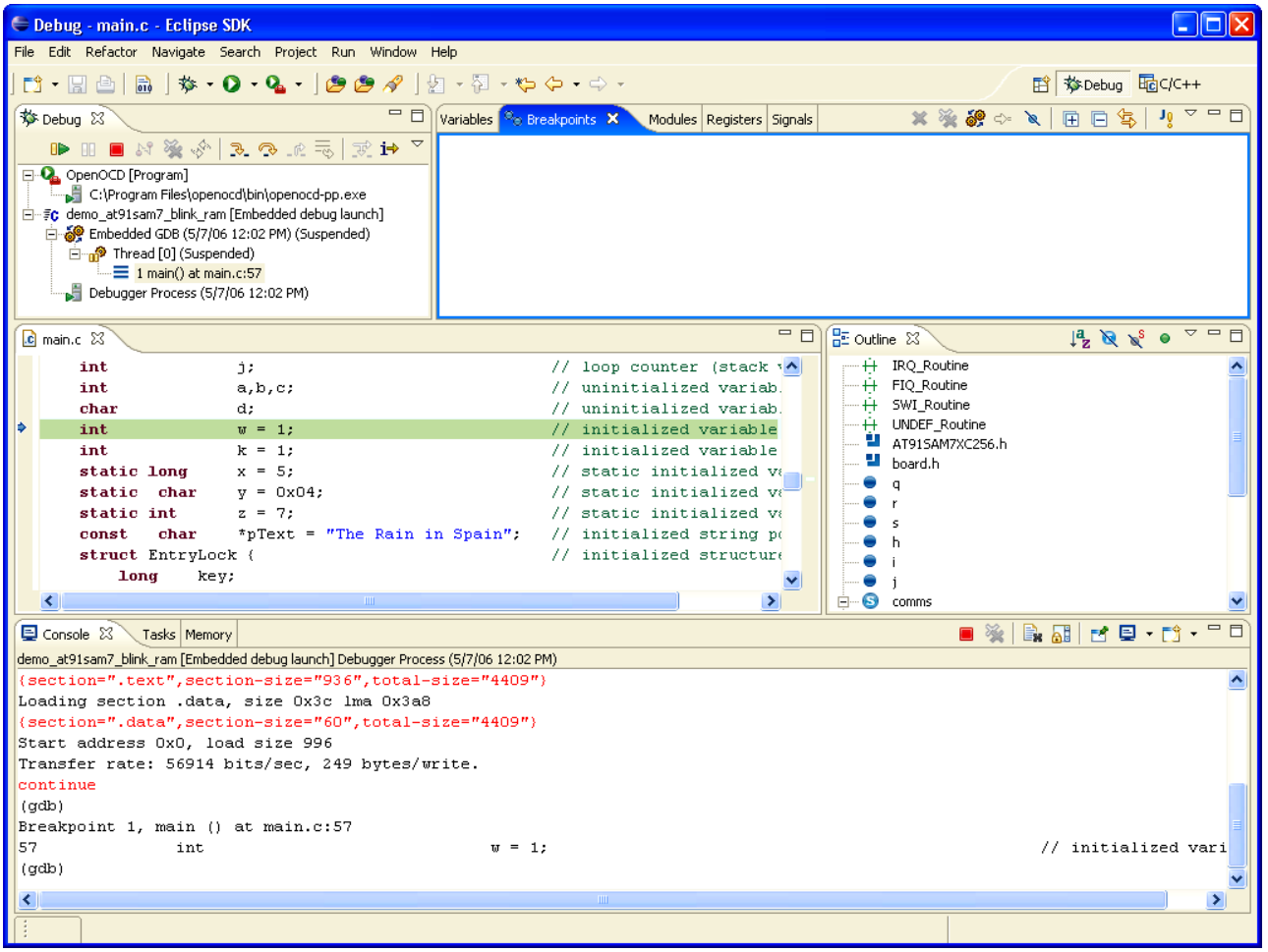
- **Start the Eclipse Debugger**

  Using the Debug toolbar button, find and start the at91sam7_blink_ram debug configuration.



- **Repeat your Debugging Session**

  Now the Eclipse debugger is stopped at the function main( ), awaiting your next instructions. Once you have this procedure committed to memory, you will find RAM-based debugging a real pleasure.



When we are debugging a FLASH-based application, we have to start the SAM-BA flash programmer and program the flash before starting the OpenOCD and the Eclipse debugger. Since the SAM-BA uses the USB port, that operation is still quite speedy. Nonetheless, it's an extra step which explains the popularity of debugging entirely within RAM memory.

Amontec JTAGkey

For the student or hobbyist, the $19 ARM-JTAG interface (wiggler) is an inexpensive way to start developing and debugging software for Atmel AT91SAM7S processors. However, the PC printer port limits the device's ability to toggle the JTAG pins to about 500 kHz and thus large downloads will take a long time. In Dominic Rath's thesis about the OpenOCD project, a USB-JTAG interface based on the FTDI FT2232C engine was described with a schematic.

Fortunately, there's no need to build this since the Swiss engineering firm Amontec has already developed and marketed a nice version of this USB JTAG interface called the **JTAGkey**. Its price is €139 (euros) or $177 (us). The **JTAGkey** is professionally designed and manufactured with additional bells and whistles, such as status LEDs and ESD protection. **JTAGkey** also automatically senses and adjusts the level shifters for the ARM voltage level; this will come in handy when lower voltage versions (e.g. 1.8 volts) of the ARM become available. The Amontec **JTAGkey** can be purchased online from here:

http://www.amontec.com/JTAGkey.shtml

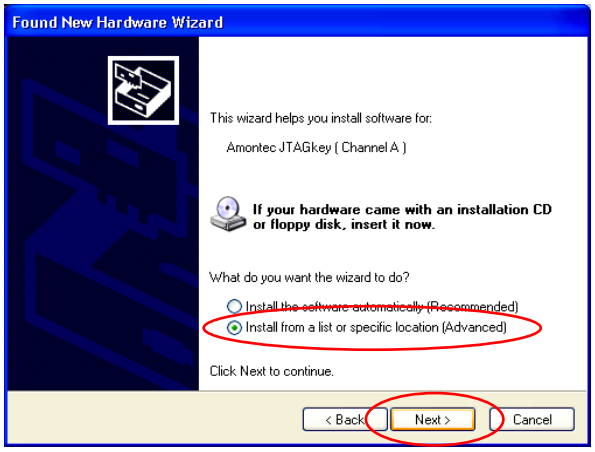
Install the JTAGkey USB Virtual Device Drivers

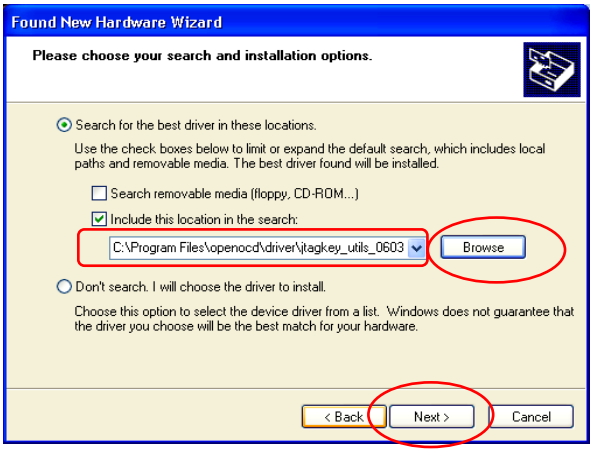
Plug in the Amontec JTAGkey into the USB port. You should hear the familiar USB "beep" sound followed by the following screen indicating that new USB hardware has been detected.

The virtual device drivers are already on our c:\Program Files\openocd folder thanks to Michael Fischer's OpenOCD installation program we ran earlier in this tutorial. Therefore, advise Windows NOT to search for the drivers by clicking on "**No, not this time**" as shown below. Click "**Next**" to continue.

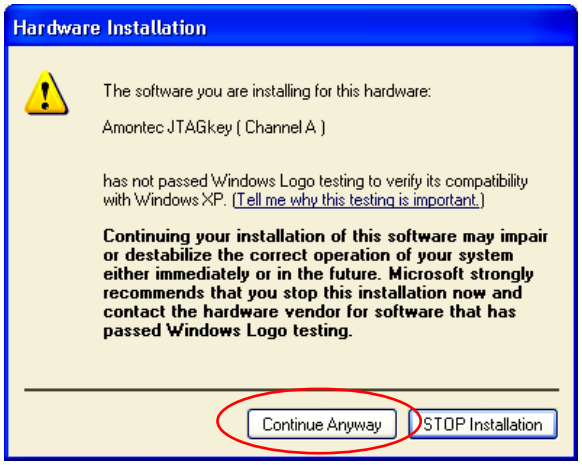
Instruct Windows to “**Install from a list or specific location (Advanced)**” as shown on the left hand screen below. Click “**Next** to continue.

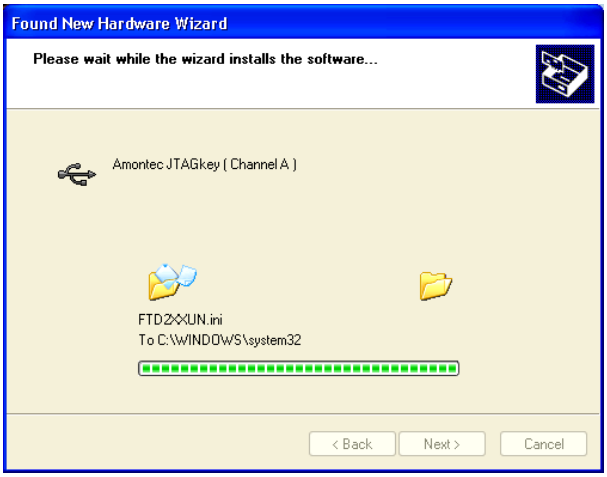
Now use the “**Browse**” button to find the directory “**c:\Program Files\openocd\driver\jtagkey_utils_0603**” as shown below on the right hand screen. Click “**Next** to continue.



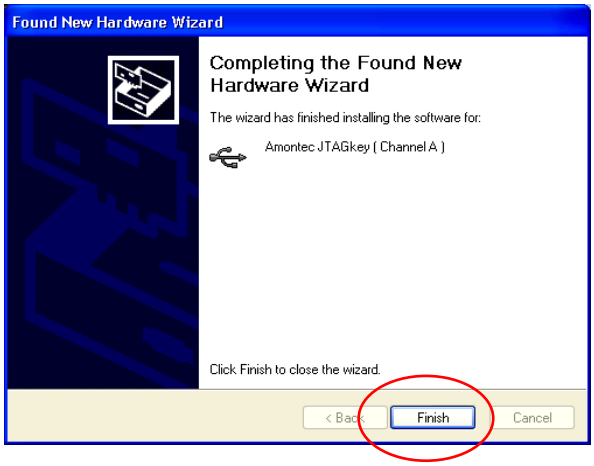


Pay no attention to Windows complaints about Logo testing by clicking on “**Continue Anyway**” on the left screen below. The virtual device driver for Channel A will now run to completion.

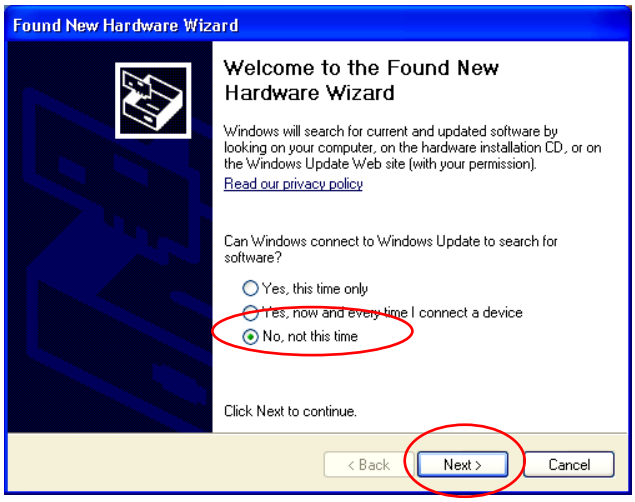


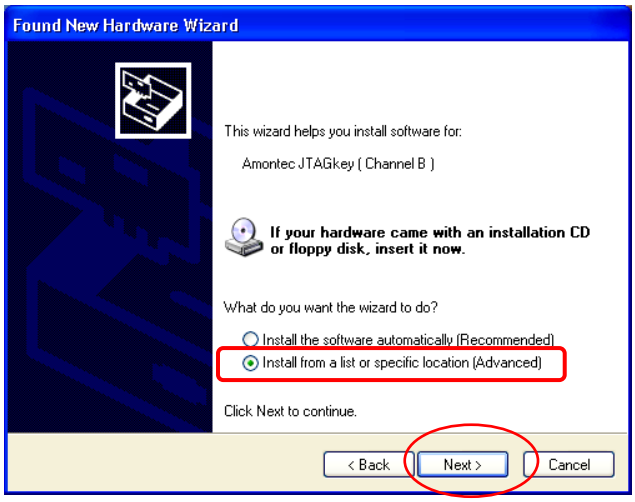


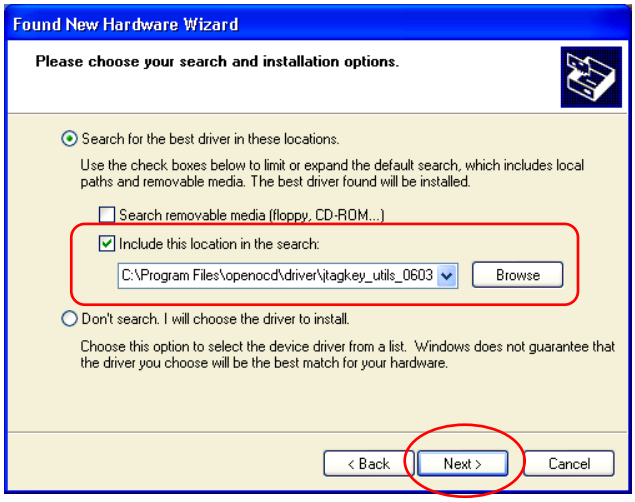
When the Channel A driver has completed, you will see the screen below indicating successful installation. Click “**Finish**” to exit the channel A installation as shown below.

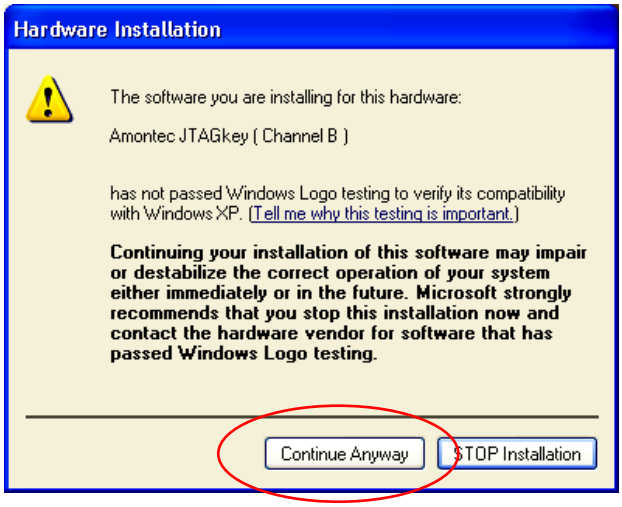


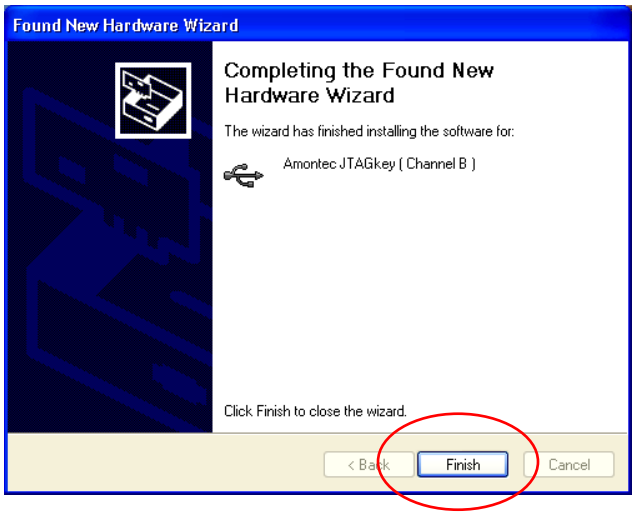
The JTAGkey is built around the FTDI FT2232C engine which has two channels. Exactly the same installation sequence is required for channel B. Follow the screens on this page in sequence, exactly like the channel A virtual device driver installation.



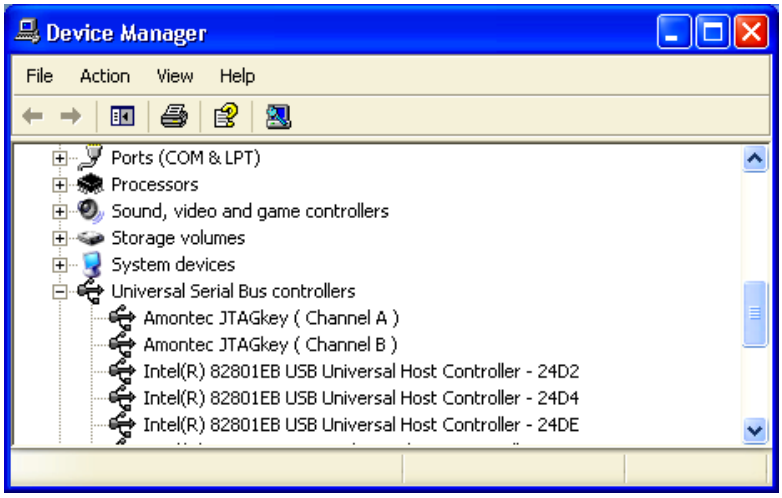






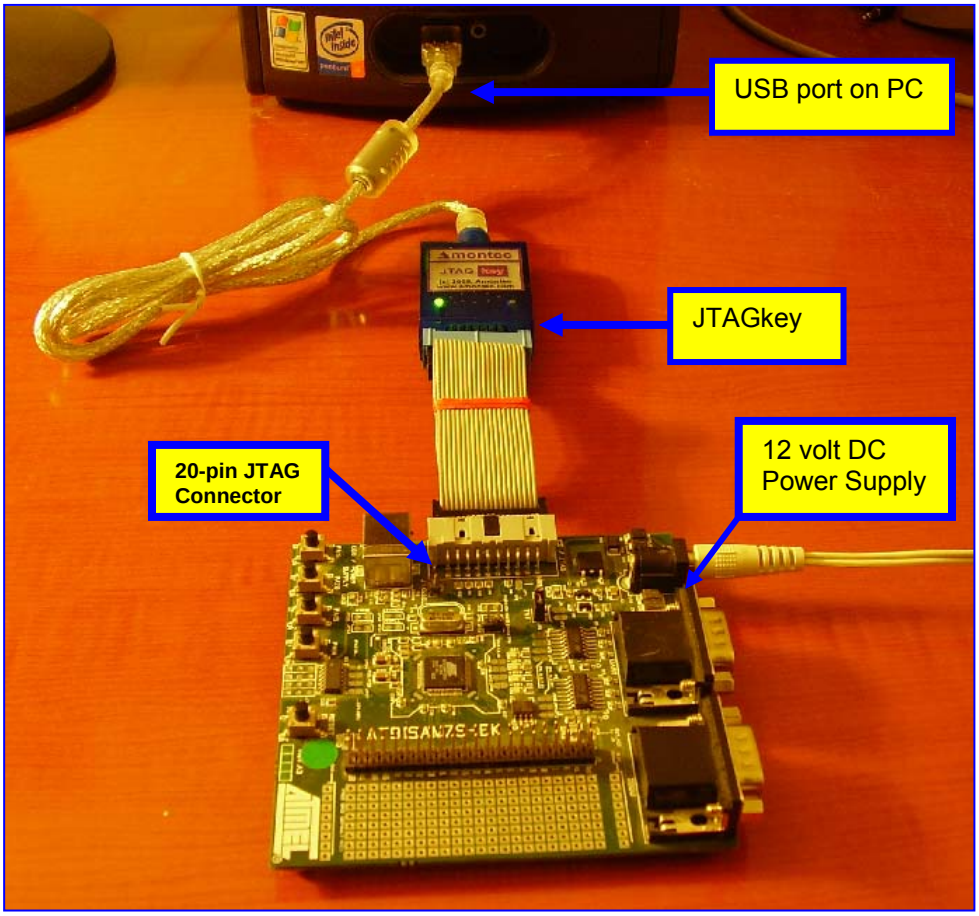


To be sure of successful installation of these JTAGkey virtual device drivers, use the Windows Start menu to look at the "**Control Panel – System – Hardware - Device Manager**", inspecting carefully the USB controllers. As can be seen below, the Amontec JTAGkey channel A and channel B USB ports are successfully installed.



Set Up the Hardware

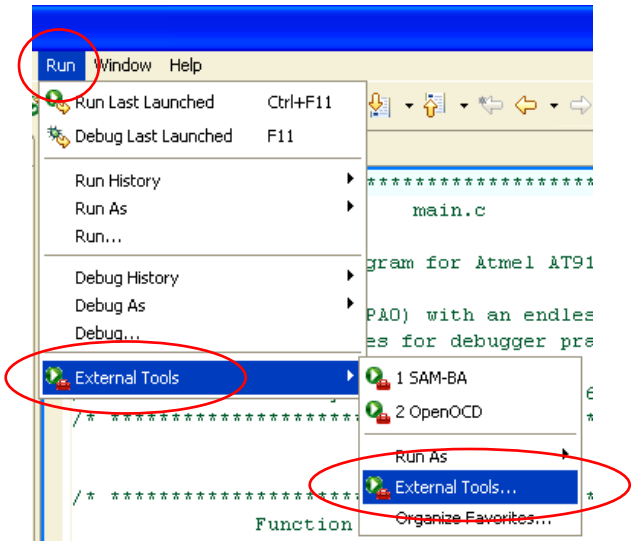
The hardware setup includes the Amontec JTAGkey plugged into the 20-pin JTAG header on the AT91SAM7S-EK target board and also into the PC's USB port. The JTAG does not supply board power, so in this example a 12-volt DC "wall wart" power supply is fitted to the power connector. If you have a spare USB port on your PC, you could use another USB cable to supply board power instead.



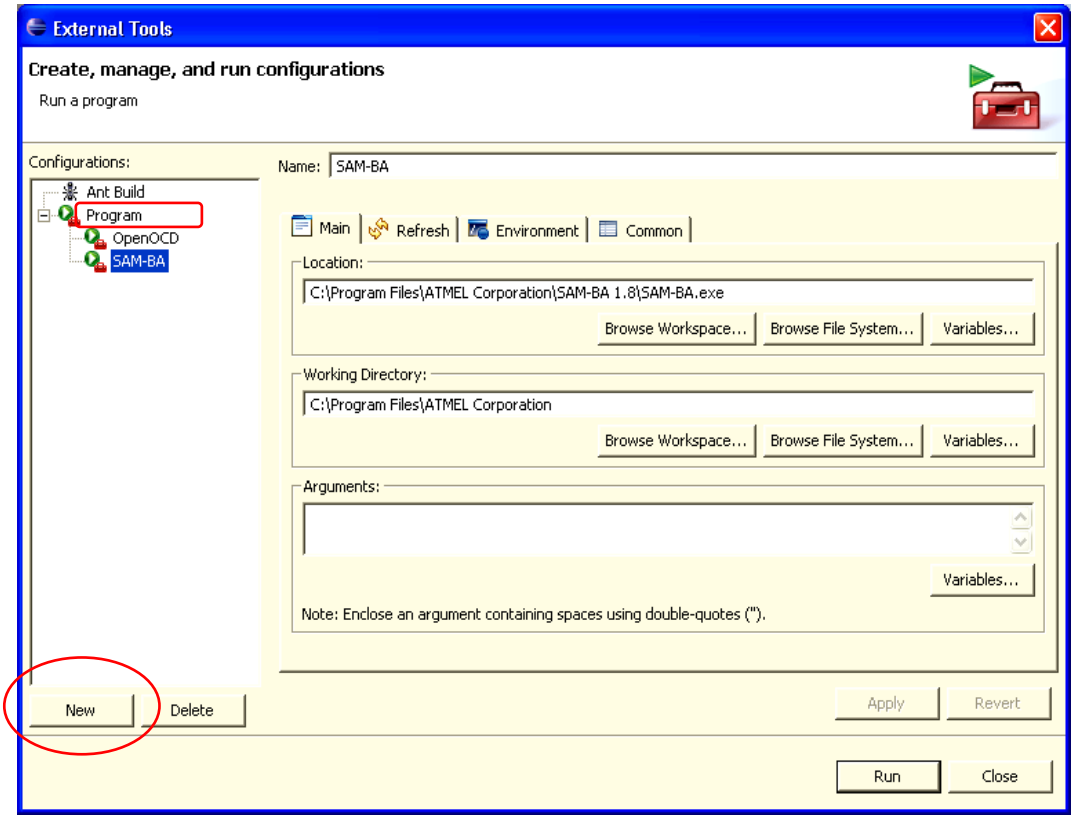
Install OpenOCD USB Version as an External Tool

A different OpenOCD executable is used for the USB JTAGkey. It must be installed as another "**external tool**" in the Eclipse RUN pull-down menu.

Click on the "**Run – External Tools – External Tools…**" pull-down menu as shown below.



The "**External Tools**" window will appear. Click on "**Program**" followed by "**New**".

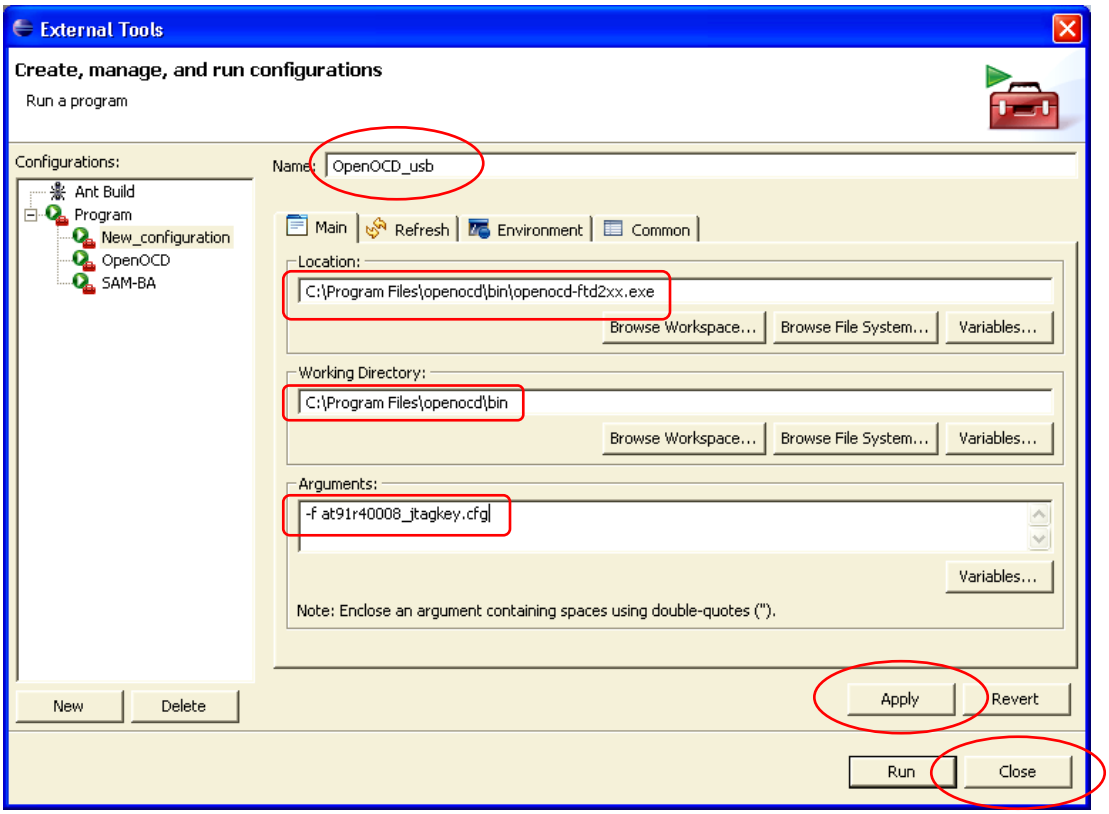


Fill out the New Configuration exactly as shown below.

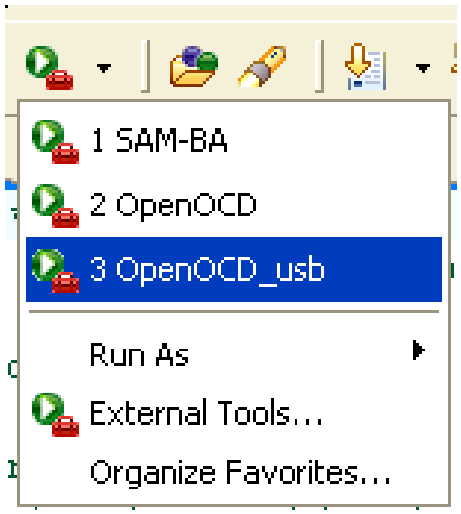
The name can be anything; we used "**OpenOCD_usb**" in this example. The location of the OpenOCD executable for the JTAGkey is in the folder "c:\Program Files\openocd\bin\" and the executable name is "**openocd-ftd2xx.exe**". Use the "**Browse File System…**" button to locate this file and place it into the Location: text box. Likewise, use the other "**Browse**" button to enter the folder name into the Working Directory: text box.

A single argument is needed; it is the configuration file "**at91r40008_jtagkey.cfg**" and it is entered with the switch –f in the Arguments: text window as shown below.

Click on "**Apply**" followed by "**Close**" to complete specification of this new External Tool.



Now use the techniques explained earlier in this tutorial to enter the External Tool "**OpenOCD_usb**" into the "**favorites**" list. Now when you use the "**External Tools**" toolbar button, there will be an additional choice of an **OpenOCD_usb** configured for the Amontec JTAGkey hardware.



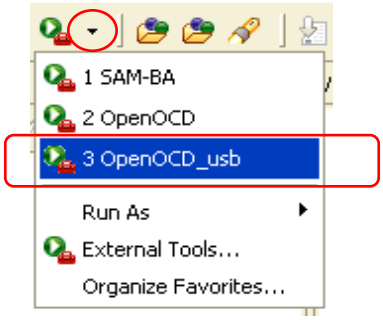
- **Debug the Application using the JTAGkey**

  There are no changes to the Eclipse debug launch configurations to use the JTAGkey interface. In the following example, we have started Eclipse and chosen the project “demo_at91sam7_blink_ram” which will build the application for execution entirely within RAM memory. Rebuild the project and switch to the Debug perspective.

Start OpenOCD_usb

Remember to cycle the target board power.

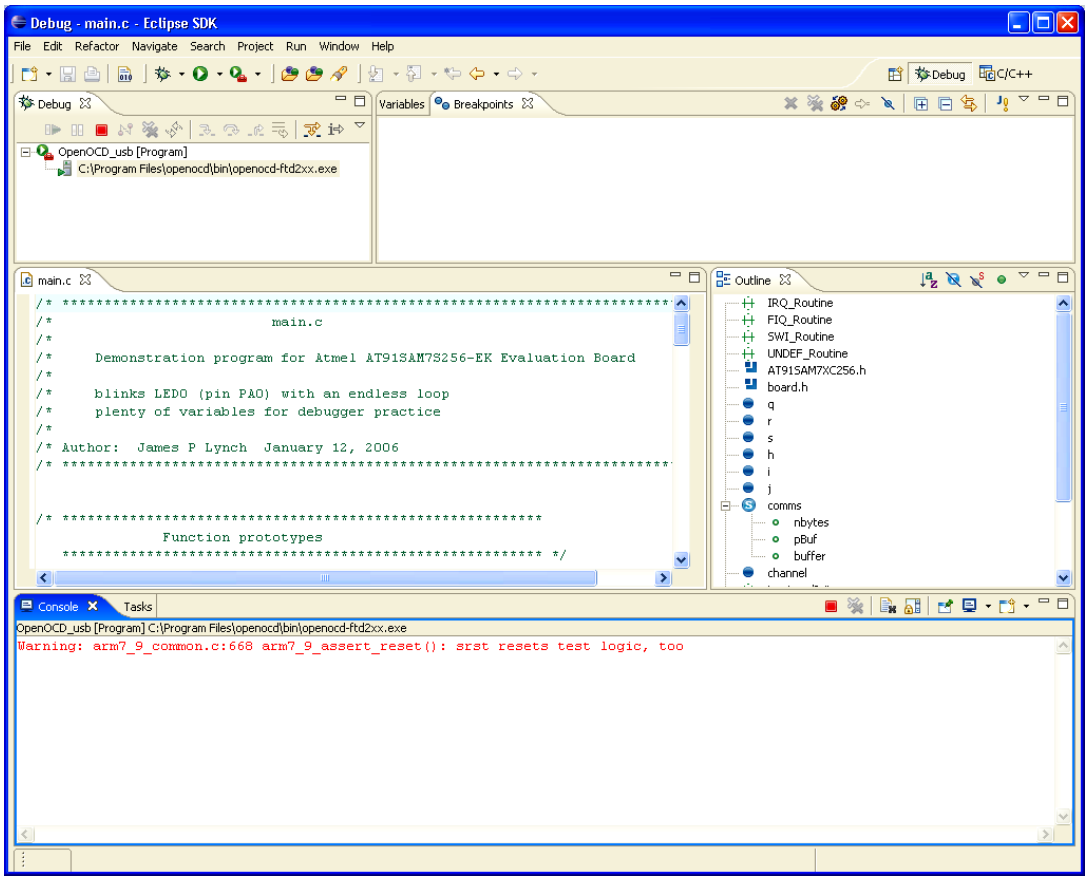
Click on the “**External Tools**” toolbar button’s down arrowhead and then select “**OpenOCD_usb**”.



In the Eclipse Debug Perspective window below, you can see that **OpenOCD_usb** has started up.

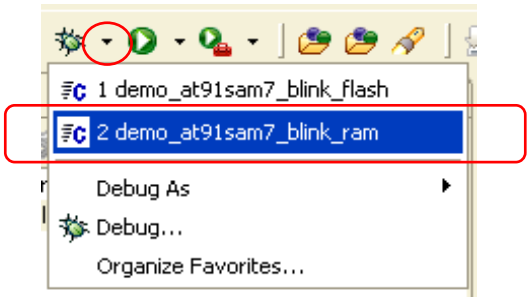
The “**Debug Control**” window on the upper left shows it running.

The “**Console View**” at the bottom only shows a minor warning – this is OK.

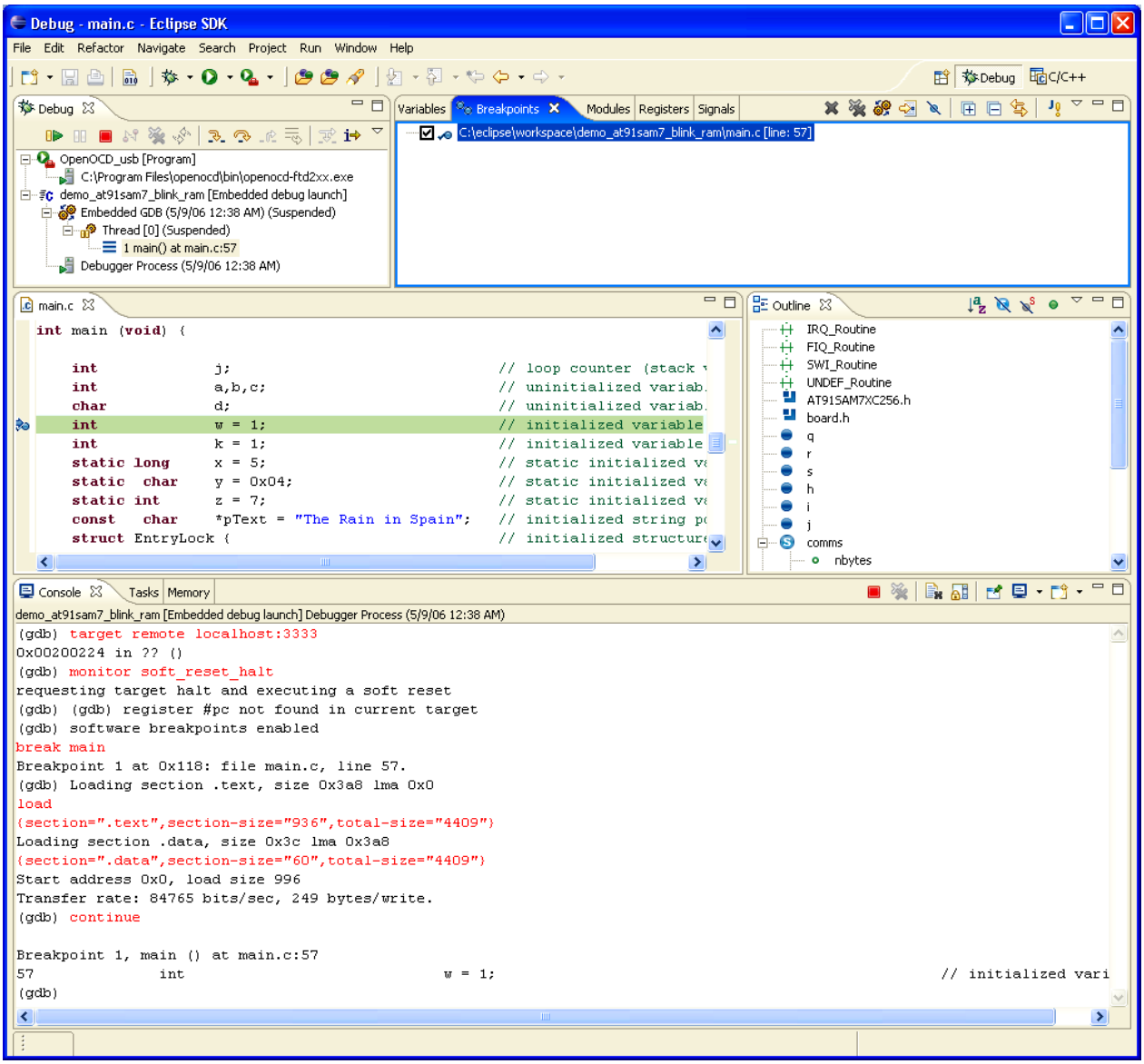


Start the Eclipse Debugger

Using the Debug Toolbar button's down arrowhead, click on the "**demo_at91sam7_blink_ram**" debug launch configuration. This is the very same one used earlier in the tutorial; it will work fine with the JTAGkey.



The Eclipse debugger will start up, under the auspices of the JTAGkey interface and run through the GDB startup commands until the debugger "stops at main()" as shown below.



Now you can run through all the debugger operations covered earlier in this tutorial. The operation of the debugger with a flash-based application is exactly the same as before also. Considering that modern desktop PCs and laptops are being manufactured without serial or parallel ports, a USB-based JTAG interface will soon be the only way to debug target boards.

Conclusions

Professional embedded software development packages from Rowley, IAR, Keil and ARM are complete, efficient, and easy-to-install and have telephone support if you encounter problems. For the professional programmer, they are worth the expense since “time is money”. Some of these companies offer “kick start” versions of their packages for free, albeit with some reduced functionality such as a 32K code limit, etc.

The Open Source tools described herein are an attractive alternative and are free. It’s obvious from the size of this tutorial that the acquisition and installation of Open Source tools is complex and time-consuming. The reader needs a high speed internet connection to download the various components and an afternoon’s time to install and test the lot.

Still, many thousands have managed successful application of Open Source tools for embedded software development. The GNU compilers are very close to the code efficiency of the professional compilers from Keil, IAR and ARM. The Eclipse and GNU Open Source tools bring the world of embedded software development to anyone on the planet that has imagination, skill and dedication but not the corporate bank account. Promoting the involvement of everyone in microprocessor development, not just an elite few, will allow us all to profit from their accomplishments.

About the Author

Jim Lynch lives in Grand Island, New York and is a software developer 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 grown 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, enjoys woodworking and hopes to write a book very soon that will teach students and hobbyists how to use these high-powered ARM microcontrollers. Lynch can be reached via e-mail at: lynch007@gmail.com

Appendix 1. **SOFTWARE COMPONENTS**

One common problem in setting up a software development system composed of disparate modules downloaded from multiple sources is ensuring that the various components will work harmoniously with each other.

To ensure that readers have a compatible set of software components, a zip file (**atmel_tutorial_components.zip**) has been prepared that includes components chosen by the author for their compatibility with each other.

The software components zip file may be downloaded from here.

www.address_to_be_determined.com

Everything the reader needs is in this zip file. The included components are:

Cygwin	includes the cygwin Ver 1.5.19-4 download and installer
Gnuarm ARM compiler suite	includes gcc_4.0.2, binutls_2.16.1, newlib_1/14.0, insight_6.4
Eclipse IDE	includes version Eclipse_SDK_3.1-win32
Zylin CDT for Embedded Applications	includes embeddedcdt-20050810 and zylincdt-20050810 plug-ins
Atmel SAM-BA	includes version 1.8 SAM-BA USB flash loader software
OpenOCD installer	includes 2006 Rev 55 build of OpenOCD plus installer and USB drivers
Source Code	set of example Eclipse C projects as described in the tutorial

Installing these components will create a development system that will compile, link and debug ARM applications. If you want to try to download the “latest and greatest” from the various web sites, you must research each intended component for its suitability. For instance, choosing the very latest Zylin build may require that you utilize the latest Eclipse “stream build”.

A safe approach is to build the ARM software development system using the included packages above, get it to work and become familiar with it. Then you can monitor the Eclipse, Zylin, GNUARM and OpenOCD web sites for new versions and choose at a later time if you want to upgrade.