Using Open Source Tools for AT91SAM7S Cross Development

Revision 2

Author:

James P. Lynch
Grand Island, New York, USA

October 8, 2006

TABLE OF CONTENTS

Introduction

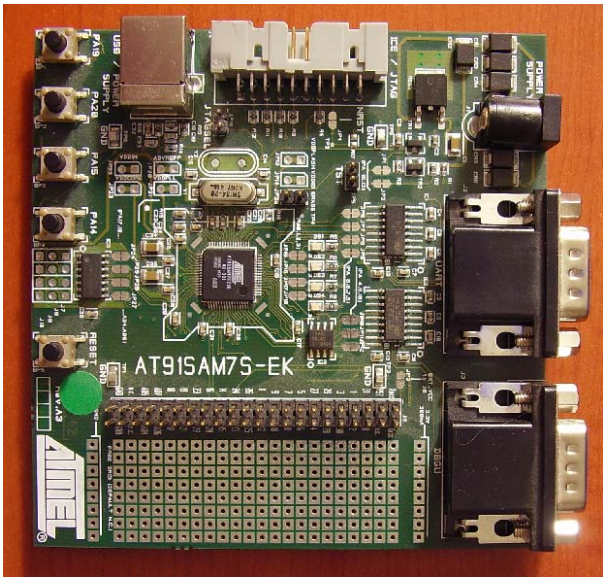
For those on a limited budget, use of open source tools to develop embedded software for the Atmel AT91SAM7S family of microcontrollers may be a very attractive approach. Professional software development packages from Keil, IAR, Rowley Associates, etc are convenient, easy to install, well-supported 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 hour's work. It does require a high-speed internet connection and a bit of patience.

Three major open source software tools make up the ARM cross development system. Eclipse provides the Integrated Development Environment (IDE) which includes a superior source code editor and symbolic debugger. YAGARTO provides a recent version of the GNU C/C++ compiler suite natively compiled for Windows. OpenOCD interfaces the Eclipse symbolic debugger with the JTAG port available on the Atmel ARM7 microcontrollers. The only hardware required is a $12.00 ARM-JTAG interface (wiggler) or a USB-based JTAG interface ($37.00 JTAGKey-Tiny or $69.95 ARM-USB-OCD) and an AT91SAM7S-EK evaluation board to serve as the hardware platform.

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.



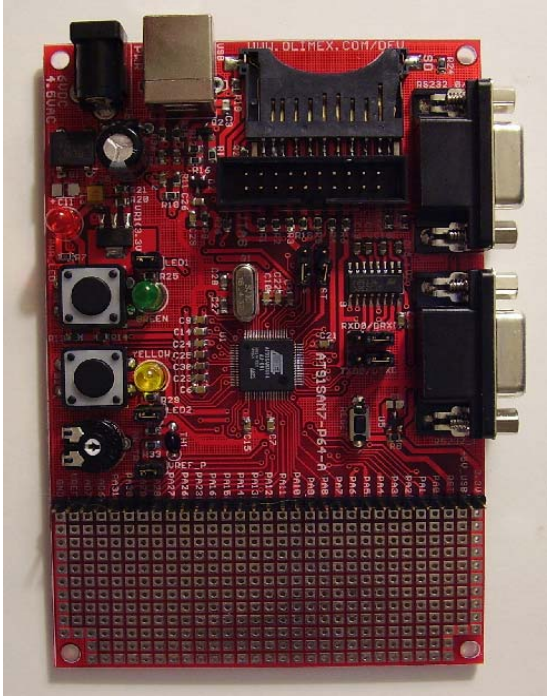
This board includes two serial ports, a USB port, an Atmel Crypto memory, JTAG connector, four buffered analog inputs, four pushbuttons, four LEDs and a prototyping area.

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 SAM7-P256 shown directly below (Olimex SAM7-P64 board shown, SAM7-P256 board is very similar).



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 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 Olimex, Spark Fun Electronics and Microcontrollershop; it retails for $69.95

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 since the Olimex board uses different I/O ports for the LEDs. See Appendix 1 for additional instructions.

Open Source Tools Required

To build this ARM cross development tool chain, we need the following components:

- **Eclipse IDE version 3.2**
- **Eclipse CDT 3.1 Plug-in for C++/C Development (Zylin custom version)**
- **Native GNU C++/C Compiler suite for ARM Targets**
- **OpenOCD version 93 or later for JTAG debugging**

There are several places on the web where we can download all the above tools from a single source with convenient installers.

YAGARTO

The **YAGARTO** ARM Cross Development Package was assembled by Michael Fischer of Germany. It includes the latest Eclipse release 3.2 and the Zylin-modified CDT (C/C++ Development Toolkit). The ARM compiler tool chain runs as a Windows native application with no Cygwin DLL required. Michael has also modified the GDB debugger to improve its performance in an embedded debug environment. Rounding out the package is the latest version of OpenOCD (the JTAG debugger). **YAGARTO** is

packaged as three downloads with a fool-proof installer for each. Michael's **YAGARTO** web site is non-commercial with no affiliation with any manufacturer.

Yagarto may be downloaded from here: www.yagarto.de

SDK4ARM

The Swiss engineering company Amontec has crafted a version of YAGARTO, made some improvements and renamed the package **SDK4ARM**. This system also includes three download packages with a fool-proof installer for each. When Eclipse is run, there is a short commercial "splash screen" but it's hardly intrusive.

SDK4ARM can be downloaded from a very fast server here:
http://www.amontec.com/sdk4arm.shtml

ARM GCC for Windows Dummies

The Bulgarian hardware manufacturer Olimex packages the "**ARM GCC for Windows Dummies**" cross development system as a CDROM. To get it, you have to purchase the ARM-USB-OCD JTAG debugger for $69.00. It uses the WinARM tool chain, Eclipse, CDT and OpenOCD. Its major advantage is that it uses a single fool proof installer to install everything.

ARM GCC for Windows Dummies can be ordered from the following web site:
http://www.olimex.com/dev/index.html

WinARM

WinARM was developed by Martin Thomas and is a very popular downloadable ARM cross development tool chain. **WinARM** runs as a Windows native application. Thomas has the best collection of sample programs also. The only downside is that the **WinARM** package, by itself, does not include Eclipse.

WinARM may be downloaded from the following web site:
http://gandalf.arubi.uni-kl.de/avr_projects/arm_projects/#winarm

The author has tried every one of the above tool chains and they all work very well indeed. To be fair, every one of them is based on the Free Software Foundation GNU tool chain and you can be sure of the compiler's efficacy and support.

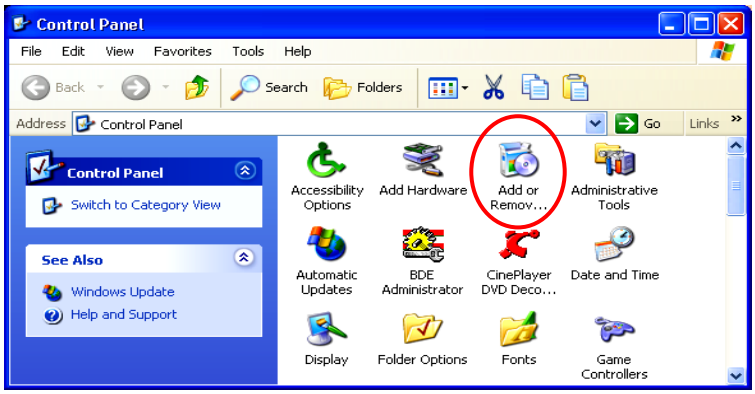
For this tutorial, the author has taken the non-political approach: demonstrating the **YAGARTO** tool chain. **YAGARTO** is complete, installs quickly and easily and is not associated with any hardware vendor. Michael Fischer keeps it up-to-date with the latest revisions of Eclipse, the GNU tool chain and OpenOCD.

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

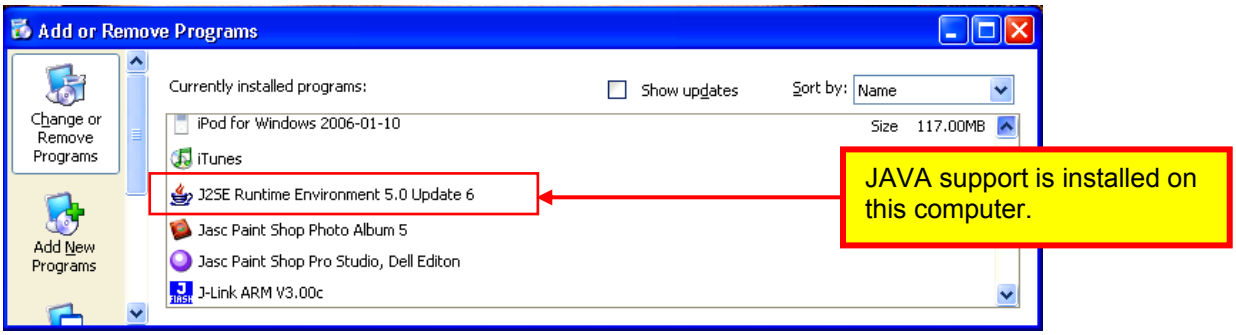
Check for JAVA Support

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 runtime support 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.



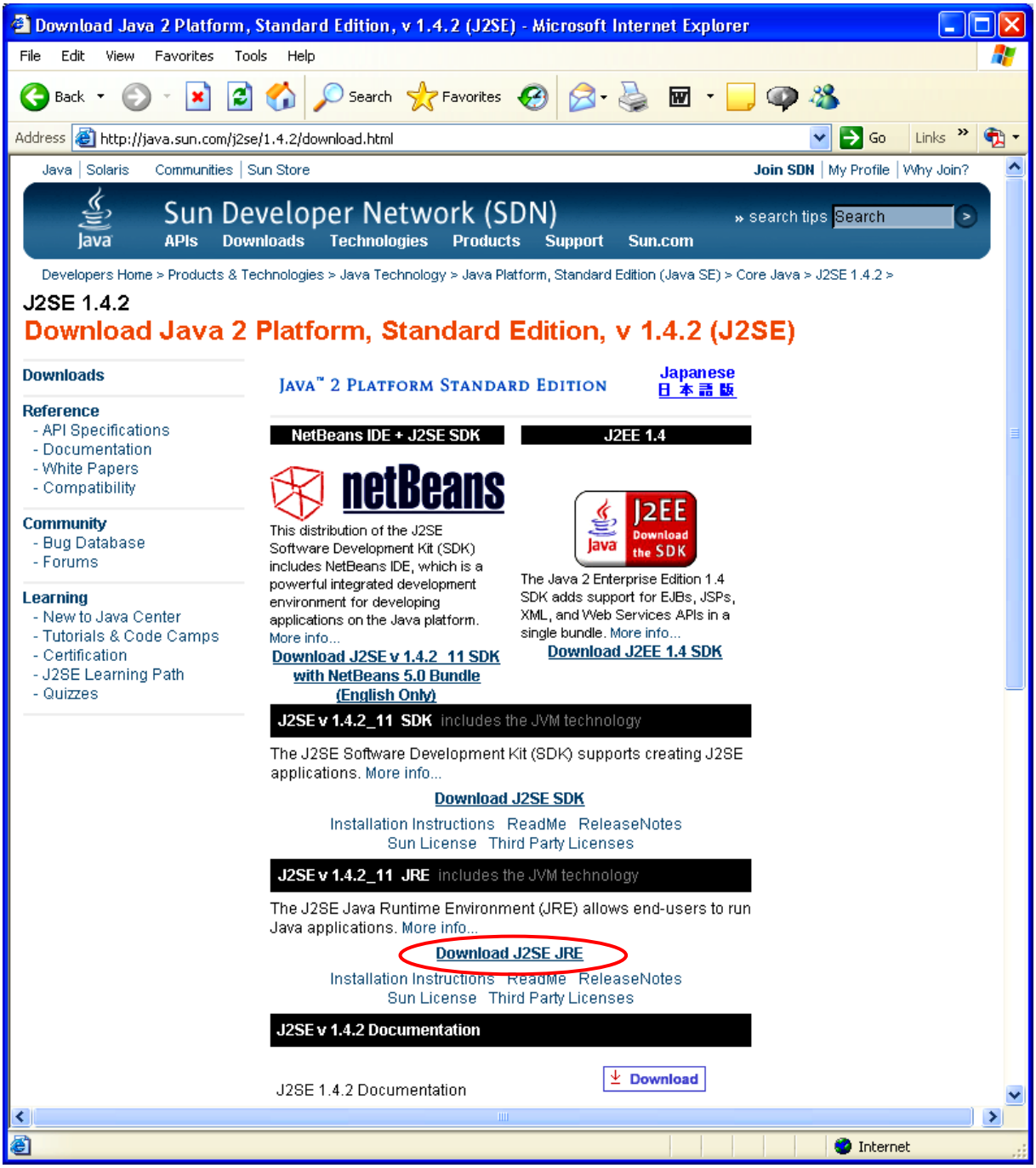
To install the JAVA Runtime Environment, go to the SUN web site and download it.

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

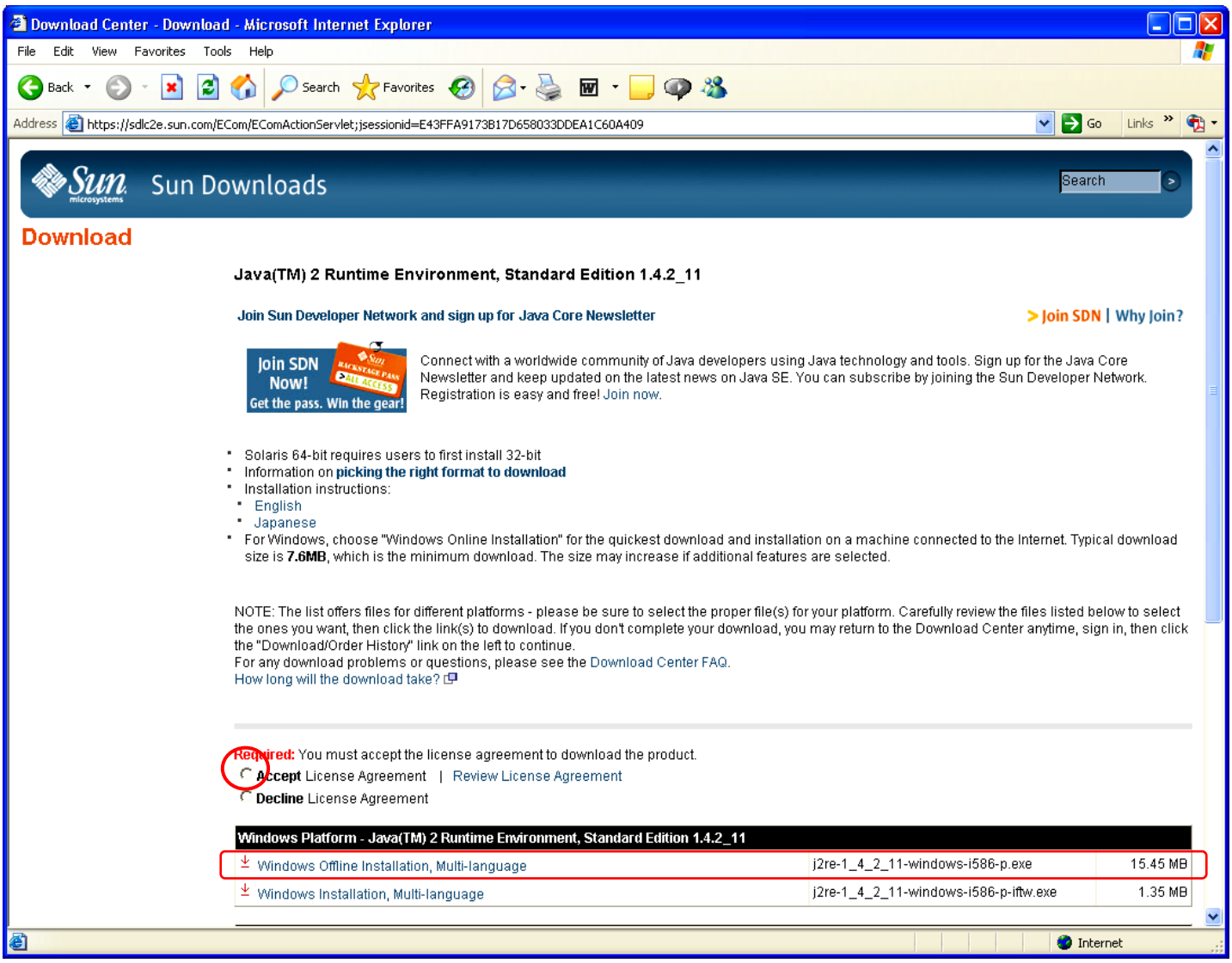
Java is not Open Source but Sun has always allowed free downloads of the JAVA package. If you build a commercial product that uses JAVA and the Java Virtual Machine, they may expect royalties. This would not typically be the case in using Eclipse in a cross-development environment.

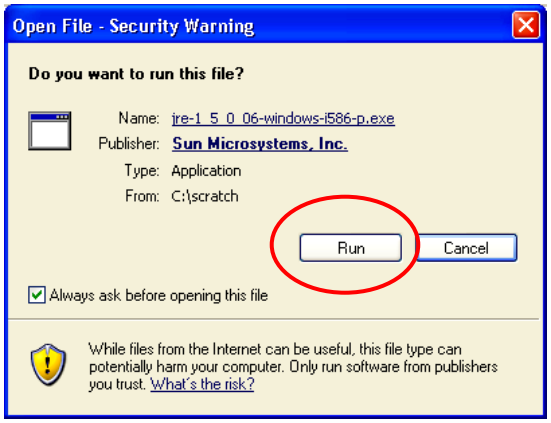
The Sun JAVA web site is very dynamic so don't be surprised if the JAVA run time download screens differ slightly from this tutorial.

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



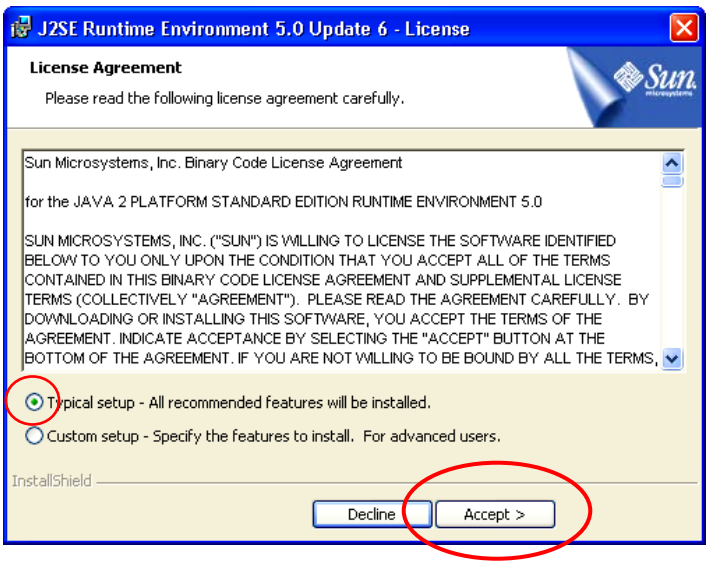
In the next download screen, shown below, click the radio button "**Accept License Agreement**" and then click on "**Windows Offline Installation, Multi-language**".



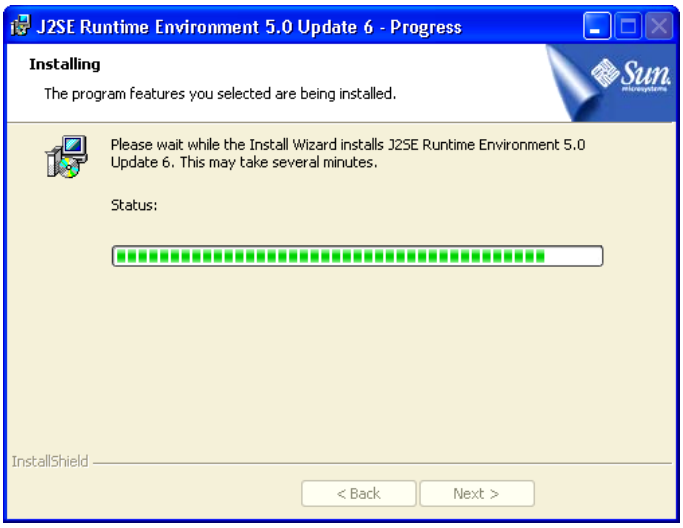


Now the Sun JAVA runtime installation engine will start. Click "**Run**" to start the installer.

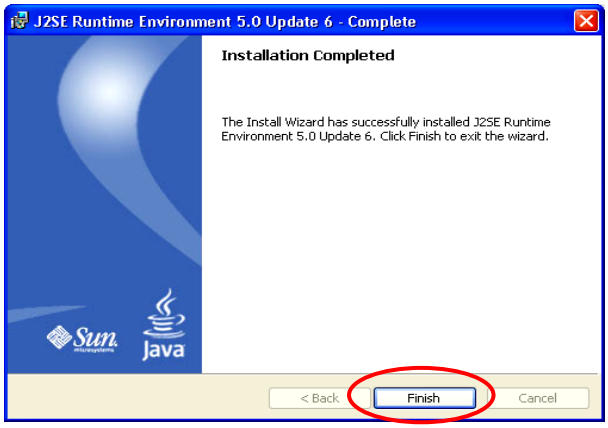
Click on the "**Typical Setup**" radio button and then accept the license terms. JAVA is free; if you build some fabulous product with JAVA integrated into it, then you may have to pay Sun royalties.



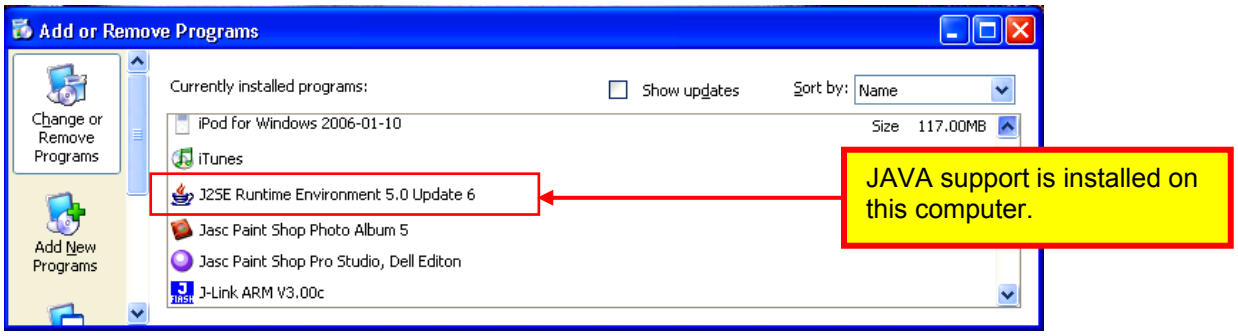
A series of installation progress screens will appear. Installation only takes a couple of minutes.



When the JAVA runtime installation completes, click on "**Finish**" to exit the installer.



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



Downloading the YAGARTO Tool Chain

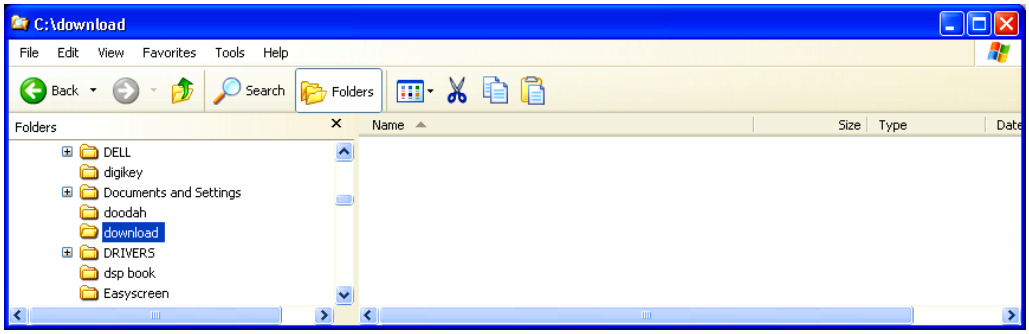
Michael Fischer of Lohfelden, Germany has put together a native version of the GNU compiler tool chain for ARM targets based on MinGW (Minimalist GNU for Windows) and called it YAGARTO (**Y**ET **A**NOTHER **G**NU **AR**M **TO**OL CHAIN). The compiler suite does not require the Cygwin package and is therefore a bit more efficient running in a Windows environment.

Eclipse, a superior open-source Integrated Development Environment (IDE), coupled with the C Development Toolkit (CDT) plug-in provides an editor and source code debugger.

The OpenOCD JTAG debugger, developed by German student Dominic Rath, interfaces the Eclipse GDB source code debugger with the AT91SAM7S JTAG port. OpenOCD supports run/stop control, memory and register inspection, software and hardware breakpoints and can also be used to program the AT91SAM7S internal FLASH memory.

Each of these three components (compiler, Eclipse IDE and OpenOCD) are downloaded separately and each has its own automatic installer that is fool-proof and convenient.

Before accessing the YAGARTO web site, use Windows Explorer to create an empty folder to hold the three forthcoming downloads. Note below that a folder **c:\download** has been created for this purpose.



Michael Fischer's YAGARTO web site can be accessed at the following link.

http://www.yagarto.de

The YAGARTO web site should look something like this, shown below.

YAGARTO Yet another GNU ARM toolchain

HOME | HOW TO | EXAMPLES | PROJECTS | FAQ | IMPRINT

HOME

Why?
Download
How to?
Support
License information
Note

Why another GNU ARM toolchain?

Initially I was searching for a toolchain with the following features:

- not based on Cygwin
- works with Eclipse

I found some native Windows toolchains based on MinGW, but the GDB of these toolchains doesn't work properly under Eclipse. That's why I decide to create a new toolchain suited for my requirements. YAGARTO was born...

YAGARTO is divided in three packages with the following components:

- Open On-Chip Debugger
- Binutils, Newlib, GCC compiler, and the Insight debugger
- Eclipse Platform Runtime Binary, Eclipse CDT and CDT plugin for the GDB embedded debugging.

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

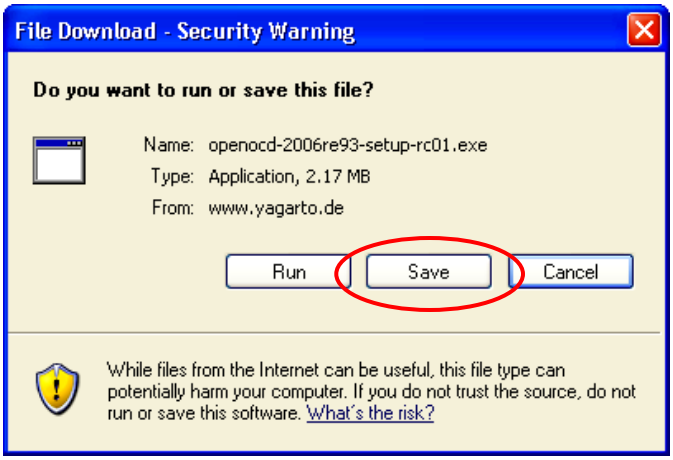
Scroll down the YAGARTO web site until you see the three download components displayed, as shown below.

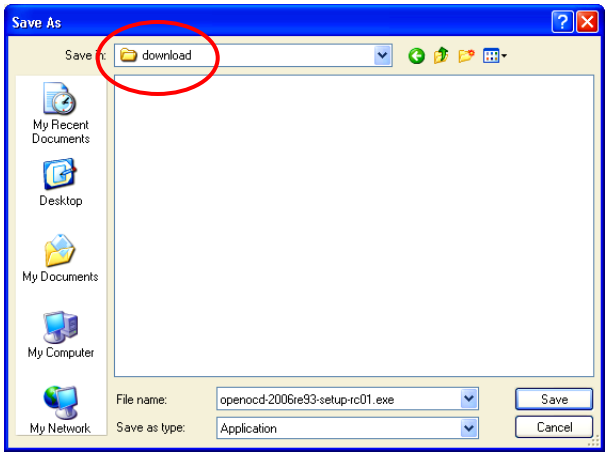
Download

The three packages of YAGARTO can be found here:

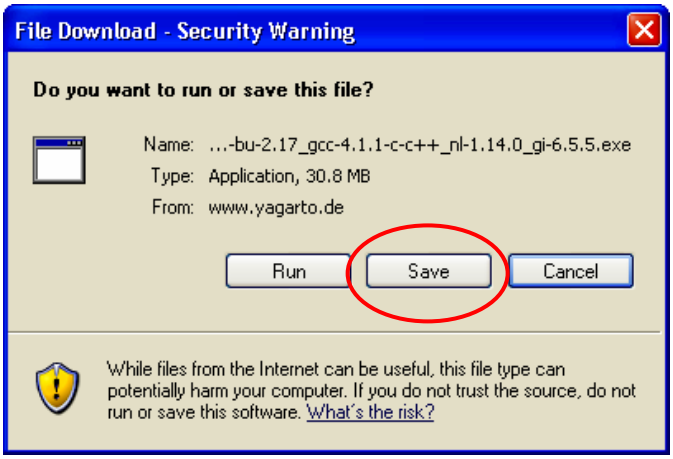
Package	Version	Last Version
Open On-Chip Debugger (2.17 MB)	r93-rc01	04.09.2006
YAGARTO GNU ARM toolchain (32 MB) (md5sum: a6e3882b582ffcc563e6c7800f186afa)	Binutils-2.17 Newlib-1.14.0 GCC-4.1.1 Insight-6.5.5.20060612	First version
Intergrated Development Environment (45 MB) (md5sum: ac100b653d93efbb4ad1a0e71b263ce4)	Eclipse 3.2 Zylin CDT 20060609 Zylin plugin 20060707	First version

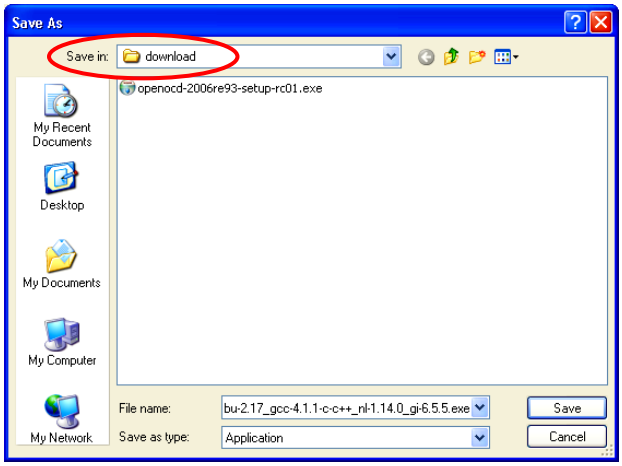
Click on the "**Open On-Chip Debugger**" above. Select "**Save**" and then specify the empty folder **c:\download** as the "**Save As**" target. The OpenOCD component will download in just a few seconds.



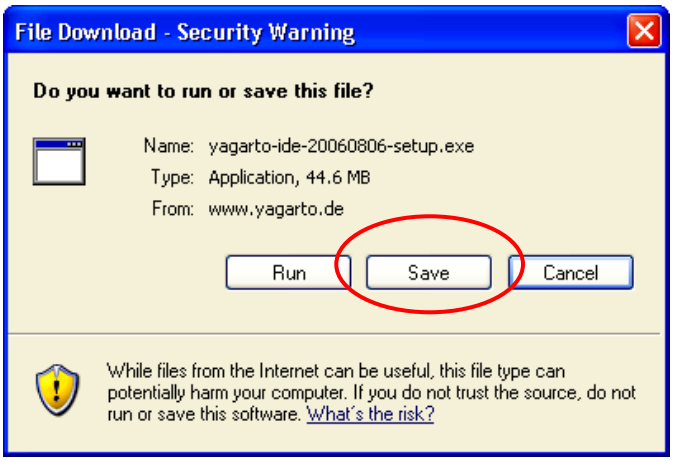


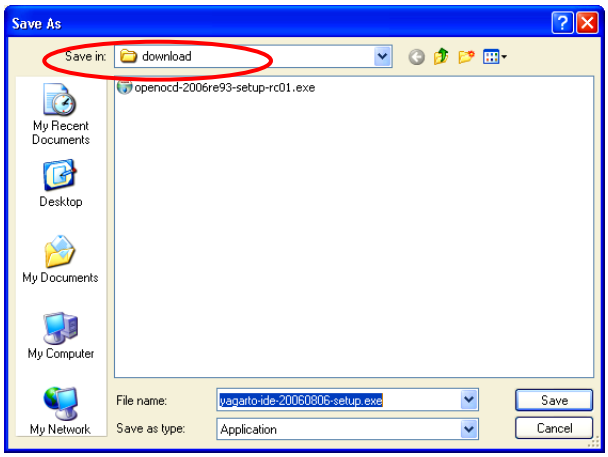
Click on the "**YAGARTO GNU ARM toolchain**" link above. Select "**Save**" and then specify the empty folder **c:\download** as the "**Save As**" target. The YAGARTO GNU ARM Tool chain component is 32 Mb and will download in a few minutes.



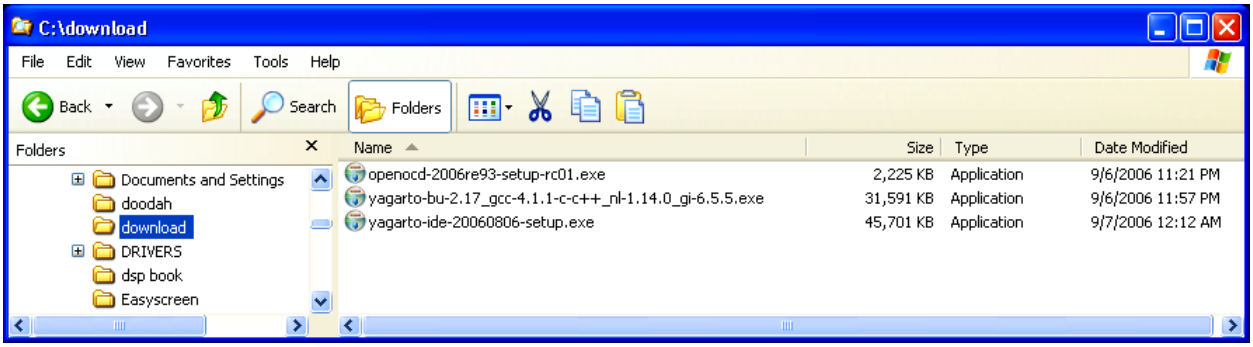


Finally, click on the "**Integrated Development Environment**" link below. Select "**Save**" and then specify the empty folder **c:\download** as the "**Save As**" target. The Eclipse IDE component is 45 Mb and will download in a few minutes.





When the three downloads have competed, the **c:\download** folder should show the following:



There are three files in the c:\download folder:

Openocd-2006re93-setup-rc01.exe	Installer for OpenOCD
Yagarto-bu-2.17_gcc-4.1.1-c-c++_nl-1.14.0_gi-6.5.5.exe	Installer for GNU compiler suite for ARM
Yagarto-ide-20060806-setup.exe	Installer for Eclipse IDE

Note to Readers:

Michael Fischer is constantly improving the YAGARTO package. If you get a newer version when you download YAGARTO, rest assured that Michael has made sure that all the components work harmoniously together.

In a few places in this tutorial you will find a folder name that has a "revision number" in it. For example, the OpenOCD executable and configuration files for this revision are stored in this folder: "**c:\Program Files\openocd-2006re93\bin**".

If Michael has posted a newer version, the folder name may change to "**c:\Program Files\openocd-2006re100\bin**", as an example.

We'll try to indicate throughout the tutorial those places where you will need to adjust the folder name to accommodate the new revision.

We are now just a few minutes away from having a complete Eclipse-based cross development system. The strategy is to double-click on each one of the above files in succession and then take the defaults on every installation screen.

Install OpenOCD

Eclipse/CDT has a fabulous graphical source code 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.

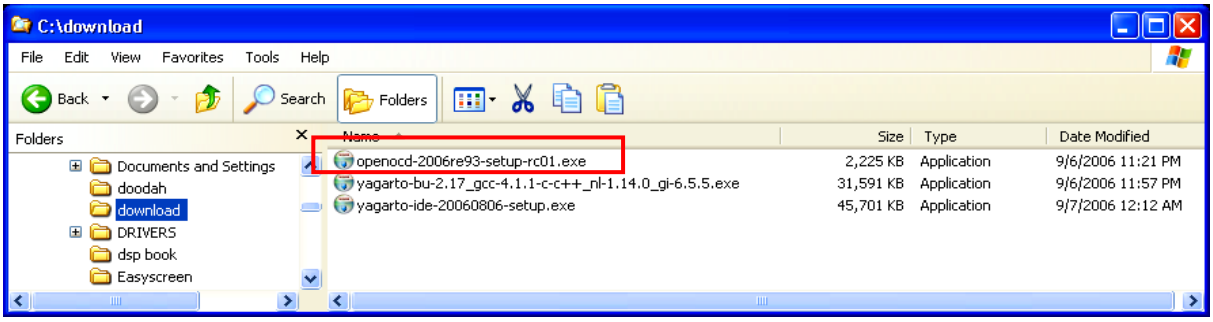
To the rescue is German college student Dominic Rath who developed an open source ARM JTAG debugger as his diploma thesis at the University of Applied Sciences, FH-Augsburg in Bavaria. Dominic's thesis can be found here: http://openocd.berlios.de/thesis.pdf

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

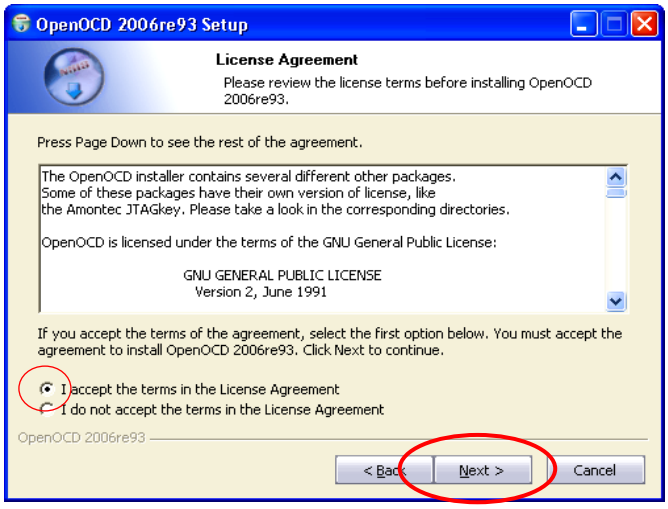
Finally, Dominic participates in the OpenOCD message board at the SparkFun site here: http://www.sparkfun.com/cgi-bin/phpbb/viewforum.php?f=18&sid=23d9b9ddd0df85507af924d6abdb54c7

OpenOCD can be used with the inexpensive "wiggler" JTAG device as well as the USB JTAG devices such as the Amontec JTAGKey, the Olimex ARM-USB-OCD and others coming on the market.

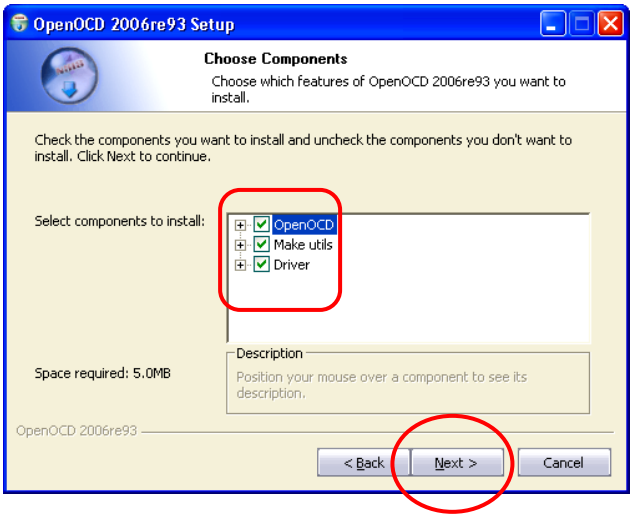
Double-click on the file **Openocd-2006re93-setup-rc01.exe** to start the OpenOCD installer.

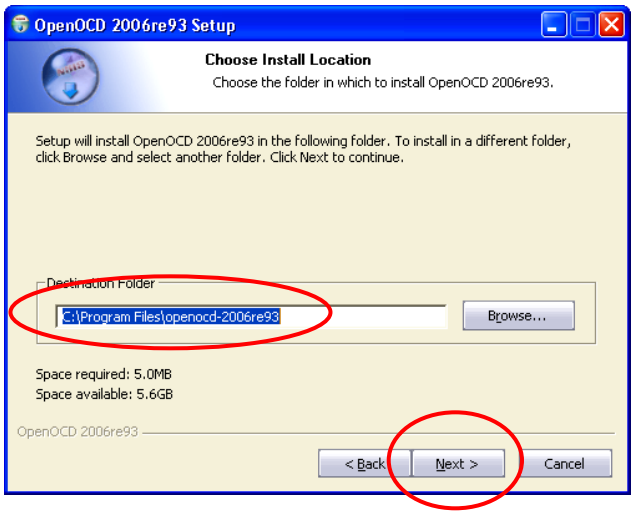


In the "**Welcome"** screen below on the left, click the "**Next**" button. The next screen is a standard GNU license agreement; click the top radio button to accept the License Agreement and click the "**Next**" button to continue.



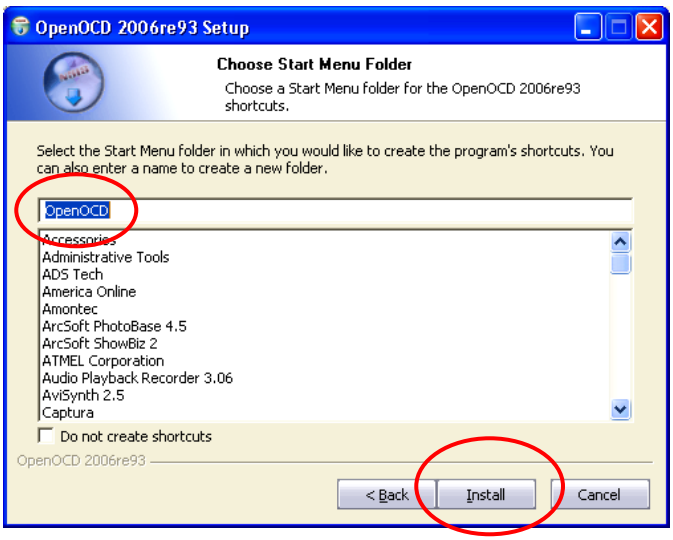
In the "**Choose Components**" screen shown below on the left, select all three components (**OpenOCD, Make Utils and Driver**). Click "**Next**" to continue. On the "**Choose Install Location**" screen below on the right, take the default location "**c:\Program Files\openocd-2006re93**" and click "**Next**" to continue.

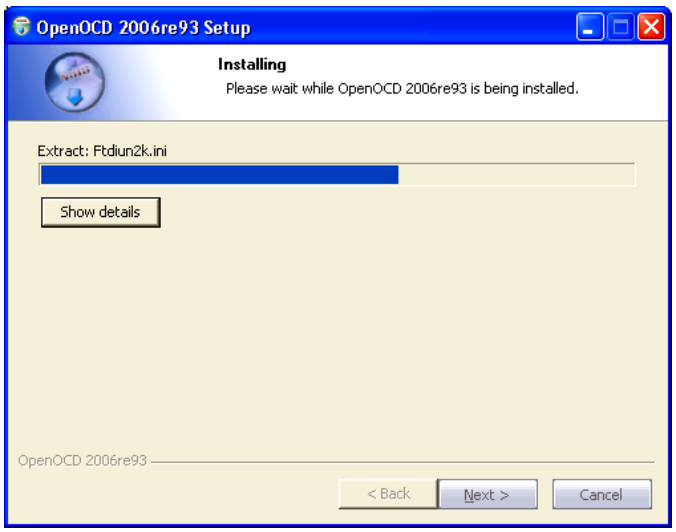




Take the default in the "**Choose Start Menu Folder**" screen shown below left. The OpenOCD debugger will be normally called from within Eclipse, so execution from the Start menu would be rare. You could choose the checkbox "Do not Create Shortcuts" if desired. Click "**Next**" to take the default and continue.

OpenOCD installs very fast (less than a minute) as shown in the "**Installing**" screen below on the right.

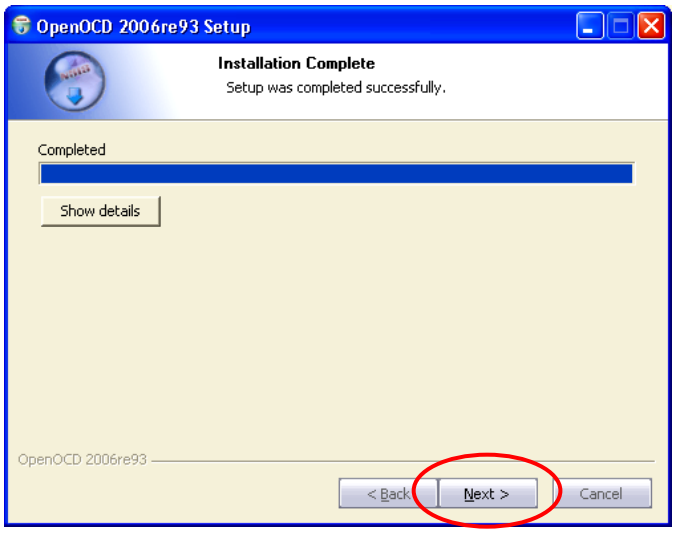


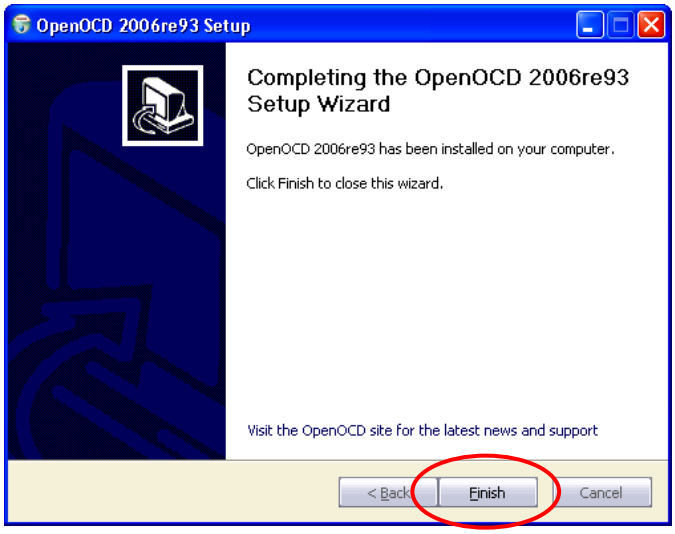


Click "**Next**" when the installation completes, as shown in the screen below on the left.

Click "**Finish**", as shown on the screen below to the right, to terminate the OpenOCD installer.

Make a mental note that the installer has placed all OpenOCD components in the following folder: **c:\Program\Files\openocd-2006re93\.** If your download includes a more recent revision of OpenOCD, make a mental note of the folder address – we will use it later in the tutorial.

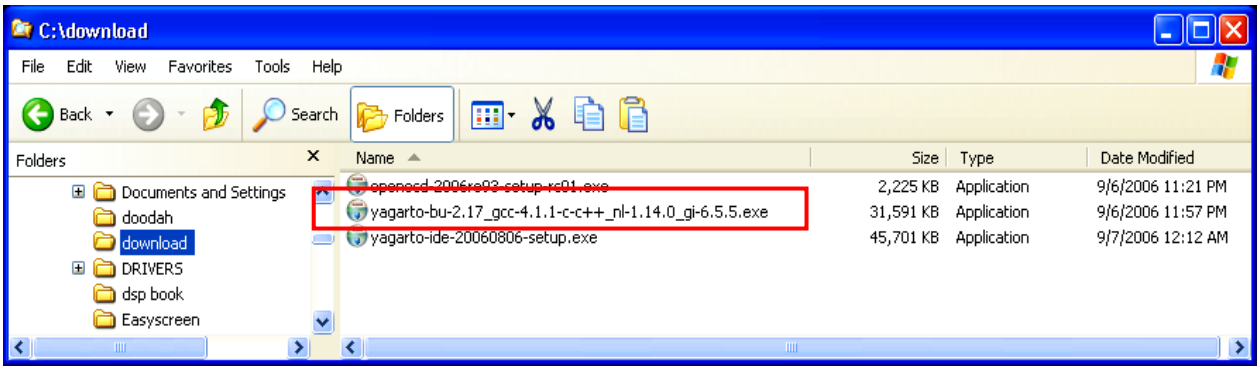




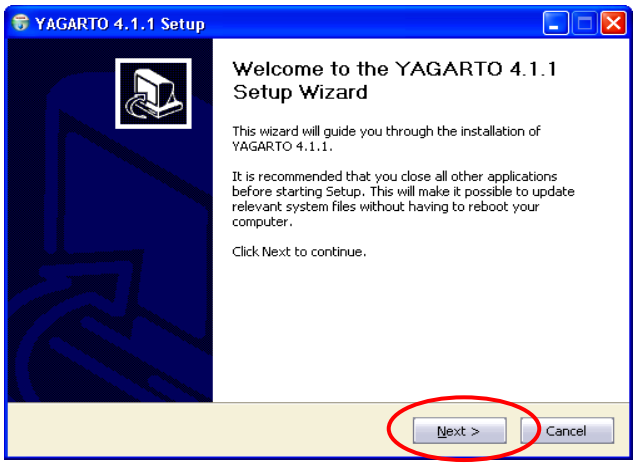
Install YAGARTO Tool Chain

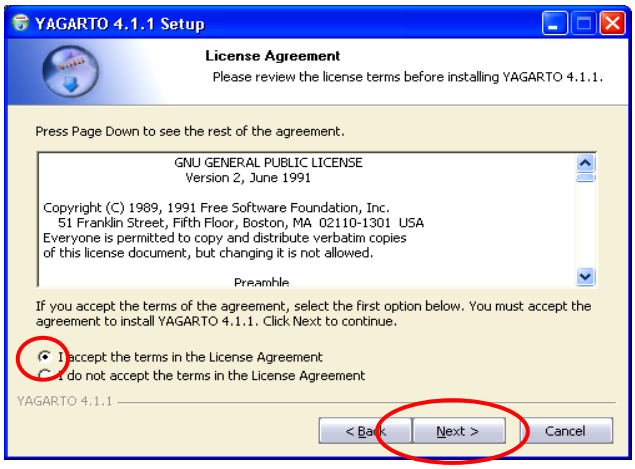
There are a number of pre-built GNU ARM compiler toolsets available on the web and they are all very good. For this tutorial, we will be using the **YAGARTO** pre-built ARM compiler tool suite developed by Michael Fischer of Lohfelden, Germany. Michael's version of the GNU compiler toolset for ARM has been natively compiled for the Intel/Windows platform; therefore the Cygwin utilities are not needed. This makes the compiler run faster and simplifies the installation. Michael has also performed some tweaks on the included GNU GDB debugger to make it perform better in the Eclipse environment.

Double-click on the file **Yagarto-bu-2.17_gcc-4.1.1-c-c++_nl-1.14.0_gi-6.5.5.exe** to start the YAGARTO tool chain installer.

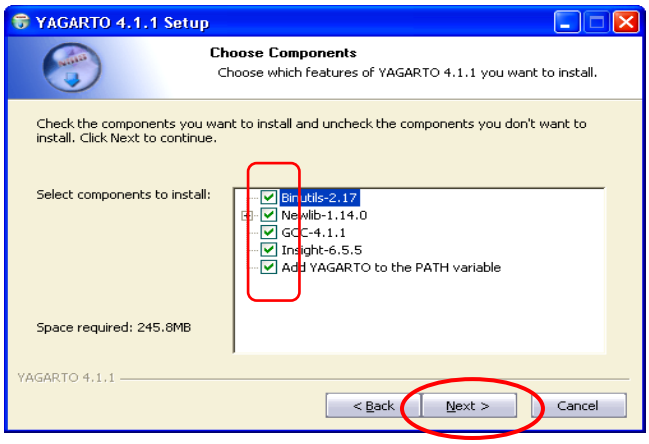


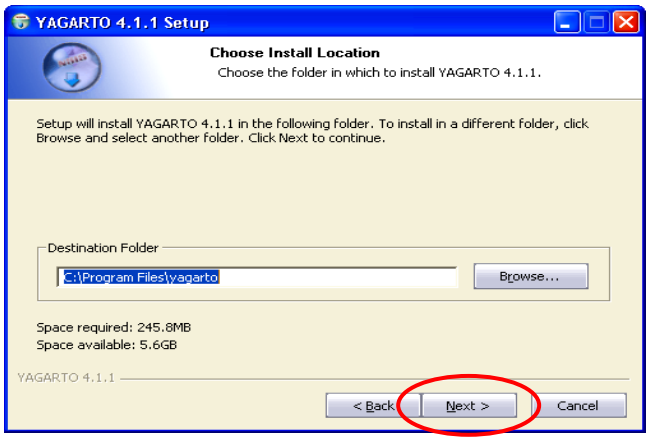
In the “**Welcome**” screen shown on the left below, click on “**Next**” to continue. Click the ”**I Accept …**” radio button on the “**License Agreement**” screen below on the right and then click “**Next**” to continue.



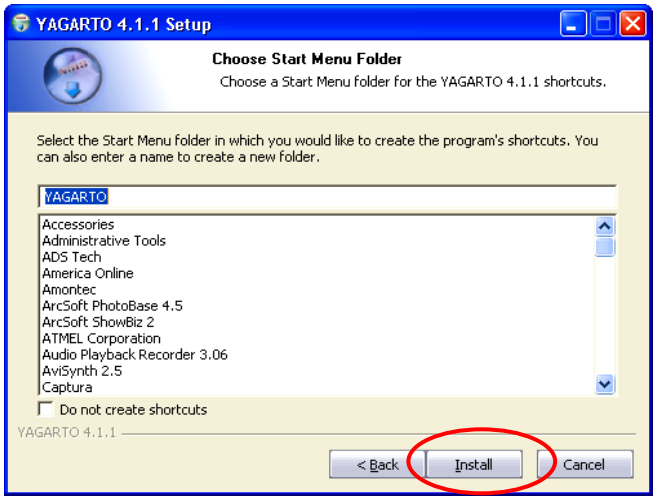


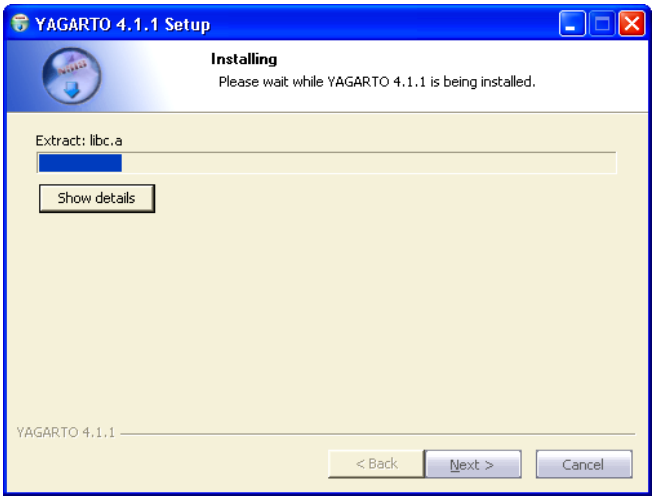
In the “**Choose Components**” screen on the left below, take all the defaults by simply clicking “**Next**” to continue. Note that this installs the Insight debugger that we will not use, but no harm is done including it. On the “**Choose Install Location**” screen on the right below, take the default again by clicking “**Next**” to continue.





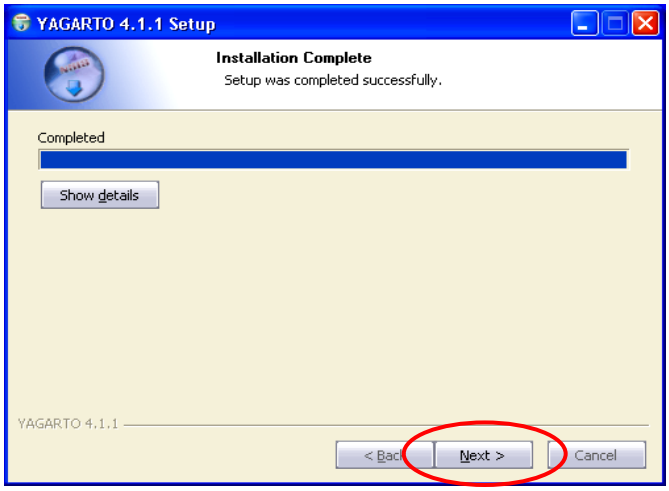
Click “**Install**” on the “**Choose Start Menu Folder**” shown below left and the YAGARTO tool chain installer will commence. This installation takes several minutes.

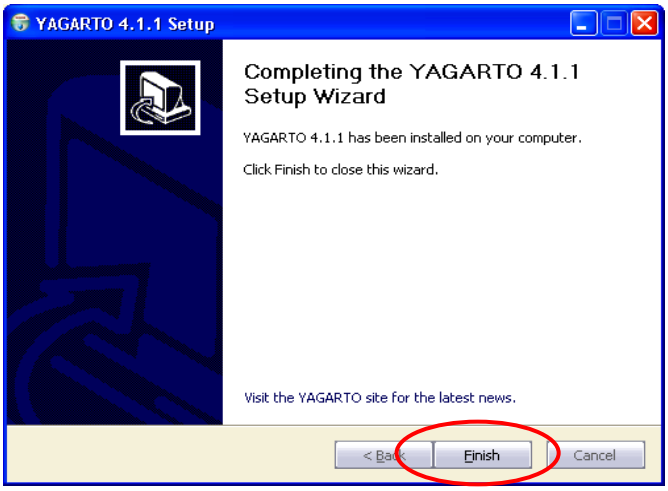




When tool chain installation completes, click "**Next**" as shown below on the left followed by clicking "**Finish**" on YAGARTO completion screen shown on the right below. This will terminate the YAGARTO installer. Make a mental note that the YAGARTO compiler tool chain is installed in the following folder:

C:\Program Files\Yagarto





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.

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.

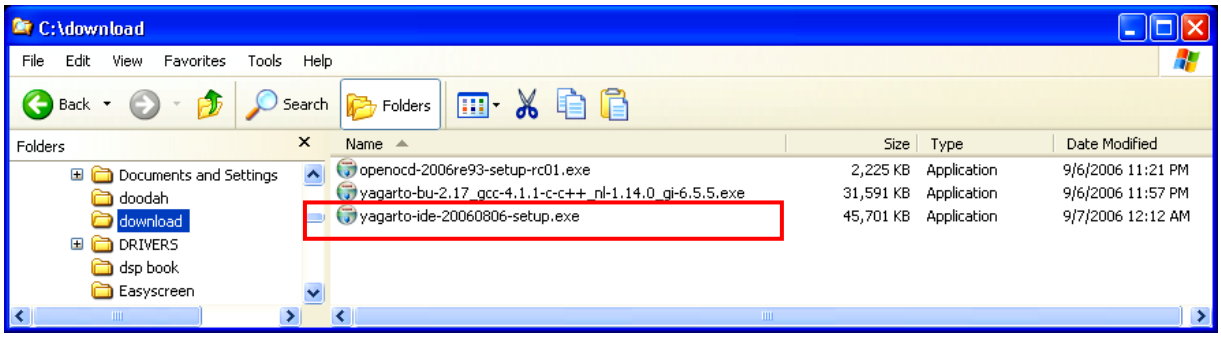
Be sure to visit the Eclipse web site: **www.eclipse.org**

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 **Eclipse/CDT** has had difficulties working with remote debuggers. Oyvind Harboe and 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.

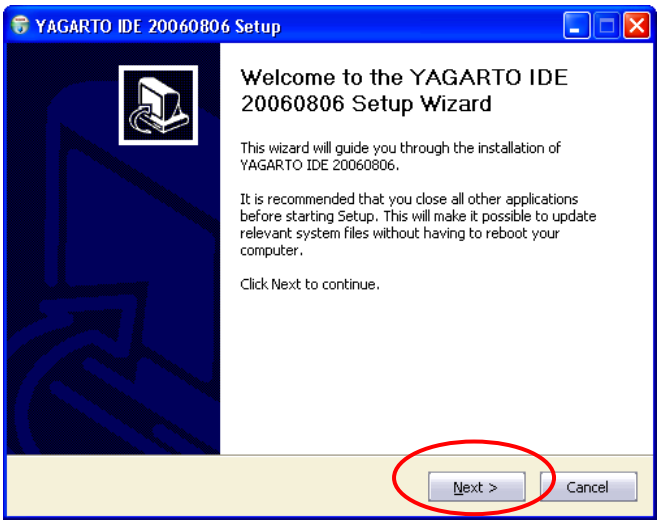
The only proviso is that we must select a version of Eclipse compatible with the Zylin **CDT** plug-in. Rest assured that the Zylin **CDT** included in the YAGARTO download was chosen for its compatibility with the new Eclipse 3.2 release.

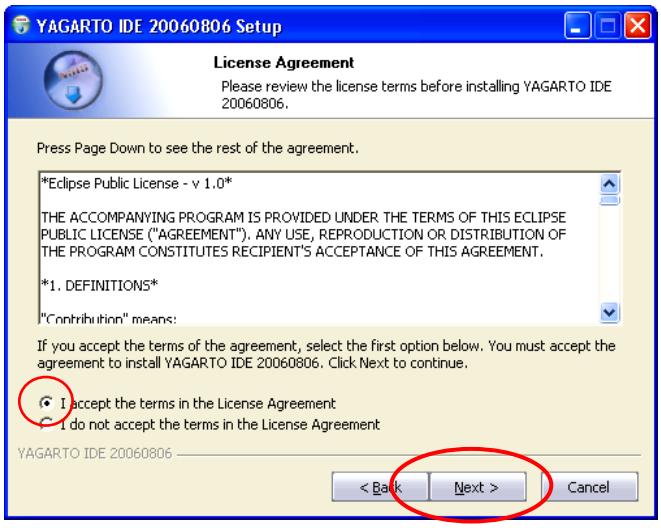
The Zylin website is at this address: **www.zylin.com**

Double-click on the file **Yagarto-ide-20060806-setup.exe** to start the Eclipse IDE installer.

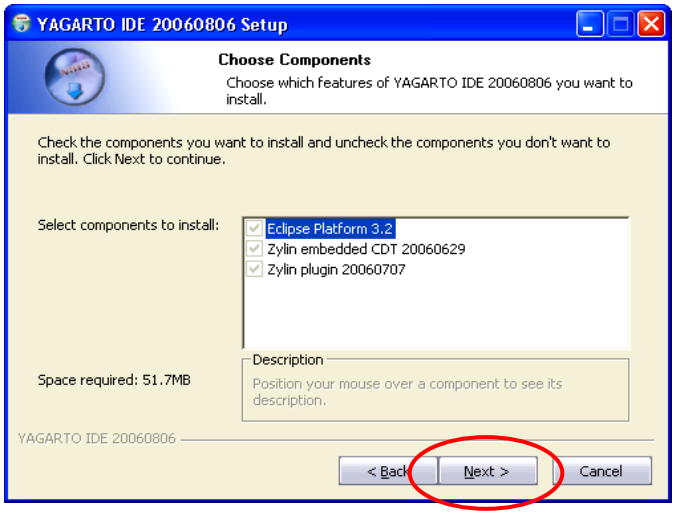


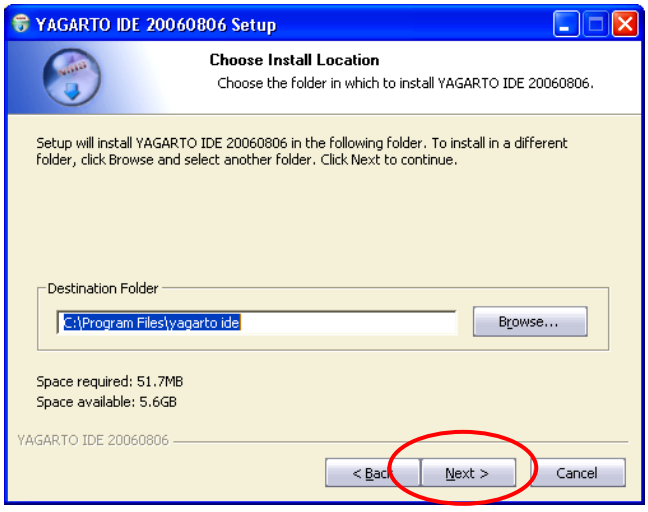
The initial "**Welcome**" screen is shown below to the left; click on "**Next**" to continue. Accept the terms of the license agreement by clicking the "**I accept …**" radio button in the screen below right and then click "**Next**" to continue.



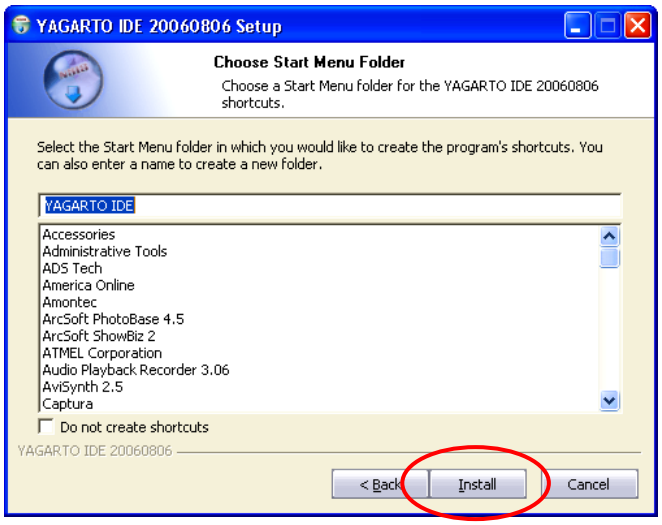


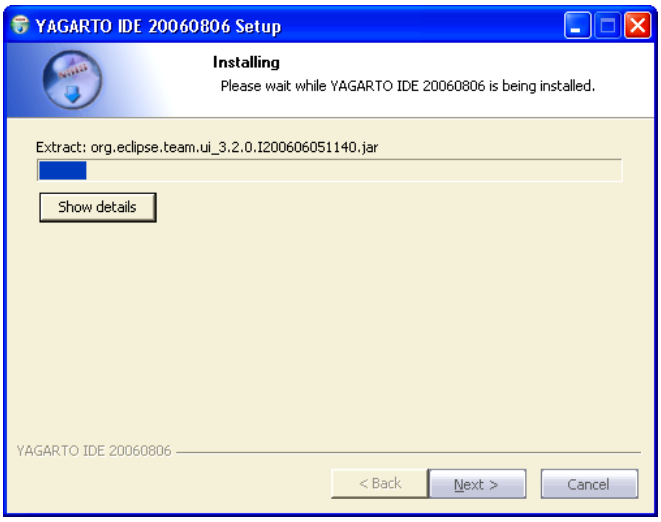
You are forced to select Eclipse and the Zylin plug-ins in the "**Choose Components**" screen shown below to the left. Click on "**Next**" to continue. Take the default in the "**Choose Install Location**" screen below on the right. Click "**Next**" to continue.



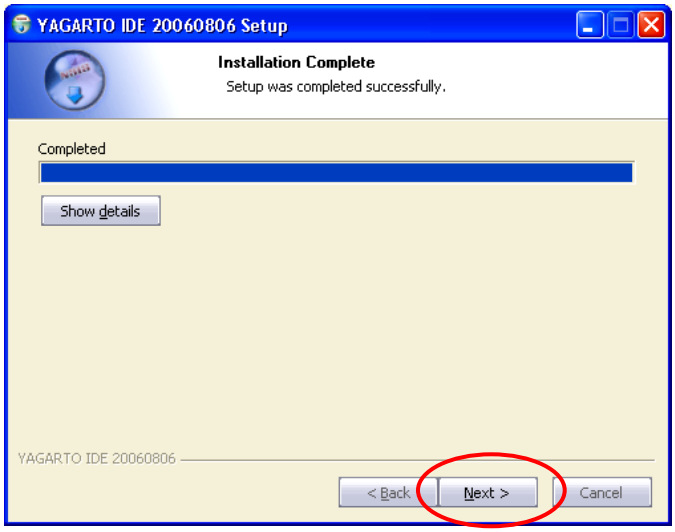


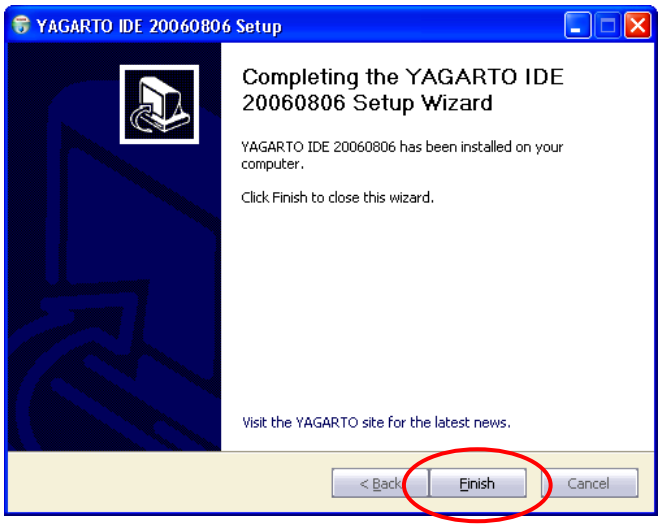
The Eclipse IDE can be added to the Start menu as shown in the "**Choose Start Menu Folder**" screen below on the left. Click "**Install**" to start the Eclipse installer. The Eclipse installer will commence and it will just take at most a couple of minutes.





Click "**Next**" when the Eclipse installer finishes, as shown below to the left. Finally, click "**Finish**" to exit the Eclipse installer as shown in the screen below on the right.





Make a mental note that YAGARTO installed the Eclipse components in the following folder:

c:\Program Files\Yagarto IDE

That's it, we're done! We now have a complete Eclipse-based ARM cross development system installed. Michael Fischer's YAGARTO installers set up the correct paths in Windows. All we have to do is set up any required drivers for the JTAG debugger, add OpenOCD as an Eclipse "external tool" and we will be ready to create our first Eclipse project.

Install the Wiggler Drivers

If you are planning to install a USB debugger, such as the Amontec **JTAGKey** or the Olimex **ARM-USB-OCD**, please skip this section.

Unless you are a perfect programmer, you will occasionally require the services of a debugger to trap and identify software bugs. The AT91SAM7S256 microprocessor has special debug circuits on chip that can start and stop execution, read and write memory, and provide two hardware-assisted breakpoints. The interface to the outside world is a standard JTAG interface (essentially a very complicated and slow serial shift register protocol). You need a device called a JTAG debugger to connect your PC to the ARM chip's JTAG pins. You also need a software program to operate that debugger and interface the JTAG protocol to the Eclipse/GDB source code debugger protocol; that is OpenOCD and you've already installed it.

One way to connect your PC to the AT91SAM7S-EK target board's JTAG connector is to use an inexpensive device called a "wiggler". This can be purchased from Olimex for $19.00 US. It's just a simple voltage level-shifter and it plugs into your PC's parallel port.

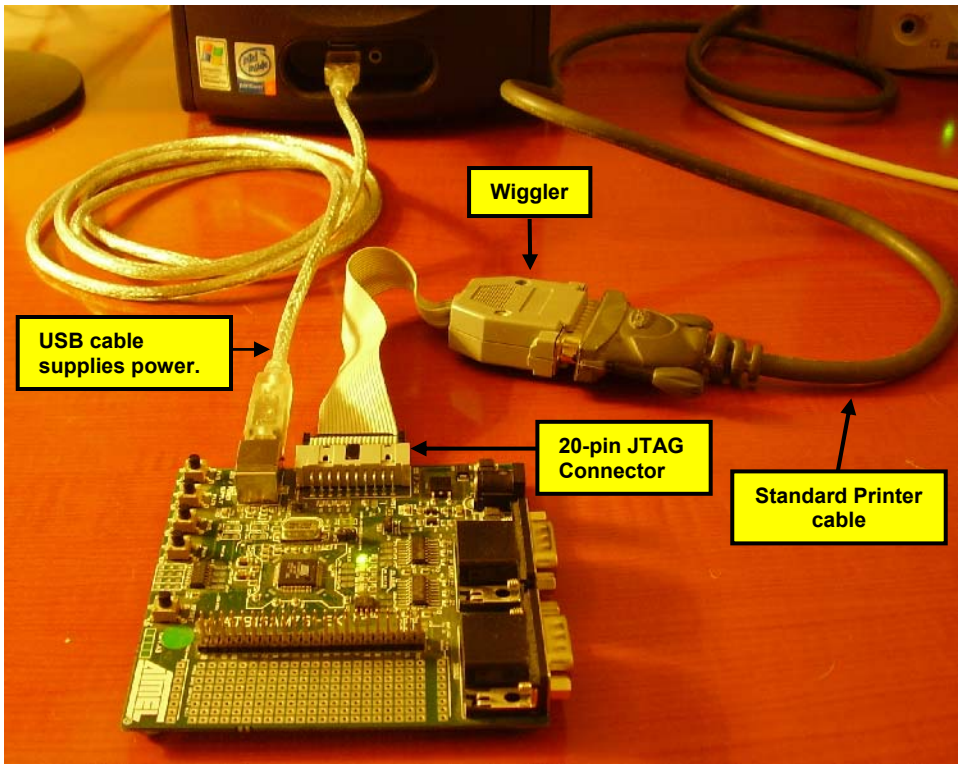
The ARM-JTAG device is available from:

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. A standard USB cable is connected to supply board power. The ARM-JTAG interface is attached to the PC's printer port; in the author's setup, a stock parallel port cable from the local computer store was employed. The JTAG 20-pin connector is keyed so it can't be inserted improperly.

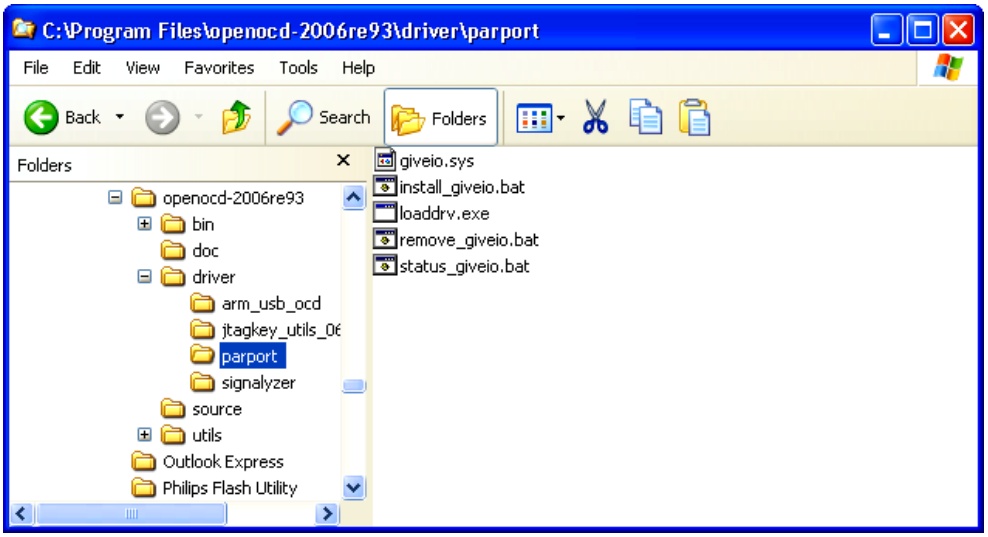


There are two well known criticisms of the “wiggler” device. First, the printer port of a PC limits operation of the JTAG to 500 Khz and this translates into slow downloading. Second, many PCs are now being manufactured without the customary serial and parallel port; the PC world is gravitating to the USB protocol.

If you are planning to use the inexpensive “wiggler” JTAG interface, a special **giveio.sys** driver has to be installed. This only needs to be done once.

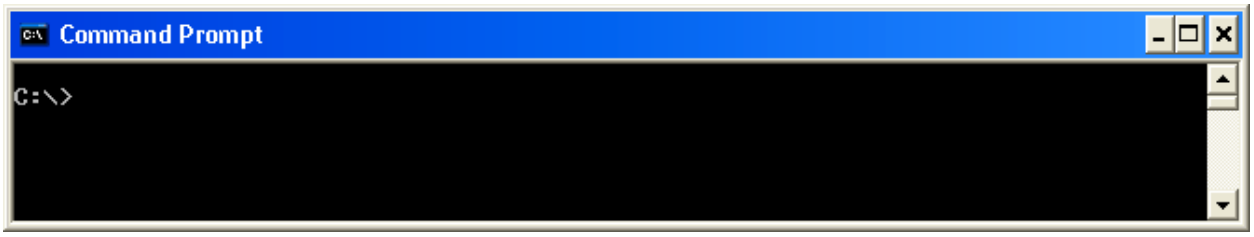
The giveio.sys driver is in the folder: **c:\Program Files\openocd-2006re93\driver\parport**

Note: check if this folder name has changed due to a newer YAGARTO release



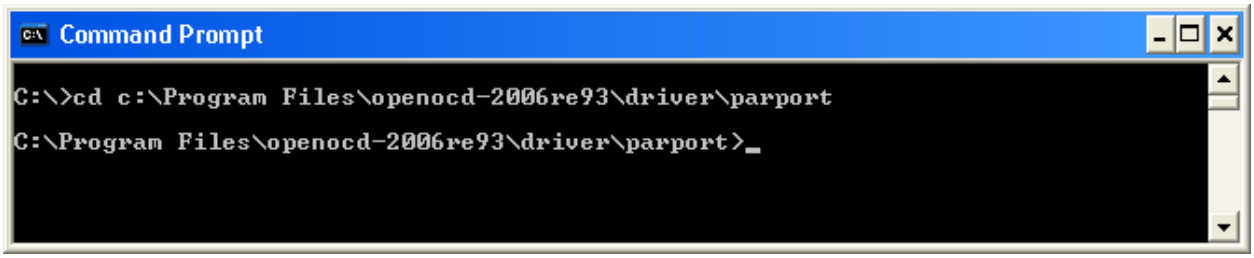
Start the installation of the giveio.sys driver by opening up a **Command Prompt** window (for really experienced readers, that’s the old DOS window). The “command prompt” can be found in your Windows start menu “**Start – All Programs – Accessories**”. If your Command Prompt window is not at the root folder **c:**, you can type the **CD ** command shown below to locate yourself at the root folder.

**>cd **

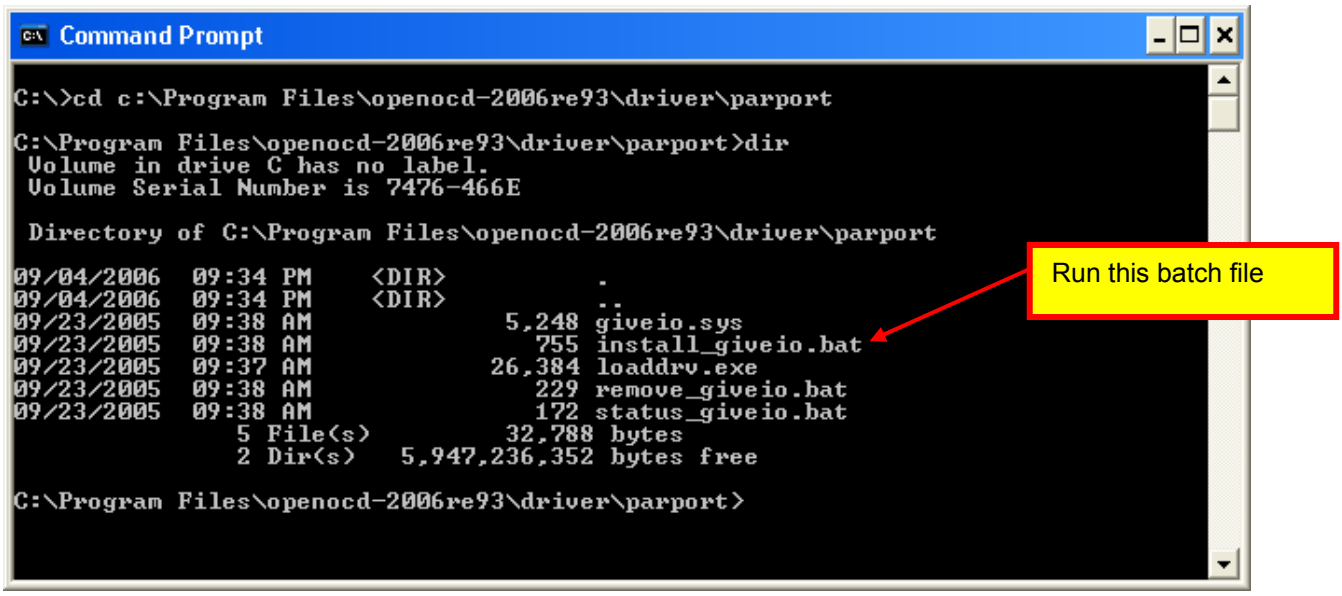


We need to change to the directory: **c:\Program Files\openocd-2006re93\driver\parport** since it contains the **giveio.bat** installation batch file. Type the **CD** command again as shown below to do this. Now the command prompt window will show that we are inside that folder.

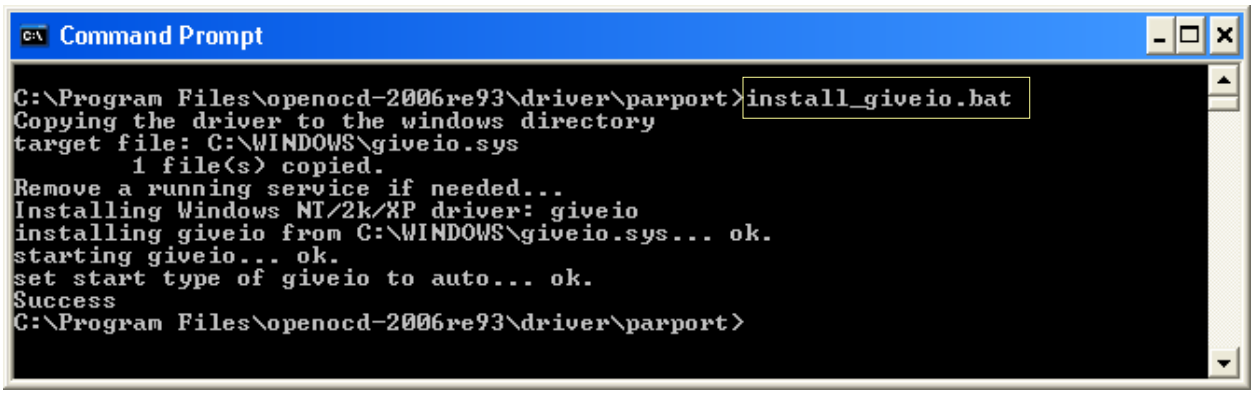
>cd c:\Program Files\openocd-2006re93\driver\parport



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. 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.



The giveio.sys driver is a permanent installation; you only have to do this once.

Install the Amontec JTAGKey USB Drivers

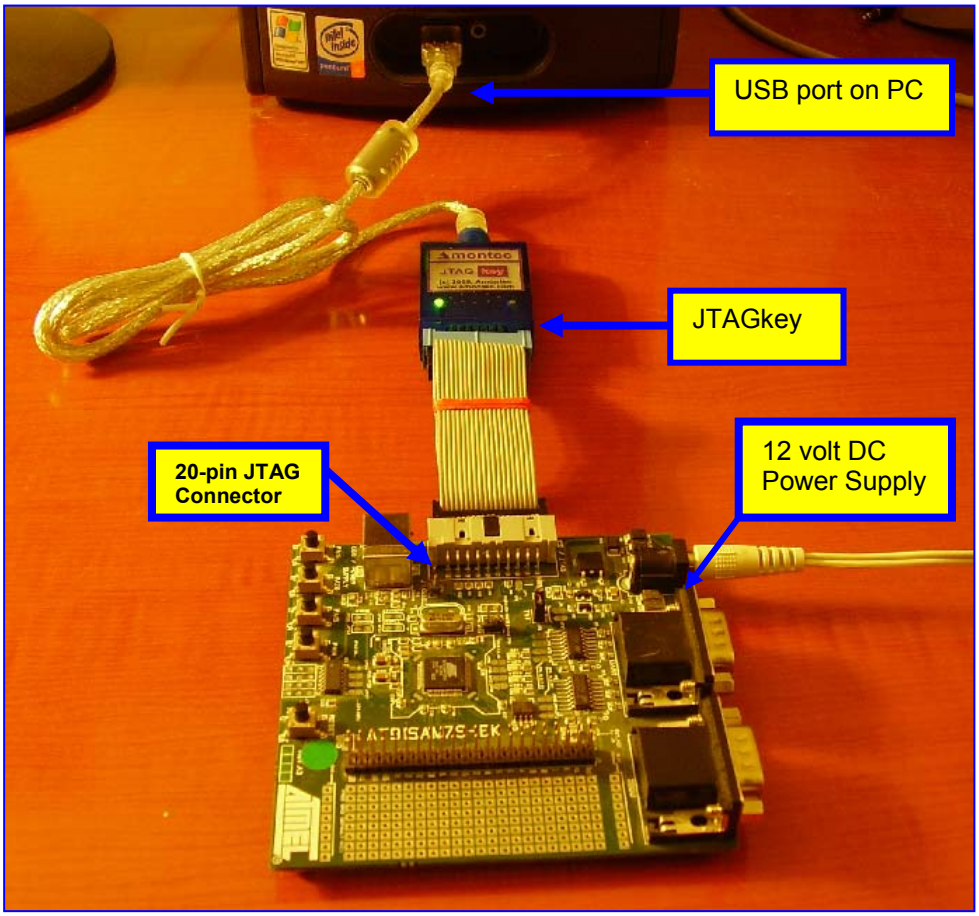
In Dominic Rath's thesis about the OpenOCD project, a USB-JTAG interface based on the FTDI FT2232C engine was described with a schematic. The Swiss engineering firm Amontec has developed and marketed a professional version of this USB JTAG interface called the **JTAGkey**. Its price is €139 (euros) or $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

Amontec has also addressed the hobbyist and student market with the **JTAGKey-Tiny** device, priced at €29 (euros) or $37 (us) and illustrated below. This smaller **JTAGKey-Tiny** device plugs directly into the 20-pin JTAG connector and uses a mini-USB cable to attach to the PC. While the author hasn't had a chance to test it prior to publication, it appears that the installation procedure is similar to that of the more expensive **JTAGKey** shown below.

Professionals would tend to select the more expensive **JTAGKey** for its ESD protection and the flat ribbon cable that attaches to the prototype system, as seen in the hardware setup coming up. It also has an integrated USB cable fitted with a ferrite filter. The **JTAGKey-Tiny** plugs directly into the application board's 20-pin JTAG connector and therefore must have the vertical clearance to permit this fitting.

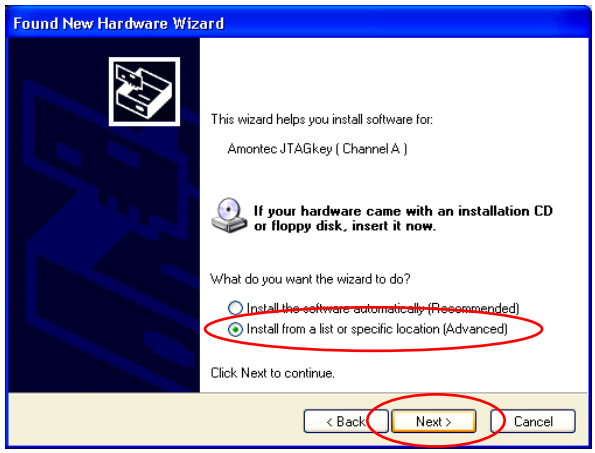
The hardware setup, shown below, includes the Amontec **JTAGKey** plugged into the 20-pin JTAG header on the AT91SAM7S-EK target board and also into the PC's USB port. The JTAG does not supply board power, so in this example a 12-volt DC "wall wart" power supply is fitted to the power connector. If you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

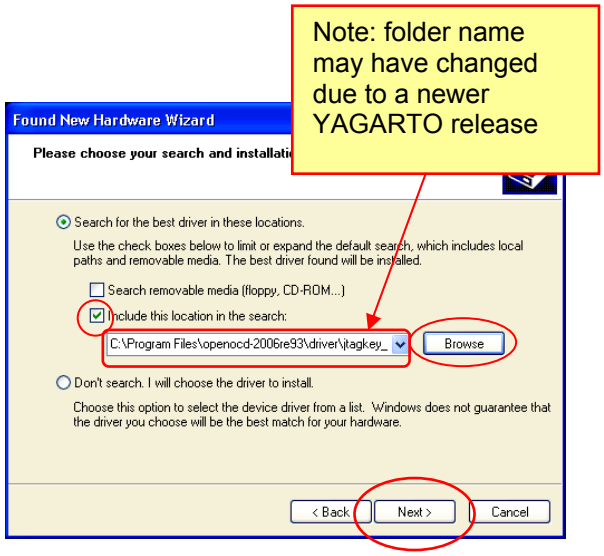


Plug in the Amontec JTAGkey into the USB port. You should hear the familiar USB "beep" sound followed by the following screen indicating that new USB hardware has been detected.

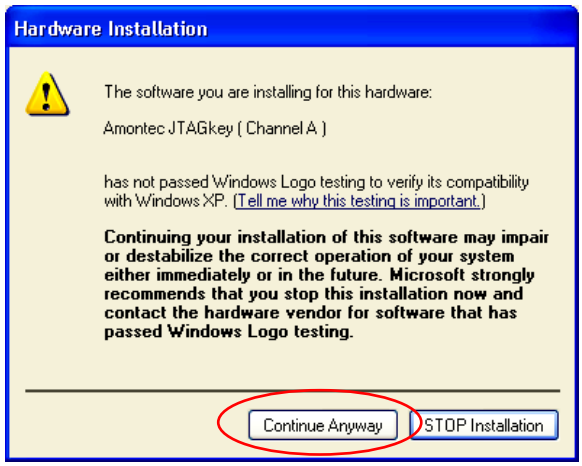
The virtual device drivers are already on our **c:\Program Files\openocd** folder thanks to Michael Fischer's OpenOCD installation program we ran earlier in this tutorial. Therefore, advise Windows NOT to search for the drivers by clicking on "**No, not this time**" as shown below. Click "**Next**" to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

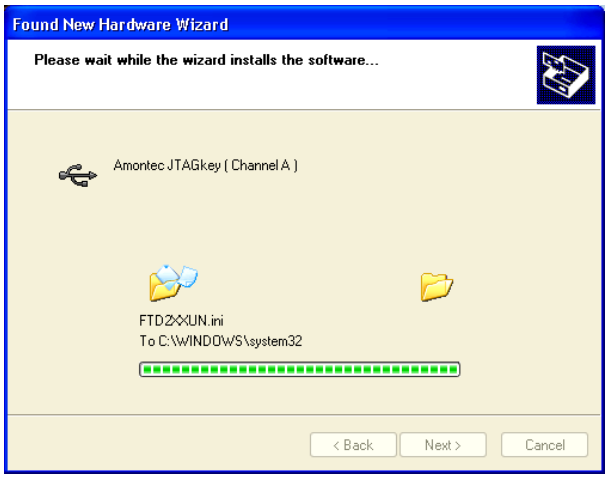
Instruct Windows to "**Install from a list or specific location (Advanced)**" as shown on the left hand screen below. Click "**Next** to continue. Now use the "**Browse**" button to find the directory "**c:\Program Files\openocd-2006re93\driver\jtagkey_utils_060307**\" as shown below on the right hand screen. Click "**Next** to continue.





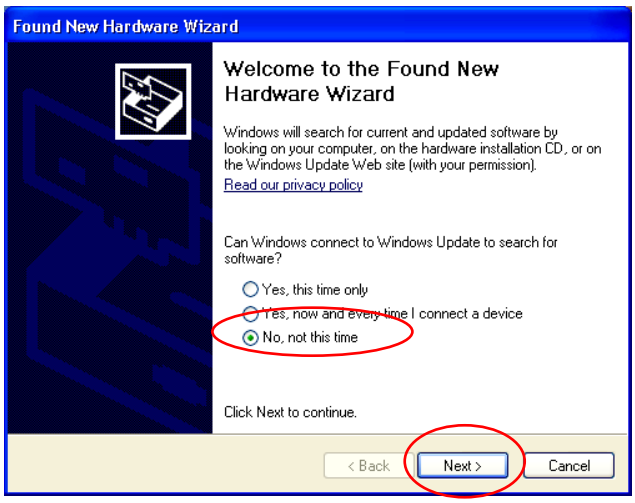
Pay no attention to Windows complaints about Logo testing by clicking on "**Continue Anyway**" on the left screen below. The virtual device driver installation for Channel A will now run to completion.

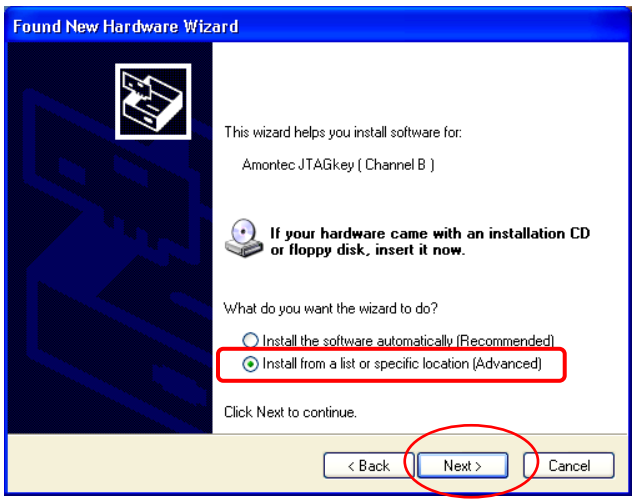


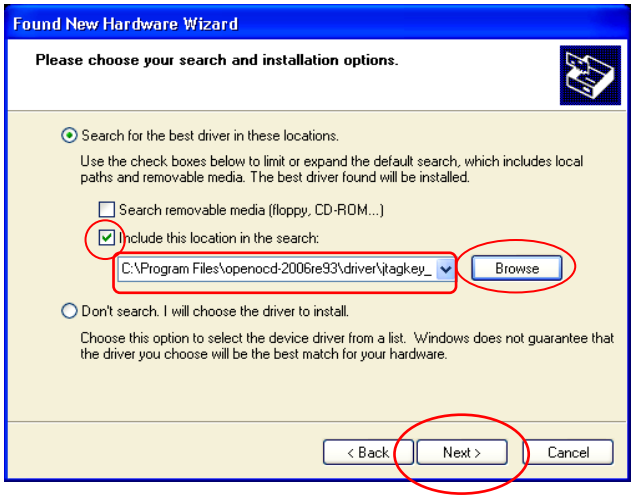


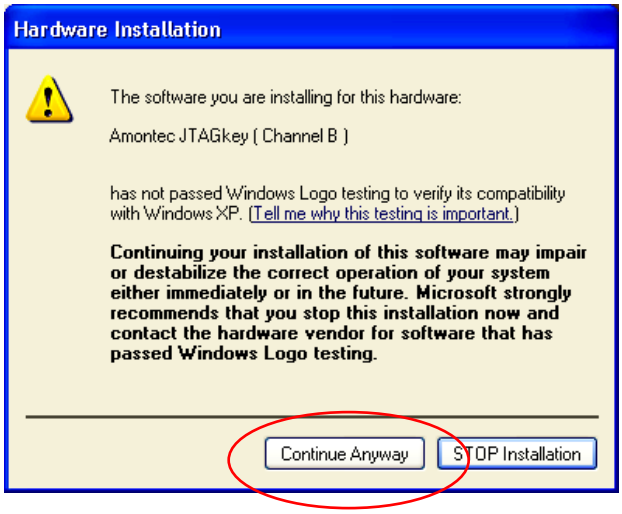
When the Channel A driver installation has completed, you will see the screen below indicating successful installation. Click "**Finish**" to exit the channel A installation as shown below.

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.

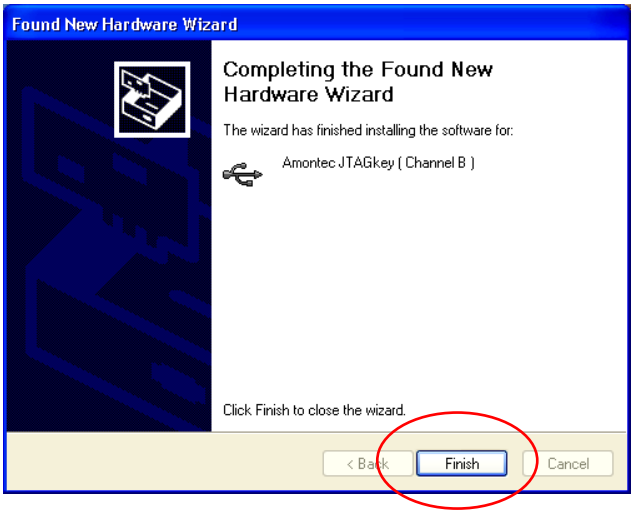




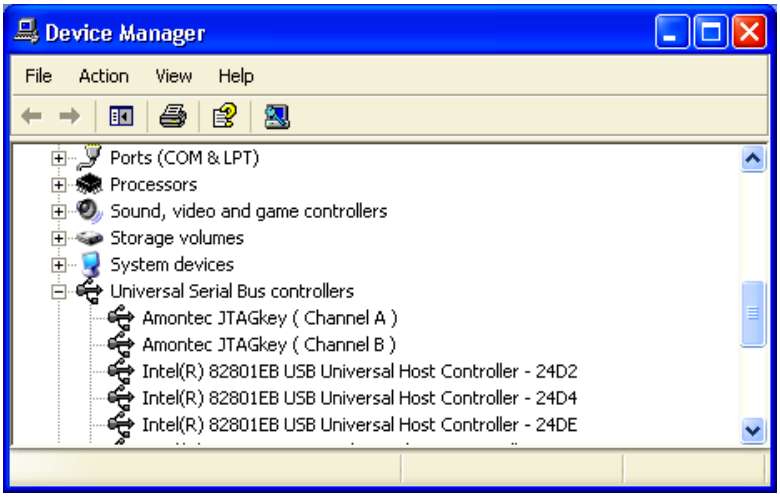




When the Channel B driver has completed, you will see the screen below indicating successful installation. Click "**Finish**" to exit the channel B installation as shown below.



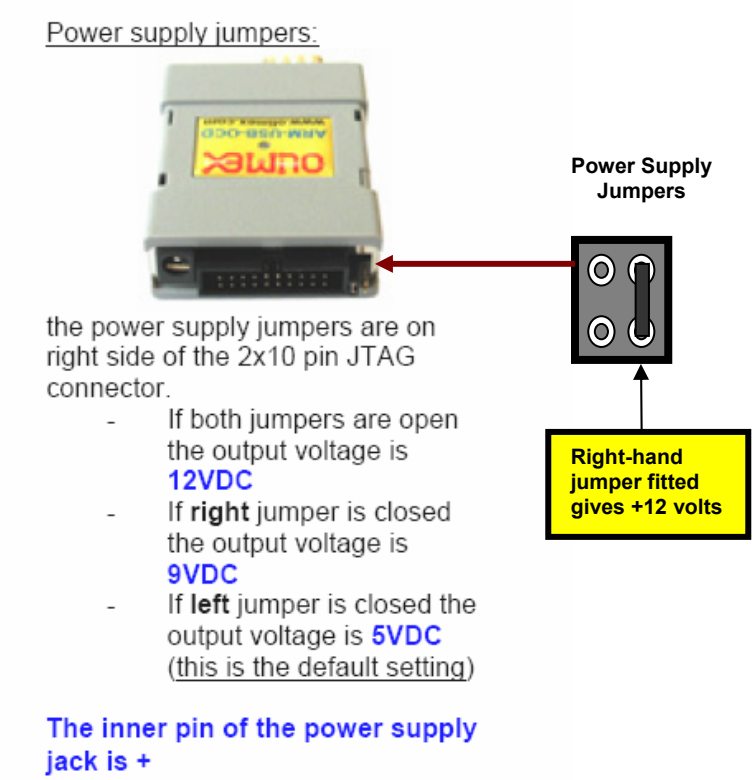
To be sure of successful installation of these JTAGkey virtual device drivers, use the Windows Start menu to look at the "**Control Panel – System – Hardware - Device Manager**", inspecting carefully the USB controllers. As can be seen below, the Amontec JTAGkey channel A and channel B USB ports are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



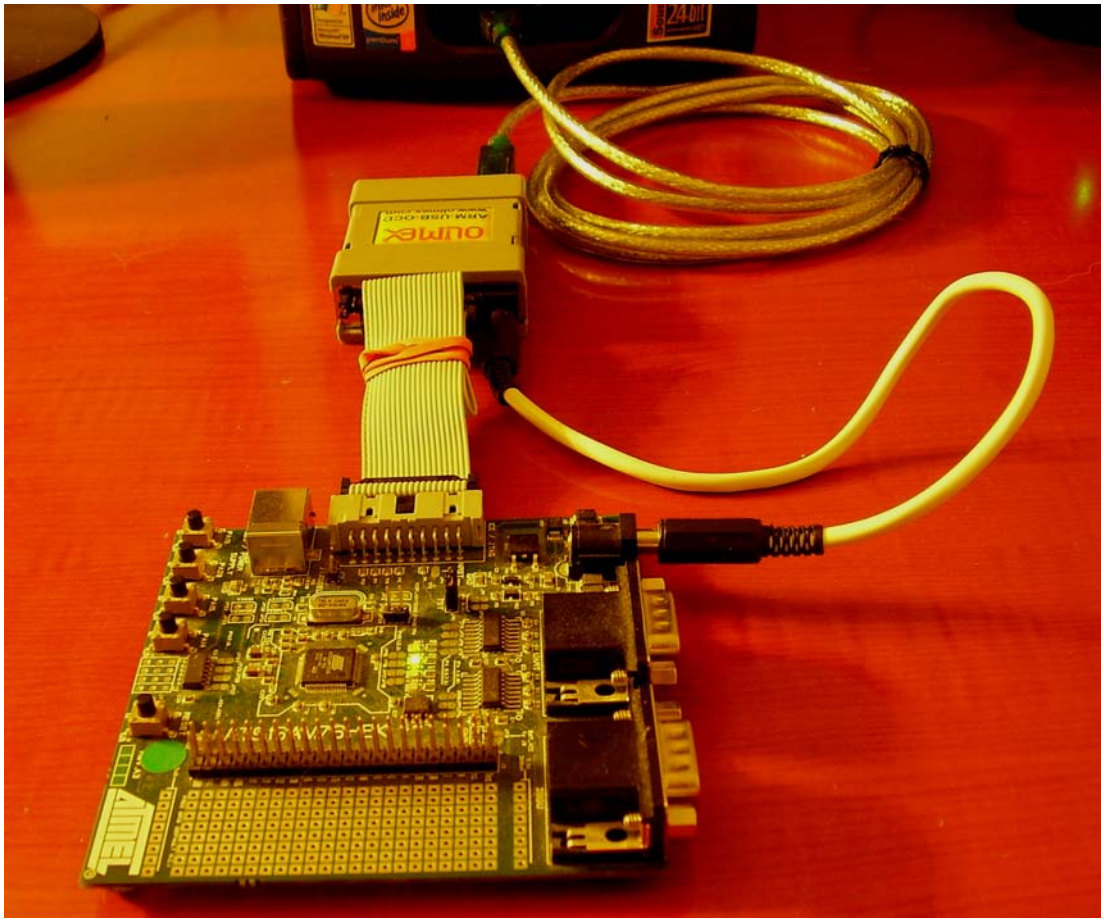
Install the Olimex ARM-USB-OCD USB Drivers

Olimex also developed a version of the USB-based JTAG debugger mentioned in Dominic Rath's OpenOCD thesis. It includes a couple of unique features such as an extra serial port (might come in handy if you have a laptop with no serial port) and a DC power supply that can be strapped for 5v, 9v or 12v operation. This DC supply includes a cable that can power your board, if needed. The Olimex ARM-USB-OCD debugger is є55 (euros) or $69.95 (US).

To use the ARM-USB-OCD power supply, there are jumpers to set the voltage. While the Atmel specification for the AT91SAM7S256-EK board is 7 – 12 volts for the DC supply, it worked for the author at all the above voltage ranges. Just to be safe, strap the Olimex ARM-USB-OCD DC supply to +9 volts (right-hand jumper installed).



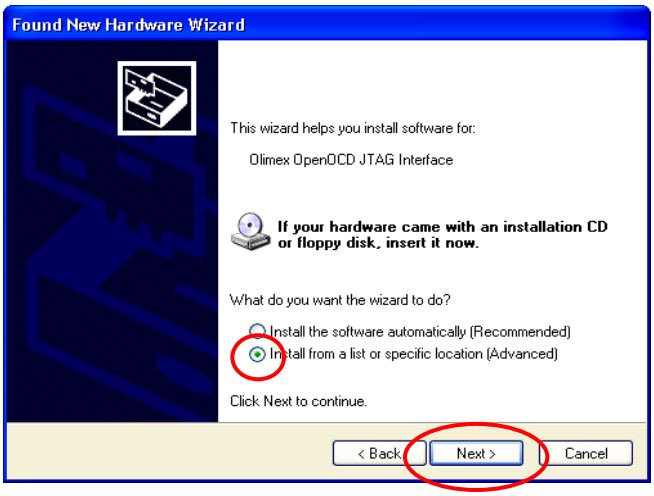
The hardware setup for the Atmel AT91SAM7S256-EK board is shown below. The 20-pin JTAG ribbon cable connectors are keyed so they can't be fitted improperly. The DC supply cable from the ARM-USB-OCD dongle powers the board.

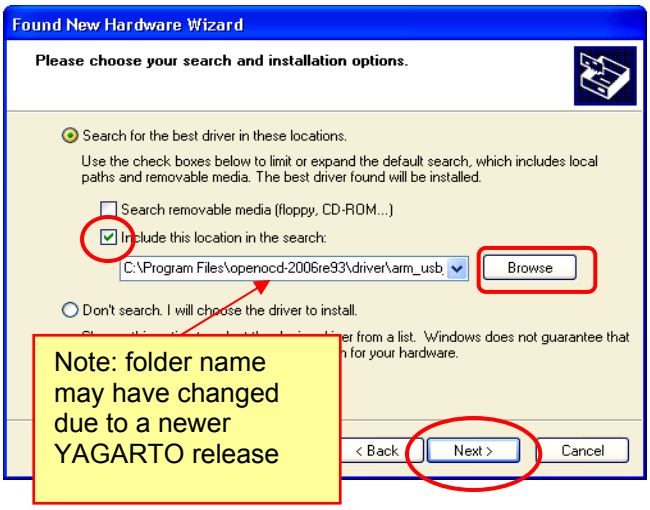


Plug in the Olimex ARM-USB-OCD dongle into the USB port. You should hear the familiar USB "beep" sound followed by the following screen indicating that new USB hardware has been detected.

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.

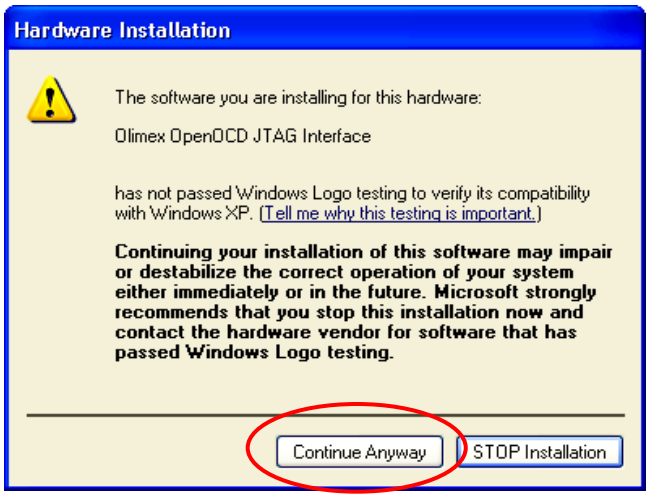
Instruct Windows to "**Install from a list or specific location (Advanced)**" as shown on the left hand screen below. Click "**Next** to continue. Now use the "**Browse**" button to find the directory "**c:\Program Files\openocd-2006re93\driver\arm_usb_ocd**" as shown below on the right hand screen. Click "**Next** to continue.

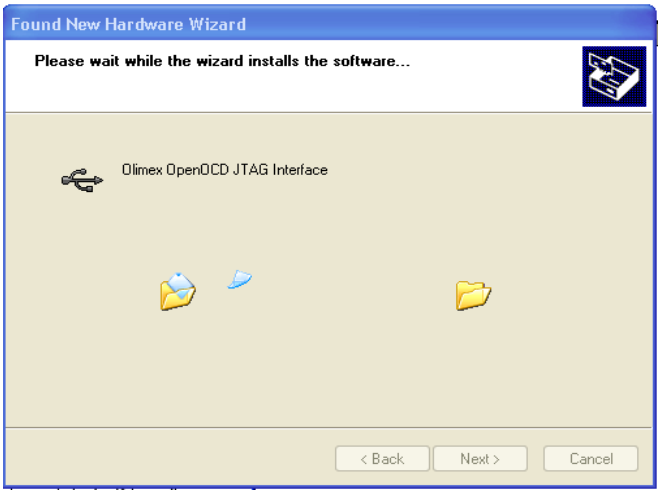




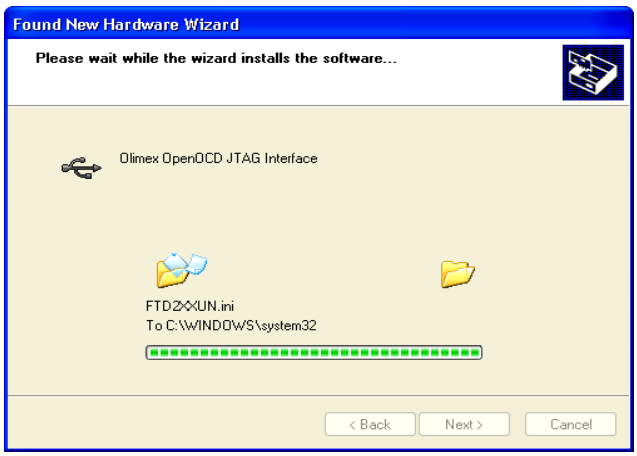
Note: folder name may have changed due to a newer YAGARTO release

Ignore the Windows XP complaint about "Logo Testing" by clicking "**Continue Anyway**" as shown on the left below. The installer will now start installation activities.



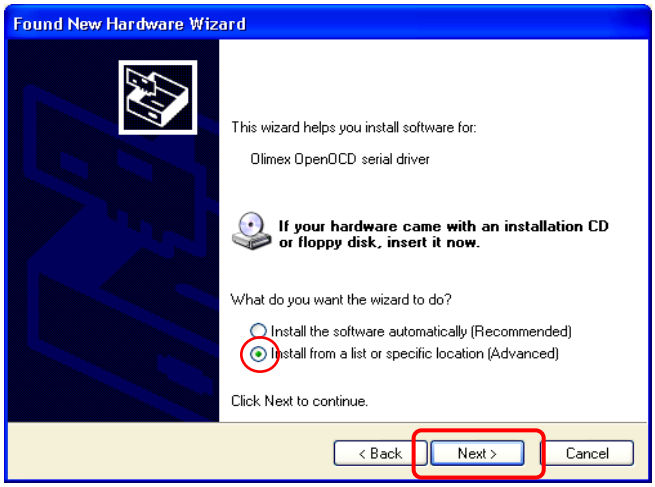


When the driver installation for the Olimex ARM-USB-OCD JTAG debugger is done, click on "Finish" on the screen shown below on the right to exit the installer.

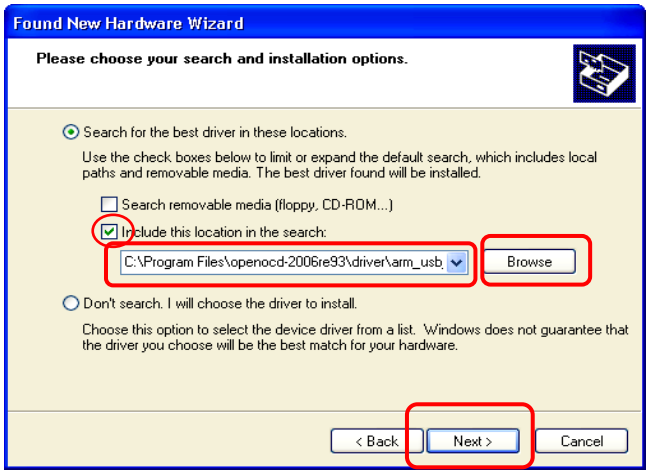


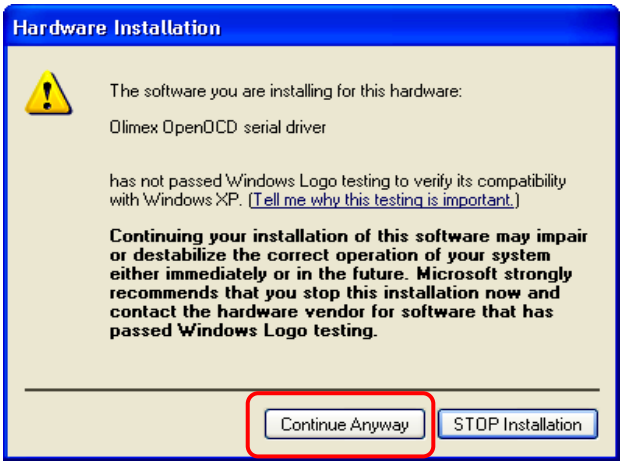
Remember that the Olimex ARM-USB-OCD also supports a auxillary serial port. Windows will now start a dialog to install that virtual driver. Since we know exactly where the driver files are, click the radio button "**No, not this time**" on the window below left and click "**Next**" to continue.

Also click the "**Install from a list or specific location (Advanced)**" radio button below on the right and then click "**Next**" to continue.



Now use the "**Browse**" button to find the directory "**c:\Program Files\openocd-2006re93\driver\arm_usb_ocd**" as shown below on the left hand screen. Click "**Next** to continue. Once again, ignore the Windows complaints about Logo testing and click "**Continue Anyway**" as shown below right.





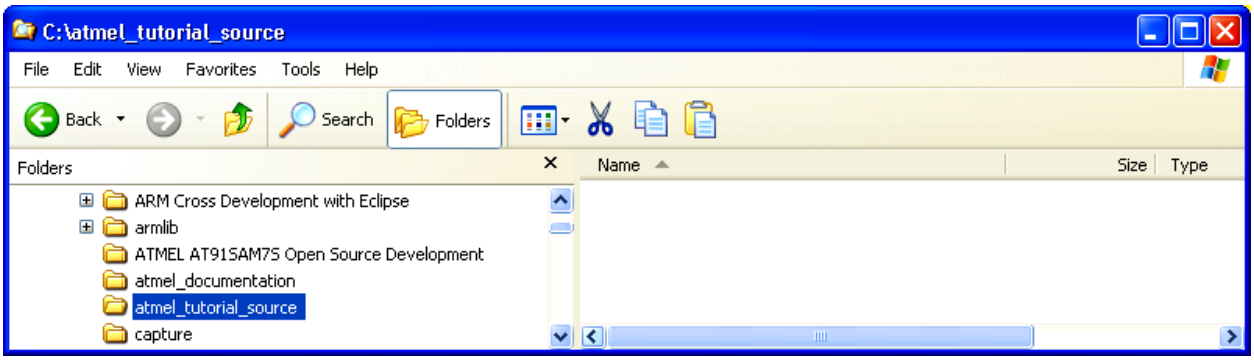
The serial driver installs very rapidly. When the "**Found New Hardware Wizard**" screen appears, click "**Finish**" to exit. Installation of the Olimex ARM-USB-OCD drivers is now completed.

Download the Sample Programs

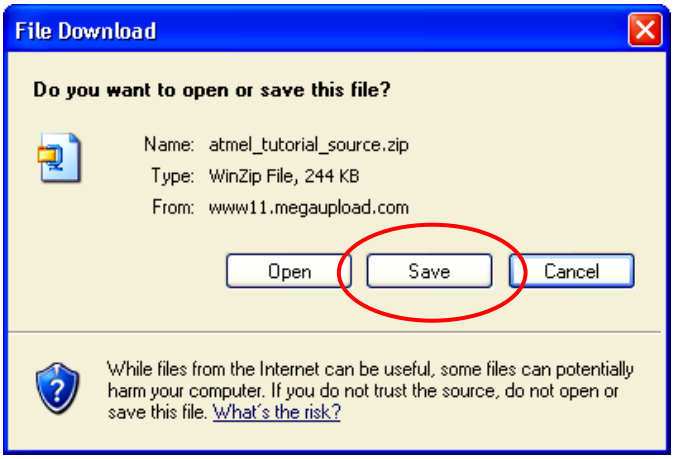
Before we start up the Eclipse IDE, let's first download the tutorial source and OpenOCD configuration files. This material may be downloaded using this link:

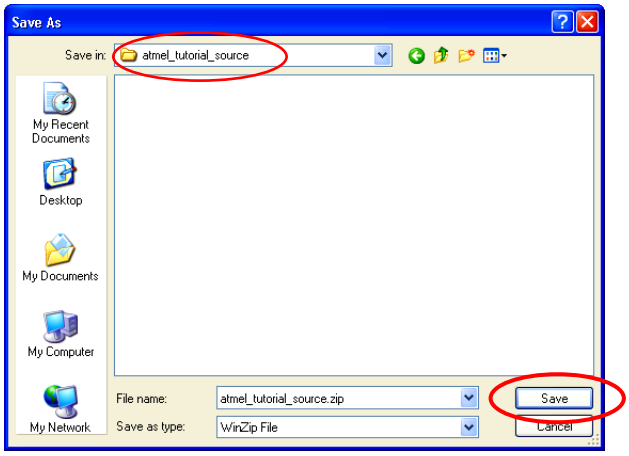
http://www.address_to_be_determined

Using Windows Explorer, create a folder **c:\atmel_tutorial_source** to hold the upcoming download.

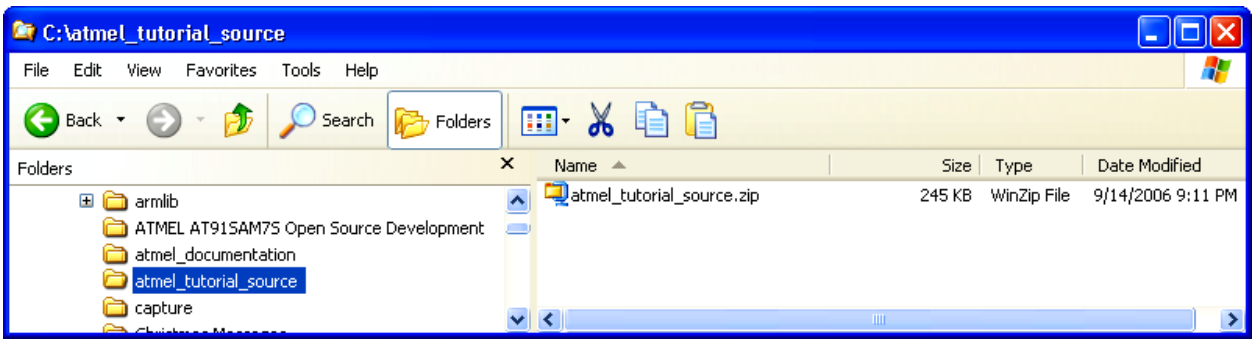


Click on the link above to start the download of the tutorial's source files. Select "**Save**" on the File Download window on the left. In the "**Save As**" window on the right, browse to the folder "**atmel_tutorial_source**" and select "**Save**" to commence the download.

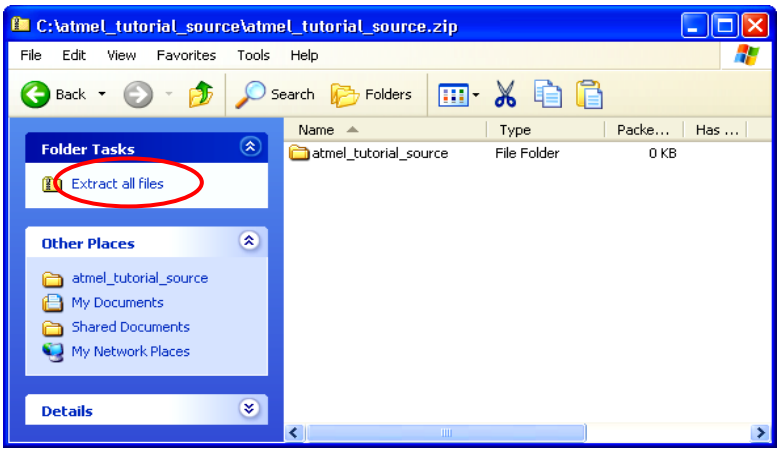




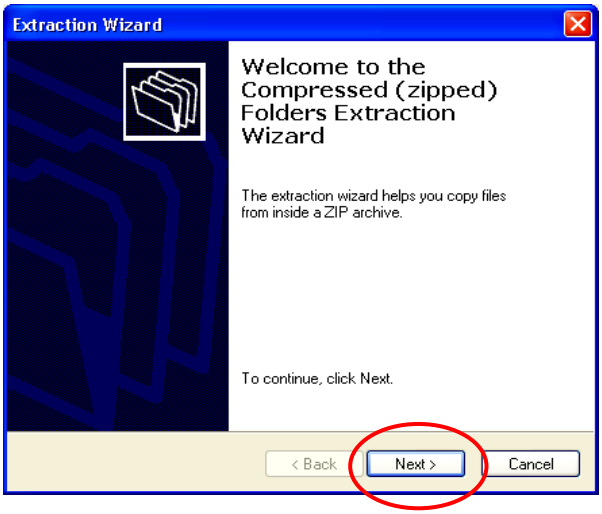
The tutorial source files will download very quickly and the folder **c:\atmel_tutorial_source** will show the zip file present.

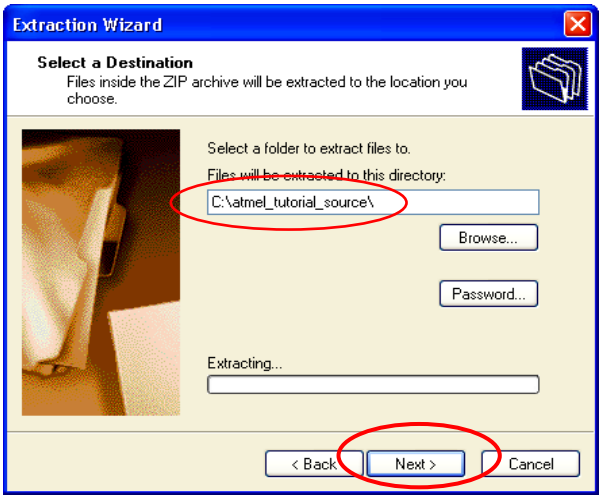


To unzip and extract the zipped file "**atmel_tutorial_source.zip**", double-click on that zip file to start the Windows XP Extraction Wizard. Then click on "**Extract all files**".

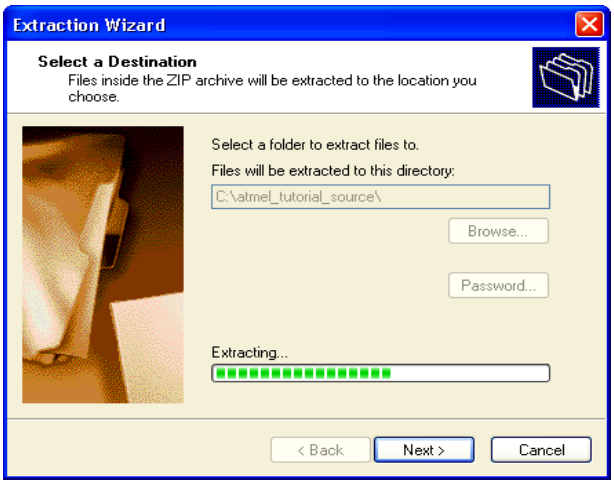


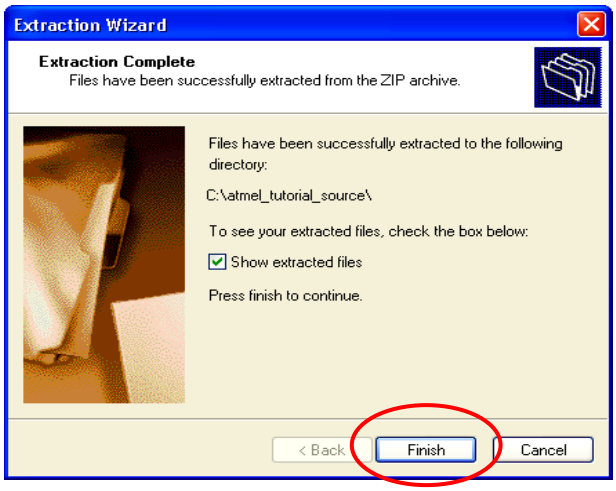
When the Extraction Wizard starts up, click "**Next**" to continue, shown on the screen below to the left. For the destination screen on the right, browse or type the same folder holding the zip file, namely "**c:\atmel_tutorial_source**" and click "**Next**" to continue.



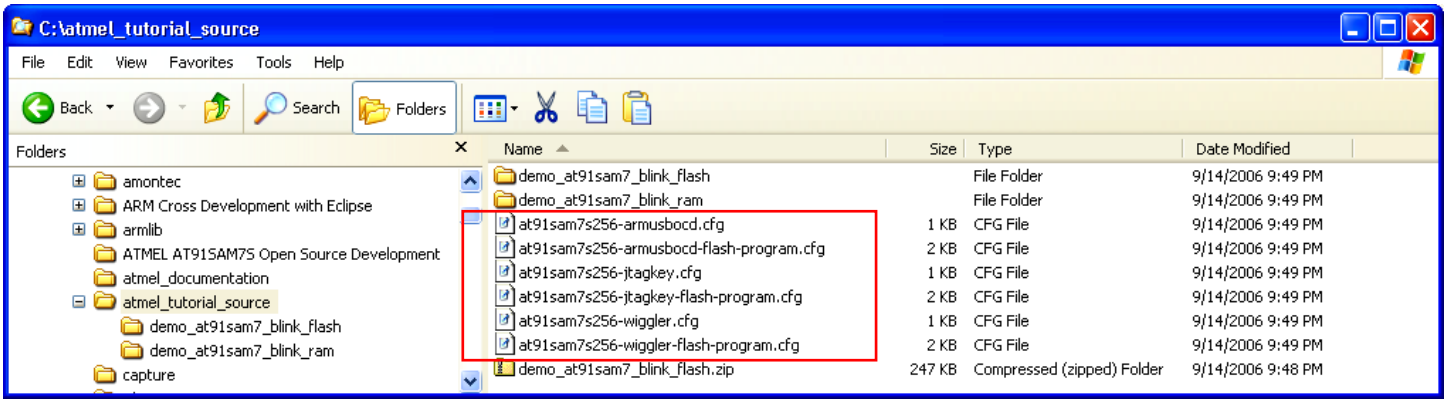


The Extraction Wizard will finish quickly, as shown on the left below. Click "**Finish**" on the "Extraction Complete" screen on the right to exit the Wizard.





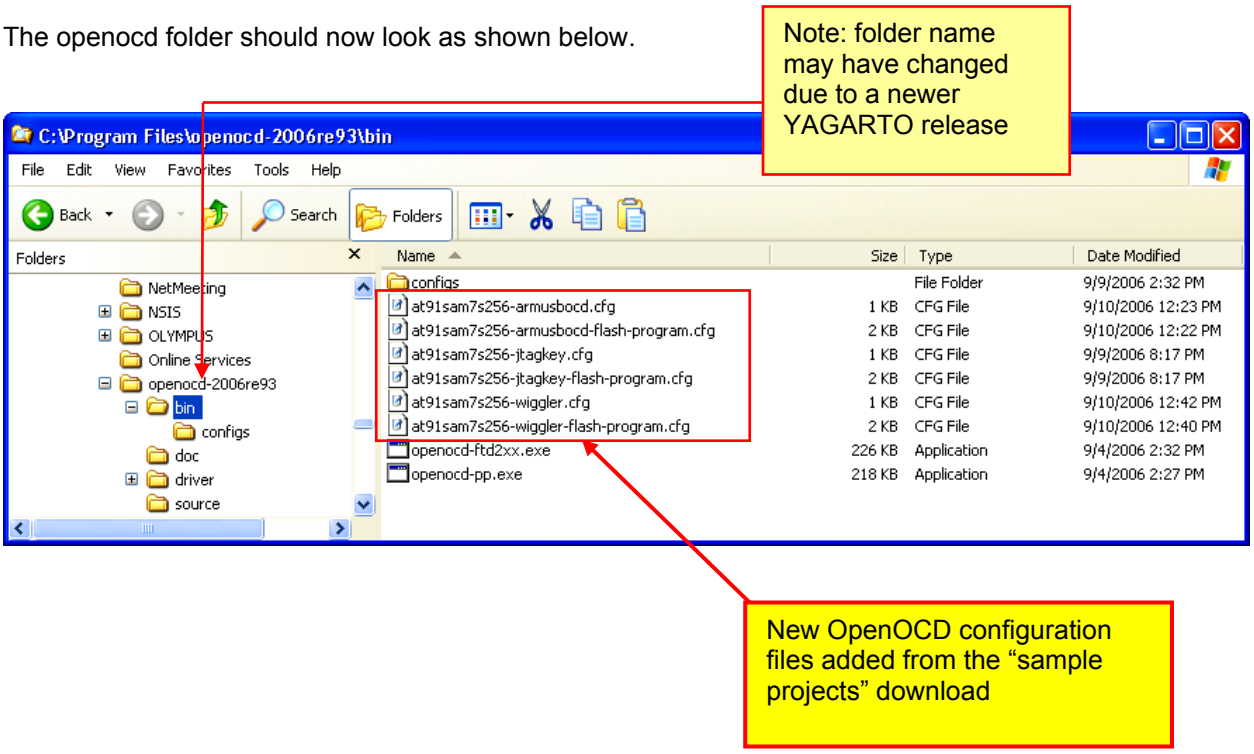
The Windows Explorer will show that the folder "**c:\atmel_tutorial_source**" has several Eclipse projects plus six OpenOCD configuration files.



Move the OpenOCD Configuration Files

Using Windows Explorer, select and move the six OpenOCD configuration files shown above into the **c:\Program Files\openocd-2006re93\bin** folder. These configuration files will be used by the sample projects later in the tutorial. Additionally, this destination folder already has a Windows path defined for it and thus simplifies setting up the OpenOCD as an external tool.

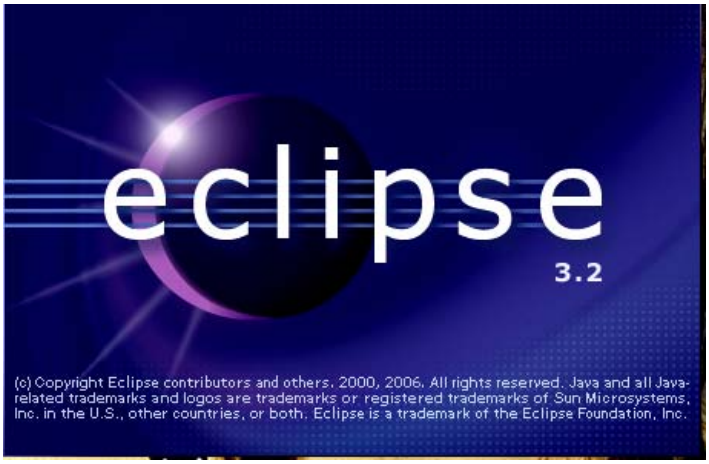
The openocd folder should now look as shown below.



Running Eclipse for the First Time

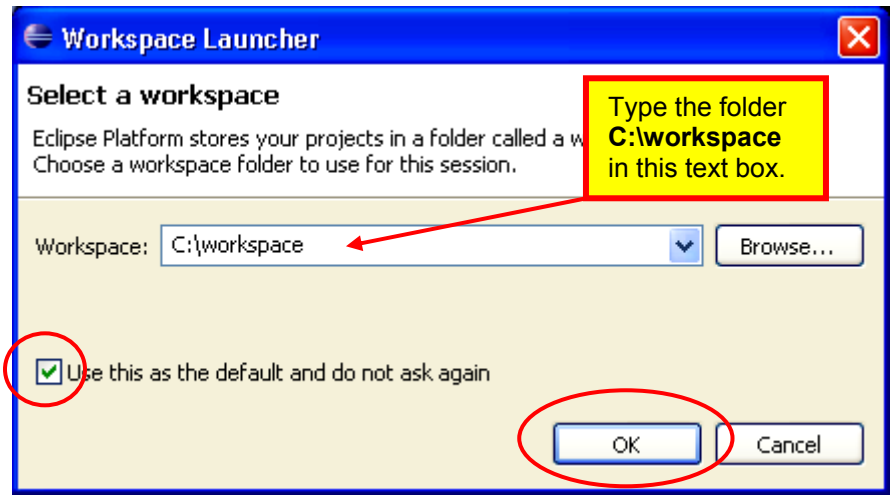
The Yagarto installer creates a desktop icon for starting Eclipse, as shown below. Click on this icon to start the Eclipse IDE.

Now the Eclipse splash screen will open up, as shown below.

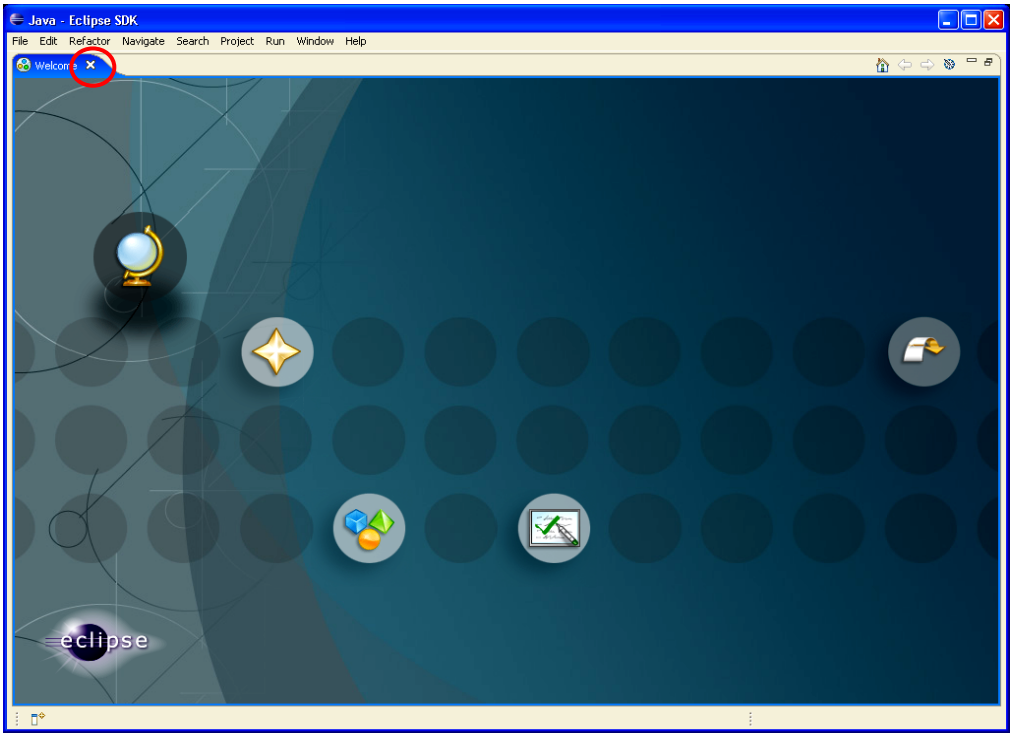


At this point, Eclipse will present a "Workspace Launcher" dialog, shown below. This is where you specify the location of the "workspace" that will hold your Eclipse/CDT projects. You may place the workspace anywhere you wish but for this tutorial I placed it in the root folder as "**C:\workspace**".

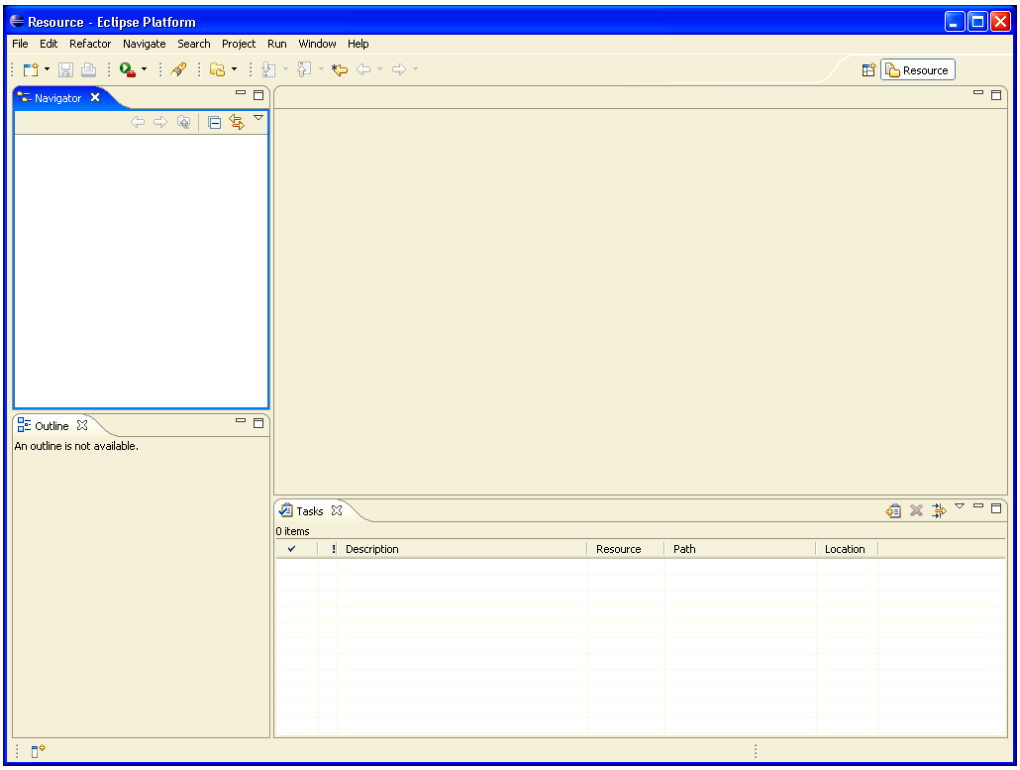
Click the check box so the folder "**C:\workspace**" can be assigned to be the default anytime you enter Eclipse. Click "**OK**" to accept the workspace assignment and continue with Eclipse start-up.



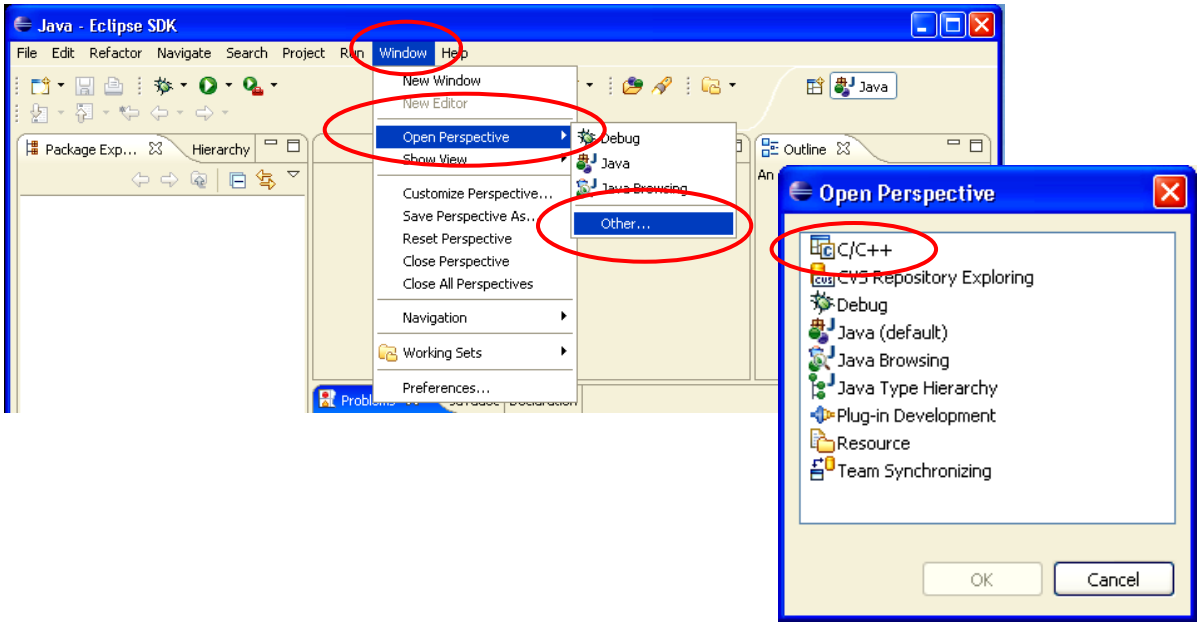
Now Eclipse will officially start and show the “Welcome” page. Since most of the informational icons refer to the JAVA aspects of Eclipse, discard the “welcome” screen by clicking on the “**X**” as shown below.



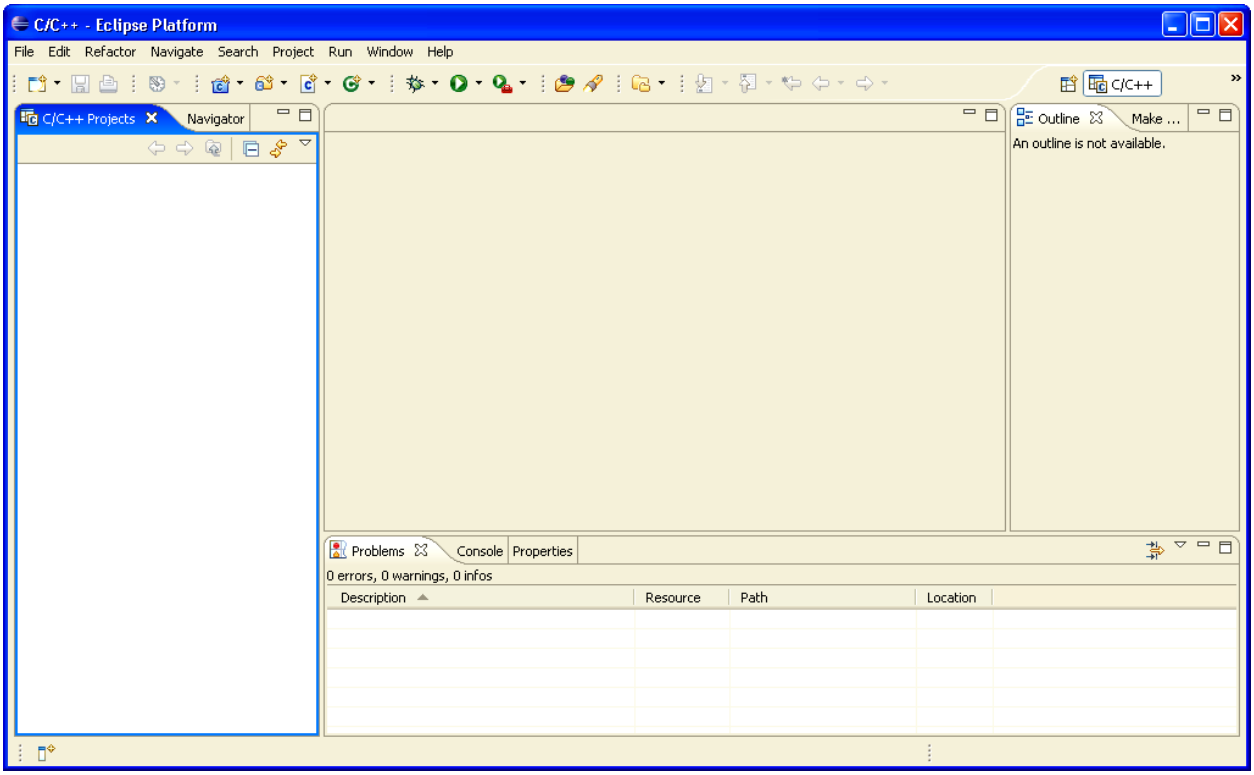
What follows is the “Resource” perspective. A perspective is simply a layout of “views” on the display surface (the Resource perspective includes “Navigator”, “Editor”, “Outline” and “Tasks” views.



Let's switch to the C/C++ perspective. Click on "**Window – Open Perspective – Other…**". Then click on "**C/C++**" to open Eclipse into the C/C++ perspective.



This is the C/C++ perspective. We will be learning more about the various component parts later in this tutorial.

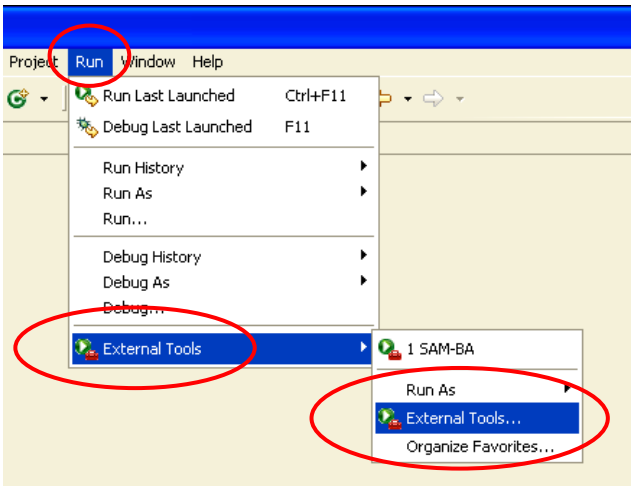


Install OpenOCD as an Eclipse External Tool (wiggler)

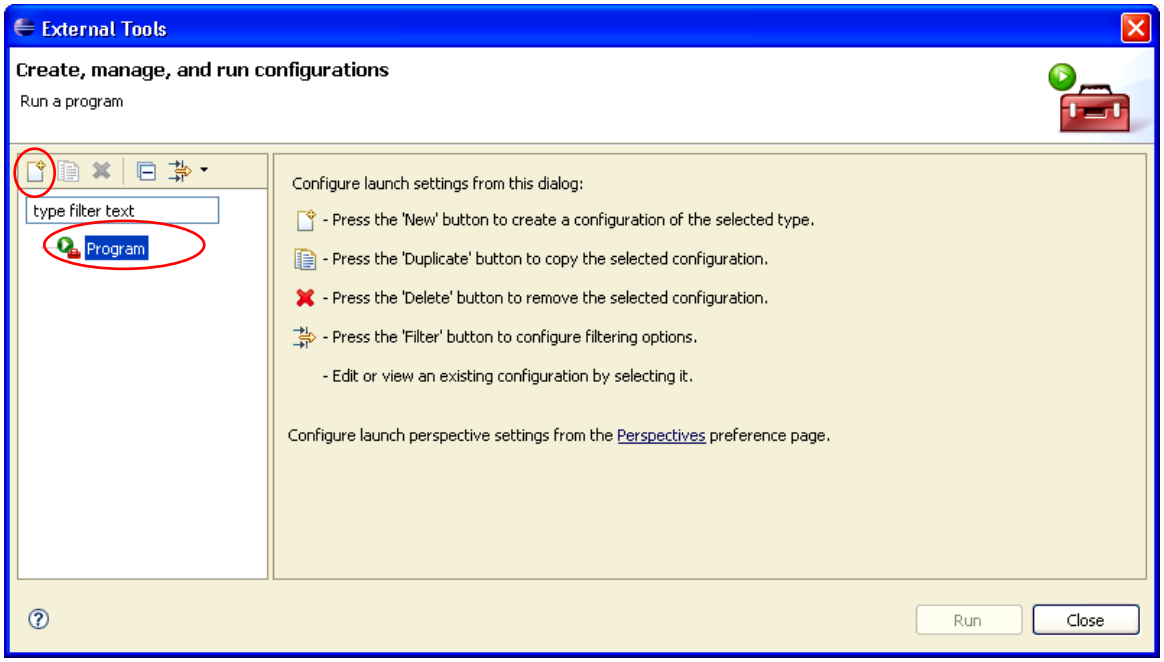
If you have a USB-based debugger, such as the Olimex ARM-USB-OCD or the Amontec JTAGKey, please skip to the next section. This section is for those with the "wiggler" device.

When it's time to debug an application, we must be able to conveniently start the OpenOCD debugger. OpenOCD runs as a daemon; a program that runs in the background waiting for commands to be submitted to it. Eclipse has a very nice "external tool" feature that allows us to add OpenOCD to the RUN pull-down menu.

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



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



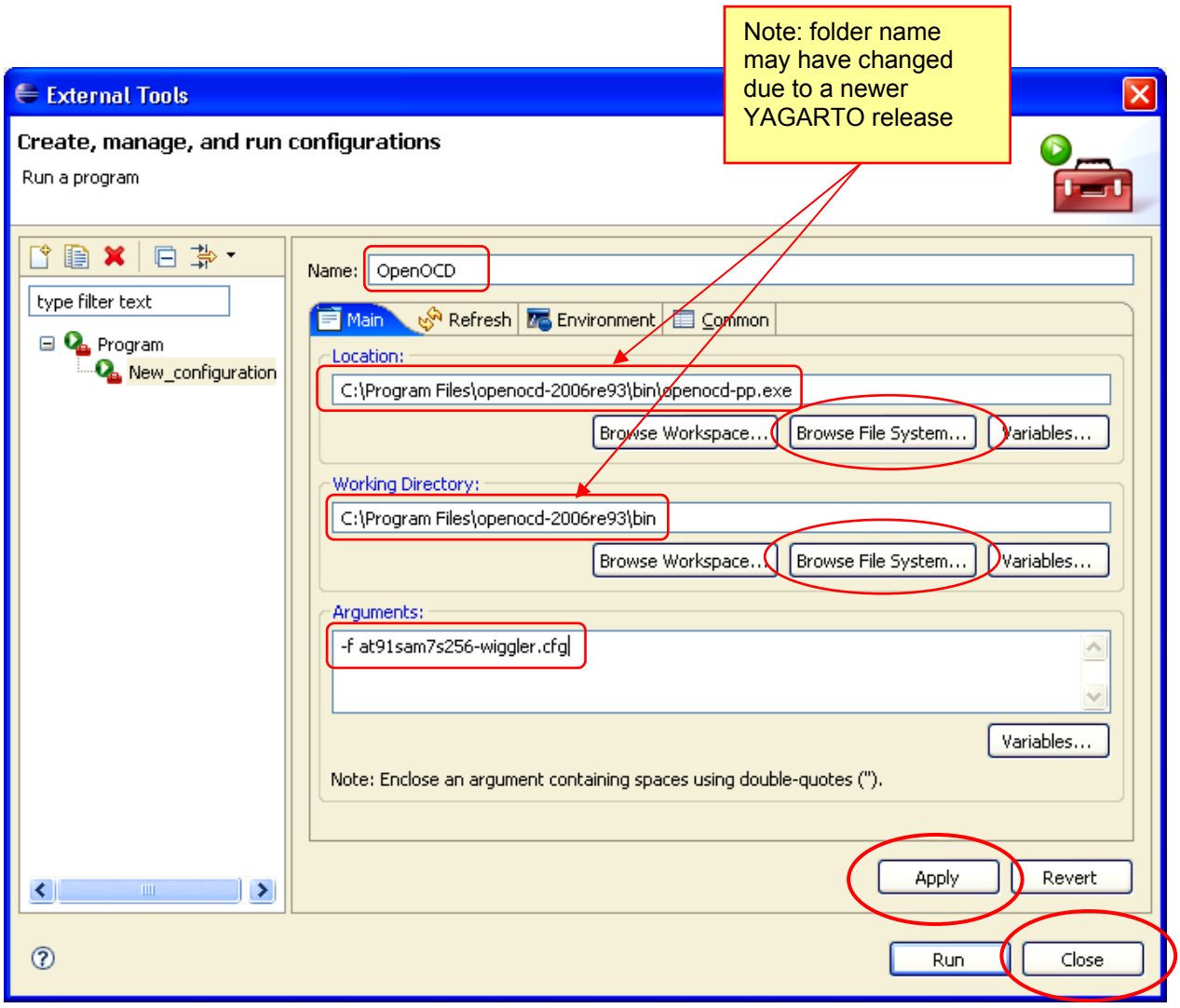
Fill out the "**External Tools**" form exactly as shown below.

There are two versions of OpenOCD; **openocd-pp.exe** supports the parallel-port "wiggler" device (ARM-JTAG from Olimex) while **openocd-ftd2xx.exe** supports the USB-based devices from Amontec and Olimex. In this section we are installing the "wiggler" version of OpenOCD as an Eclipse external tool.

In the "Location:" pane, use the "**Browse File System…**" button to search for the OpenOCD executable; it will be in this folder: **c:\Program Files\openocd-2006re93\bin\openocd-pp.exe**".

In the "Working Directory" pane, use the "**Browse File System…**" button to specify "**c:\Program Files\openocd-2006re93\bin**" as the working directory.

In the "Arguments" pane, enter the argument "**-f at91sam7s256-wiggler.cfg**" to specify the OpenOCD configuration file designed for the wiggler.



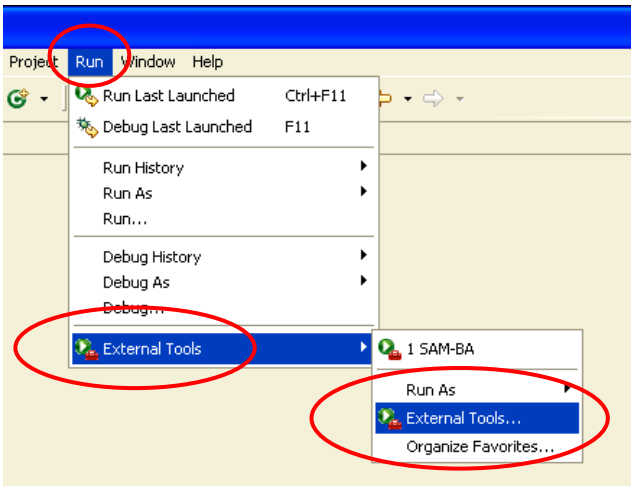
No changes are required to the other tabs in the form (Refresh, Environment, and Common).
Click on "**Apply**" and "**Close**" to register **OpenOCD** as an external tool.

Install OpenOCD as an Eclipse External Tool (ARM-USB-OCD)

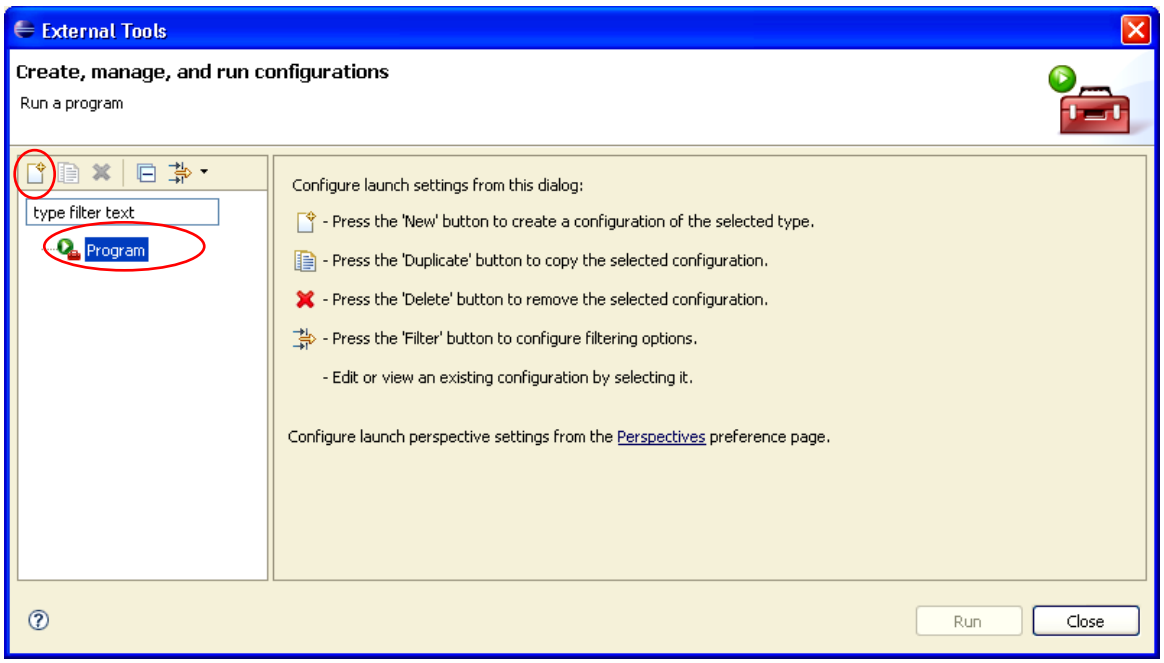
If you have the Amontec USB-based debugger, please skip to the next section. This section is for those with the Olimex ARM-USB-OCD device.

When it's time to debug an application, we must be able to conveniently start the OpenOCD debugger. OpenOCD runs as a daemon; a program that runs in the background waiting for commands to be submitted to it. Eclipse has a very nice "external tool" feature that allows us to add OpenOCD to the RUN pull-down menu.

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



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



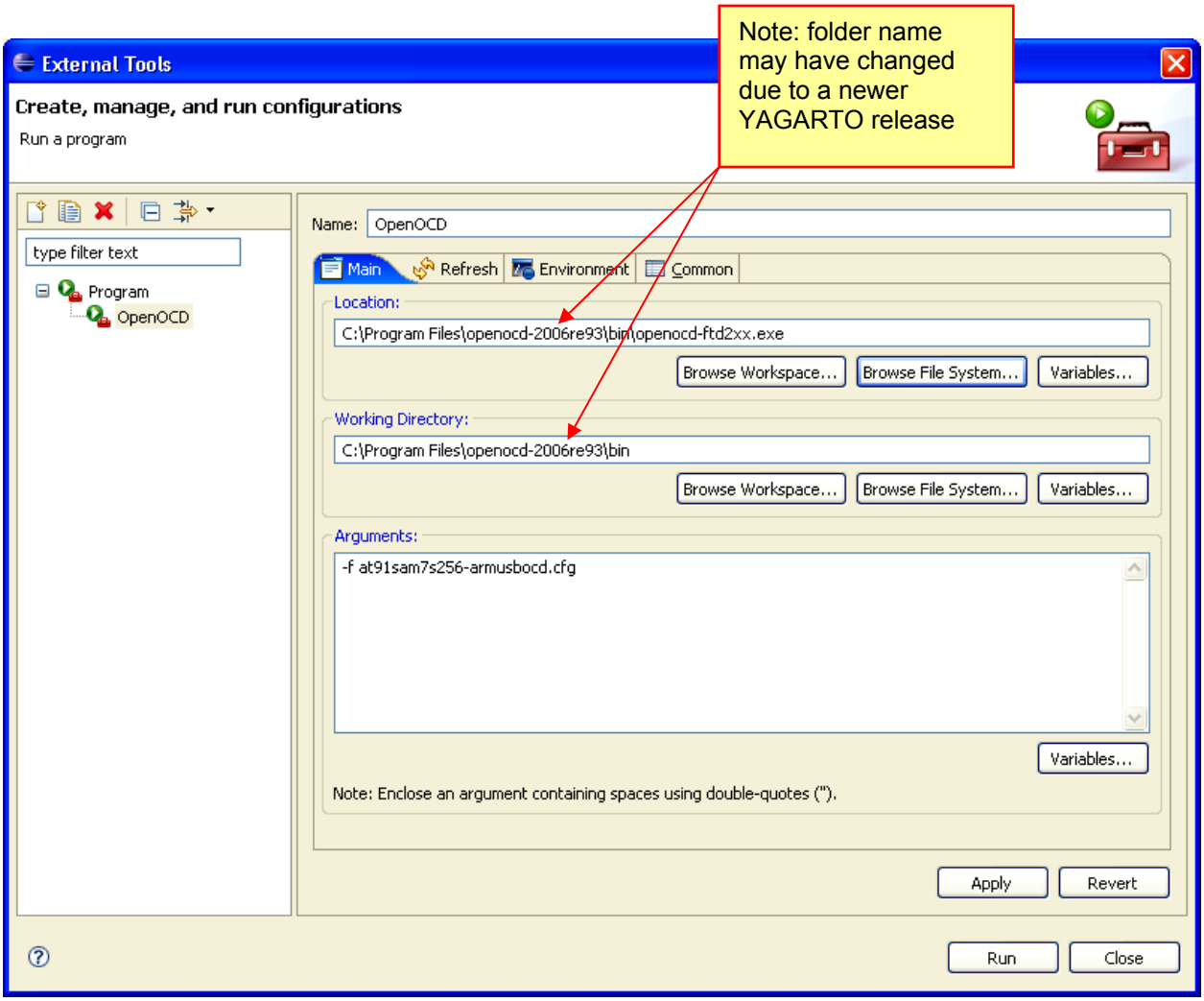
Fill out the "**External Tools**" form exactly as shown below.

There are two versions of OpenOCD; **openocd-pp.exe** supports the parallel-port "wiggler" device (ARM-JTAG from Olimex) while **openocd-ftd2xx.exe** supports the USB-based devices from Amontec and Olimex. In this section we are installing the "USB" version of OpenOCD as an Eclipse external tool.

In the "Location:" pane, use the "**Browse File System…**" button to search for the OpenOCD executable; it will be in this folder: **c:\Program Files\openocd-2006re93\bin\openocd-ftd2xx.exe**".

In the "Working Directory" pane, use the "**Browse File System…**" button to specify "**c:\Program Files\openocd-2006re93\bin**" as the working directory.

In the "Arguments" pane, enter the argument "**-f at91sam7s256-armusbocd.cfg**" to specify the OpenOCD configuration file designed for the Olimex ARM-USB-OCD.

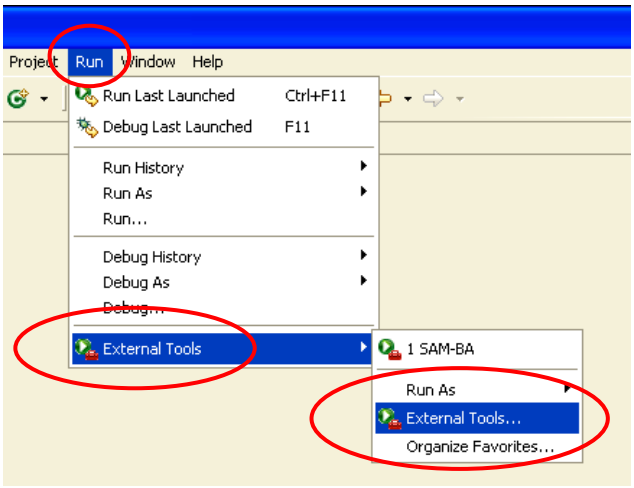


Install OpenOCD as an Eclipse External Tool (JTAGKey)

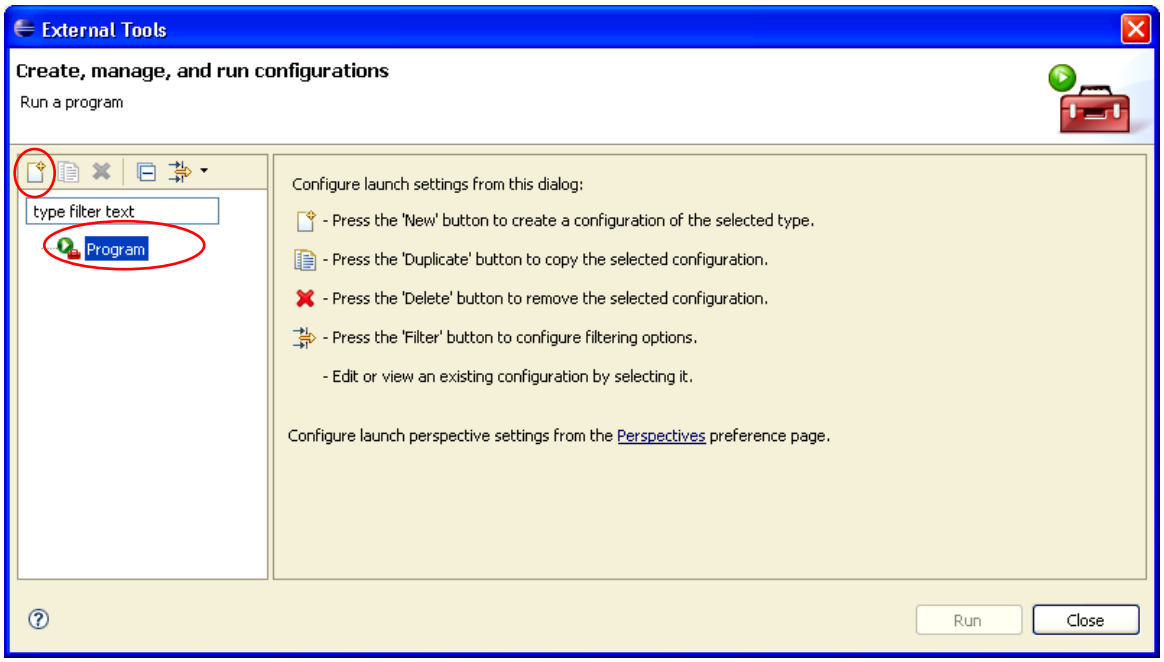
This section is for those with the Amontec JTAGKey device.

When it's time to debug an application, we must be able to conveniently start the OpenOCD debugger. OpenOCD runs as a daemon; a program that runs in the background waiting for commands to be submitted to it. Eclipse has a very nice "external tool" feature that allows us to add OpenOCD to the RUN pull-down menu.

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



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



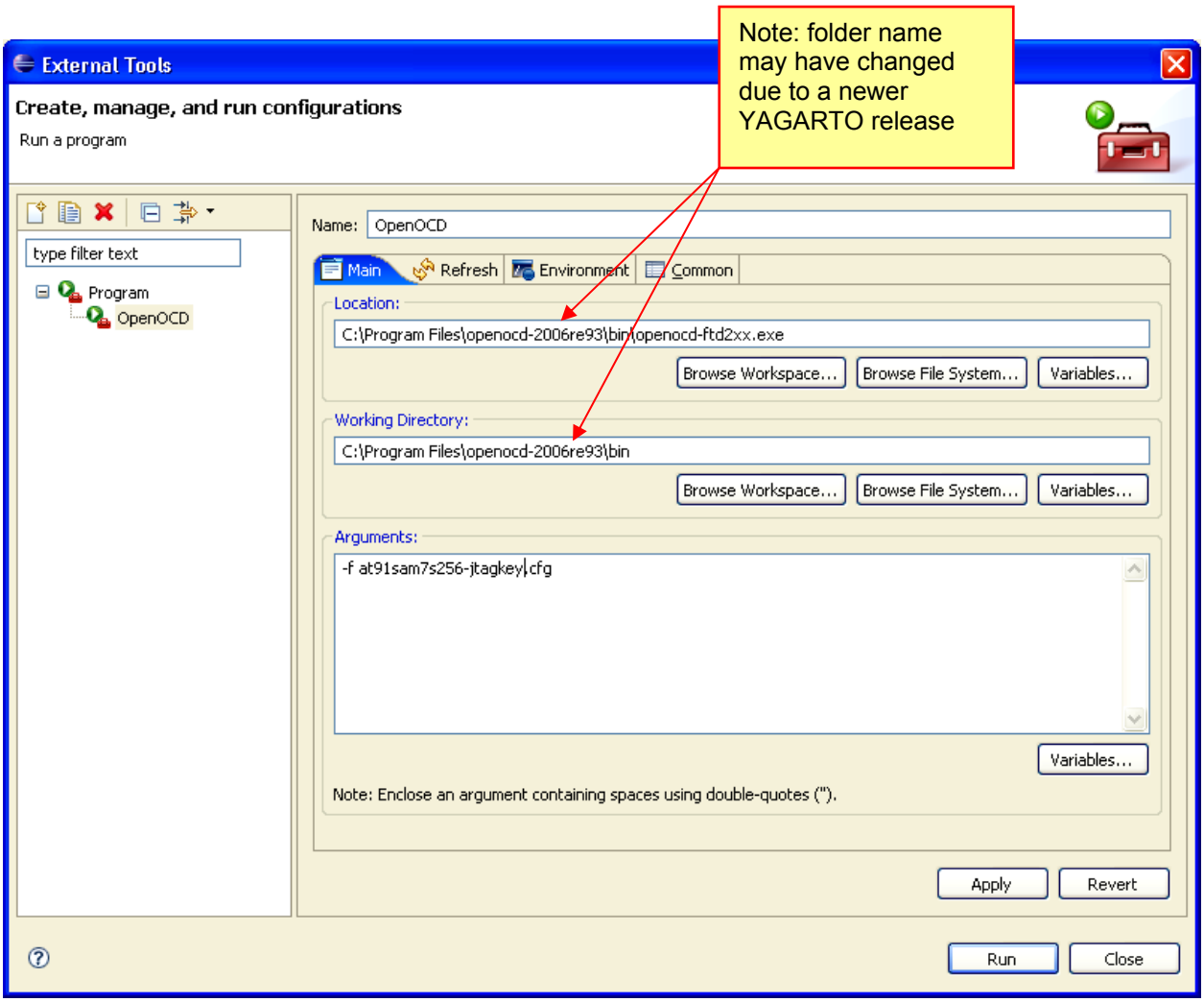
Fill out the "**External Tools**" form exactly as shown below.

There are two versions of OpenOCD; **openocd-pp.exe** supports the parallel-port "wiggler" device (ARM-JTAG from Olimex) while **openocd-ftd2xx.exe** supports the USB-based devices from Amontec and Olimex. In this section we are installing the "USB" version of OpenOCD as an Eclipse external tool.

In the "Location:" pane, use the "**Browse File System…**" button to search for the OpenOCD executable; it will be in this folder: **c:\Program Files\openocd-2006re93\bin\openocd-ftd2xx.exe**".

In the "Working Directory" pane, use the "**Browse File System…**" button to specify "**c:\Program Files\openocd-2006re93\bin**" as the working directory.

In the "Arguments" pane, enter the argument "**-f at91sam7s256-jtagkey.cfg**" to specify the OpenOCD configuration file designed for the Amontec JTAGKey and its little brother, the JTAGKey-Tiny.



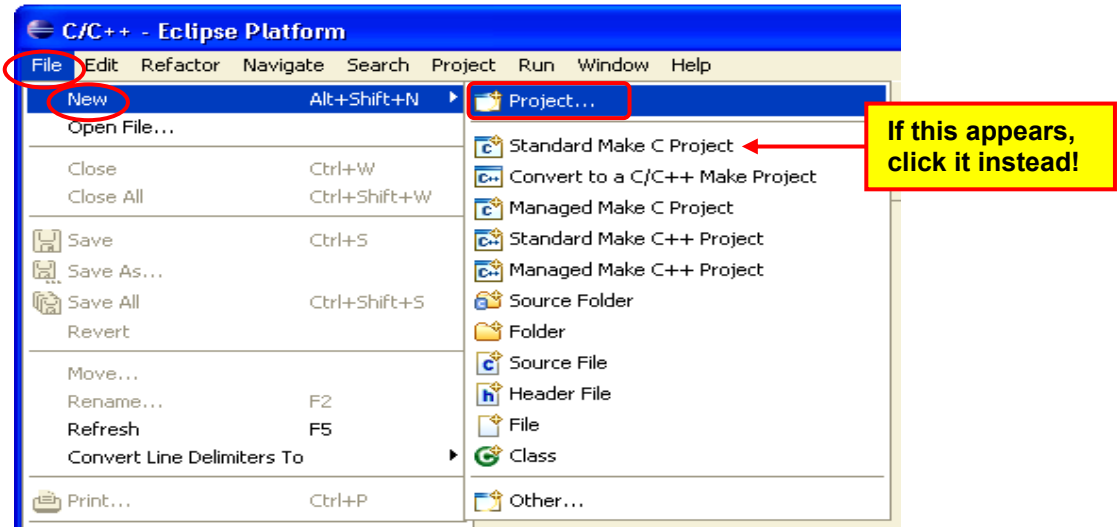
Create an Eclipse Project

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.

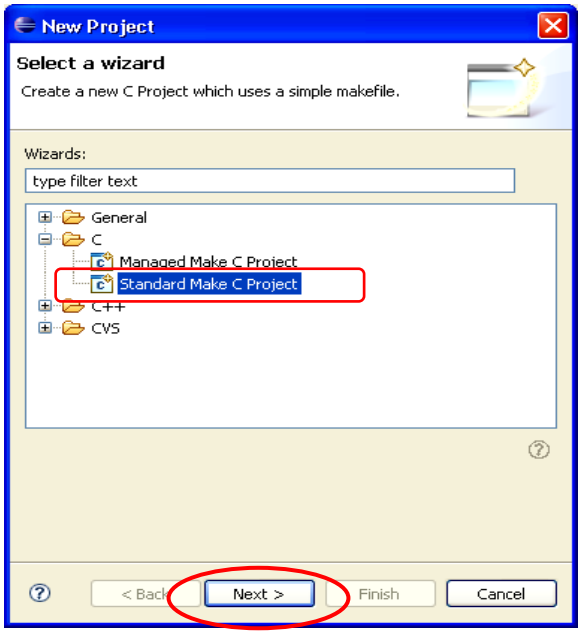
Click on the desktop Eclipse icon to start Eclipse.

Let's jump right in and create an Eclipse C/C++ project. This project will run out of FLASH memory. Specifically the project will blink LED1 in a main program background loop. It will blink LED2 on an IRQ interrupt from onboard Timer1. Finally, if you push switch SW1 it will assert a FIQ interrupt that flashes LED3 and increments a counter. There are also plenty of variables defined for debug practice.

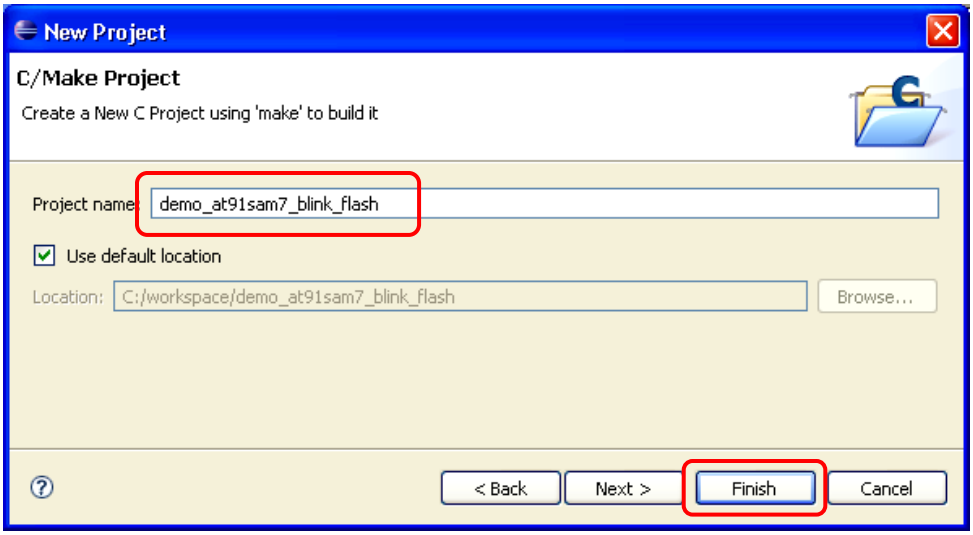
In the **File** pull-down menu, click on "**File – New – Project…**" to get started, as shown below.



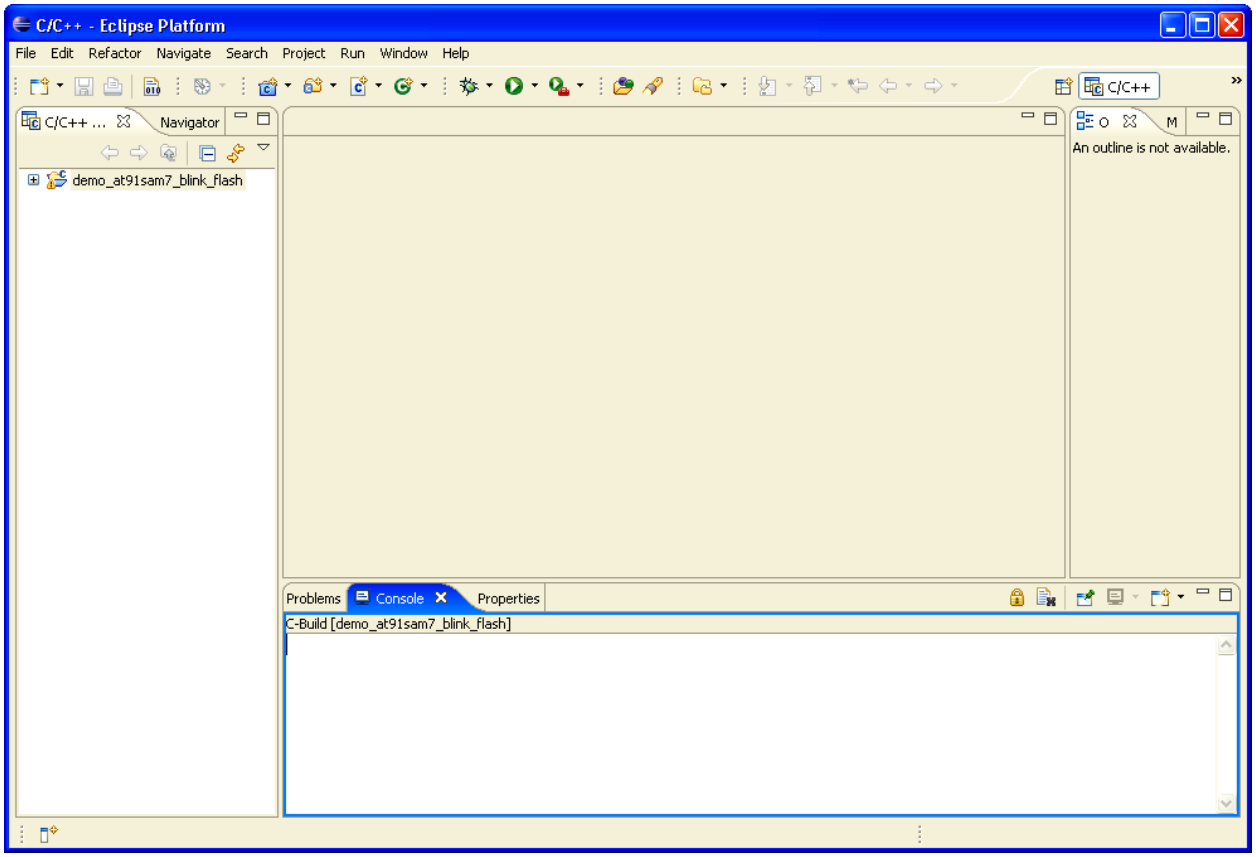
In the "New Project" wizard shown below, expand the C type by clicking on the "**+**" sign and then select "**Standard Make C Project**". Click "**Next**" to continue.



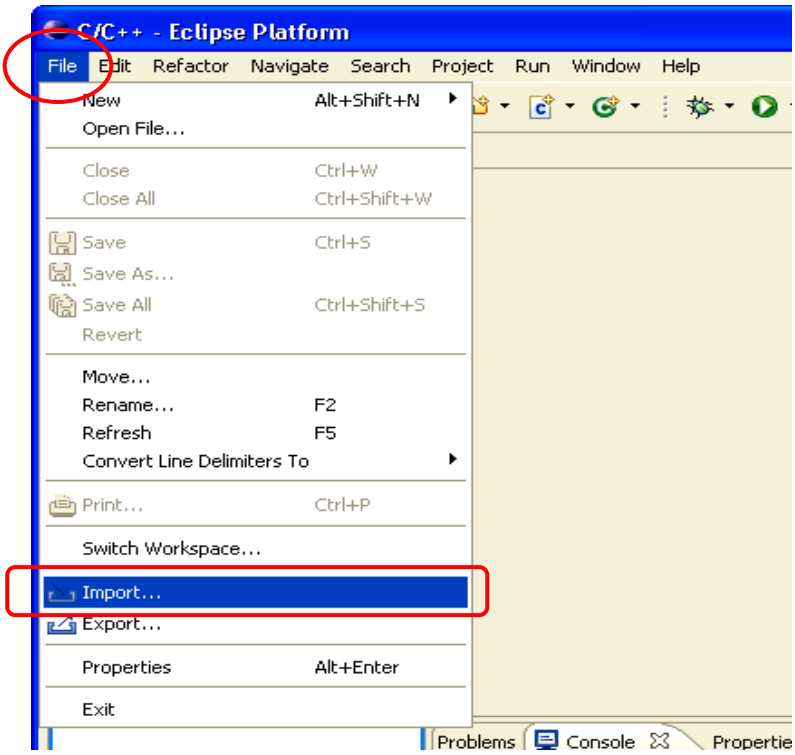
Enter the sample project name “**demo_at91sam7_blink_flash**” into the text window below. Click “**Finish**” to continue.



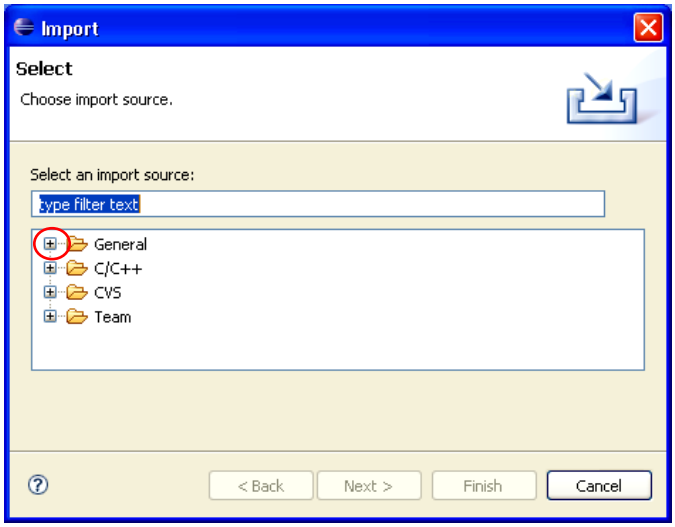
Now the C/C++ perspective shows a valid project, as shown below in the C/C++ Projects view on the left, but there are no source files in that project. Normally you would select “**File – New – Source File**” and enter a file name and start typing. This time, however, we will be importing source files already prepared by the author to demonstrate Eclipse’s features.

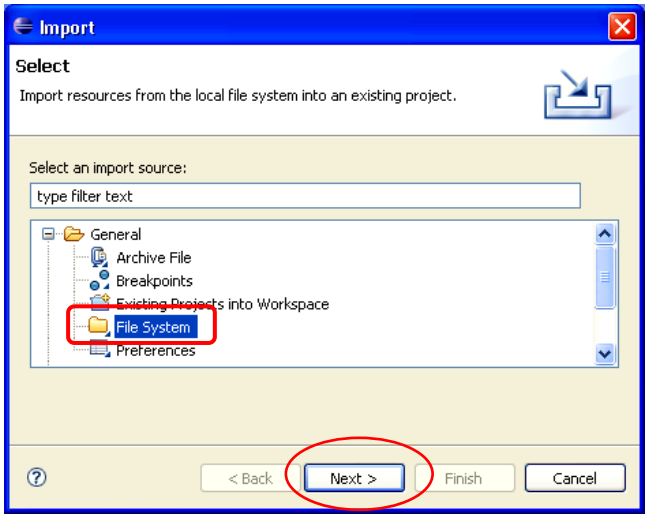


In the Eclipse screen below, click on "**File – Import…**"; this will bring up the file import dialog.



In the import source screen below left, click on the "**+**" symbol adjacent to the "**General**" folder. Click on "**File System**" on the import source screen below to the right. Click "**Next**" to continue.

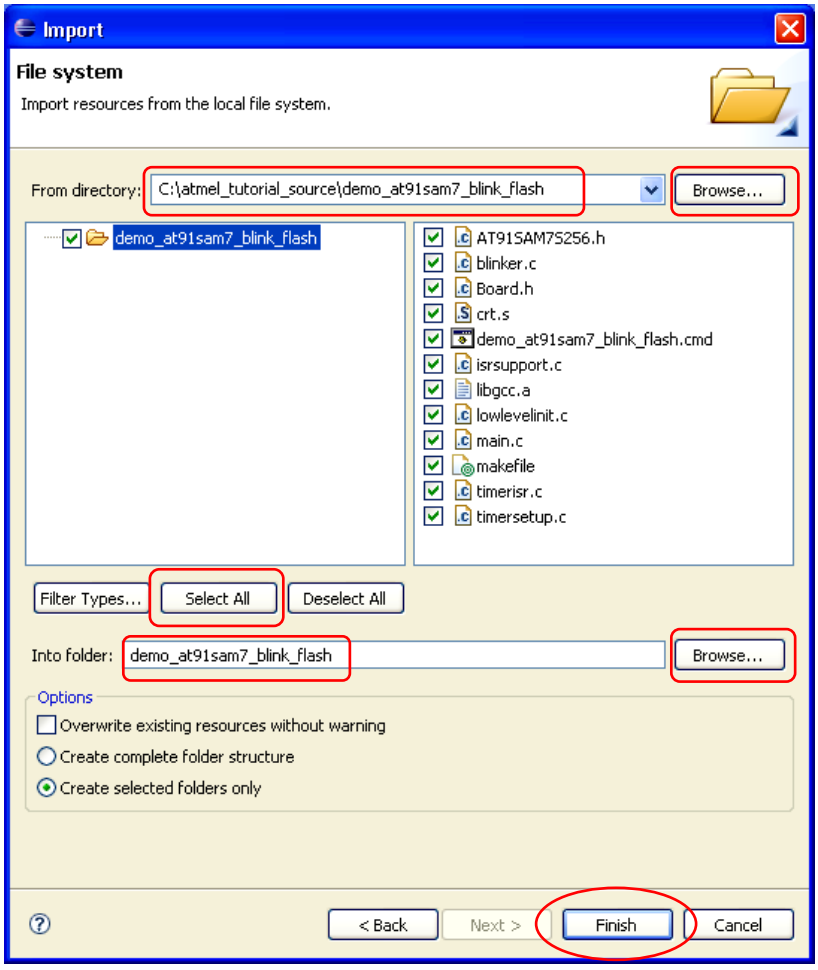




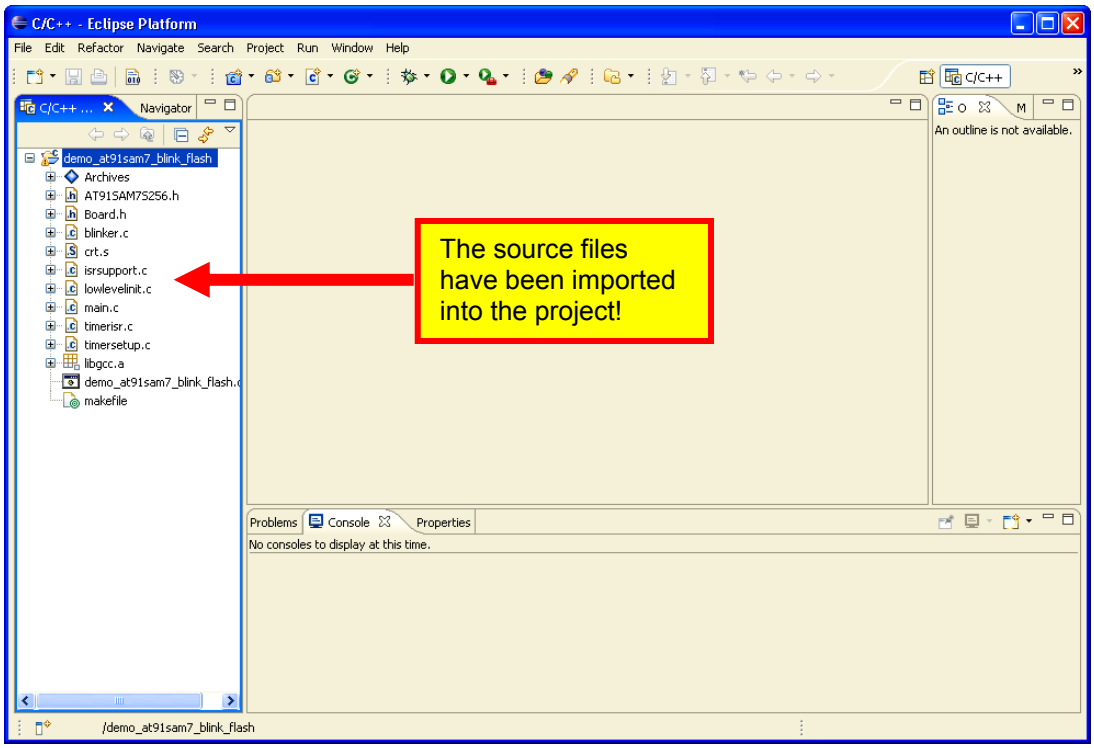
In the import – file system screen below, use the "**Browse**" button associated with the "From directory" text box to search for the sample project to be imported. In this case, it resides in the folder you created earlier: **c:\atmel_tutorial_source\demo_at91sam7_blink_flash.**

Click the "**Select All**" button because we want to import every one of these files.

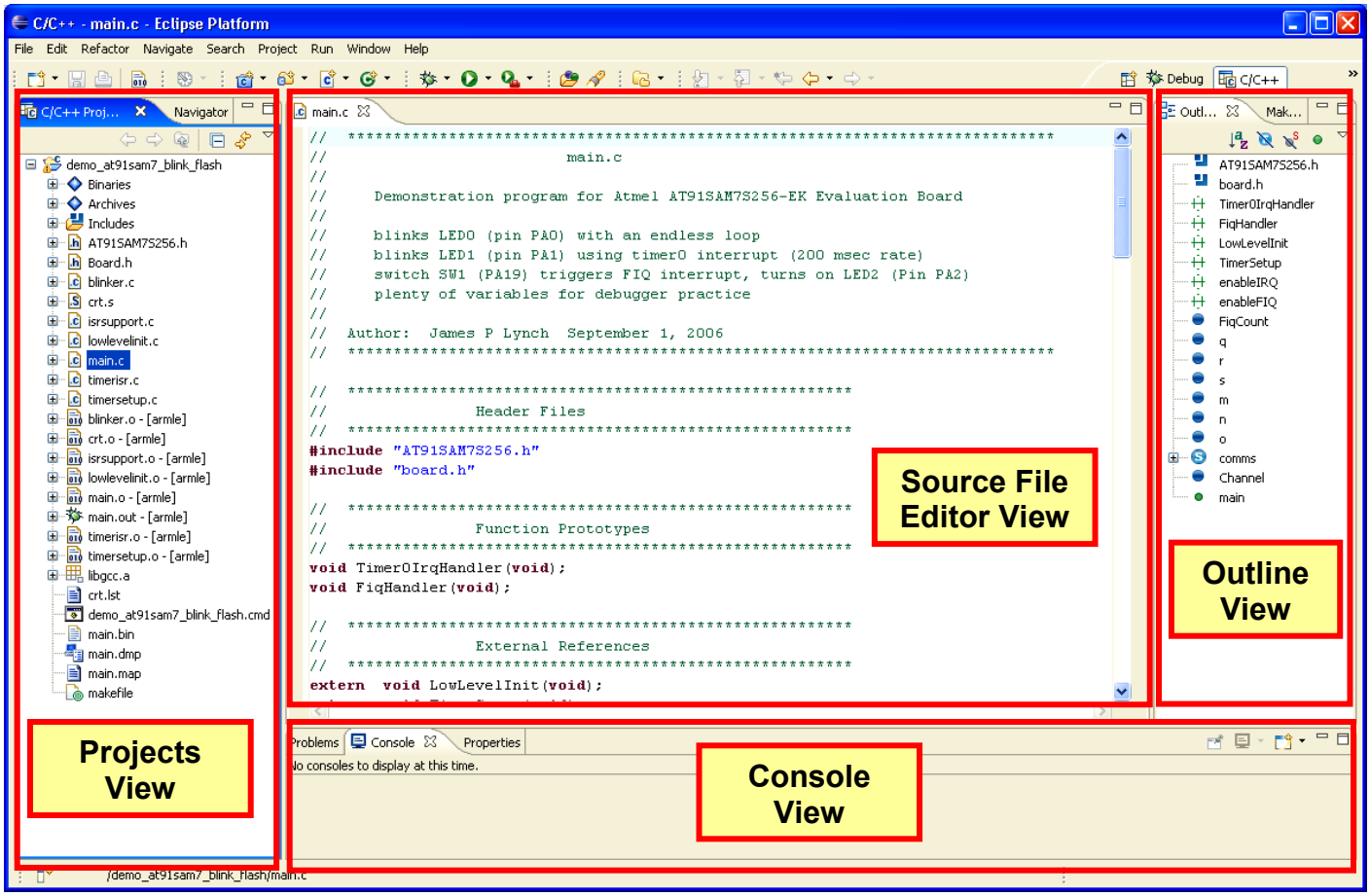
The "Into folder:" text box should already be filled in properly; if not, click the "**Browse**" button to specify the project folder "**demo_at91sam7_blink_flash**". Click "**Finish**" to start the File Import operation.



Now if you expand the **demo_at91sam7_blink_flash** project in the C/C++ Projects view below on the left, you will see that all the source files have been imported into our project.



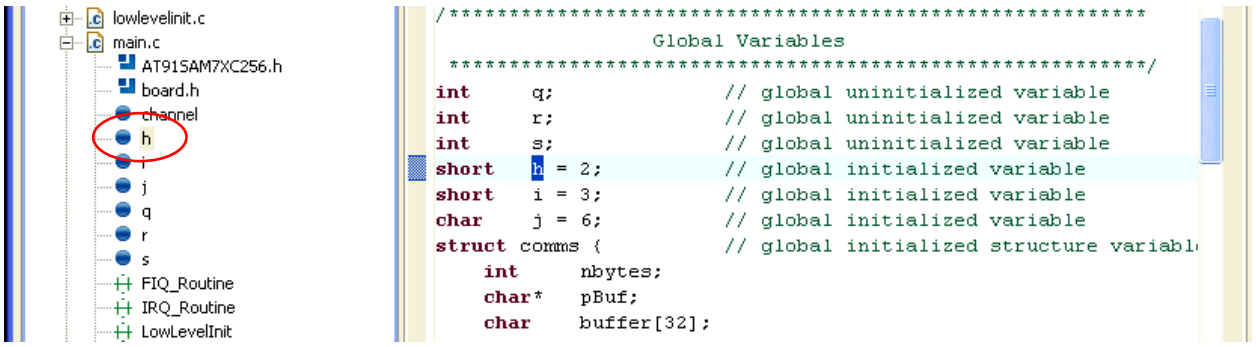
By clicking on the "**+**" sign on the project name in the C/C++ Projects panel on the left, the imported files are revealed. In the Eclipse window below, the **main.c** file has been selected by clicking on it and it thus displays in the source file editor view in the center.



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.

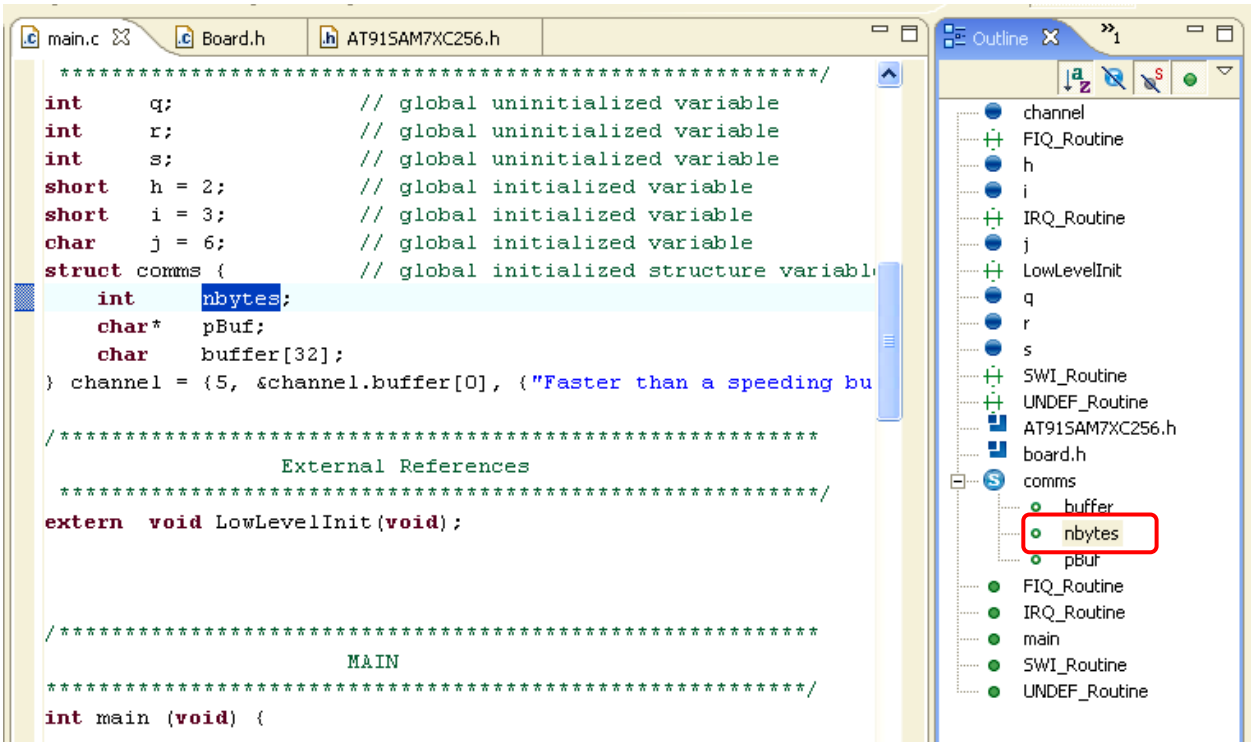
Source modules can be expanded (by clicking on the "+" expander icon) to reveal the variables and functions contained therein. This allows a very quick way to find the definition of a variable in the file.

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.

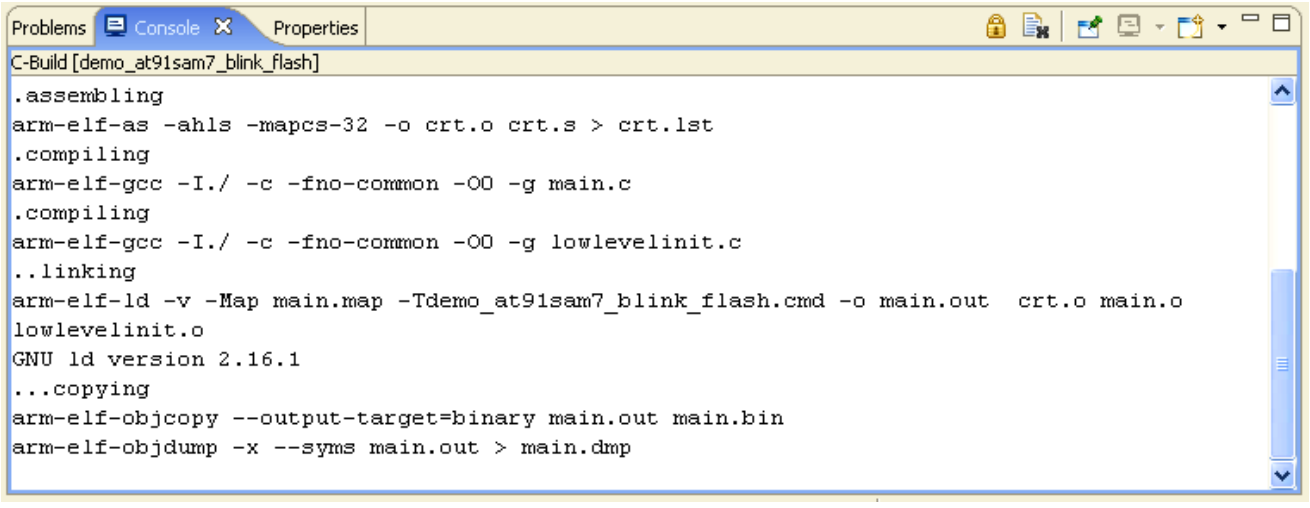


In the "**Outline**" view on the right, any C/C++ file being displayed in the source window in the center will have a tabular list of all important C/C++ elements (such as enumerations, structures, typedefs, variables, etc) to allow quick location of those elements in the source file.

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. The source files designed by the author are heavily annotated and you shouldn't have too much trouble understanding them.

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-EK 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 */
```

BLINKER.C

The blinker routine is entered if the application code crashes due to a prefetch abort interrupt, a data abort interrupt or an undefined instruction abort interrupt. The function enters an endless loop and emits a blink code identifying the source of the abort. The system must be RESET to recover.

```
//  ****************************************************************************
//                              blinker.c
//
//     Endless loop blinks a code for crash analysis
//
//         Inputs:      Code  -  blink code to display
//                      1 = undefined instruction (one blinks ........ long pause)
//                      2 = prefetch abort        (two blinks ........ long pause)
//                      3 = data abort            (three blinks ...... long pause)
```

```
//
//  Author:  James P Lynch  September 20, 2006
//  ****************************************************************************

#include "AT91SAM7S256.h"
#include "board.h"

unsigned long   blinkcount;                     // global variable


void  blinker(unsigned char   code) {
        volatile AT91PS_PIO     pPIO = AT91C_BASE_PIOA;             // pointer to PIO register structure
        volatile unsigned int   j,k;                                // loop counters

        // endless loop
        while (1)  {
                for  (j = code; j != 0; j--) {                      // count out the proper number of blinks
                        pPIO->PIO_CODR = LED1;                      // turn LED1 (DS1) on
                        for (k = 600000; k != 0; k-- );             // wait 250 msec
                        pPIO->PIO_SODR = LED1;                      // turn LED1 (DS1) off
                        for (k = 600000; k != 0; k-- );             // wait 250 msec
                }
                for (k = 5000000; (code != 0) && (k != 0); k-- );   // wait 2 seconds
                blinkcount++;
        }
}
```

CRT.S

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

The interrupt vector table is implemented as branch instructions with one interesting difference; the FIQ interrupt service routine is completely implemented right after the vector table. The designers of the ARM microprocessor purposely placed the FIQ vector last in the vector table for this very purpose. This is the most efficient implementation of a FIQ interrupt. The AT91F_Fiq_Handler routine, coded completely in assembler, turns on LED3 and increments a global variable.

The AT91F_Irq_Handler routine is derived from Atmel documentation and supports nested IRQ interrupts. For a detailed technical discussion of this topic, consult pages 336 – 342 in the book “ARM System Developer’s Guide” by Andrew Sloss et. al. Another great advantage of this technique is that the assembly language nested interrupt handler calls a standard C Language function to do most of the work servicing the IRQ interrupt. You don’t have to deal with the GNU C extensions that support ARM interrupt processing.

The start-up code called by the RESET vector sets up 128 byte stacks for the IRQ and FIQ interrupt modes and places the CPU in “System” mode with the FIQ and IRQ interrupts disabled. System mode operation allows the main( ) program to enable the IRQ and FIQ interrupts after all peripherals have been properly initialized.

The start-up code also initializes all variables that require it and clears all uninitialized variables to zero before branching to the C Language main( ) routine.

```
/* ********************************************************************************************************* */
/*                                        crt.s                                                              */
/*                                                                                                           */
/*                        Assembly Language Startup Code for Atmel AT91SAM7S256                              */
/*                                                                                                           */
/*                                                                                                           */
/*                                                                                                           */
/*                                                                                                           */
/* Author:  James P Lynch      September 20, 2006                                                            */
/* ********************************************************************************************************* */

/* 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, 0x00000080         /* stack for "SVC" supervisor mode is 128 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) */

/* Addresses and offsets of AIC and PIO  */
.set  AT91C_BASE_AIC, 0xFFFFF000         /* (AIC) Base Address                        */
.set  AT91C_PIOA_CODR, 0xFFFFF434        /* (PIO) Clear Output Data Register          */
.set  AT91C_AIC_IVR, 0xFFFFF100          /* (AIC) IRQ Interrupt Vector Register       */
.set  AT91C_AIC_FVR, 0xFFFFF104          /* (AIC) FIQ Interrupt Vector Register       */
.set  AIC_IVR, 256                       /* IRQ Vector Register offset from base above */
.set  AIC_FVR, 260                       /* FIQ Vector Register offset from base above */
.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_Fiq_Handler
.global AT91F_Default_FIQ_handler
.global AT91F_Default_IRQ_handler
.global AT91F_Spurious_handler
.global AT91F_Dabt_Handler
.global AT91F_Pabt_Handler
.global AT91F_Undef_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 0x00000000  */
_vec_undef:     b           AT91F_Undef_Handler         /* Undefined Instruction vector          */
_vec_swi:       b           _vec_swi                    /* Software Interrupt vector             */
_vec_pabt:      b           AT91F_Pabt_Handler          /* Prefetch abort vector                 */
_vec_dabt:      b           AT91F_Dabt_Handler          /* Data abort vector                     */
_vec_rsv:       nop                                     /* Reserved vector                       */
_vec_irq:       b           AT91F_Irq_Handler           /* Interrupt Request (IRQ) vector        */
_vec_fiq:                                               /* Fast interrupt request (FIQ) vector   */

/* ============================================================================ */
/* Function:                    AT91F_Fiq_Handler                               */
/*                                                                              */
/* The FIQ interrupt asserts when switch SW1 is pressed.                        */
/*                                                                              */
/* This simple FIQ handler turns on LED3 (Port PA2). The LED3 will be           */
/* turned off by the background loop in main() thus giving a visual             */
/* indication that the interrupt has occurred.                                  */
/*                                                                              */
/* This FIQ_Handler supports non-nested FIQ interrupts (a FIQ interrupt         */
/* cannot itself be interrupted).                                               */
/*                                                                              */
/* The Fast Interrupt Vector Register (AIC_FVR) is read to clear the            */
/* interrupt.                                                                   */
/*                                                                              */
/* A global variable FiqCount is also incremented.                              */
/*                                                                              */
/* Remember that switch SW1 is not debounced, so the FIQ interrupt may          */
/* occur more than once for a single button push.                               */
/*                                                                              */
/* Programmer: James P Lynch                                                    */
/* ============================================================================ */
AT91F_Fiq_Handler:

/* Adjust LR_irq */
                sub     lr, lr, #4

/* Read the AIC Fast Interrupt Vector register to clear the interrupt */
                ldr     r12, =AT91C_AIC_FVR
                ldr     r11, [r12]

/* Turn on LED3 (write 0x0008 to PIOA_CODR at 0xFFFFF434) */
                ldr     r12, =AT91C_PIOA_CODR
                mov     r11, #0x04
                str     r11, [r12]

/* Increment the _FiqCount variable */
                ldr     r12, =FiqCount
                ldr     r11, [r12]
                add     r11, r11, #1
                str     r11, [r12]

/* Return from Fiq interrupt */
                movs    pc, lr
```

```
/* ========================================================================== */
/*                                _init_reset Handler                         */
/*                                                                            */
/*       RESET vector 0x00000000 branches to here.                            */
/*                                                                            */
/*       _init_reset handler:  creates a stack for each ARM mode.             */
/*                             sets up a stack pointer for each ARM mode.     */
/*                             turns off interrupts in each mode.             */
/*                             leaves CPU in SYS (System) mode.               */
/*                                                                            */
/*                             block copies the initializers to .data section */
/*                             clears the .bss section to zero                */
/*                                                                            */
/*                             branches to main( )                            */
/* ========================================================================== */

_init_reset:
                /* Setup a 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                        */
/*                                                                       */
/* This IRQ_Handler supports nested interrupts (an IRQ interrupt can itself */
/* be interrupted).                                                      */
/*                                                                       */
/* This handler re-enables interrupts and switches to "Supervisor" mode to  */
/* prevent any corruption to the link and IP registers.                  */
/*                                                                       */
/* The Interrupt Vector Register (AIC_IVR) is read to determine the address */
/* of the required interrupt service routine. The ISR routine can be a      */
/* standard C function since this handler minds all the save/restore     */
/* protocols.                                                            */
/*                                                                       */
/*                                                                       */
/* Programmer: James P. Lynch                                            */
/* ===================================================================== */
AT91F_Irq_Handler:

/* Adjust link register and save r0 and lr on IRQ stack */
                sub     lr, lr, #4
                stmfd   sp!, {r0, lr}

/*- SPSR needs to be saved on IRQ stack for nested interrupt */
                mrs     r14, SPSR
                stmfd   sp!, {r14}

/* Read the AIC Interrupt Vector register to clear the interrupt and fetch IRQ handler address */
                ldr     r14, =AT91C_AIC_IVR
                ldr     r0 , [r14]

/* Enable IRQ interrupts and switch to Supervisor mode */
                msr     CPSR_c, #MODE_SVC

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

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

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

/* Disable IRQ interrupts and switch back to 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!, {r14}
                msr     SPSR_cxsf, r14

/* pop r0 and lr from IRQ stack and restore the SPSR_irq; this returns from IRQ interrupt */
                ldmia   sp!, {r0, pc}^




/* ===================================================================== */
/* Function:                    AT91F_Dabt_Handler                       */
/*                                                                       */
/* Entered on Data Abort exception.                                      */
/* Enters blink routine  (3 blinks followed by a pause)                  */
/* processor hangs in the blink loop forever                             */
/*                                                                       */
/* ===================================================================== */
AT91F_Dabt_Handler:     mov     R0, #3
                        b       blinker
```

```
/* ===================================================================== */
/* Function:                    AT91F_Pabt_Handler                       */
/*                                                                       */
/* Entered on Prefetch Abort exception.                                  */
/* Enters blink routine  (2 blinks followed by a pause)                  */
/* processor hangs in the blink loop forever                             */
/*                                                                       */
/* ===================================================================== */
AT91F_Pabt_Handler:      mov     R0, #2
                         b       blinker


/* ===================================================================== */
/* Function:                    AT91F_Undef_Handler                      */
/*                                                                       */
/* Entered on Undefined Instruction exception.                           */
/* Enters blink routine  (1 blinks followed by a pause)                  */
/* processor hangs in the blink loop forever                             */
/*                                                                       */
/* ===================================================================== */
AT91F_Undef_Handler:     mov     R0, #1
                                                    b               blinker


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
```

ISRSUPPORT.C

The isrsupport module is adapted from Pascal Stang's ARMLIB and contains various utility functions to enable/disable interrupts, etc.

```
//  ********************************************************************************************
//
//  File Name    : isrsupport.c
//  Title        : interrupt enable/disable functions
//  Author       : Pascal Stang - Copyright (C) 2004
//  Created      : 2004.05.05
//  Revised      : 2004.07.12
//  Version      : 0.1
//  Target MCU   : ARM processors
//  Editor Tabs  : 4
//
//  NOTE: This code is currently below version 1.0, and therefore is considered
//  to be lacking in some functionality or documentation, or may not be fully
//  tested.  Nonetheless, you can expect most functions to work.
//
//  This code is distributed under the GNU Public License
//  which can be found at http://www.gnu.org/licenses/gpl.txt
//
//  Note from Jim Lynch:
//  This is an abbreviated version of Pascal Stang's processor.c routine from the ARMLIB
//      http://hubbard.engr.scu.edu/embedded/arm/armlib/index.html
//
//  ********************************************************************************************
```

```
#define IRQ_MASK 0x00000080
#define FIQ_MASK 0x00000040
#define INT_MASK (IRQ_MASK | FIQ_MASK)


static inline unsigned __get_cpsr(void)
{
  unsigned long retval;
  asm volatile (" mrs  %0, cpsr" : "=r" (retval) : /* no inputs */  );
  return retval;
}

static inline void __set_cpsr(unsigned val)
{
  asm volatile (" msr  cpsr, %0" : /* no outputs */ : "r" (val)  );
}

unsigned disableIRQ(void)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr(_cpsr | IRQ_MASK);
  return _cpsr;
}

unsigned restoreIRQ(unsigned oldCPSR)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr((_cpsr & ~IRQ_MASK) | (oldCPSR & IRQ_MASK));
  return _cpsr;
}

unsigned enableIRQ(void)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr(_cpsr & ~IRQ_MASK);
  return _cpsr;
}



unsigned disableFIQ(void)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr(_cpsr | FIQ_MASK);
  return _cpsr;
}

unsigned restoreFIQ(unsigned oldCPSR)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr((_cpsr & ~FIQ_MASK) | (oldCPSR & FIQ_MASK));
  return _cpsr;
}

unsigned enableFIQ(void)
{
  unsigned _cpsr;

  _cpsr = __get_cpsr();
  __set_cpsr(_cpsr & ~FIQ_MASK);
  return _cpsr;
}
```

LOWLEVELINIT.S

This function, developed by Atmel Technical Support, initializes the PLL clock system. Some annotation has been corrected by the author.

```
//  ----------------------------------------------------------------------------
//           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 "AT91SAM7S256.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 = 48054841 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 48 054 841
    // 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));



        // PMC Clock Generator PLL Register setup
        //
        // The following settings are used:  DIV = 14
        //                                   MUL = 72
        //                                   PLLCOUNT = 10
        //
        // Main Clock (MAINCK from crystal oscillator) = 18432000 hz (see AT91SAM7-EK schematic)
        // MAINCK / DIV = 18432000/14 = 1316571 hz
        // PLLCK = 1316571 * (MUL + 1) = 1316571 * (72 + 1) = 1316571 * 73 = 96109683 hz
        //
```

```
        // PLLCOUNT = number of slow clock cycles before the LOCK bit is set
        //            in PMC_SR after CKGR_PLLR is written.
        //
        // PLLCOUNT = 10
        //
        // OUT = 0 (not used)
        pPMC->PMC_PLLR = ((AT91C_CKGR_DIV & 14) |
                    (AT91C_CKGR_PLLCOUNT & (10<<8)) |
                    (AT91C_CKGR_MUL & (72<<16)));

        // Wait the startup time (until PMC Status register LOCK bit is set)
        while(!(pPMC->PMC_SR & AT91C_PMC_LOCK));

        // PMC Master Clock (MCK) Register setup
        //
        // CSS  = 3  (PLLCK clock selected)
        //
        // PRES = 1  (MCK = PLLCK / 2) = 96109683/2 = 48054841 hz
        //
        // Note: Master Clock  MCK = 48054841 hz  (this is the CPU clock speed)
        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

The Main() program, designed by the author, provides a background wait loop that flashes LED1 at approximately a 1 Hz rate, flashes LED2 at a 10 Hz rate triggered by a Timer0 IRQ interrupt, and flashes LED3 whenever you push switch SW1 which triggers a FIQ interrupt . There are also plenty of variables for debugging practice.

There are code snippets, currently commented out, that can trigger an ABORT interrupt that results in a crash blinker code that will identify the source of the abort.

```
//  ******************************************************************************
//                                          main.c
//
//     Demonstration program for Atmel AT91SAM7S256-EK Evaluation Board
//
//     blinks LED0 (pin PA0) with an endless loop
//     blinks LED1 (pin PA1) using timer0 interrupt (50 msec rate)
//     switch SW1 (PA19) triggers FIQ interrupt, turns on LED2 (Pin PA2)
//     plenty of variables for debugger practice
//
//  Author:  James P Lynch  September 23, 2006
//  ******************************************************************************

//  ************************************************
//               Header Files
//  ************************************************
#include "AT91SAM7S256.h"
#include "board.h"
```

```
//  *****************************************************
//               Function Prototypes
//  *****************************************************
void Timer0IrqHandler(void);
void FiqHandler(void);

//  *****************************************************
//               External References
//  *****************************************************
extern  void      LowLevelInit(void);
extern  void      TimerSetup(void);
extern  unsigned  enableIRQ(void);
extern  unsigned  enableFIQ(void);

//  *****************************************************
//              Global Variables
//  *****************************************************
unsigned int    FiqCount = 0;           // global uninitialized variable
int             q;                      // global uninitialized variable
int             r;                      // global uninitialized variable
int             s;                      // global uninitialized variable
int             m = 2;                  // global initialized variable
int             n = 3;                  // global initialized variable
int             o = 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"}};


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

        // lots of variables for debugging practice
        int             a, b, c;        // uninitialized variables
        char            d;              // uninitialized variable
        int             w = 1;          // initialized variable
        int             k = 2;          // initialized variable
        static long     x = 5;          // static initialized variable
        static char     y = 0x04;       // static initialized variable
        unsigned long   j;              // loop counter (stack variable)
        unsigned long   IdleCount = 0;  // idle loop blink counter (2x)
        int             *p;             // pointer to 32-bit word
        typedef void    (*FnPtr)(void); // create a "pointer to function" type
        FnPtr           pFnPtr;         // pointer to a function

        const char      *pText = "The rain in Spain";    // initialized string pointer variable

        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();


        // Turn on the peripheral clock for Timer0
        // ---------------------------------------

        // pointer to PMC data structure
        volatile AT91PS_PMC     pPMC = AT91C_BASE_PMC;

        // enable Timer0 peripheral clock
        pPMC->PMC_PCER = (1<<AT91C_ID_TC0);
```

```
// Set up the PIO ports
// --------------------

// pointer to PIO data structure
volatile AT91PS_PIO     pPIO = AT91C_BASE_PIOA;

// PIO Enable Register - allow PIO to control pins P0 - P3 and pin 19
pPIO->PIO_PER = LED_MASK | SW1_MASK;

// PIO Output Enable Register - sets pins P0 - P3 to outputs
pPIO->PIO_OER = LED_MASK;

// PIO Set Output Data Register - turns off the four LEDs
pPIO->PIO_SODR = LED_MASK;

// Select PA19 (pushbutton) to be FIQ function (Peripheral B)
pPIO->PIO_BSR = SW1_MASK;


// Set up the Advanced Interrupt Controller AIC for Timer 0
// --------------------------------------------------------

// pointer to AIC data structure
volatile AT91PS_AIC     pAIC = AT91C_BASE_AIC;

// Disable timer 0 interrupt in AIC Interrupt Disable Command Register
pAIC->AIC_IDCR = (1<<AT91C_ID_TC0);


// Set the TC0 IRQ handler address in AIC Source Vector Register[12]
pAIC->AIC_SVR[AT91C_ID_TC0] = (unsigned int)Timer0IrqHandler;



// Set the interrupt source type and priority in AIC Source Mode Register[12]
pAIC->AIC_SMR[AT91C_ID_TC0] = (AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL | 0x4 );

// Clear the TC0 interrupt in AIC Interrupt Clear Command Register
pAIC->AIC_ICCR = (1<<AT91C_ID_TC0);

// Remove disable timer 0 interrupt in AIC Interrupt Disable Command Register
pAIC->AIC_IDCR = (0<<AT91C_ID_TC0);

// Enable the TC0 interrupt in AIC Interrupt Enable Command Register
pAIC->AIC_IECR = (1<<AT91C_ID_TC0);


// Set up the Advanced Interrupt Controller AIC for FIQ (pushbutton SW1)
// ---------------------------------------------------------------------

// Disable FIQ interrupt in AIC Interrupt Disable Command Register
pAIC->AIC_IDCR = (1<<AT91C_ID_FIQ);

// Set the interrupt source type in AIC Source Mode Register[0]
pAIC->AIC_SMR[AT91C_ID_FIQ] = (AT91C_AIC_SRCTYPE_INT_POSITIVE_EDGE);

// Clear the FIQ interrupt in AIC Interrupt Clear Command Register
pAIC->AIC_ICCR = (1<<AT91C_ID_FIQ);

// Remove disable FIQ interrupt in AIC Interrupt Disable Command Register
pAIC->AIC_IDCR = (0<<AT91C_ID_FIQ);

// Enable the FIQ interrupt in AIC Interrupt Enable Command Register
pAIC->AIC_IECR = (1<<AT91C_ID_FIQ);



// Setup timer0 to generate a 50 msec periodic interrupt
// -----------------------------------------------------

TimerSetup();
```

```
        // enable interrupts
        // -----------------

        enableIRQ();
        enableFIQ();


        // endless background blink loop
        // -----------------------------

        while (1) {
                if  ((pPIO->PIO_ODSR & LED1) == LED1)    // read previous state of LED1
                        pPIO->PIO_CODR = LED1;          // turn LED1 (DS1) on
                else
                        pPIO->PIO_SODR = LED1;          // turn LED1 (DS1) off

                for (j = 2000000; j != 0; j-- );        // wait 1 second

                IdleCount++;                            // count # of times through the idle loop
                pPIO->PIO_SODR = LED3;                  // turn LED3 (DS3) off, in case FIQ turned it on

                // uncomment following four lines to cause a data abort(3 blink code)
                //if  (IdleCount >= 10) {               // let it blink 5 times then crash
                //      p = (int *)0x800000;            // this address doesn't exist
                //      *p = 1234;                      // attempt to write data to invalid address
                //}

                // uncomment following four lines to cause a prefetch abort (two blinks)
                //if  (IdleCount >= 10) {               // let it blink 5 times then crash
                //      pFnPtr = (FnPtr)0x800000;       // this address doesn't exist
                //      pFnPtr();                       // attempt to call a function at a illegal address
                //}
        }
}
```

TIMERISR.C

The Timer0 interrupt service routine is called by the AT91F_Irq_Handler in the crt.s assembly language start-up module. The AT91F_Irq_Handler in the start-up routine supports “nested” IRQ interrupts and thus calls a standard C function do most of the interrupt work.

The C language IRQ support routine below clears the interrupt by reading the TCO status register. It then updates a global variable tickcount; which can be inspected by the debugger. Finally it toggles LED2 to give a visual indication that the timer interrupt is functioning properly.

```
//  ****************************************************************************
//                                          timerisr.c
//
//     Timer 0 Interrupt Service Routine
//
//         entered when Timer0 RC compare interrupt asserts (200 msec period)
//     blinks LED2 (pin PA2)
//
//  Author:  James P Lynch  July 1, 2006
//  ****************************************************************************

#include "AT91SAM7S256.h"
#include "board.h"

unsigned long   tickcount = 0;                              // global variable counts interrupts
```

```
void Timer0IrqHandler (void) {

        volatile AT91PS_TC      pTC = AT91C_BASE_TC0;           // pointer to timer channel 0 register structure
        volatile AT91PS_PIO     pPIO = AT91C_BASE_PIOA;         // pointer to PIO register structure
        unsigned int            dummy;                          // temporary

        dummy = pTC->TC_SR;                                     // read TC0 Status Register to clear it
        tickcount++;                                            // increment the tick count

        if  ((pPIO->PIO_ODSR & LED2) == LED2)
                pPIO->PIO_CODR = LED2;                          // turn LED2 (DS2) on
        else
                pPIO->PIO_SODR = LED2;                          // turn LED2 (DS2) off
}
```

TIMERSETUP.C

All the peripherals on the Atmel AT91SAM7S256 chip are complex; there is no substitute for a careful and thorough study of the Atmel documentation. In this application, we are using the Timer0 counter/timer to count out a 50 msec time interval. The timersetup.c routine shown below is extensively annotated to make it clear how the clock frequencies and count-match values were determined to get the 50 msec repetition rate. The timer counts up, comparing at each tick the current count with the value in the timer compare register C. When the values match, the IRQ interrupt is asserted. Timer 0 has been set up to automatically restart the timer beginning at zero for the next interval.

```
// ********************************************************************************
//                                              timersetup.c
//
//     Purpose:  Set up the 16-bit Timer/Counter
//
//     We will use Timer Channel 0 to develop a 50 msec interrupt.
//
//     The AT91SAM7S-EK board has a 18,432,000 hz crystal oscillator.
//
//     MAINCK  = 18432000 hz
//     PLLCK = (MAINCK / DIV) * (MUL + 1) = 18432000/14 * (72 + 1)
//     PLLCLK = 1316571 * 73 = 96109683 hz
//
//     MCK = PLLCLK / 2 = 96109683 / 2 = 48054841 hz
//
//     TIMER_CLOCK5 = MCK / 1024  = 48054841 / 1024  =  46928 hz
//
//     TIMER_CLOCK5 Period = 1 / 46928  =  21.309239686 microseconds
//
//     A little algebra:  .050 sec = count * 21.3092396896*10**-6
//                        count =  .050 / 21.3092396896*10**-6
//                        count =  2346
//
//
//     Therefore:  set Timer Channel 0 register RC to 9835
//                 turn on capture mode WAVE = 0
//                 enable the clock  CLKEN = 1
//                 select TIMER_CLOCK5  TCCLKS = 100
//                 clock is NOT inverted CLKI = 0
//                 enable RC compare CPCTRG = 1
//                 enable RC compare interrupt  CPCS = 1
//                 disable all the other timer 0 interrupts
//
// Author:  James P Lynch  September 23, 2006
// ********************************************************************************


/*****************************************************
                Header files
 *****************************************************/
#include "AT91SAM7S256.h"
#include "board.h"
```

```
void    TimerSetup(void) {

        //              TC Block Control Register TC_BCR    (read/write)
        //
        //      |-------------------------------------------------------------------|------|
        //      |                                                                   |SYNC  |
        //      |-------------------------------------------------------------------|------|
        //   31                                                                 1    0
        //
        //      SYNC = 0 (no effect)    <===== take  default
        //      SYNC = 1 (generate software trigger for all 3 timer channels simultaneously)
        //
        AT91PS_TCB pTCB = AT91C_BASE_TCB;               // create a pointer to TC Global Register structure
        pTCB->TCB_BCR = 0;                              // SYNC trigger not used




        //              TC Block Mode Register  TC_BMR    (read/write)
        //
        //      |-------------------------------------|-----------|-----------|-----------|
        //      |                                     | TC2XC2S   | TCXC1S    |  TC0XC0S  |
        //      |-------------------------------------|-----------|-----------|-----------|
        //   31                                    5       4   3       2   1         0
        //
        //      TC0XC0S Select = 00 TCLK0 (PA4)
        //                     = 01 none          <===== we select this one
        //                  = 10 TIOA1 (PA15)
        //                  = 11 TIOA2 (PA26)
        //
        //      TCXC1S  Select = 00 TCLK1 (PA28)
        //                     = 01 none          <===== we select this one
        //                  = 10 TIOA0 (PA15)
        //                  = 11 TIOA2 (PA26)
        //
        //      TC2XC2S Select = 00 TCLK2 (PA29)
        //                     = 01 none          <===== we select this one
        //                  = 10 TIOA0 (PA00)
        //                  = 11 TIOA1 (PA26)
        //
        pTCB->TCB_BMR = 0x15;                   // external clocks not used


        //              TC Channel Control Register  TC_CCR    (read/write)
        //
        //      |-----------------------------------|--------------|------------|-----------|
        //      |                                   |   SWTRG      |   CLKDIS   |   CLKENS  |
        //      |-----------------------------------|--------------|------------|-----------|
        //   31                                        2             1           0
        //
        //      CLKEN = 0    no effect
        //      CLKEN = 1    enables the clock     <===== we select this one
        //
        //      CLKDIS = 0   no effect             <===== take  default
        //      CLKDIS = 1   disables the clock
        //
        //  SWTRG = 0    no effect
        //  SWTRG = 1    software trigger aserted counter reset and clock starts   <===== we select this one
        //
        AT91PS_TC pTC = AT91C_BASE_TC0;         // create a pointer to channel 0 Register structure
        pTC->TC_CCR = 0x5;                      // enable the clock      and start it
```

```
//              TC Channel Mode Register  TC_CMR    (read/write)
//
//      |-----------------------------------|------------|---------------|
//      |                                   |  LDRB      |     LDRA      |
//      |-----------------------------------|------------|---------------|
//   31                                  19        18 17           16
//
//      |----------|---------|--------------|------------|---------------|
//      |WAVE = 0  | CPCTRG  |              |  ABETRG    |    ETRGEDG    |
//      |----------|---------|--------------|------------|---------------|
//      15         14     13          11      10      9             8
//
//      |----------|---------|--------------|------------|---------------|
//      |  LDBDIS  | LDBSTOP |   BURST      |  CLKI      |     TCCLKS    |
//      |----------|---------|--------------|------------|---------------|
//       7          6     5           4      3       2             0
//
//  CLOCK SELECTION
//      TCCLKS = 000  TIMER_CLOCK1    (MCK/2    =  24027420 hz)
//               001  TIMER_CLOCK2    (MCK/8    =   6006855 hz)
//               010  TIMER_CLOCK3    (MCK/32   =   1501713 hz)
//               011  TIMER_CLOCK4    (MCK/128  =    375428 hz)
//               100  TIMER_CLOCK5    (MCK/1024 =     46928 hz)   <===== we select this one
//               101  XC0
//               101  XC1
//               101  XC2
//
```

```
//  CLOCK INVERT
//      CLKI    = 0   counter incremented on rising clock edge    <===== we select this one
//      CLKI    = 1   counter incremented on falling clock edge
//
//  BURST SIGNAL SELECTION
//  BURST  = 00  clock is not gated by any external system   <===== take  default
//           01  XC0 is anded with the clock
//           10  XC1 is anded with the clock
//           11  XC2 is anded with the clock
//
//  COUNTER CLOCK STOPPED WITH RB LOADING
//  LDBSTOP  = 0  counter clock is not stopped when RB loading occurs   <===== take  default
//           = 1  counter clock is stopped when RB loading occur
//
//  COUNTER CLOCK DISABLE WITH RB LOADING
//  LDBDIS   = 0  counter clock is not disabled when RB loading occurs   <===== take  default
//           = 1  counter clock is disabled when RB loading occurs
//
//  EXTERNAL TRIGGER EDGE SELECTION
//  ETRGEDG  = 00  (none)              <===== take default
//             01  (rising edge)
//             10  (falling edge)
//             11  (each edge)
//
//  TIOA OR TIOB EXTERNAL TRIGGER SELECTION
//  ABETRG   = 0  (TIOA is used)      <===== take default
//             1  (TIOB is used)
//
//  RC COMPARE TRIGGER ENABLE
//  CPCTRG   = 0  (RC Compare has no effect on the counter and its clock)
//             1  (RC Compare resets the counter and starts the clock)      <===== we select this one
//
//  WAVE
//  WAVE     = 0  Capture Mode is enabled     <===== we select this one
//             1  Waveform Mode is enabled
//
//  RA LOADING SELECTION
//  LDRA  = 00 none)                  <===== take default
//          01 (rising edge of TIOA)
//          10 (falling edge of TIOA)
//          11 (each edge of TIOA)
//
```

```
//  RB LOADING SELECTION
//  LDRB  = 00 (none)                   <===== take default
//          01 (rising edge of TIOA)
//          10 (falling edge of TIOA)
//          11 (each edge of TIOA)
//
pTC->TC_CMR = 0x4004;           // TCCLKS = 1 (TIMER_CLOCK5)
                                // CPCTRG = 1 (RC Compare resets the counter and restarts the clock)
                                // WAVE   = 0 (Capture mode enabled)


//              TC Register C     TC_RC   (read/write)   Compare Register 16-bits
//
//      |----------------------------------|----------------------------------------|
//      |           not used               |               RC                       |
//      |----------------------------------|----------------------------------------|
//   31                               16 15                                      0
//
//      Timer Calculation:   What count gives 50 msec time-out?
//
//     TIMER_CLOCK5 = MCK / 1024  = 48054841 / 1024  =  46928 hz
//
//     TIMER_CLOCK5 Period = 1 / 46928  =  21.309239686 microseconds
//
//     A little algebra:  .050 sec = count * 21.3092396896*10**-6
//                        count =  .050 / 21.3092396896*10**-6
//                        count =  2346
//
pTC->TC_RC = 2346;




//              TC Interrupt Enable Register  TC_IER    (write-only)
//
//
//      |------------|-------|-------|-------|-------|--------|--------|--------|--------|
//      |            | ETRGS | LDRBS | LDRAS | CPCS  | CPBS   |  CPAS  |  LOVRS |  COVFS |
//      |------------|-------|-------|-------|-------|--------|--------|--------|--------|
//   31         8    7       6       5       4        3        2        1        0
//
//  COVFS    = 0  no effect    <===== take  default
//             1  enable counter overflow interrupt
//
//  LOVRS    = 0  no effect    <===== take  default
//             1  enable load overrun interrupt
//
//  CPAS     = 0  no effect    <===== take  default
//             1  enable RA compare interrupt
//
//  CPBS     = 0  no effect    <===== take  default
//             1  enable RB compare interrupt
//
//  CPCS     = 0  no effect
//             1  enable RC compare interrupt    <===== we select this one
//
//  LDRAS    = 0  no effect    <===== take  default
//             1  enable RA load interrupt
//
//  LDRBS    = 0  no effect    <===== take  default
//             1  enable RB load interrupt
//
//  ETRGS    = 0  no effect    <===== take  default
//             1  enable External Trigger interrupt
//
pTC->TC_IER = 0x10;                             // enable RC compare interrupt
```

```
        //              TC Interrupt Disable Register  TC_IDR    (write-only)
        //
        //
        //      |------------|-------|-------|-------|-------|--------|--------|--------|--------|
        //      |            | ETRGS | LDRBS | LDRAS | CPCS  | CPBS   |  CPAS  |  LOVRS |  COVFS |
        //      |------------|-------|-------|-------|-------|--------|--------|--------|--------|
        //   31         8    7       6       5       4        3        2        1        0
        //
        //  COVFS    = 0  no effect
        //             1  disable counter overflow interrupt    <===== we select this one
        //
        //  LOVRS    = 0  no effect
        //             1  disable load overrun interrupt    <===== we select this one
        //
        //  CPAS     = 0  no effect
        //             1  disable RA compare interrupt    <===== we select this one
        //
        //  CPBS     = 0  no effect
        //             1  disable RB compare interrupt    <===== we select this one
        //
        //  CPCS     = 0  no effect    <===== take  default
        //             1  disable RC compare interrupt
        //
        //  LDRAS    = 0  no effect
        //             1  disable RA load interrupt    <===== we select this one
        //
        //  LDRBS    = 0  no effect
        //             1  disable RB load interrupt    <===== we select this one
        //
        //  ETRGS    = 0  no effect
        //             1  disable External Trigger interrupt    <===== we select this one
        //
        pTC->TC_IER = 0xEF;                             // disable all except RC compare interrupt
}
```

DEMO_AT91SAM7_BLINK_FLASH.CMD

The Linker command script instructs the linker where to place the various parts of your program into FLASH and RAM.

The layout of memory and the subsequent specification of the TOS (top of stack) are critical. In the snippet below we specify 256K of FLASH starting at address 0x00000000 and 64K of RAM starting at address 0x00200000. Given the RAM starting at 0x00200000 and being 65536 bytes in length, the Top of Stack is placed 4 bytes from the end of RAM at 0x0020FFFC. The specification of the "top of stack" (_stack_end = 0x20FFFC) is used by the start-up routine, crt.s, to create the stacks for the various interrupt modes. The statements extracted below are the ones that you would modify when moving to a different memory layout.

```
/* specify the AT91SAM7S256s  */
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 = 0x20FFFC;
```

It's a good idea to remind ourselves that the executable code (.text section) goes into FLASH memory and therefore the FLASH must be programmed before attempting execution. I can't tell you how many times the author has built an application and forgotten to program the FLASH with the new code before starting the debugger.

```
/* ********************************************************************************************************
     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 0x00000000


     The top of the stack (_stack_end) is (last_byte_of_ram +1) - 4

     Therefore:   _stack_end = (0x00020FFFF + 1) - 4  =  0x00021000 - 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          |
                .         |                                 |
                .         |---------------------------------|0x0020FFEC
                .         |                                 |
                .         |    ABT Stack  16 bytes          |
                .         |                                 |
                .         |---------------------------------|0x0020FFDC
                .         |                                 |
                .         |                                 |
                .         |    FIQ Stack  128 bytes         |
                .         |                                 |
                .         |                                 |
               RAM        |---------------------------------|0x0020FF5C
                .         |                                 |
                .         |                                 |
                .         |    IRQ Stack  128 bytes         |
                .         |                                 |
                .         |                                 |
                .         |---------------------------------|0x0020FEDC
                .         |                                 |
                .         |    SVC Stack  16 bytes          |
                .         |                                 |
                .         |---------------------------------|0x0020FECC
                .         |                                 |
                .         |     stack area for user program |
                .         |                                 |
                .         |                                 |
                .         |                                 |
                .         |          free ram               |
                .         |                                 |
                .         |.................................|0x002006D8 <---------- _bss_end
                .         |                                 |
                .         |  .bss   uninitialized variables |
                .         |.................................|0x002006D0 <---------- _bss_start, _edata
                .         |                                 |
                .         |  .data  initialized variables   |
                .         |                                 |
                .-------->|_________________________________|0x00200000
```

```
             .-------->|---------------------------------|0x00100000
             .         |                                 |
             .         |                                 |
             .         |        free flash               |
             .         |                                 |
             .         |                                 |
             .         |---------------------------------|0x000006D0 <---------- _bss_start, _edata
             .         |                                 |
             .         |  .data  initialized variables   |
             .         |                                 |
             .         |---------------------------------|0x000006C4 <---------- _etext
             .         |                                 |
             .         |           C code                |
             .         |                                 |
             .         |                                 |
             .         |---------------------------------|0x00000118  main()
             .         |                                 |
             .         |    Startup Code  (crt.s)        |
             .         |         (assembler)             |
             .         |                                 |
             .         |---------------------------------|0x00000020
             .         |                                 |
             .         | Interrupt Vector Table          |
             .         |          32 bytes               |
             .-------->|---------------------------------|0x00000000 _vec_reset

    Author:  James P. Lynch

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

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

/* specify the AT91SAM7S256s  */
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 = 0x20FFFC;

/* 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 initializer 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

The makefile was kept intentionally simple so that a beginner need only read the first chapter of the "GNU Make" document by Richard Stallman and Roland McGrath to understand everything in the makefile.

The makefile is composed of two parts; the part that assembles, compiles and links your program to create a .bin file that you can load into flash, and a special "program" target that is used to independently program the FLASH on chip memory using the OpenOCD JTAG utility.

The essential component of a Makefile is the "rule". The rule is composed of a target file and dependent files on a single line. If any of the dependent files are newer than the target file, then the commands below are executed. The one or more commands MUST be indented by a TAB character (this little nuance beleaguers many novices). For example:

```
main.o:   main.c    AT91SAM7S256.h   board.h
          arm-elf-gcc   -I./  -c  -fno-common  -O0  –g   main.c
```

In the example rule above, if you edit either the main.c source file or the AT91SAM7S256.h or the board.h include files, they will then be "newer" than the main.o target file. Therefore, the commands below must be executed. The single compile command shown updates the target object file so that the target and the dependent files now have the same creation date.

The Make utility checks the rules from top to bottom and this has the effect of only compiling those source files that are "out of date".

If you click the Eclipse "**Project - Clean**" pull-down menu option, the "clean" target below is performed first followed by the "all" target. This has the effect of recompiling everything since all the objects and binary files are erased first.

If you click the Eclipse "**Project – Build Project**" pull-down menu option, the "all" target is performed and only those source files that are out-of-date are recompiled. In a large application with many source files, this is a real convenience and time saver.

Note that the "clean" and "all" targets are NOT files. In this case, Make will only process them unless you specifically direct it to do so (make clean all or make all). This also explains why in scanning from top to bottom during a "make all", make stops when it encounters the "program" target (used to program the FLASH). This is explained in more detail in a section to follow.

The ARM7 architecture supports two instruction sets, ARM and THUMB. The ARM instruction set is composed of 32-bit instructions and is very fast (most instructions execute in a single clock cycle). The THUMB instruction set is composed of 16-bit instructions that require less memory space but take longer to execute. To keep this tutorial simple, we've set up the project exclusively for the ARM 32-bit instruction set. If you would like to see a good example of mixing ARM and THUMB instruction sets in an ARM7 application, take a look at Richard Barry's FreeRTOS kernel at www.freertos.com.

```
# ****************************************************************
# *     Makefile for Atmel AT91SAM7S256 - flash execution        *
# *                                                              *
# *                                                              *
# * James P Lynch  September 22, 2006                            *
# ****************************************************************

NAME   = demo_at91sam7_blink_flash

# variables
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

OBJECTS = crt.o  main.o timerisr.o timersetup.o isrsupport.o lowlevelinit.o blinker.o




# make target called by Eclipse (Project -> Clean ...)
clean:
        -rm $(OBJECTS) crt.lst main.lst main.out main.bin main.hex main.map main.dmp


# make target called by Eclipse  (Project -> Build Project)
all:  main.out
        @ echo "...copying"
        $(CP) $(CPFLAGS) main.out main.bin
        $(OD) $(ODFLAGS) main.out > main.dmp

main.out: $(OBJECTS) demo_at91sam7_blink_flash.cmd
        @ echo "..linking"
        $(LD) $(LFLAGS) -o main.out $(OBJECTS) libgcc.a

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

main.o: main.c  AT91SAM7S256.h board.h
        @ echo ".compiling main.c"
        $(CC) $(CFLAGS) main.c

timerisr.o: timerisr.c  AT91SAM7S256.h board.h
        @ echo ".compiling timerisr.c"
        $(CC) $(CFLAGS) timerisr.c



lowlevelinit.o: lowlevelinit.c AT91SAM7S256.h board.h
        @ echo ".compiling lowlevelinit.c"
        $(CC) $(CFLAGS) lowlevelinit.c

timersetup.o: timersetup.c AT91SAM7S256.h board.h
        @ echo ".compiling timersetup.c"
        $(CC) $(CFLAGS) timersetup.c



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

blinker.o: blinker.c AT91SAM7S256.h board.h
        @ echo ".compiling blinker.c"
        $(CC) $(CFLAGS) blinker.c



# ********************************************************************************************
#                            FLASH PROGRAMMING      (using OpenOCD and Amontec JTAGKey)
#
# Alternate make target for flash programming only
#
# You must create a special Eclipse make target (program) to run this part of the makefile
# (Project -> Create Make Target...  then set the Target Name and Make Target to "program")
#
# OpenOCD is run in "batch" mode with a special configuration file and a script file containing
# the flash commands. When flash programming completes, OpenOCD terminates.
#
# Note that the make file below creates the script file of flash commands "on the fly"
#
# Programmers: Martin Thomas, Joseph M Dupre, James P Lynch
# ********************************************************************************************
```

```
# specify output filename here (must be *.bin file)
TARGET = main.bin

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = 'c:\Program Files\openocd-2006re93\bin\'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
#OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe


# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-wiggler-flash-program.cfg
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-jtagkey-flash-program.cfg
OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-armusbocd-flash-program.cfg

# specify the name and folder of the flash programming script file
OPENOCD_SCRIPT = c:\temp\temp.ocd

# program the AT91SAM7S256 internal flash memory
program: $(TARGET)
        @echo "Preparing OpenOCD script..."
        @cmd /c 'echo wait_halt > $(OPENOCD_SCRIPT)'
        @cmd /c 'echo armv4_5 core_state arm >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo flash write 0 $(TARGET) 0x0 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo mww 0xfffffd08 0xa5000401 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo reset >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo shutdown >> $(OPENOCD_SCRIPT)'
        @echo "Flash Programming with OpenOCD..."
        $(OPENOCD) -f $(OPENOCD_CFG)
        @echo "Flash Programming Finished."
```

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 already done in the makefile above; note that we modified the CPFLAGS macro substitution as follows:

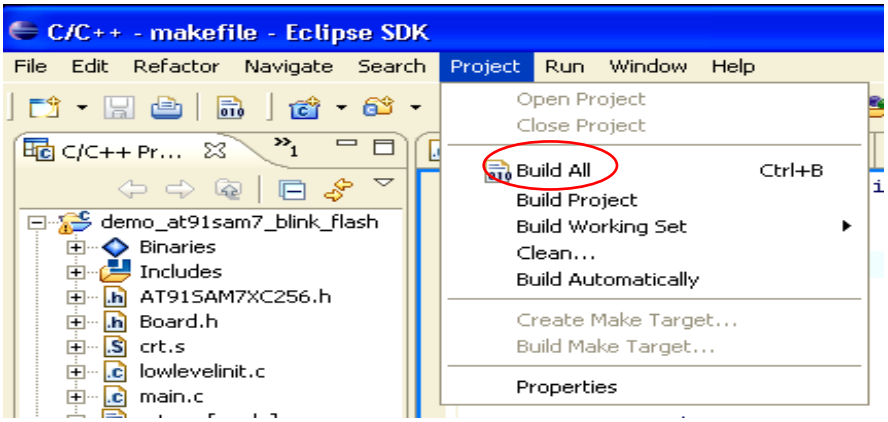
CFLAGS = -I./ -c -fno-common **-O0** –g Where the switch: -O0 means no optimization.

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

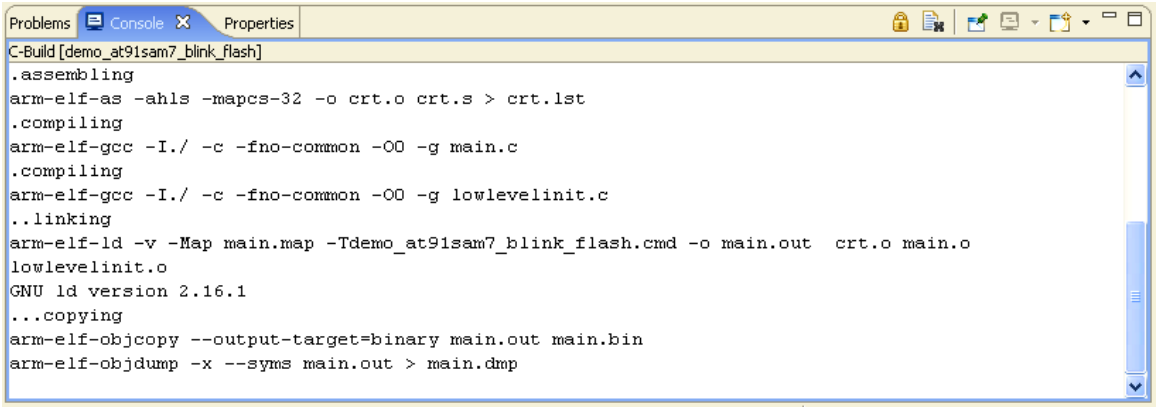
The “**Project**” pull-down menu has several options to build the application. “**Clean**” will erase all object, list, map, and output and dump files, thus forcing Eclipse to compile, assemble and link every file. This may be time-consuming in a large project with many files. “**Build All**” will only 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 OpenOCD flash programming Utility. The makefile also creates a “**main.out**” file that has symbol information; this is used in debugging and also is “loaded” into RAM when you create a RAM-based executable.

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.

Using OpenOCD to Program the FLASH memory

OpenOCD is a utility that converts Eclipse/GDB remote serial protocol to the JTAG protocol supported by the AT91SAM7 on chip debugging unit. In this role, it acts as a “daemon” which is a program that operates in the background, waiting for you to supply a command. We will see plenty of examples of that when we run the debugger shortly.

The other role for OpenOCD is to program the on chip FLASH using the JTAG. In this role, OpenOCD is run in a “batch” mode where the program is executed with a special configuration file and a “script” file with the flash programming commands.

The OpenOCD configuration files to support flash programming on an Atmel AT91SAM7S are as follows. If you are interested in understanding every nuance of these files, refer to the OpenOCD Wiki here:

http://openfacts.berlios.de/index-en.phtml?title=Open_On-Chip_Debugger

It’s worth mentioning that the non-flash-programming versions of these configuration files are simply the part that’s above the FLASH programming commands. When FLASH programming is completed, OpenOCD is automatically terminated.

OpenOCD Configuration File for Wiggler (FLASH programming version)

```
#define our ports
telnet_port 4444
gdb_port 3333

#commands specific to the Wiggler
interface parport
parport_port 0x378
parport_cable wiggler
jtag_speed 0
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#reset_config <signals> [combination] [trst_type] [srst_type]
reset_config srst_only srst_pulls_trst

#jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
jtag_device 4 0x1 0xf 0xe

#daemon_startup <'attach'|'reset'>
daemon_startup reset

#target <type> <endianess> <reset_mode> <jtag#> [variant]
target arm7tdmi little run_and_init 0 arm7tdmi_r4

#run_and_halt_time <target#> <time_in_ms>
run_and_halt_time 0 30

# commands below are specific to AT91sam7 Flash Programming
# ---------------------------------------------------------

#target_script specifies the flash programming script file
target_script 0 reset c:\temp\temp.ocd

#working_area <target#> <address> <size> <'backup'|'nobackup'>
working_area 0 0x40000000 0x4000 nobackup

#flash bank at91sam7 0 0 0 0 <target#>
flash bank at91sam7 0 0 0 0 0
```

OpenOCD Configuration File for JTAGKey (FLASH programming version)

```
#define our ports
telnet_port 4444
gdb_port 3333

#commands specific to the Amontec JTAGKey
interface ft2232
ft2232_device_desc "Amontec JTAGkey A"
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
jtag_speed 2
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#reset_config <signals> [combination] [trst_type] [srst_type]
reset_config srst_only srst_pulls_trst

#jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
jtag_device 4 0x1 0xf 0xe

#daemon_startup <'attach'|'reset'>
daemon_startup reset

#target <type> <endianess> <reset_mode> <jtag#> [variant]
target arm7tdmi little run_and_init 0 arm7tdmi_r4

#run_and_halt_time <target#> <time_in_ms>
run_and_halt_time 0 30

# commands below are specific to AT91sam7 Flash Programming
# ---------------------------------------------------------

#target_script specifies the flash programming script file
target_script 0 reset c:\temp\temp.ocd

#working_area <target#> <address> <size> <'backup'|'nobackup'>
working_area 0 0x40000000 0x4000 nobackup

#flash bank at91sam7 0 0 0 0 <target#>
flash bank at91sam7 0 0 0 0 0
```

OpenOCD Configuration File for ARMUSBOCD (FLASH programming version)

```
#define our ports
telnet_port 4444
gdb_port 3333

#commands specific to the Olimex ARM-USB-OCD
interface ft2232
ft2232_device_desc "Olimex OpenOCD JTAG A"
ft2232_layout "olimex-jtag"
ft2232_vid_pid 0x15BA 0x0003
jtag_speed 2
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#reset_config <signals> [combination] [trst_type] [srst_type]
reset_config srst_only srst_pulls_trst

#jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
jtag_device 4 0x1 0xf 0xe
```

```
#daemon_startup <'attach'|'reset'>
daemon_startup reset

#target <type> <endianess> <reset_mode> <jtag#> [variant]
target arm7tdmi little run_and_init 0 arm7tdmi_r4

#run_and_halt_time <target#> <time_in_ms>
run_and_halt_time 0 30

# commands below are specific to AT91sam7 Flash Programming
# ---------------------------------------------------------

#target_script specifies the flash programming script file
target_script 0 reset c:\temp\temp.ocd

#working_area <target#> <address> <size> <'backup'|'nobackup'>
working_area 0 0x40000000 0x4000 nobackup

#flash bank at91sam7 0 0 0 0 <target#>
flash bank at91sam7 0 0 0 0 0
```

Note that all three of the configuration files (for FLASH programming) have the following command line:

target_script 0 reset c:\temp\temp.ocd

This is directing OpenOCD to execute the script file "**temp.ocd**" which has the flash programming commands. This script file typically has the following contents.

```
wait_halt                    <-- let cpu run a bit, then halt it
armv4_5 core_state arm       <-- make sure it's in ARM mode, not THUMB
flash write 0 main.bin 0x0   <-- erase then program the on chip Flash
mww 0xfffffd08 0xa5000401    <-- enable user reset
reset                        <-- reset cpu after programming is done
shutdown                     <-- terminate OpenOCD
```

If the flash programming doesn't work, it may well be that you have accidentally set the "lock" bits on the bottom two pages of flash. You can easily do this by powering up the board with the TST jumper installed; this installs a USB support program in your FLASH memory to enable the board to communicate with the SAM-BA flash programming utility.

In this case, you could add two additional commands to clear the lock bits. Be forewarned that the lock bits can only be set or cleared 100 times, so don't leave these two commands in the script file.

```
wait_halt                    <-- let cpu run a bit, then halt it
armv4_5 core_state arm       <-- make sure it's in ARM mode, not THUMB
mww 0xffffff64 0x5a000004    <-- clear lock bit 0
mww 0xffffff64 0x5a002004    <-- clear lock bit 1
flash write 0 main.bin 0x0   <-- erase then program the on chip Flash
mww 0xfffffd08 0xa5000401    <-- enable user reset
reset                        <-- reset cpu after programming is done
shutdown                     <-- terminate OpenOCD
```

Martin Thomas, guru of the WinARM tool chain, suggested that the flash programming using OpenOCD could be integrated into the makefile as an additional target. Joseph M. Dupre contacted the author with the excellent suggestion to let the makefile create the flash programming script file “on the fly”.

Let’s review again the part of the makefile that programs the flash.

```
# ****************************************************************************************
#                            FLASH PROGRAMMING      (using OpenOCD and Amontec JTAGKey)
#
# Alternate make target for flash programming only
#
# You must create a special Eclipse make target (program) to run this part of the makefile
# (Project -> Create Make Target...  then set the Target Name and Make Target to "program")
#
# OpenOCD is run in "batch" mode with a special configuration file and a script file containing
# the flash commands. When flash programming completes, OpenOCD terminates.
#
# Note that the make file below creates the script file of flash commands "on the fly"
#
# Programmers: Martin Thomas, Joseph M Dupre, James P Lynch
# ****************************************************************************************

# specify output filename here (must be *.bin file)
TARGET = main.bin

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# Note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = 'c:\Program Files\openocd-2006re93\bin\'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
#OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-wiggler-flash-program.cfg
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-jtagkey-flash-program.cfg
OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-armusbocd-flash-program.cfg

# specify the name and folder of the flash programming script file
OPENOCD_SCRIPT = c:\temp\temp.ocd

# program the AT91SAM7S256 internal flash memory
program: $(TARGET)
        @echo "Preparing OpenOCD script..."
        @cmd /c 'echo wait_halt > $(OPENOCD_SCRIPT)'
        @cmd /c 'echo armv4_5 core_state arm >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo flash write 0 $(TARGET) 0x0 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo mww 0xfffffd08 0xa5000401 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo reset >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo shutdown >> $(OPENOCD_SCRIPT)'
        @echo "Flash Programming with OpenOCD..."
        $(OPENOCD) -f $(OPENOCD_CFG)
        @echo "Flash Programming Finished."
```

There are three places in the above makefile excerpt that you must customize.

First, you must correctly specify the folder name where the OpenOCD executable and the configuration files reside as this can change if a newer version of YAGARTO is downloaded.

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
# Note: you may have to adjust this if a newer version of YAGARTO has been downloaded
OPENOCD_DIR = 'c:\Program Files\openocd-2006re93\bin\'

Second, you must choose which version of OpenOCD you are running (wiggler or USB).

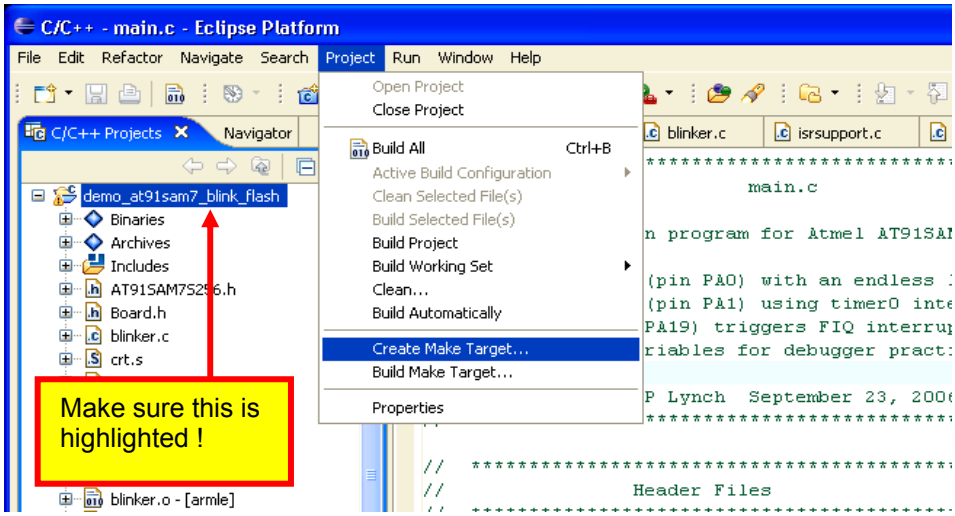
```
# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
#OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe
```

Next, you must choose which OpenOCD configuration file you will be using (wiggler, JTAGKey or ARMUSBOCD).

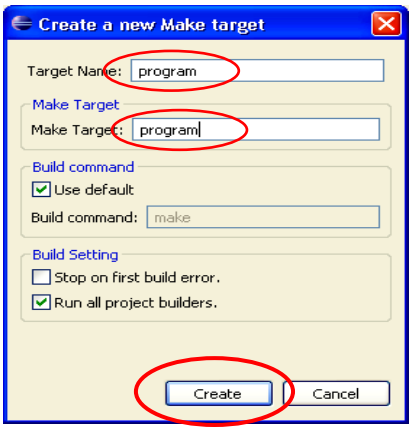
```
# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-wiggler-flash-program.cfg
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-jtagkey-flash-program.cfg
OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-armusbocd-flash-program.cfg
```

Assuming that you have already performed a "Build All" on the sample program and have an output file (main.bin) to program into the FLASH plus you have set up the hardware as shown earlier, you can now program the FLASH by running the "**program**" target in the makefile.

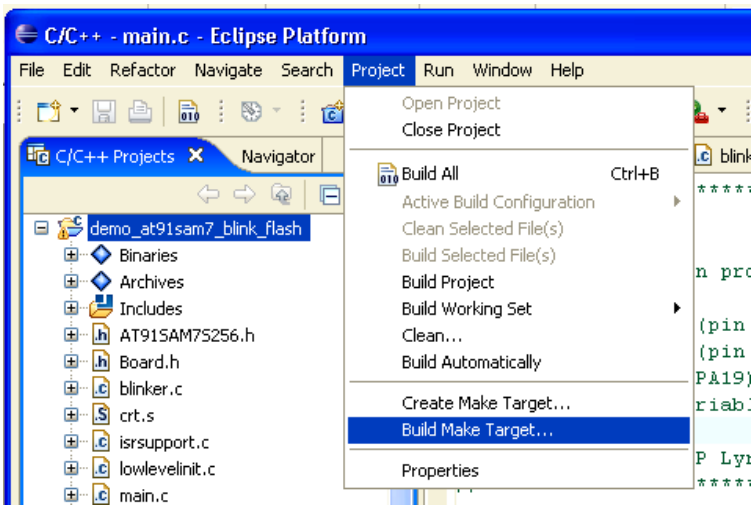
To prepare to do this, we need to establish "**program**" as a secondary make target. Click "**Project – Create Make Target…**" as shown below. Note that you must have the project itself highlighted in the "**Projects**" view to enable this.



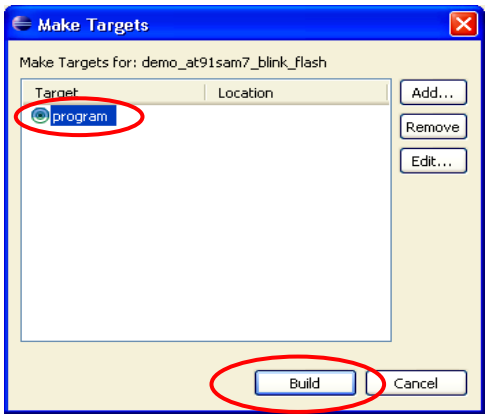
In the "Create a New Make Target" dialog, enter "**program**" into the Target Name text box. Enter "**program**" into the Make Target text box also. Click "**Create**" to finish.



There are two ways to execute the alternate Make target. The first way is to use the Project pull-down menu. Click on "**Project – Build Make Target**" as shown below.

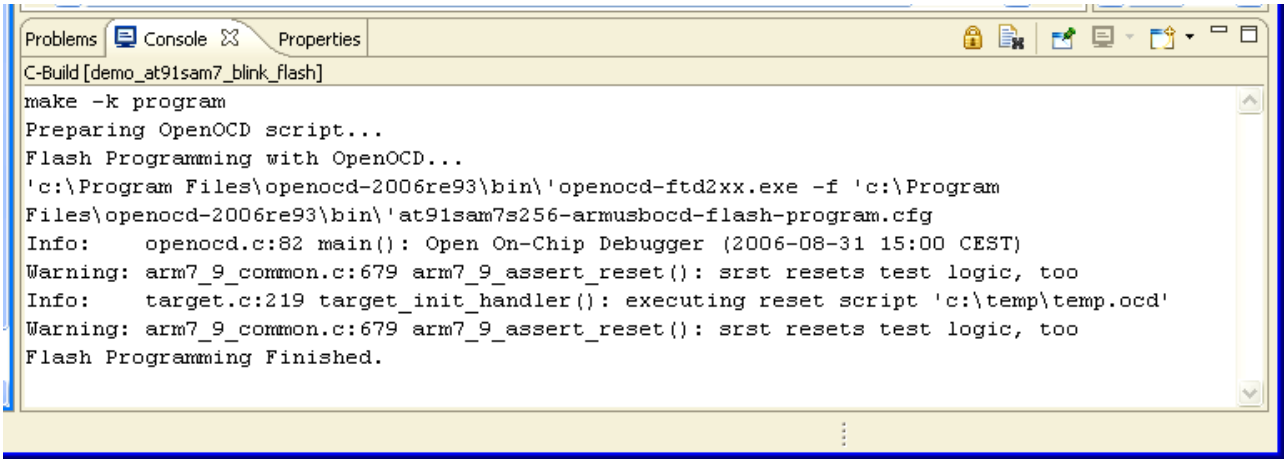


Click on the "**program**" icon below to highlight it (there can be multiple alternate targets defined) and then click "**Build**" to execute the makefile alternate target called "program" and thereby program the FLASH.

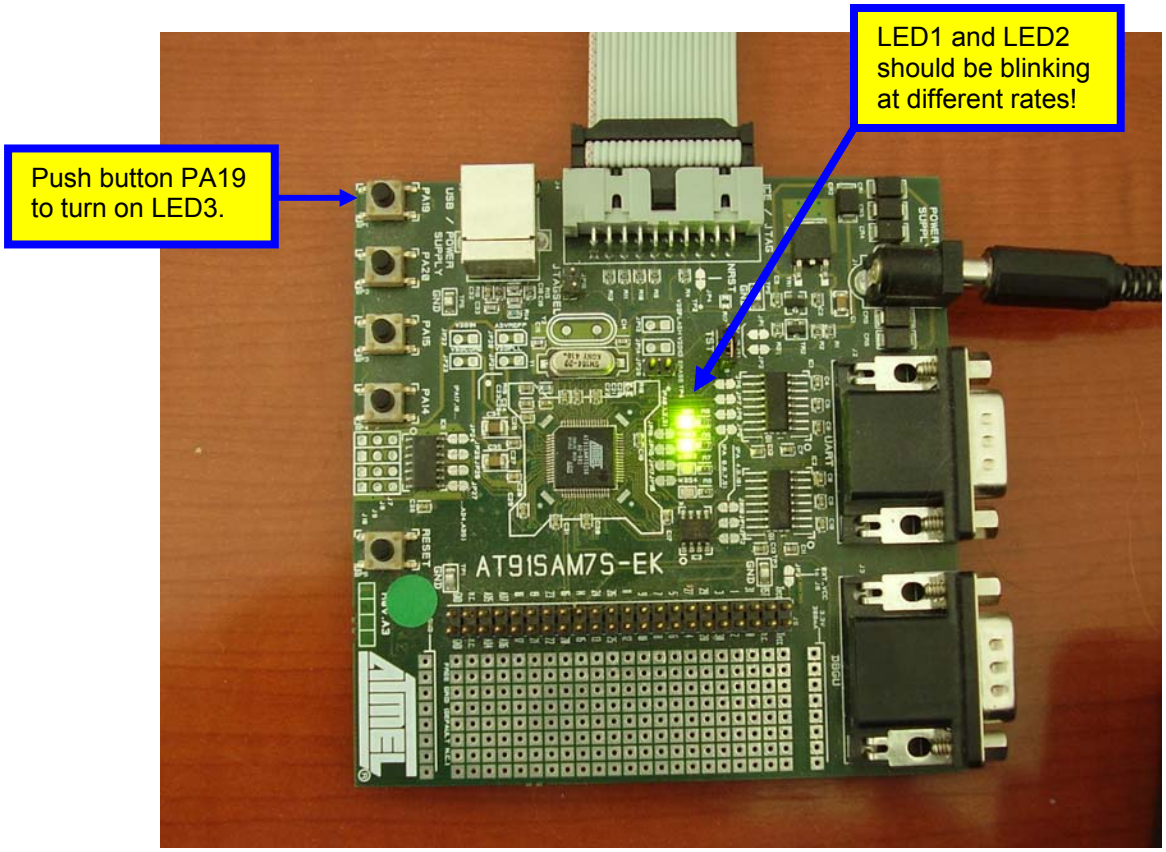


The FLASH programming algorithm built into OpenOCD will now start. Since the sample program is relatively small, this will run through to completion in just a few seconds.

The results of the FLASH programming activity are displayed in the "Console" view as shown below.

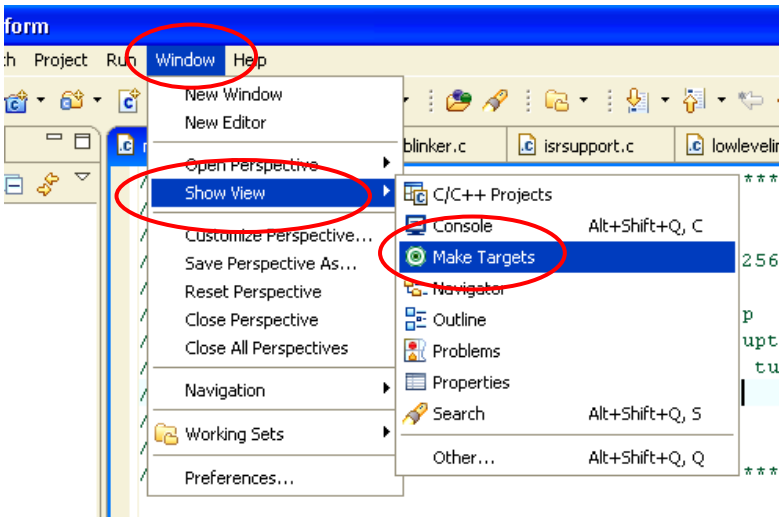


To test the application, hit the "**Reset**" button on the Atmel AT91SAM7S-EK. LED1 should be blinking at about a 1 Hz rate. LED2 should be blinking at a 10 Hz rate (Timer0 interrupt). If you push switch SW1, labeled PA19 on the AT91SAM7S256-EK board, you should see LED3 light up (it will turn off as part of the background loop).

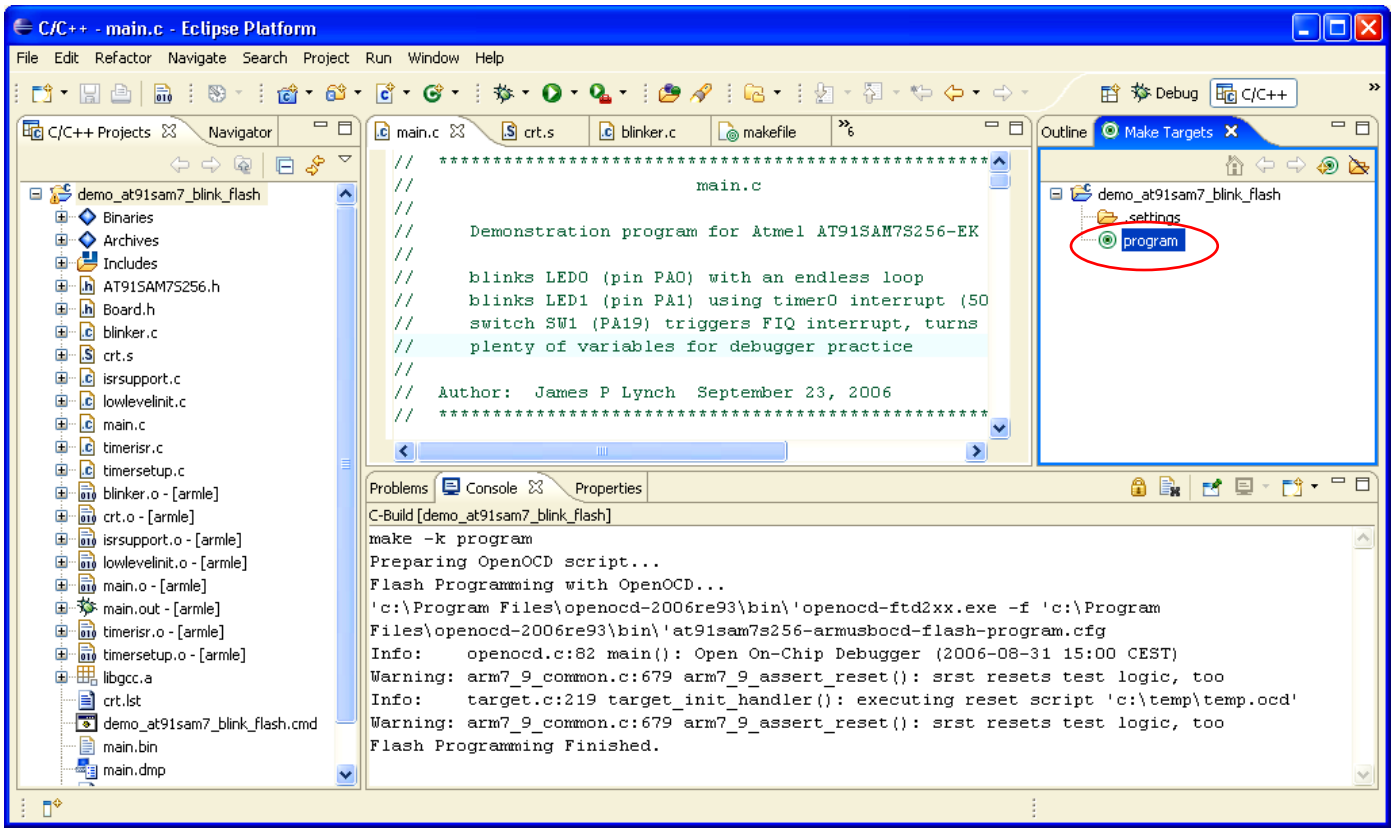


Congratulations! You now have a full-fledged ARM cross development system operational and it didn't cost a thing!

There's one other way to conveniently launch the alternate make target to program the flash. We can display a special "Make Targets" view. Click "**Window – Show View – Make Targets**" as shown below.



Now the "**Make Targets**" view is presented with the "**Outline**" view on the right and if you double-click on "**program'** the alternate make file target will immediately run and the FLASH will be programmed.



Debugging the FLASH Application

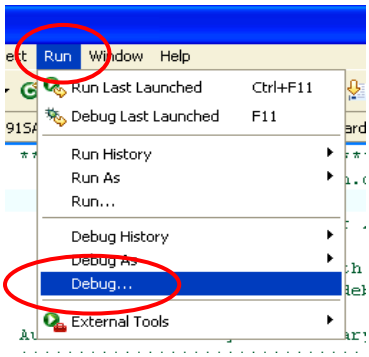
Unless you are an infallible programmer, you will occasionally require the services of a debugger to trap and identify software bugs. Eclipse has a wonderful visual source code debugger that interfaces to the GNU GDB debugger.

You can debug an application programmed into on chip FLASH; the built-in on chip JTAG debug circuits allow this. There is only one restriction; you are limited to just two breakpoints. Attempting to specify more than two hardware breakpoints at a time may cause the debugger to malfunction. Otherwise all Eclipse debugging features work properly, such as single-stepping, inspection and modification of variables, memory dumps, etc.

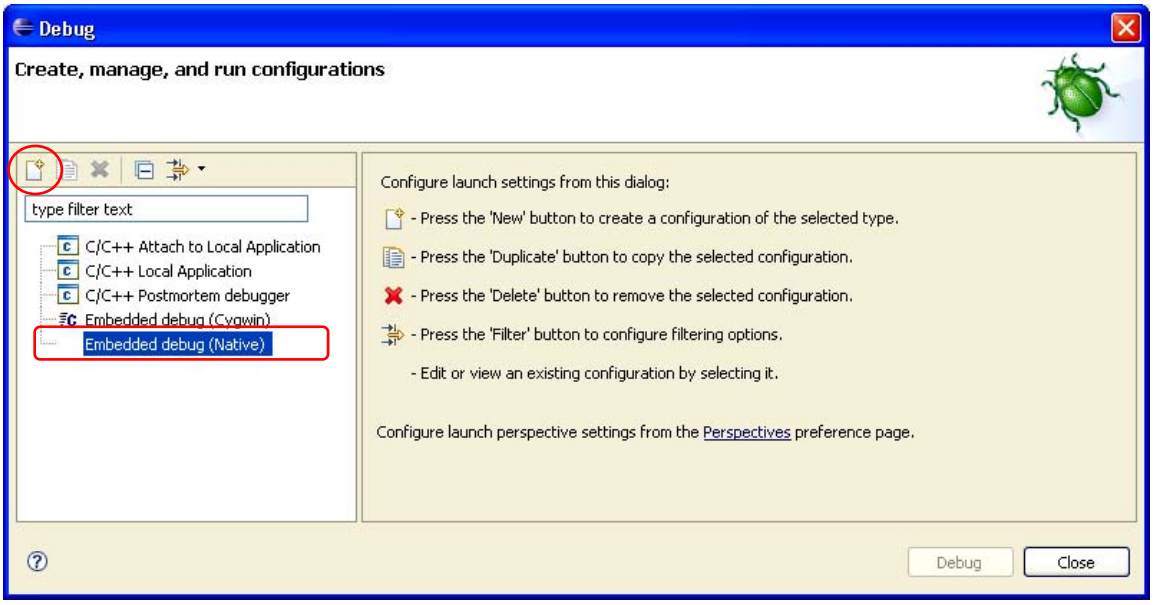
Create a Debug Launch Configuration

Before we can debug the FLASH application, we have to create a Debug Launch Configuration for this project. The Debug Launch Configuration locates the GDB debugger for Eclipse, locates the project's 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. Most people will define a "**Debug Launch Configuration**" for each project they create.

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



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



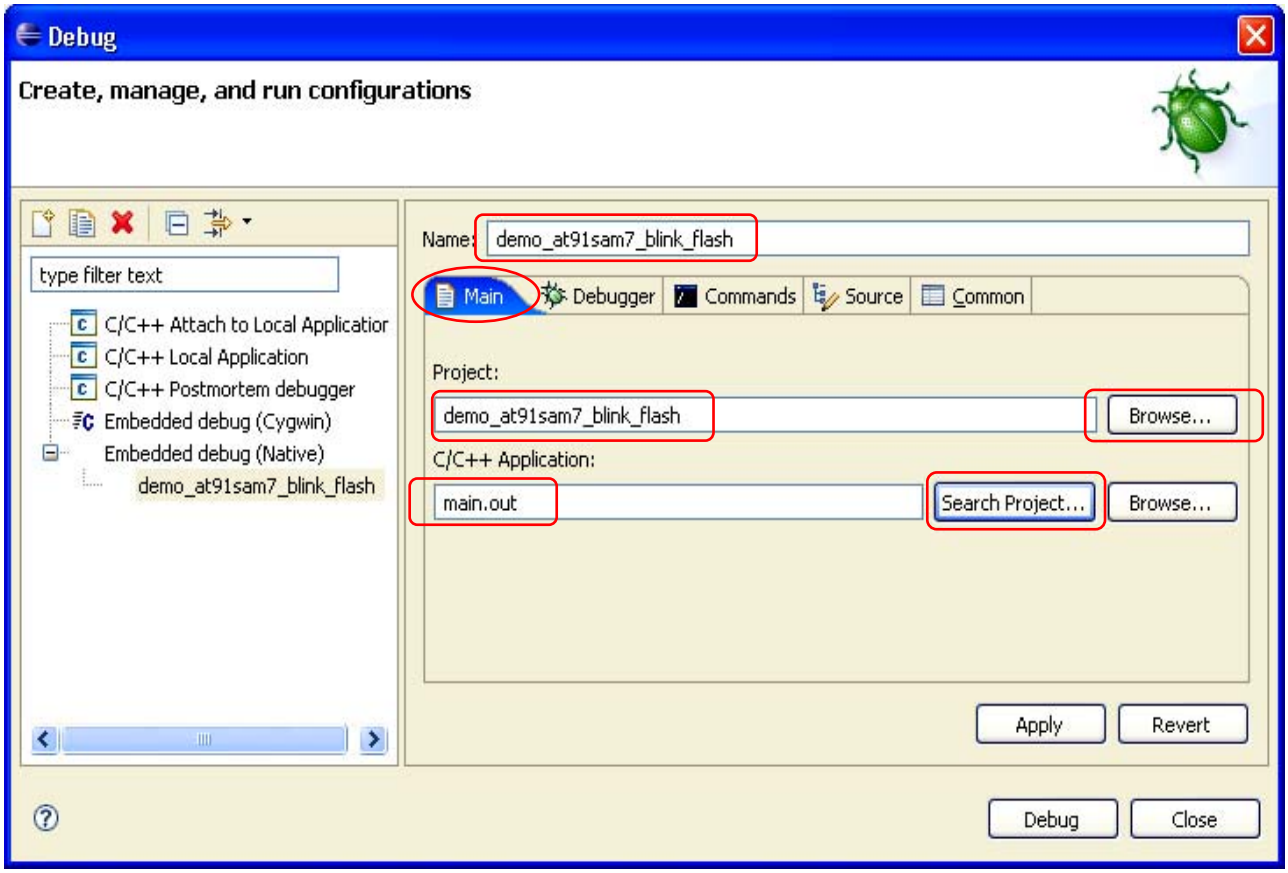
The Debug “Create, manage and run configurations” window changes to the dialog shown below. Start by making sure that the “**Main**” tab is selected.

In the “Name:” text box, enter the name of this debug launch configuration. 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. In this example, we simply use the project name “**demo_at91sam7_blink_flash**” for this purpose.

In the “Project” text box, use the “**Browse**” button to find the project “**demo_at91sam7_blink_flash**”.

In the “C/C++ Application” text box, use the “**Search Project…**” button to find the application file “**main.out”**.

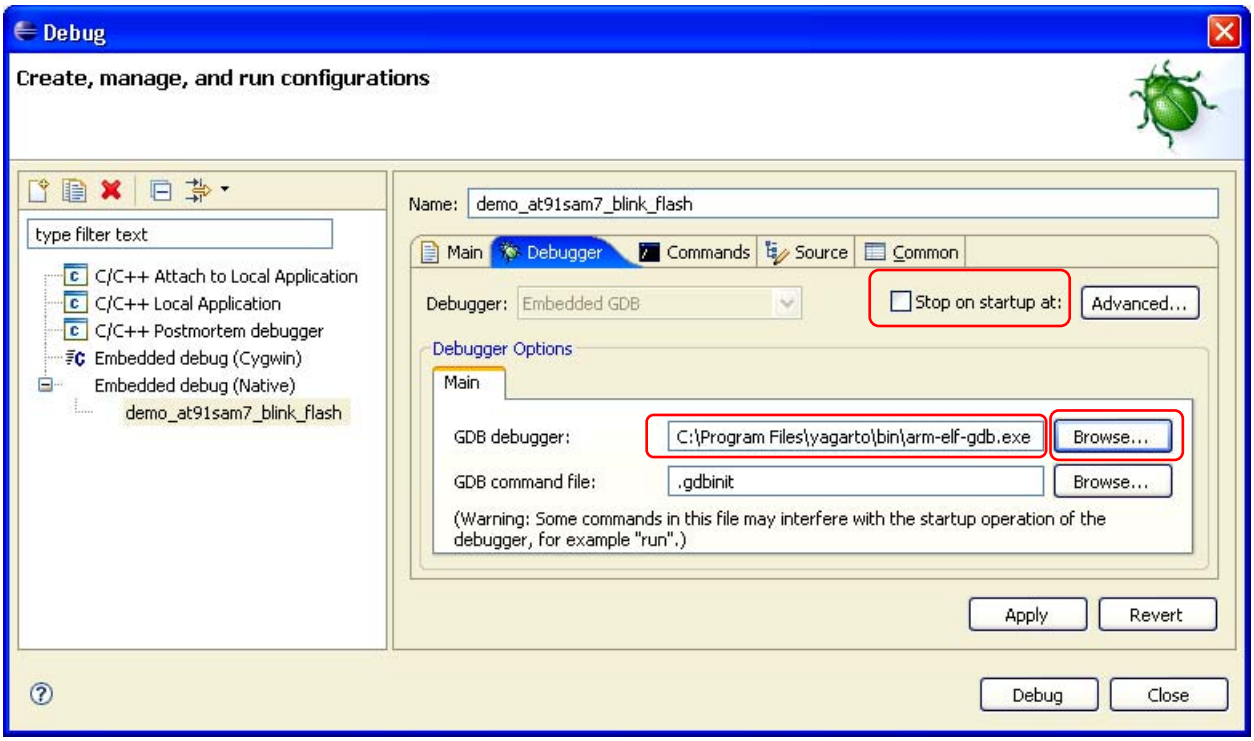
You might be inclined to ask why this is not the “**main.bin**” file? The binary file was used earlier to program the flash, but the debugger needs the application file that has the symbols; this is the “**main.out**” file. While the “**main.out**” file also has the executable code within it, the debugger only uses the symbol information for FLASH debugging.



Now select the “**Debugger**” tab as shown below.

Un-check the check box that says “**Stop on startup at...**”. We will be using a “**thbreak main**” command in our GDB startup commands to accomplish the same thing.

In the text box labeled “**GDB Debugger:**”, use the “**Browse**” button to locate the “**arm-elf-gdb.exe**” file. It will be found in the “**c:\Program Files\yagarto\bin**” folder.

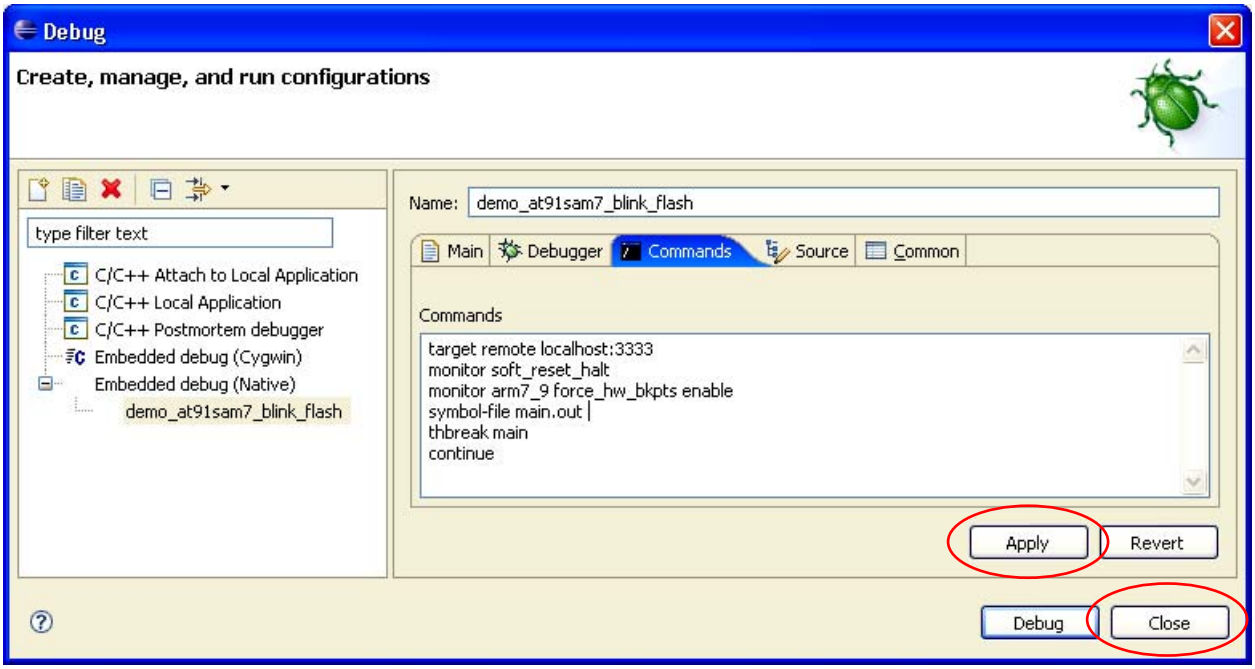


Now select the "**Comands**" tab as shown below.

In the Commands text window, enter the six GDB startup commands exactly as shown.

When finished, click on "**Apply**" followed by "**Close**" to finish specification of the debug launch configuration for FLASH debugging. Eclipse may ask you if you want to save this configuration, answer "**Yes**".

The "**Source**" and "**Common**" tabs can be left in their default state.



The six GDB startup commands shown above 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

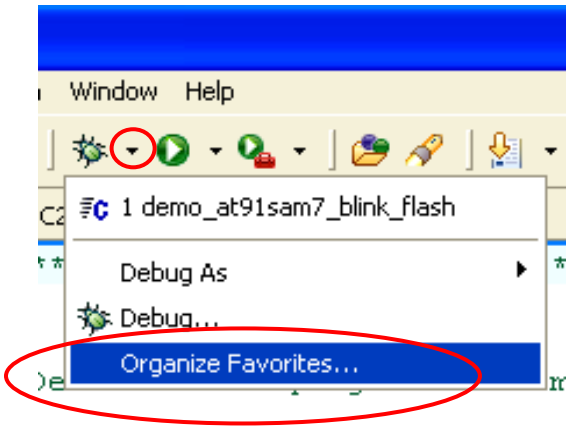
This is a **GDB** command. It sets a temporary hardware breakpoint at the entry point main(). Once the debugger breaks at main(), this breakpoint is automatically removed.

continue

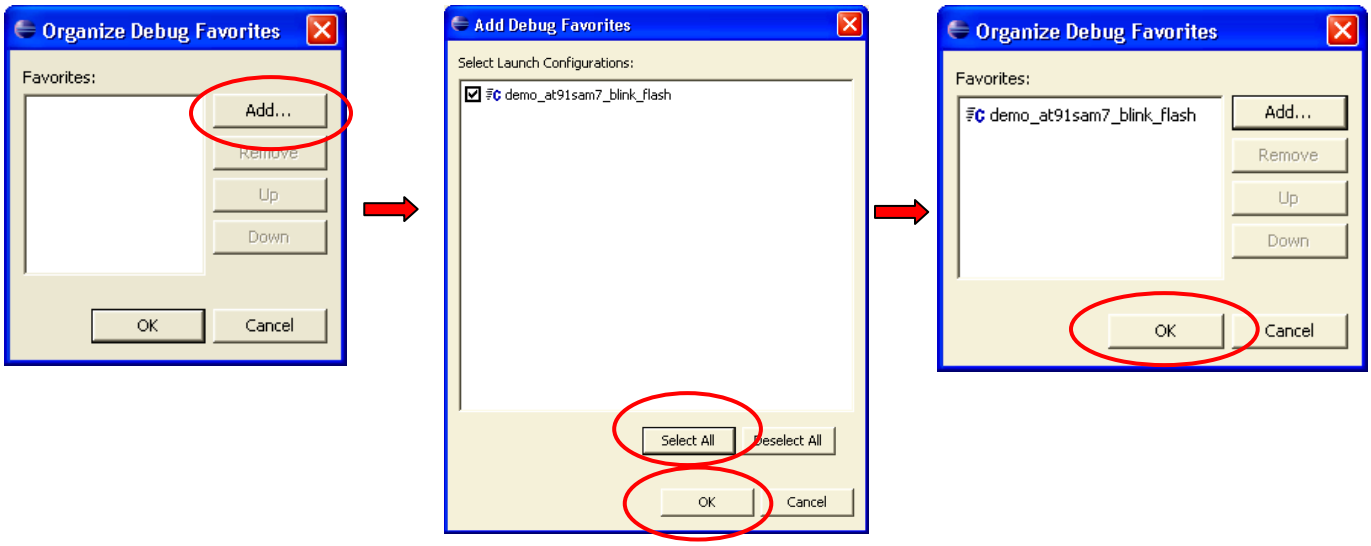
This is a **GDB** command. It forces the ARM processor out of breakpoint/halt state and resumes execution from the reset vector till it breaks at main().

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.

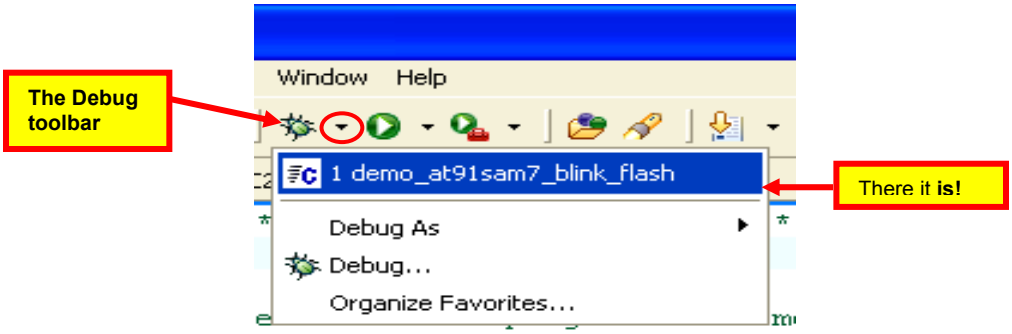
In the toolbar, click on the **down arrowhead** next to the debug symbol and then click "**Organize Favorites…**".



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.



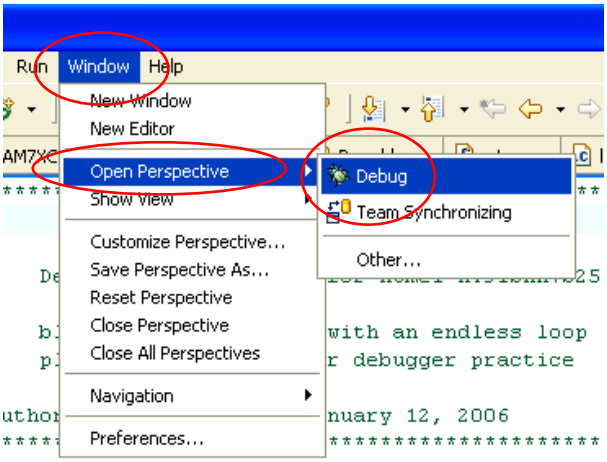
Now when you click on the Debug Toolbar button's down arrowhead, you will see the "**demo_at91SAM7_blink_flash**" debug launch configuration installed as a favorite, as shown below.



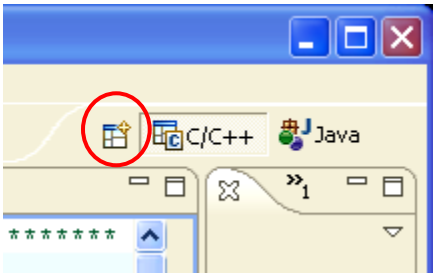
Now everything is in place to debug the project that we loaded into FLASH memory via OpenOCD.

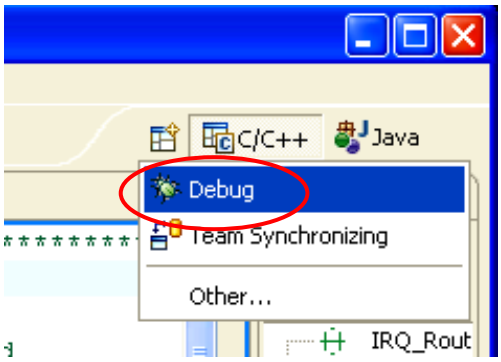
Open the Eclipse Debug Prespective

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.

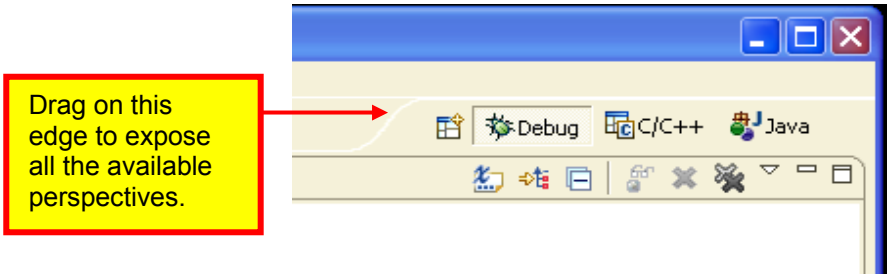


A more convenient way to switch perspectives is to click on the "perspective" buttons at the Eclipse upper-right window location. Click on the "**OpenPerspective**" toolbar button below on the left and then choose "**Debug**" when the other perspectives are displayed.

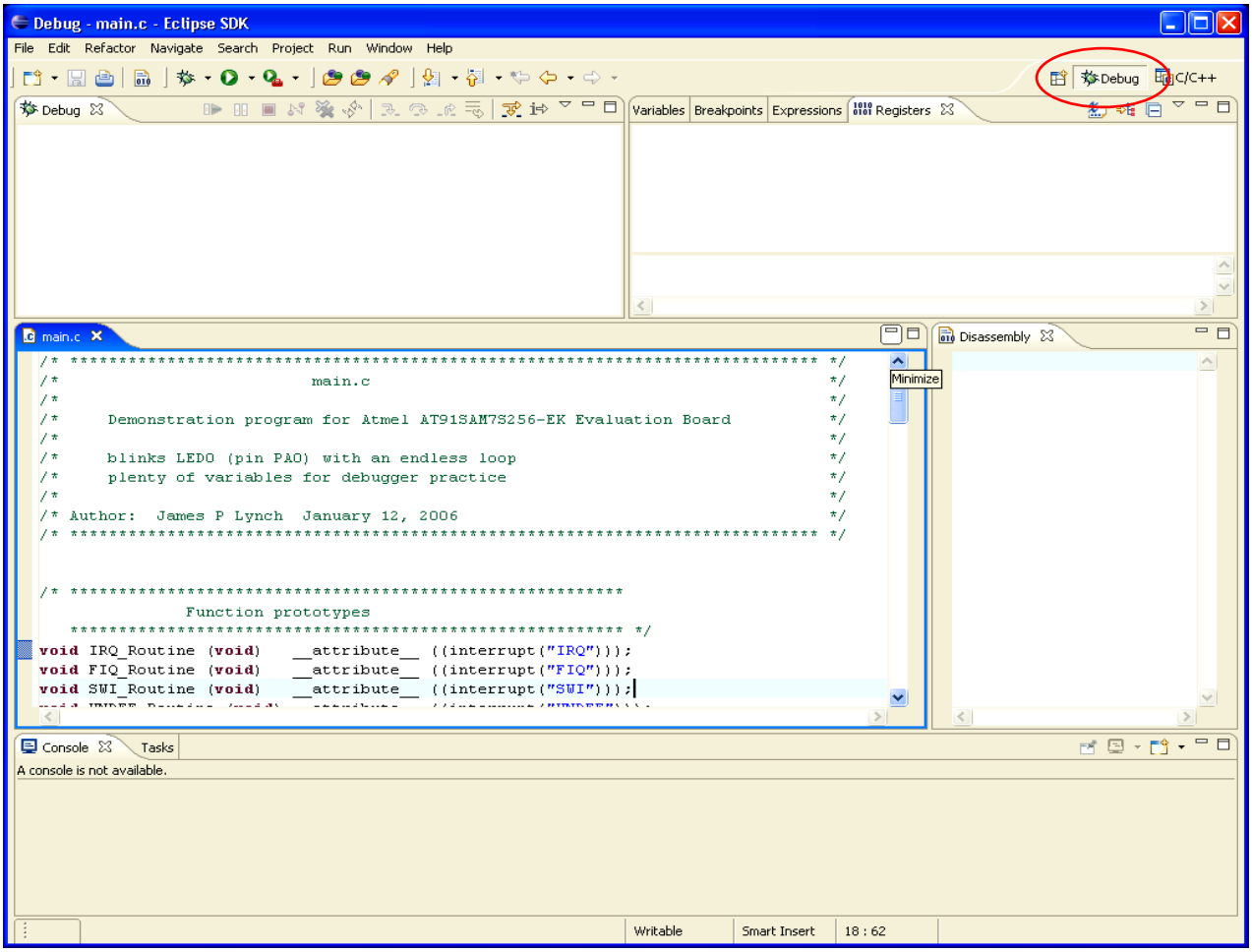




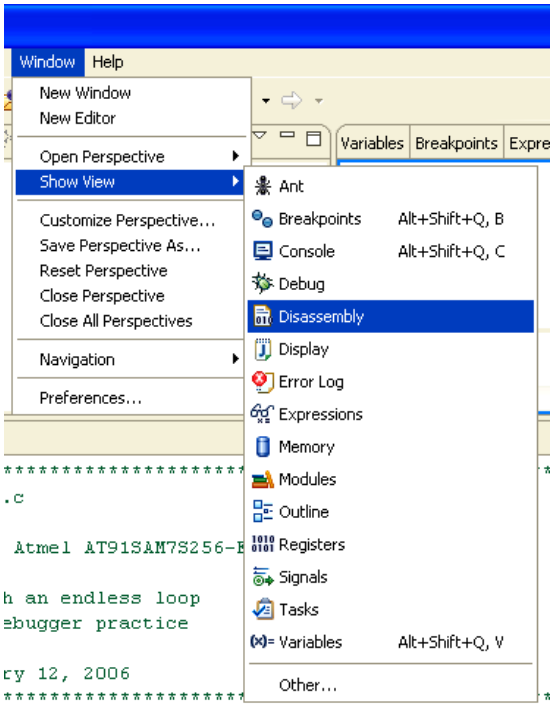
Now we have a "Debug" button as shown below. You may have to drag on the edge to expose all the perspective buttons. You can also right-click on any of the buttons and "Close" them to narrow the display to only the perspectives you are interested in.



Click on the "Debug" perspective button at the upper-right to open the Debug Perspective display, shown below.

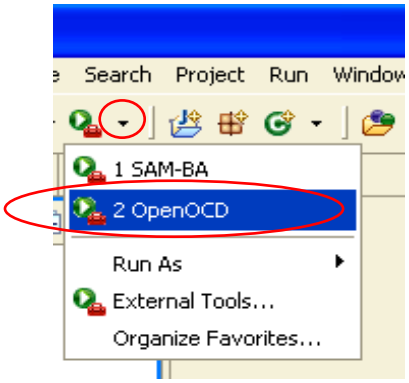


If your display doesn't look exactly like the debug display above, click on "**Window – Show View**" and select any of the missing elements.



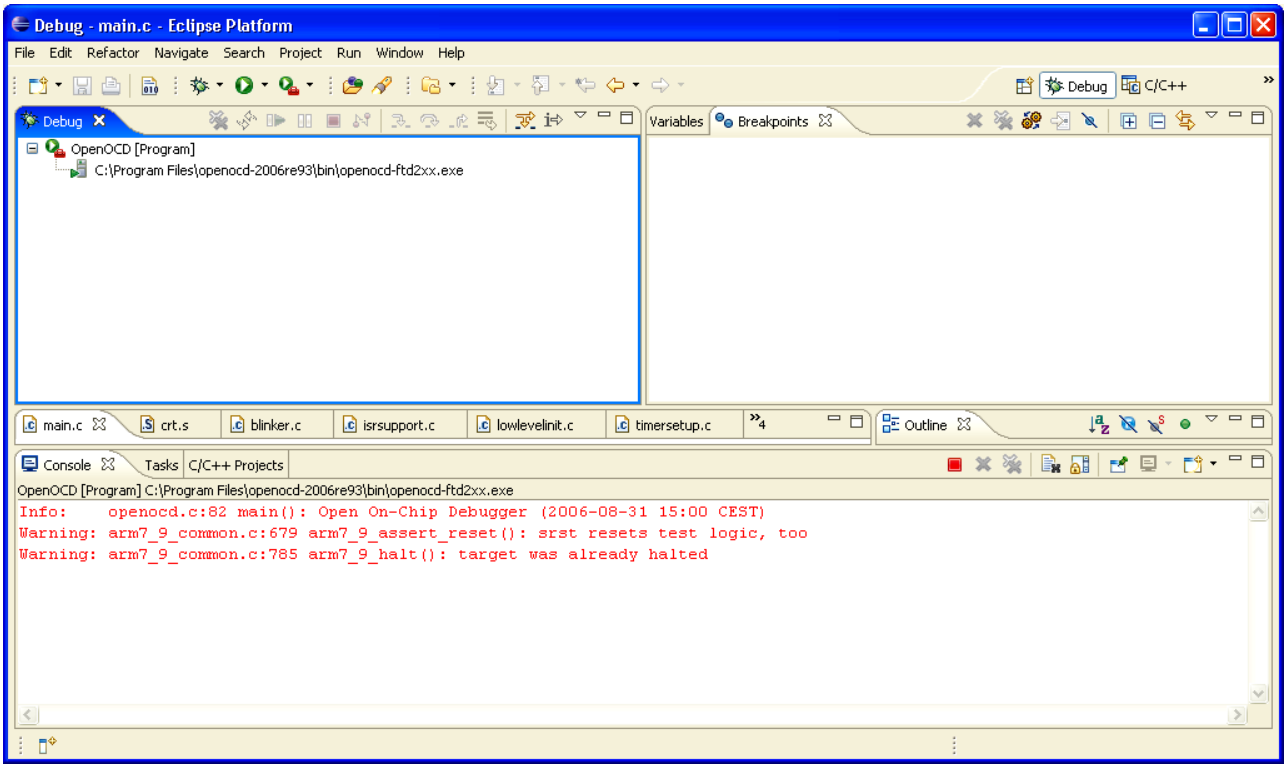
Starting OpenOCD

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**".



Eclipse remembers the last button you selected, so you can usually just click on the red toolbox button itself to start OpenOCD. If you're not sure what "external tool" will be selected, just hover the cursor over the toolbox icon and the "hints" feature will show that "OpenOCD" will be selected.

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

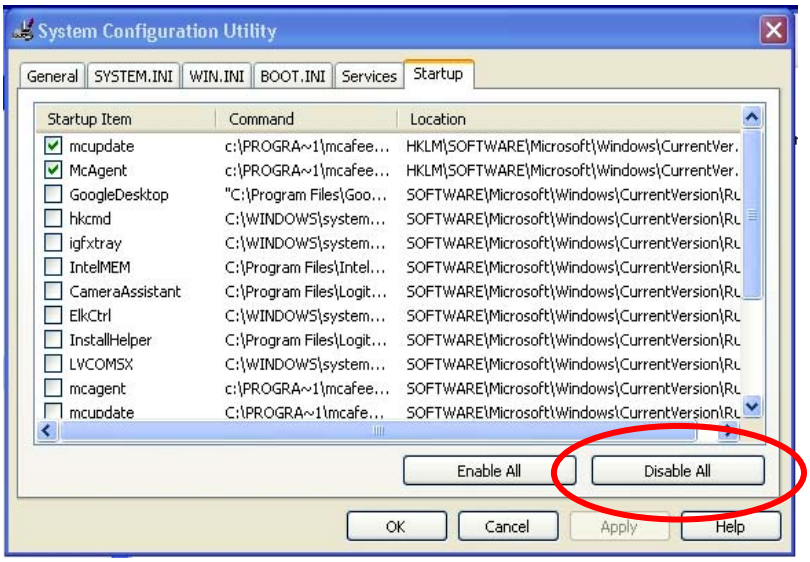


If for some reason, **OpenOCD** will not properly start in your system, you can try the following things.

- 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.

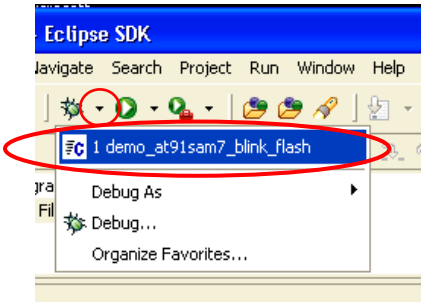
  For Windows XP users, here is a simple way to get rid of all those background programs. Click "**Start – Help and Support – Use Tools… - System Configuration Utility – Open System Configuration Utility – Startup Tab**". Click on "**Disable All**". Windows will ask you to re-boot and the PC will restart with none of the start-up programs running. Use the same procedure to reverse this action.



Start the Eclipse Debugger

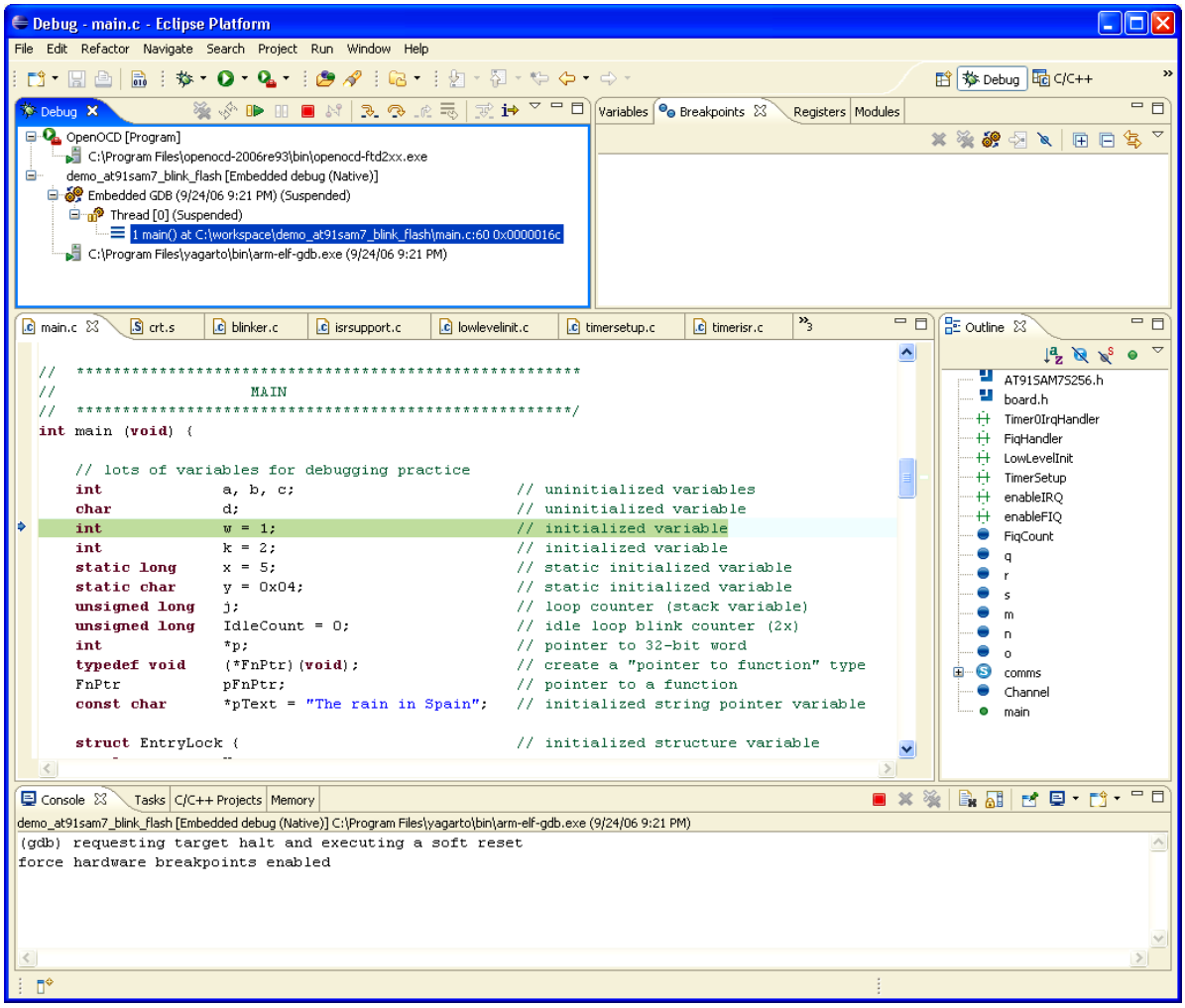
To start the Eclipse debugger, click on the "**Debug**" toolbar button's down arrowhead and select the debug launch configuration "**demo_at91sam7_blink_flash**" as shown below.

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.



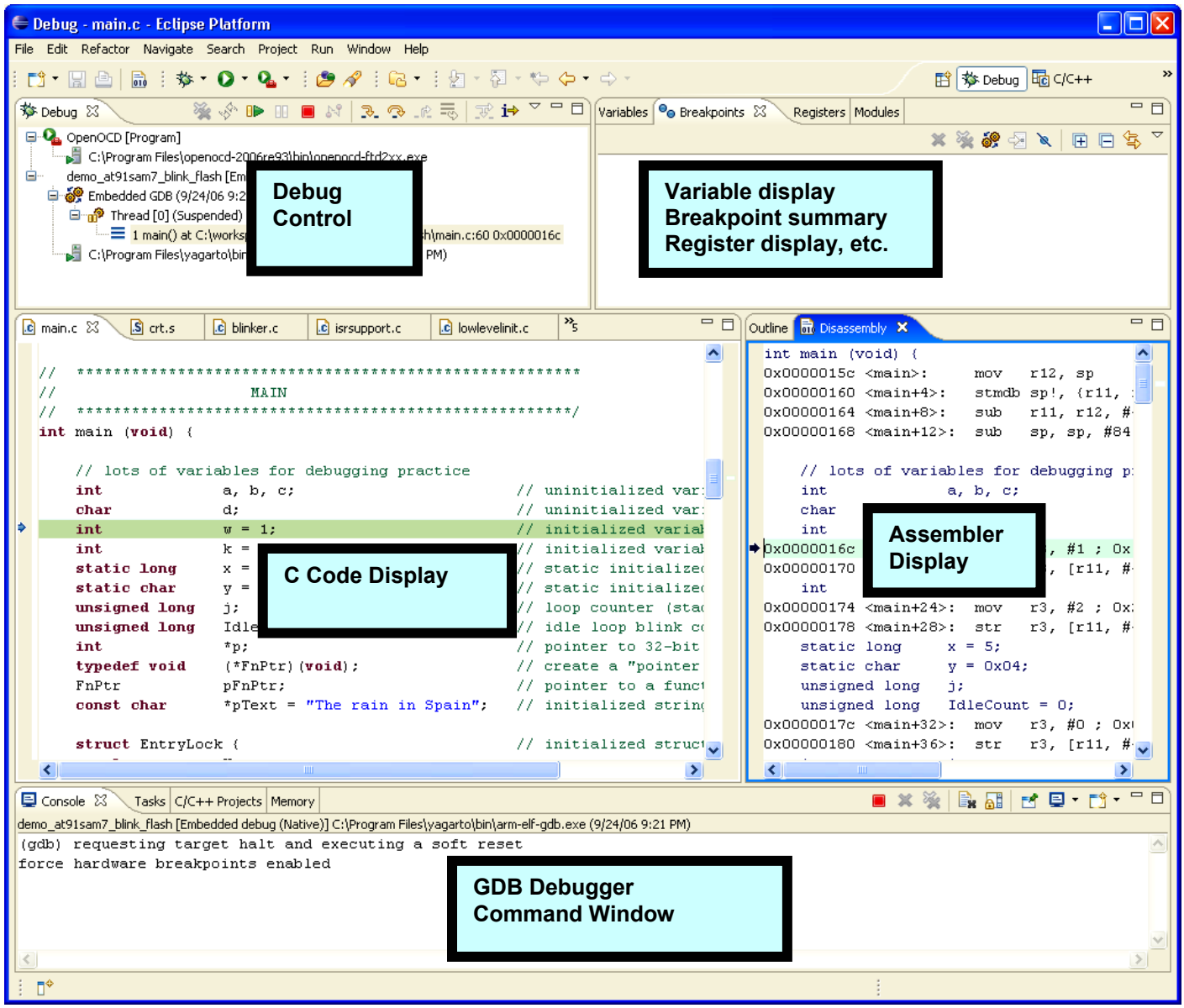
If the Eclipse debugger starts properly, the debug view (upper left) shows that the debugger has stopped at line 60 in main(). The console view (bottom) shows each GDB command that was executed in red letters and some **OpenOCD** messages after some of the commands.

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.

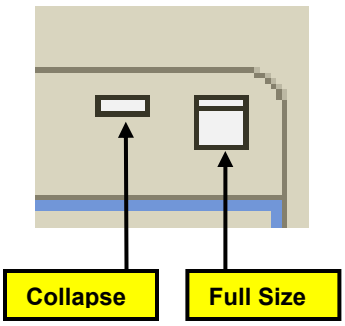


Components of the DEBUG Perspective

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

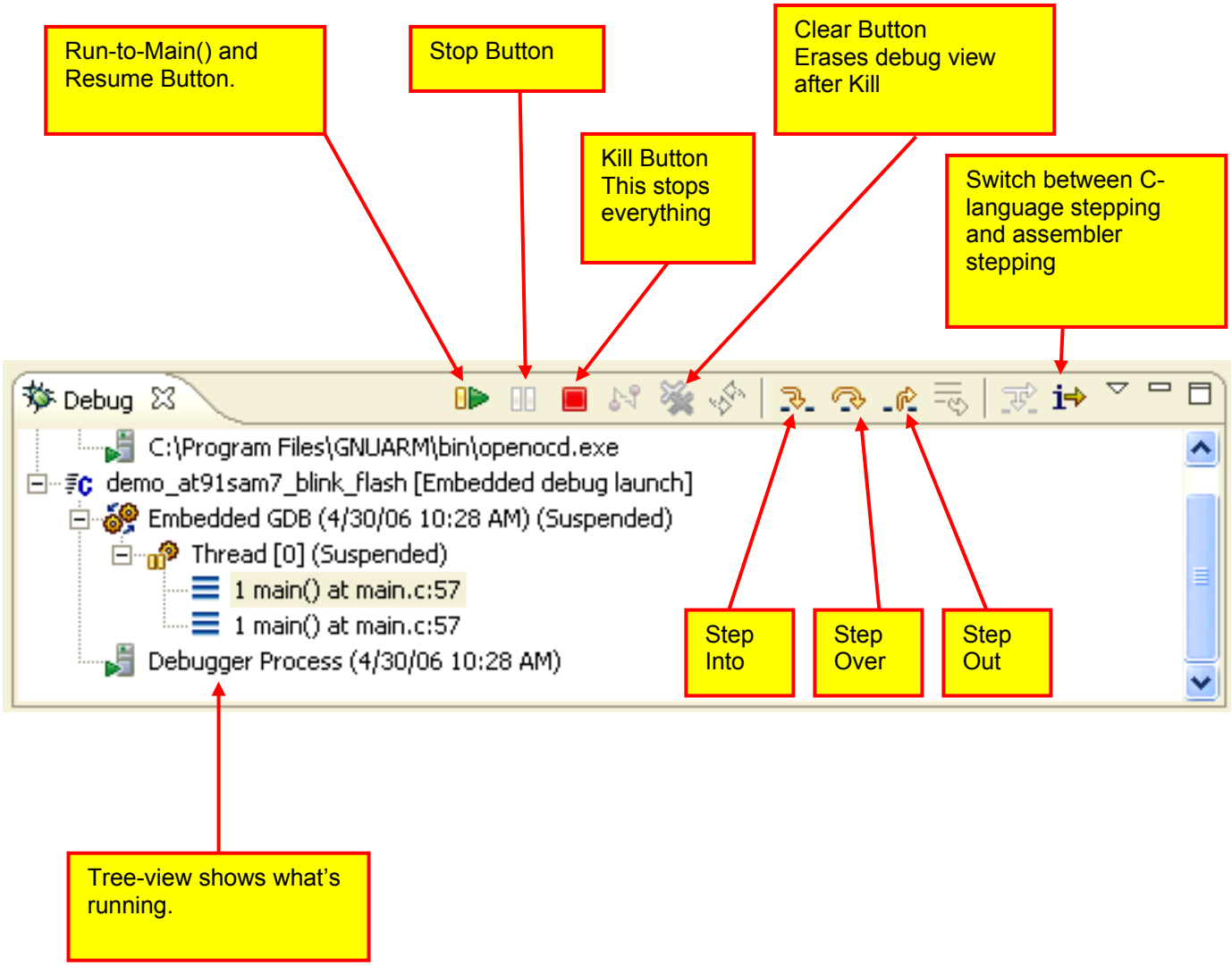


While this may be obvious to most, you can minimize and restore any of the windows in the Debug perspective by clicking on the "maximize" and "minimize" buttons at the top right corner of each window.



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 **OpenOCD** 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, it behooves you to have no breakpoints set since Eclipse needs one of the hardware breakpoints for single-stepping.

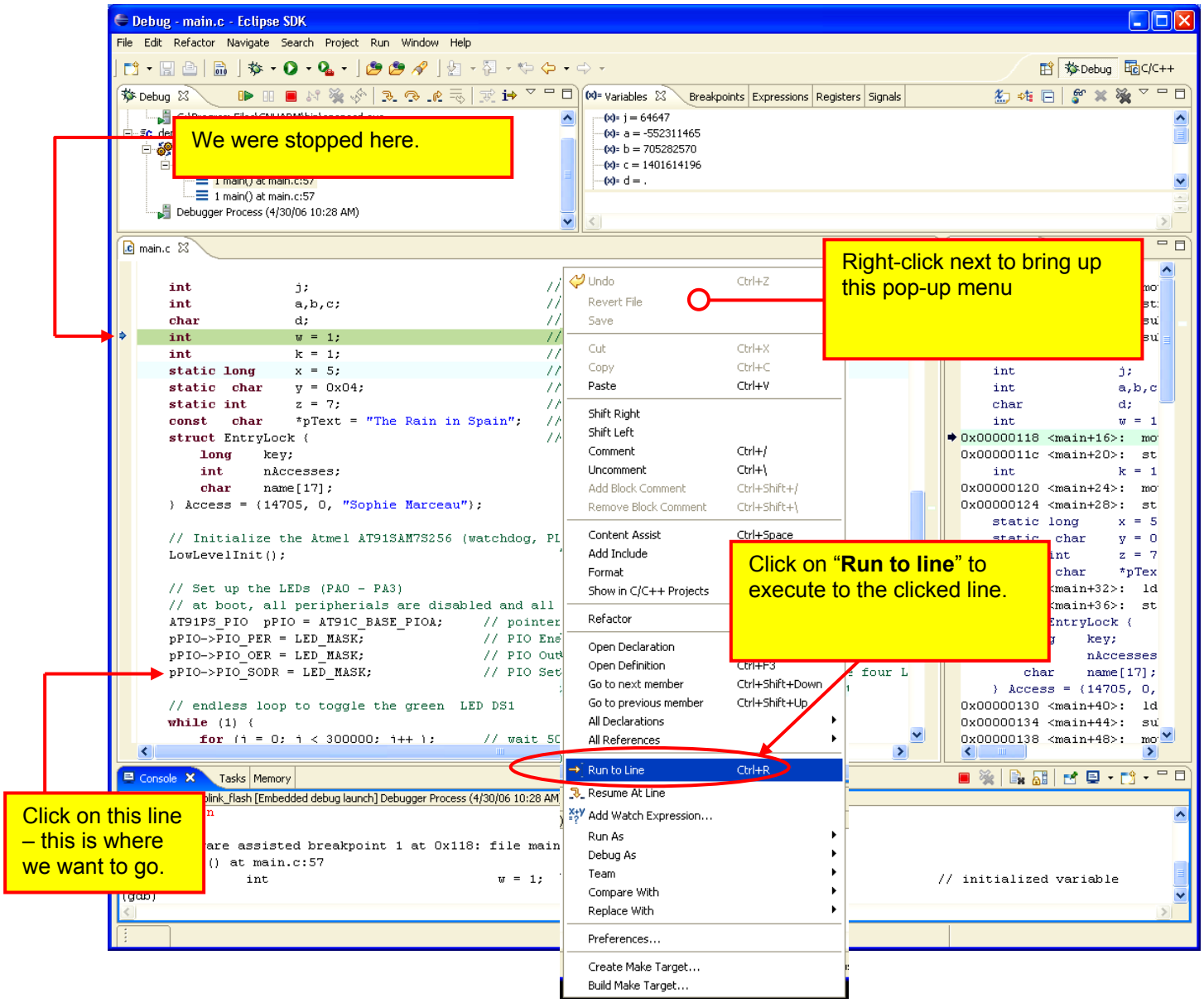
- If you re-compile your application, you must stop the debugger and OpenOCD, re-build and burn the main.bin file into FLASH using the OpenOCD FLASH programming facility.

Run and Stop with the Right-Click Menu

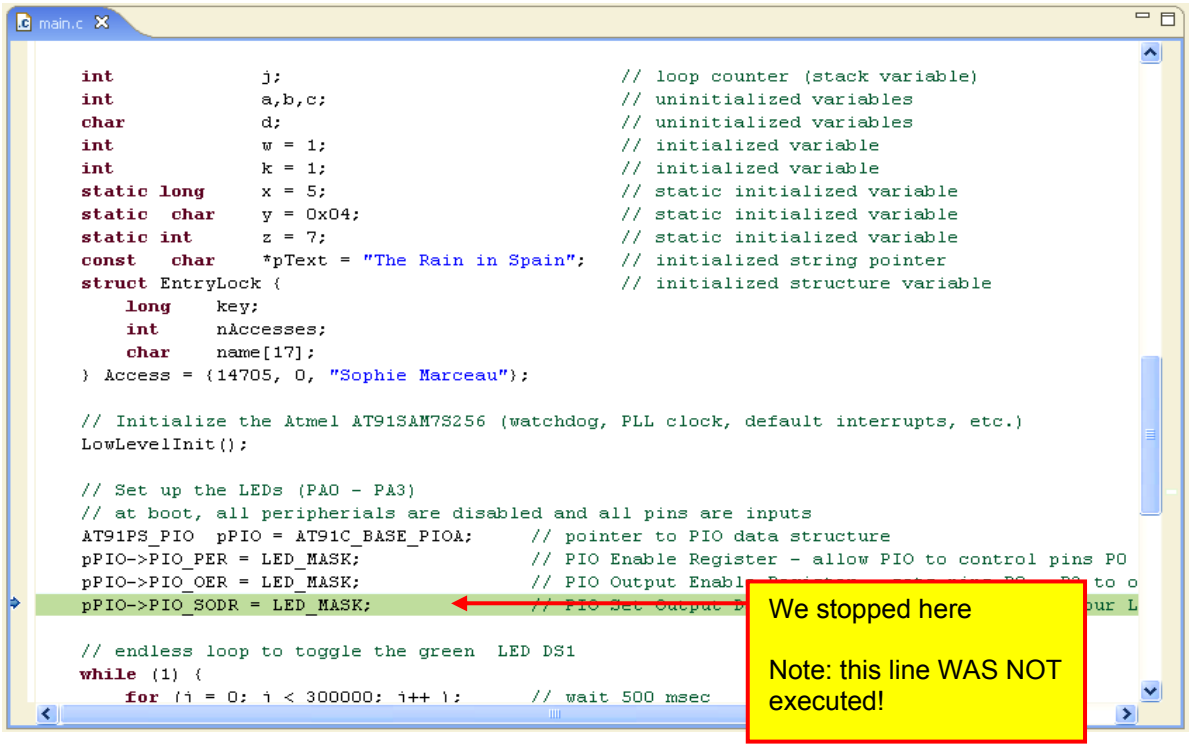
The easiest method of running is to employ the right-click menu. In the example below, the blue arrowhead cursor indicates where the program is currently stopped - just after main( ).

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.



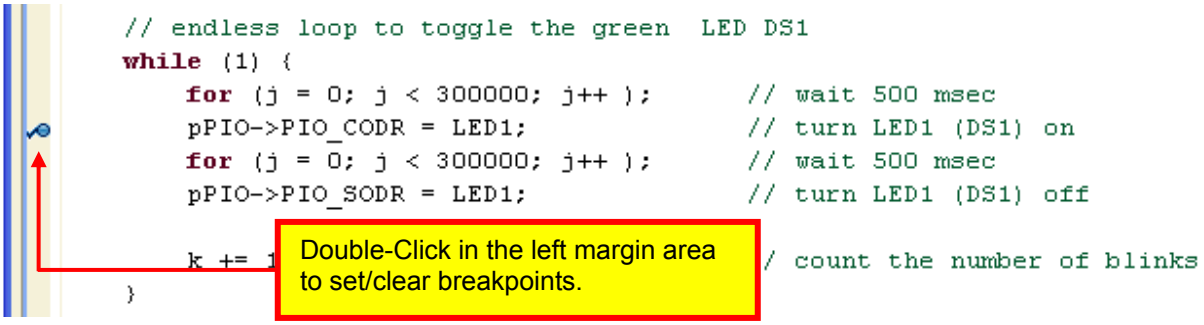
When you click on the "**Run to line**" choice, the program will execute to the line the cursor resides on and then stop (N.B. it will not execute the line).



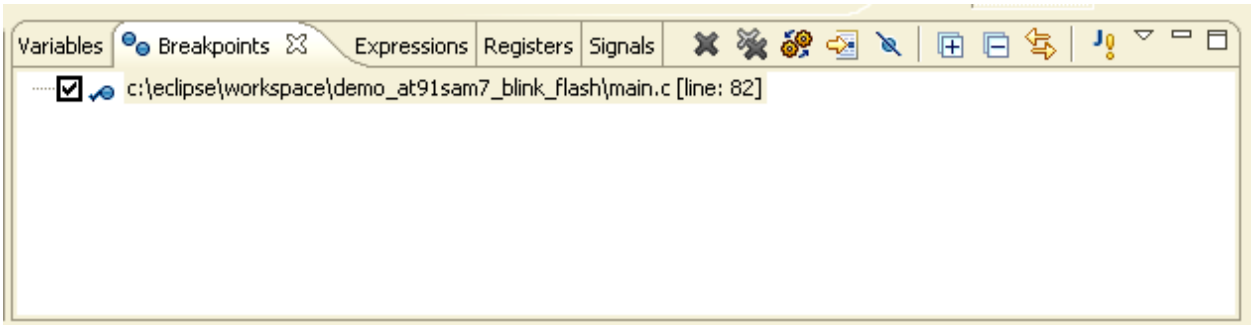
You can right-click the "**Resume at Line**" choice to continue execution from that point. If there are no other breakpoints set, then the Blink application will start blinking continuously.

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.

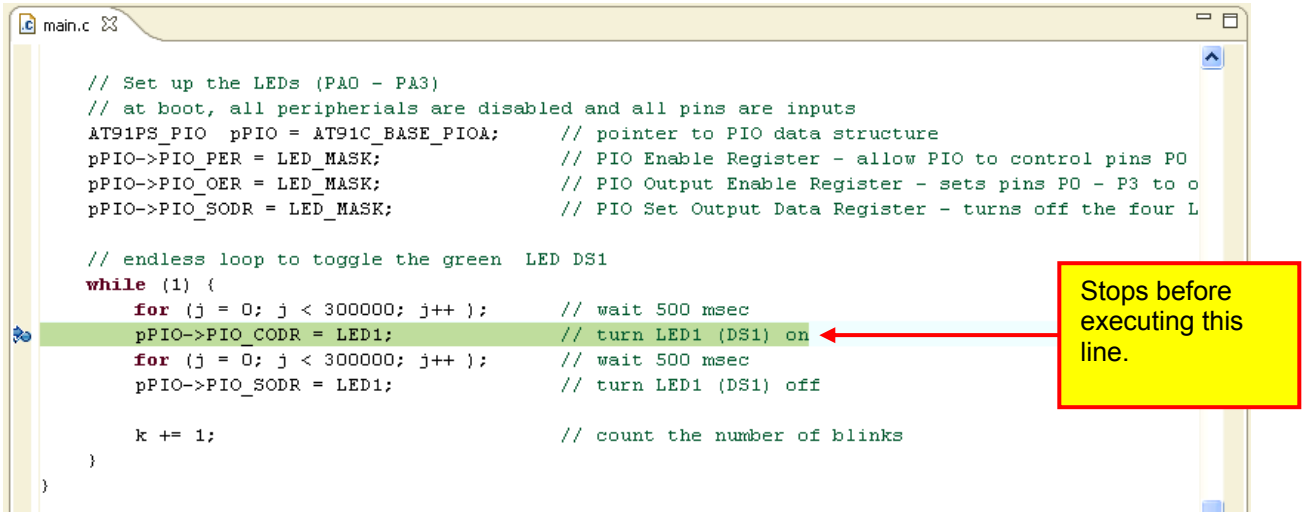


Note in the upper right "Breakpoint Summary" pane, the new breakpoint at line 82 has been indicated, as shown below.



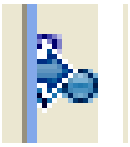
Now click on the "**Run/Continue**" button in the Debug view.

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

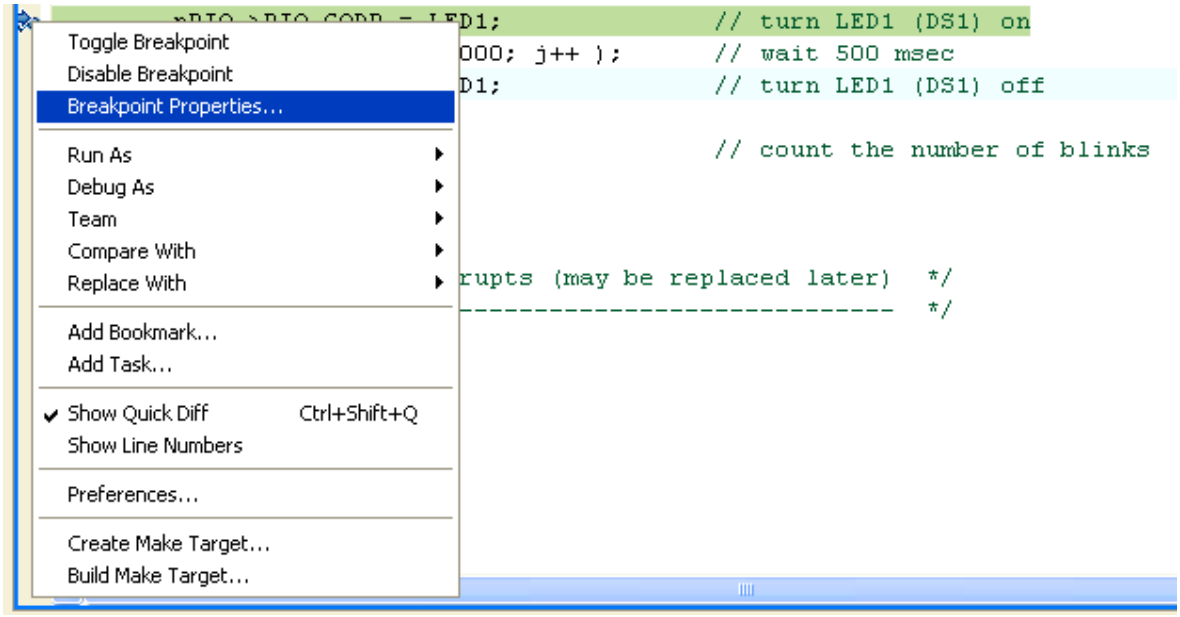


Since this is a FLASH application and breakpoints are "hardware" breakpoints, you are limited to **only two breakpoints specified at a time**. Setting more than two breakpoints will cause the debugger to malfunction!

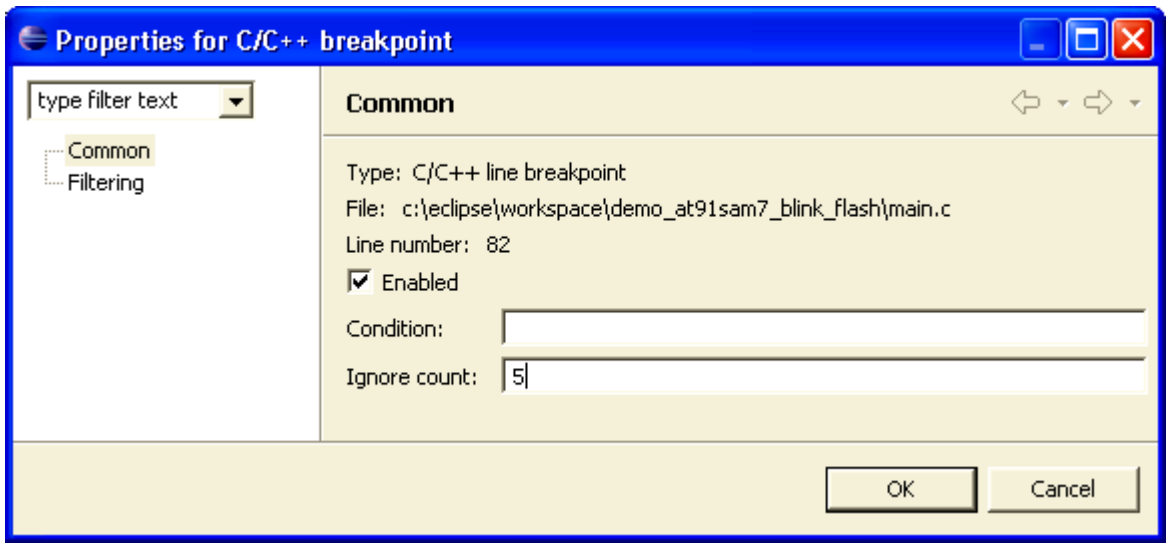
The breakpoints can be more complex. For example, to ignore the breakpoint 5 times and then stop, right-click on the breakpoint symbol on the far left.



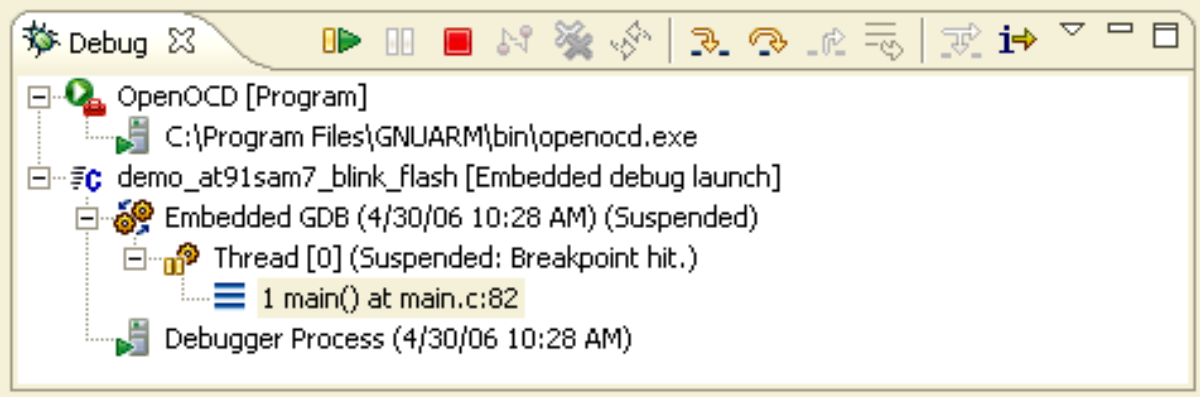
This brings up the pop-up menu below; click on "**Breakpoint Properties …**".



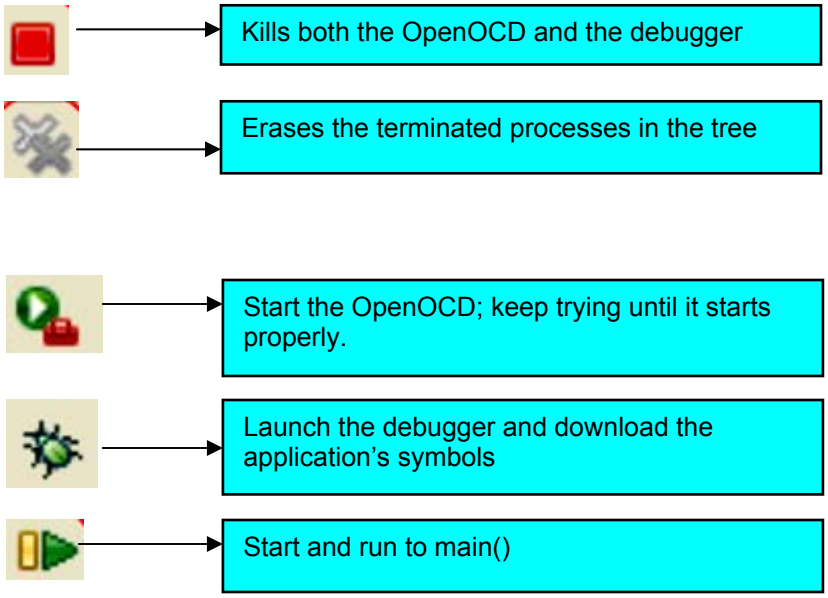
In the "**Properties for C/C++ breakpoint**" window, set the **Ignore Count** to 5. This means that the debugger will ignore the first five times it encounters the breakpoint and then stop.



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



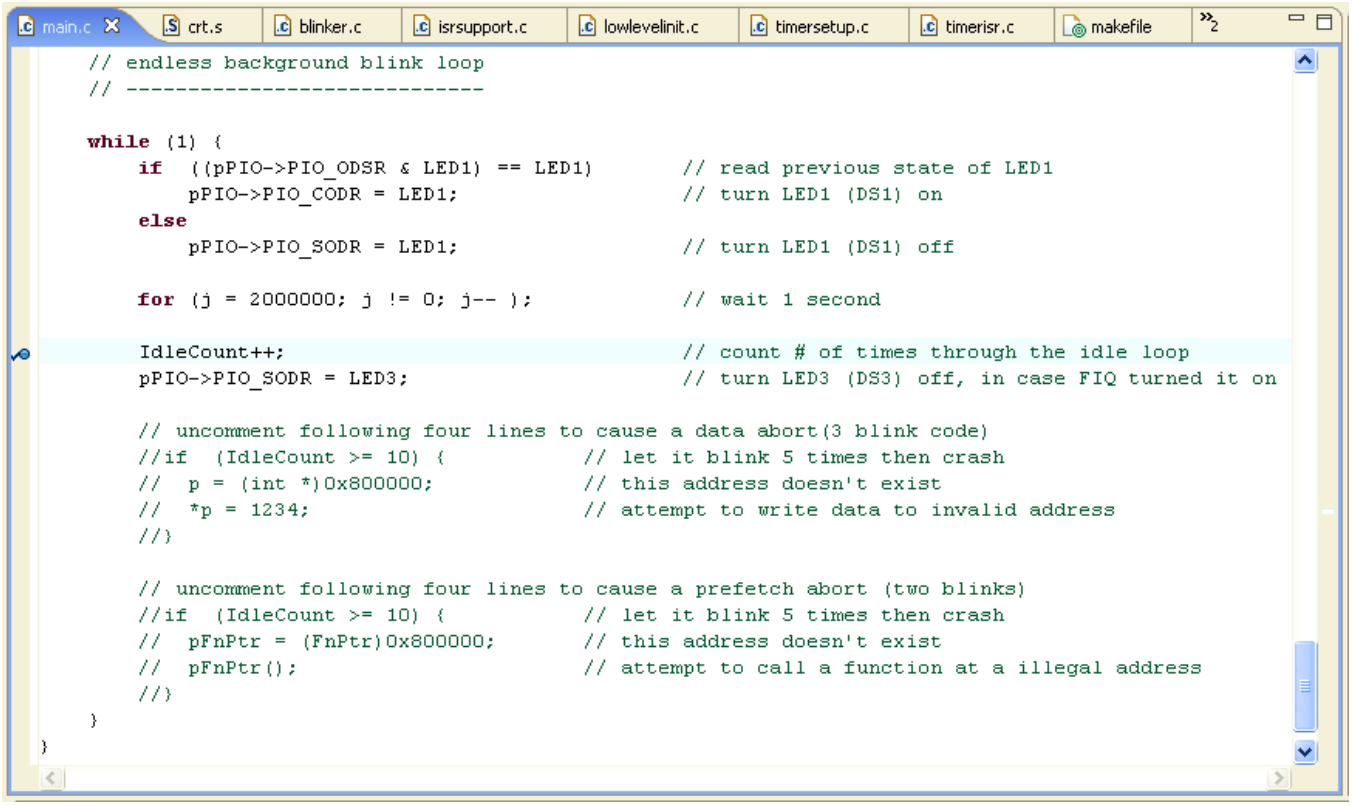
Get used to this sequence:



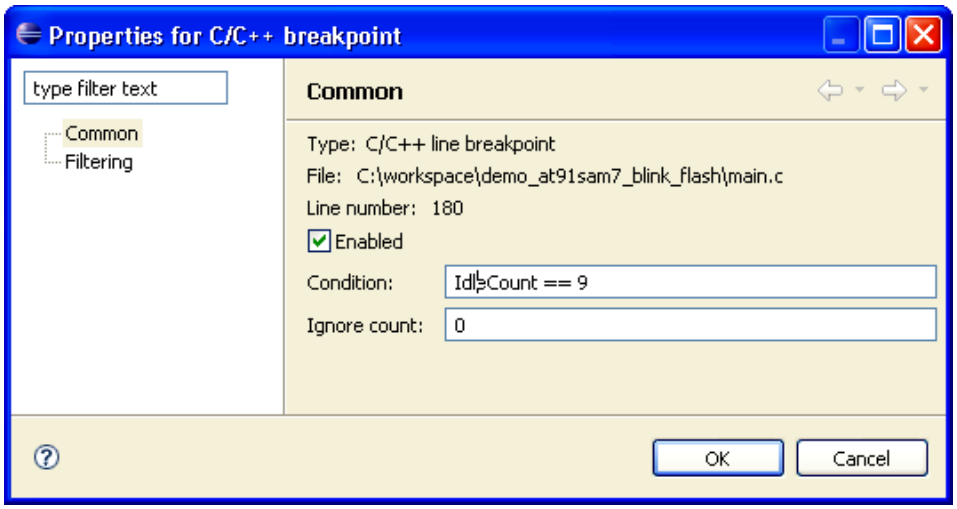
Now when you hit the **Run/Continue** button again, the program will blink 5 times and stop. Don't expect this feature to run in real-time. Each time the breakpoint is encountered the debugger will automatically continue until the "ignore" count is reached. This involves quite a bit of debugger communication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

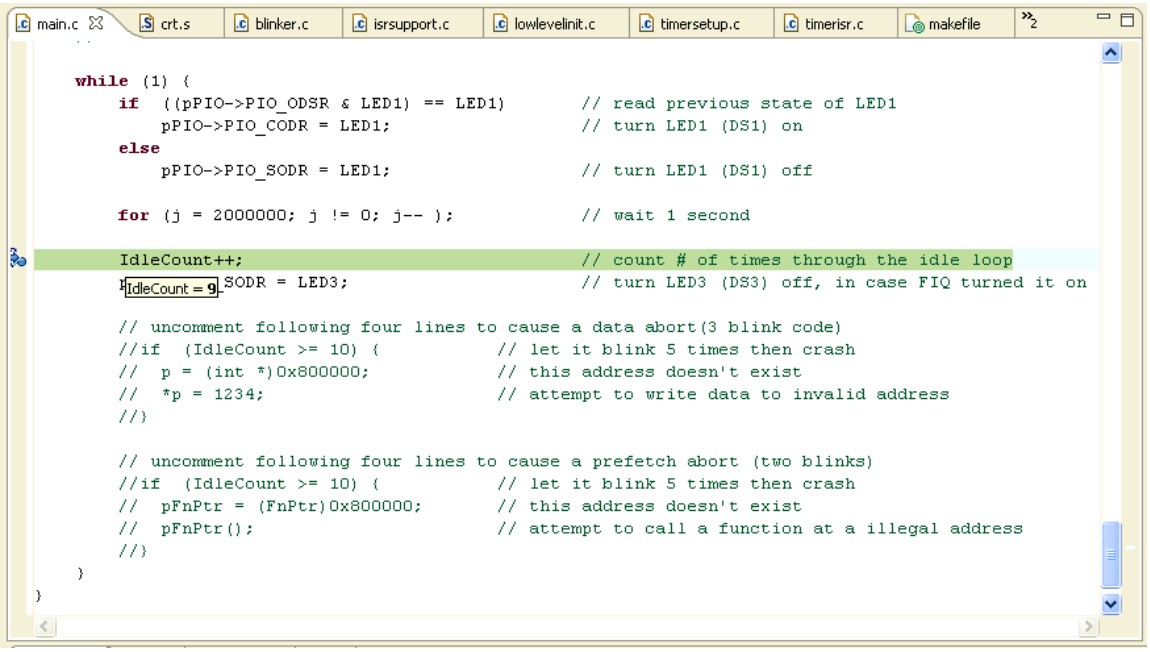


Right click on the breakpoint symbol and select "**Breakpoint Properties**". In the Breakpoint Properties window, set the condition text box to "**IdleCount == 9**".



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 IdleCount after the program has suspended on the breakpoint, it will display that the current value is 9.



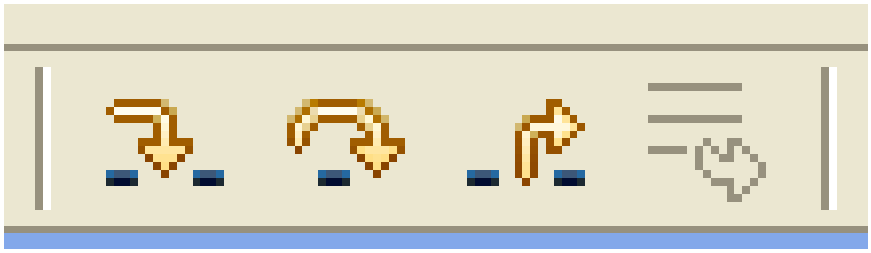

```
    while (1) {
        if  ((pPIO->PIO_ODSR & LED1) == LED1)       // read previous state of LED1
            pPIO->PIO_CODR = LED1;                  // turn LED1 (DS1) on
        else
            pPIO->PIO_SODR = LED1;                  // turn LED1 (DS1) off

        for (j = 2000000; j != 0; j-- );            // wait 1 second

        IdleCount++;                                // count # of times through the idle loop
        pPIO->PIO_SODR = LED3;                      // turn LED3 (DS3) off, in case FIQ turned it on

        // uncomment following four lines to cause a data abort(3 blink code)
        //if  (IdleCount >= 10) {                   // let it blink 5 times then crash
        //  p = (int *)0x800000;                    // this address doesn't exist
        //  *p = 1234;                              // attempt to write data to invalid address
        //}

        // uncomment following four lines to cause a prefetch abort (two blinks)
        //if  (IdleCount >= 10) {                   // let it blink 5 times then crash
        //  pFnPtr = (FnPtr)0x800000;               // this address doesn't exist
        //  pFnPtr();                               // attempt to call a function at a illegal address
        //}
    }
}
```

If you specify that it should break when IdleCount == 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 IdleCount. When IdleCount==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, 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.

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.

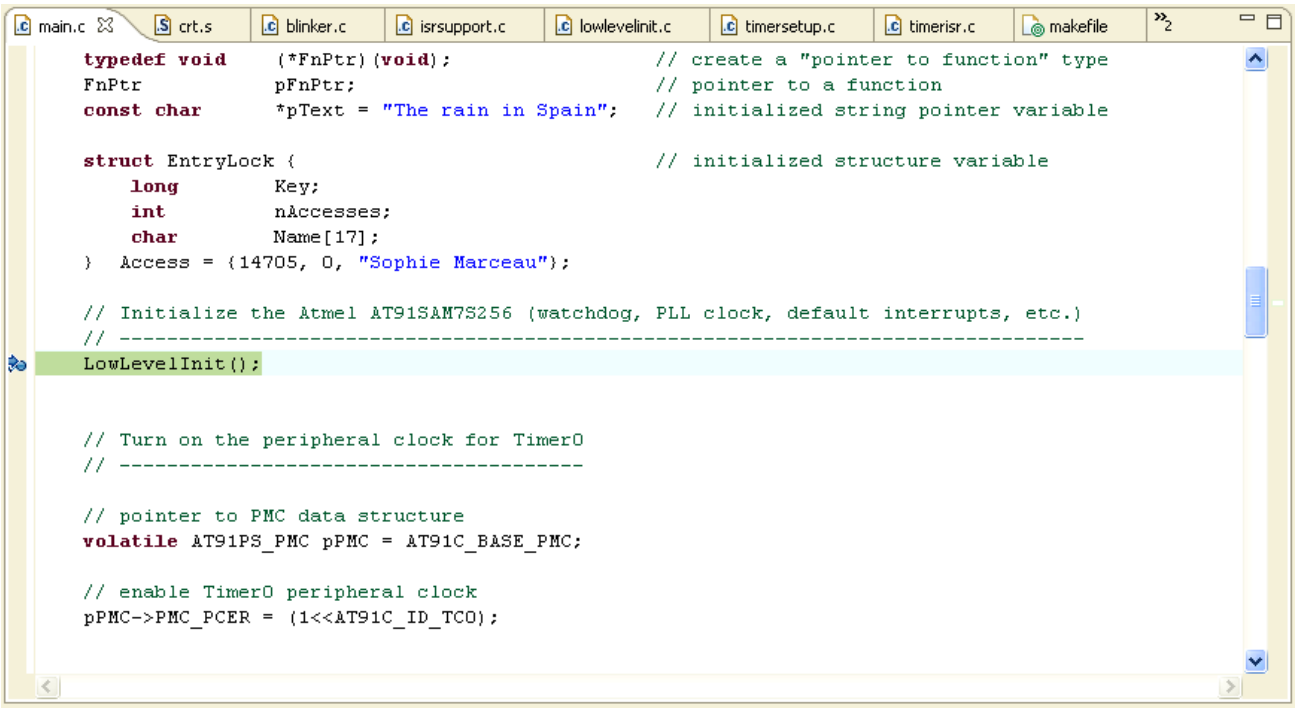
If cursor is on any other line, this will execute one instruction

Step Out Of

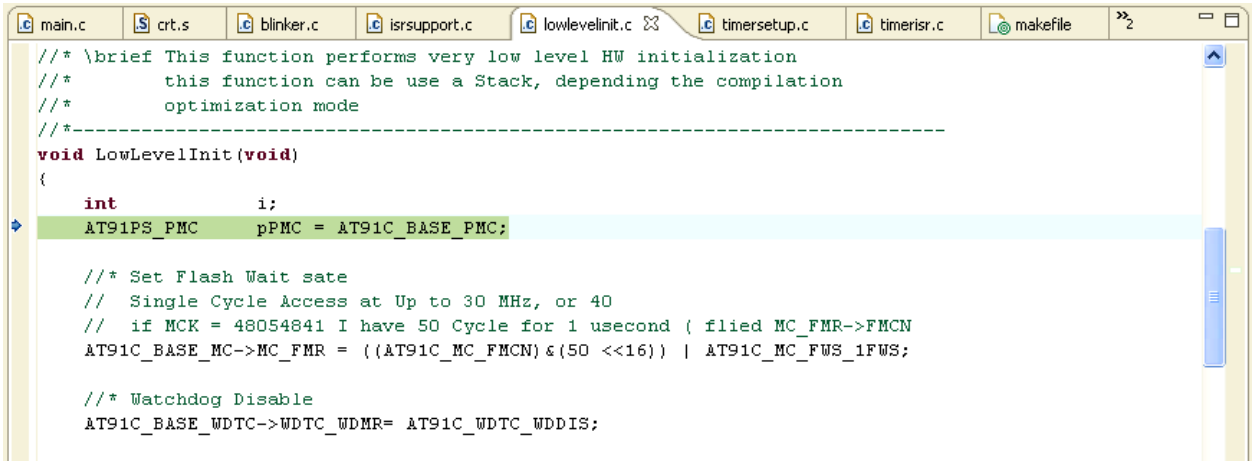
If the cursor is within a function, this will execute the remaining instructions in the function and stop on the next instruction after the function call.

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

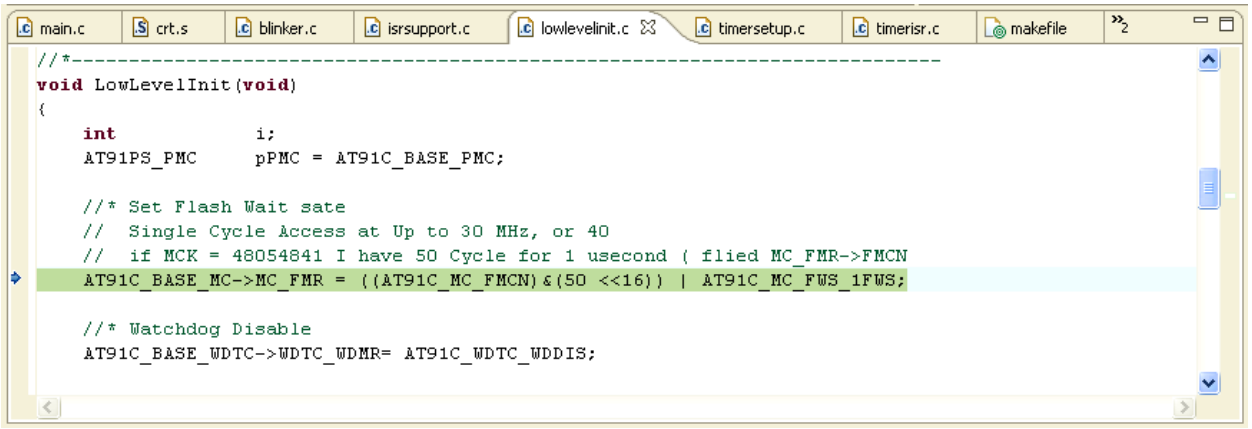
As a simple example, restart the debugger and set a breakpoint on the line that calls the **LowLevelInit( )** function. Hit the **Start** button to go to that breakpoint.



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

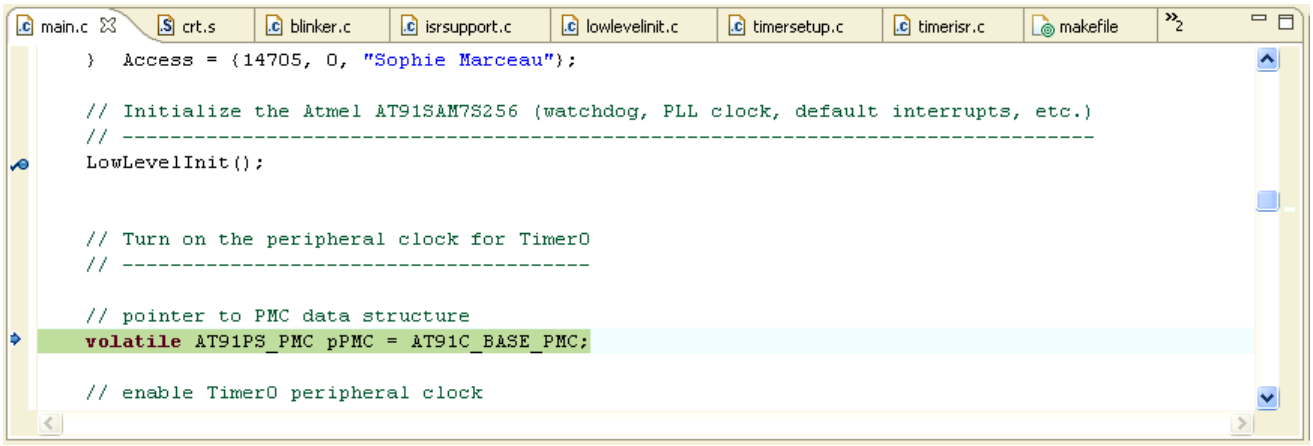


Click the “**Step Over**” button  The debugger will execute one instruction.



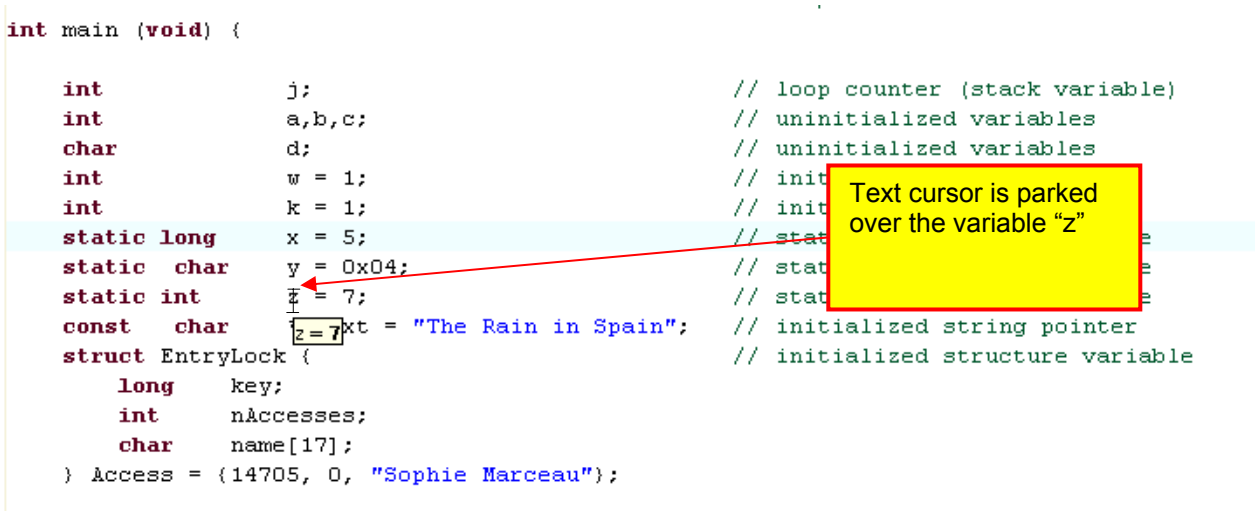
Notice that the “**Step Out Of**” button is illuminated. Click the “**Step Out Of**” button 

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



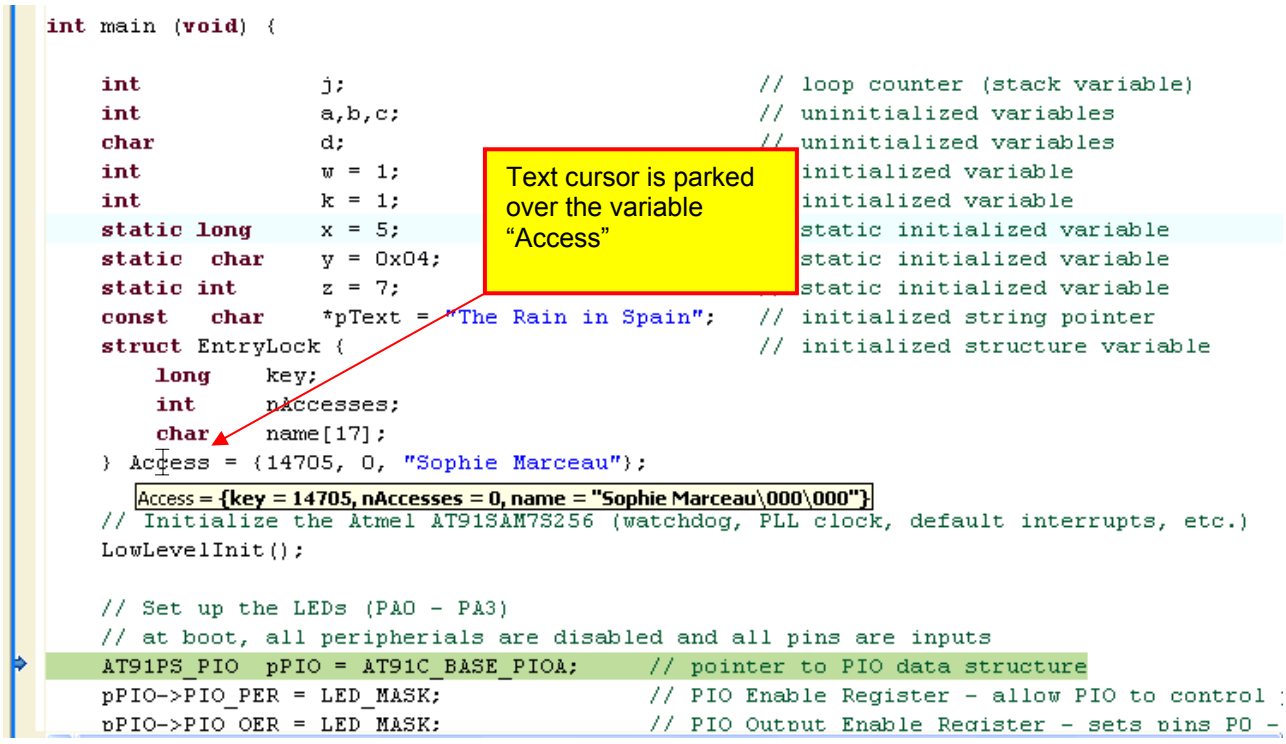
Inspecting and Modifying Variables

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 within the variable name.




```
int main (void) {

    int             j;                                  // loop counter (stack variable)
    int             a,b,c;                              // uninitialized variables
    char            d;                                  // uninitialized variables
    int             w = 1;                              // init
    int             k = 1;                              // init
    static long     x = 5;                              // stat
    static  char    y = 0x04;                           // stat
    static int      z = 7;                              // stat
    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"};
```

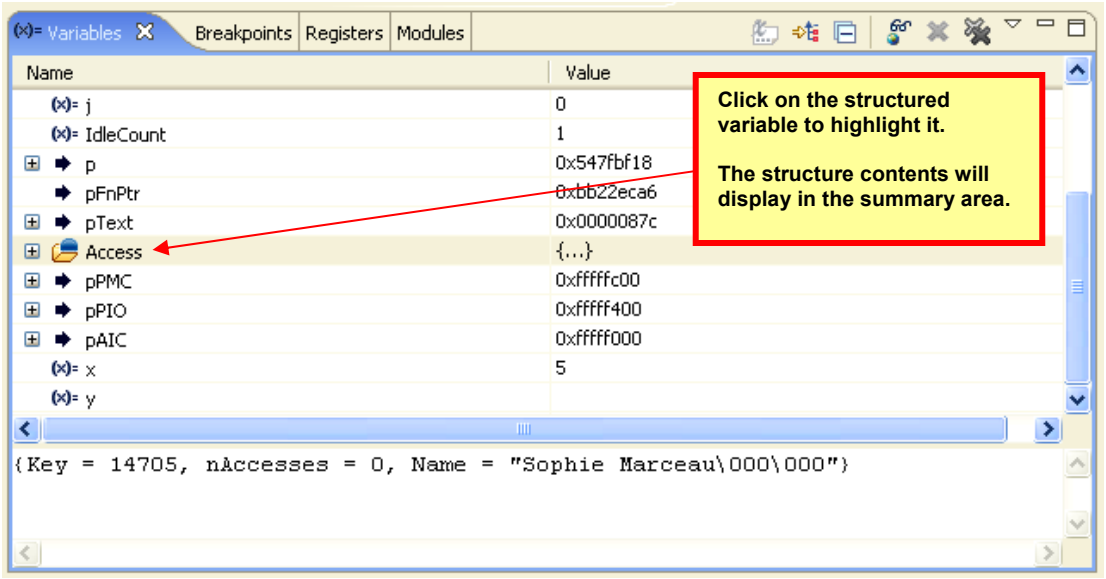
For a structured variable, parking the cursor over the variable name will show the values of all the internal component parts.



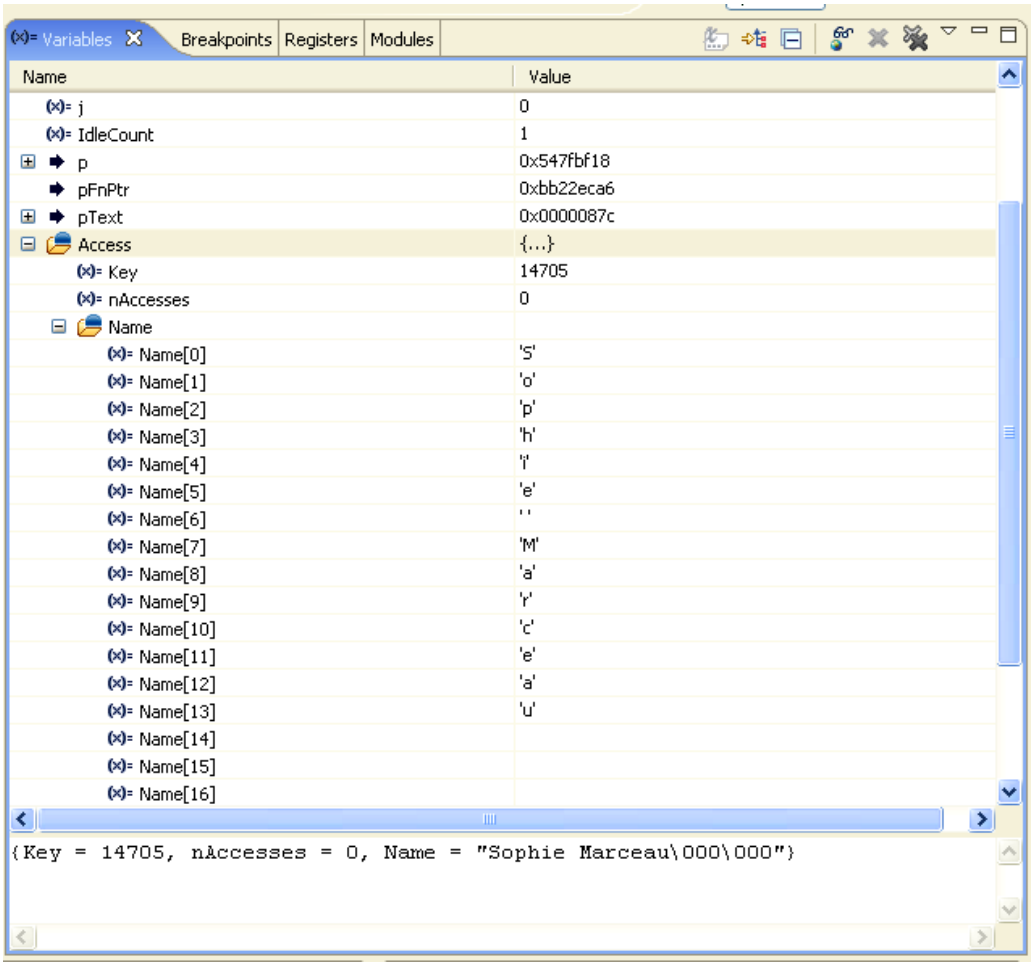

```
int main (void) {

    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
    pPIO->PIO_OER = LED_MASK;                // PIO Output Enable Register - sets pins PO -
```

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.



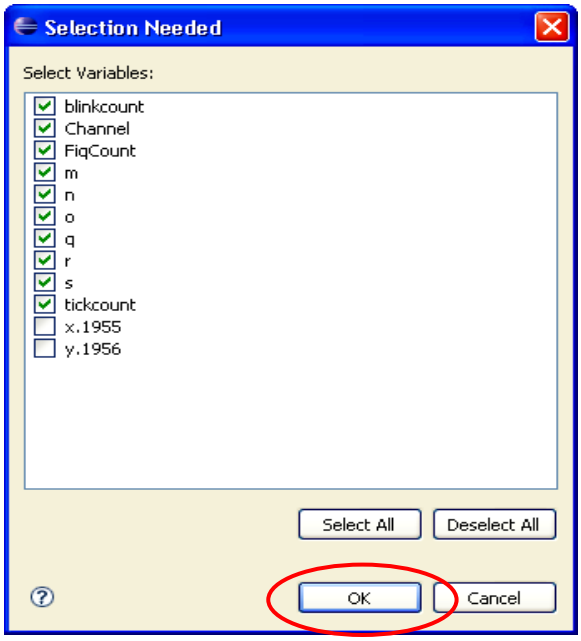
The Variables view can also expand structures. Just click on any "+" signs you see to expand the structure and view its contents.



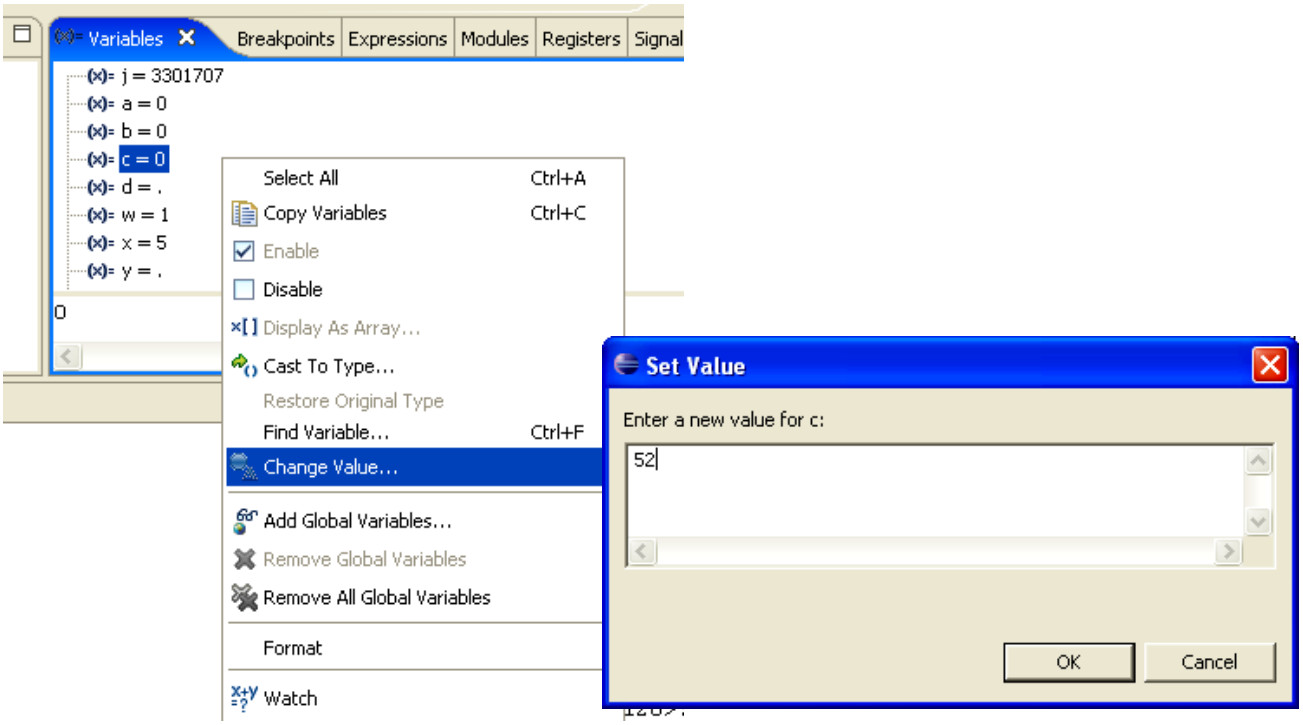
Global variables have to be individually selected for display within the “Variables” view.

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

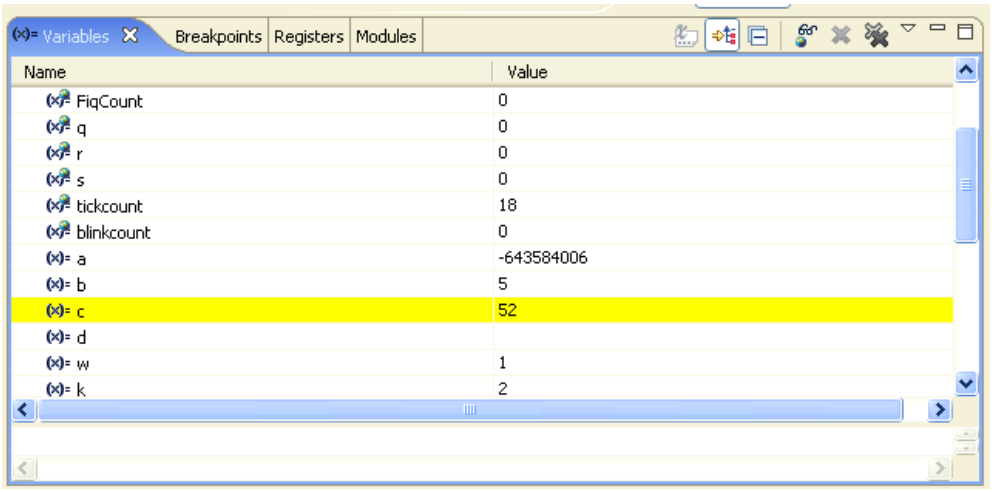
Check the variables you want to display and then click “**OK**” to add them to the **Variables** view,



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.



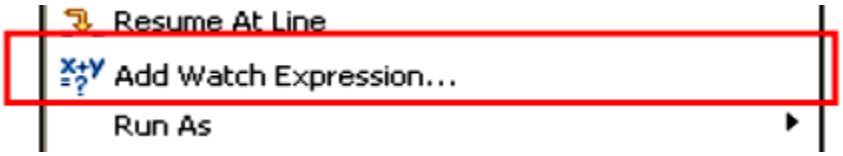
Now the "**Variables**" view should show the new value for the variable "c". Note that it has been backlighted to the color yellow to indicate that it has been changed.



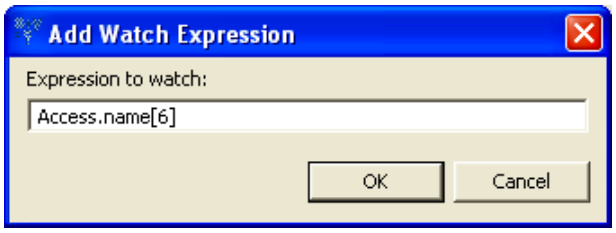
Watch Expressions

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.

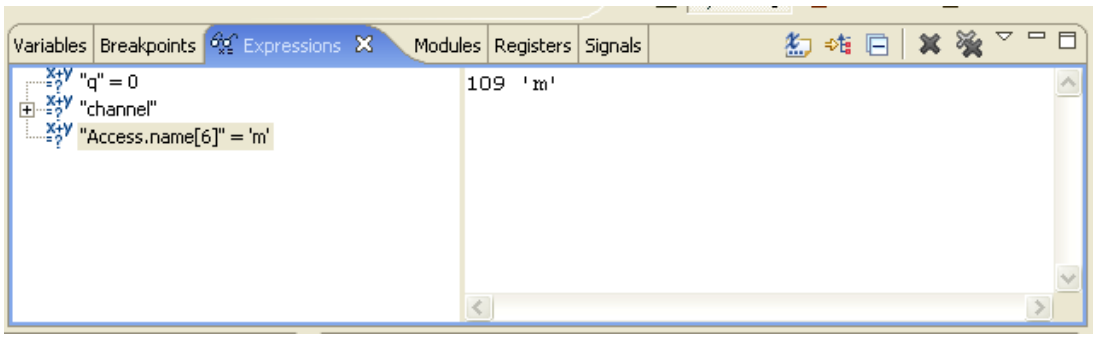
For example, to display the 6th character of the name in the structured variable "**Access**", bring up the right-click menu and select "**Add Watch Expression…**".



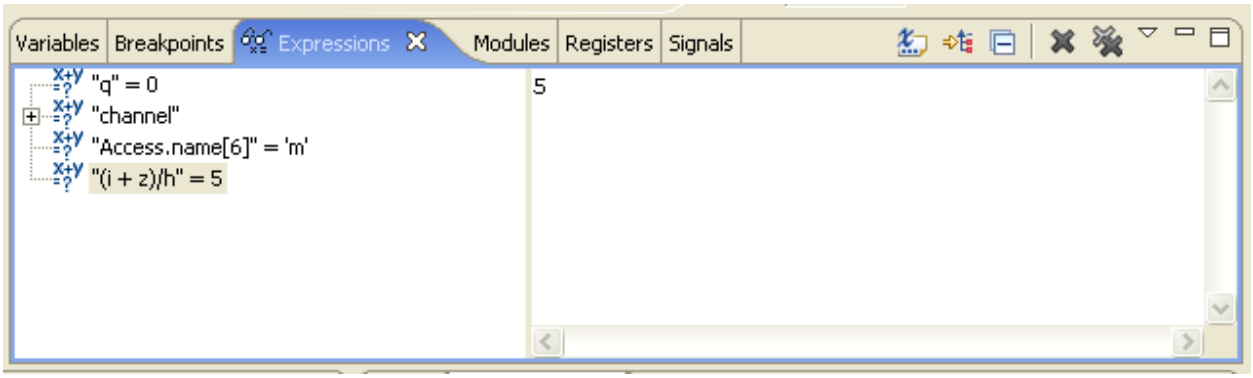
Enter the fully qualified name of the 6th character of the name[] array.



Note that it now appears in the "Expressions" view.

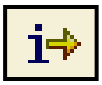


You can type in very complicated expressions. Here we defined the expression (i + z)/h



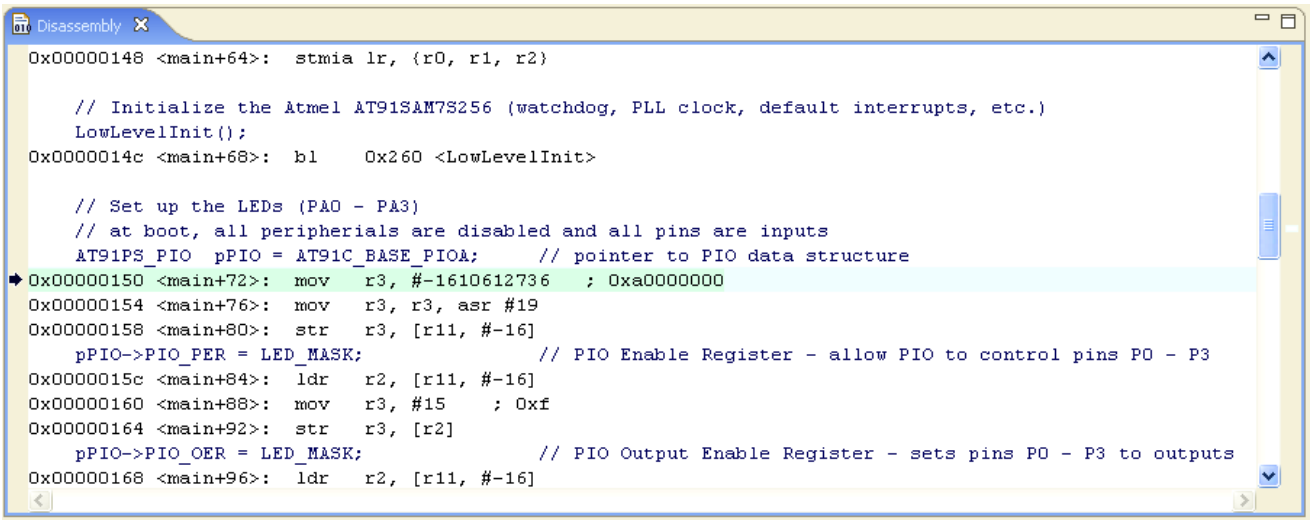
Assembly Language Debugging

The Debug perspective includes an Assembly Language view.

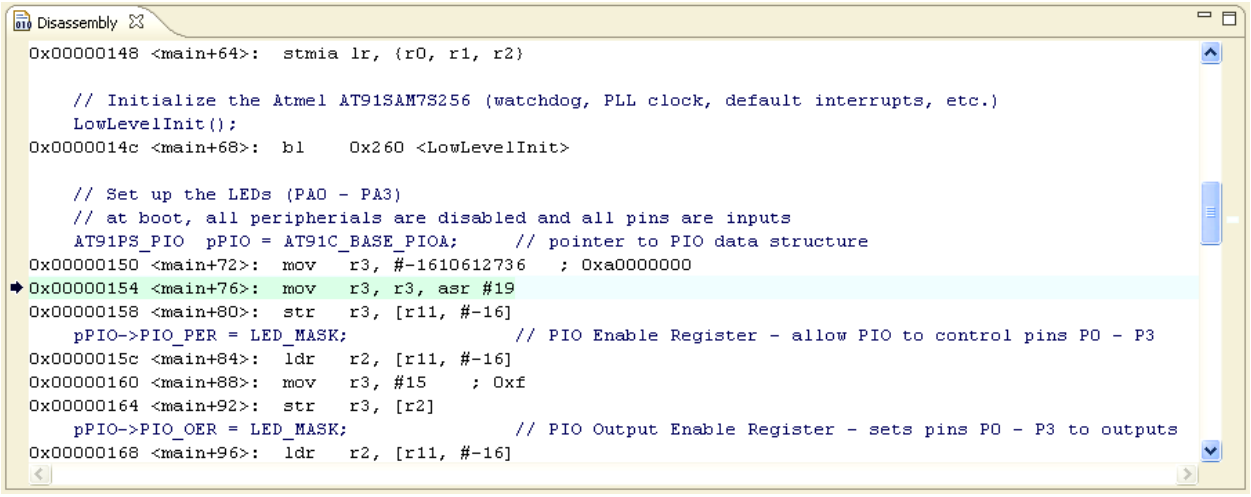
If you click on the Instruction Stepping Mode toggle button in the Debug view, 

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.



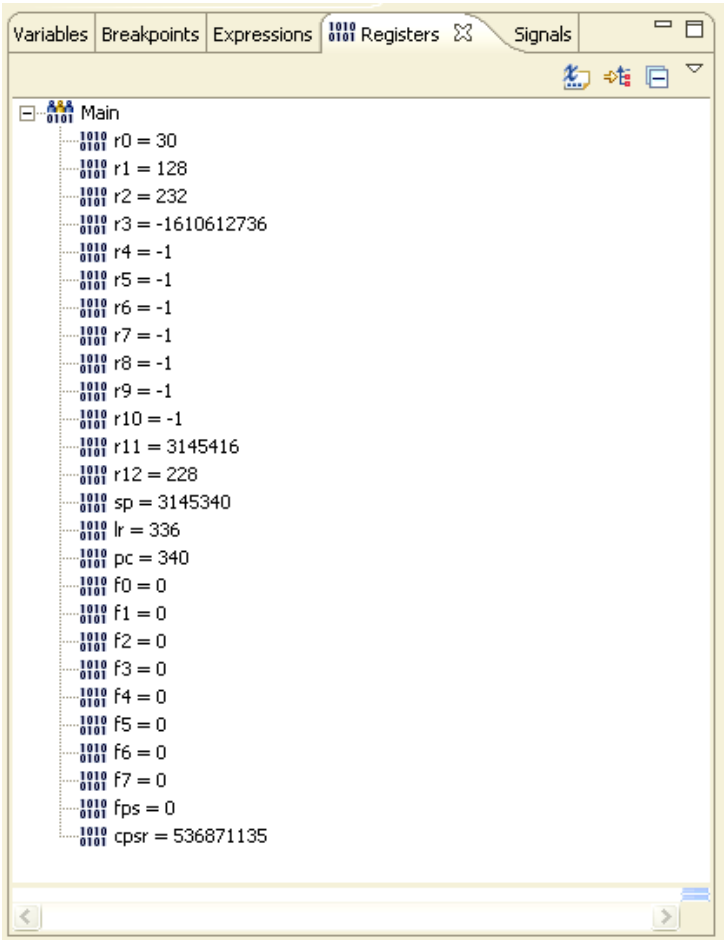
The "**Step Into**" and "**Step Out Of**" buttons work in the same was as for C code.

Note: It pains the author greatly to report that the Eclipse 3.2 release has a bug wherein assembly language breakpoints do not function. Monitor the AT91 chat board on Yahoo to see when this is resolved. Truthfully, you shouldn't be programming in assembly language anyway!

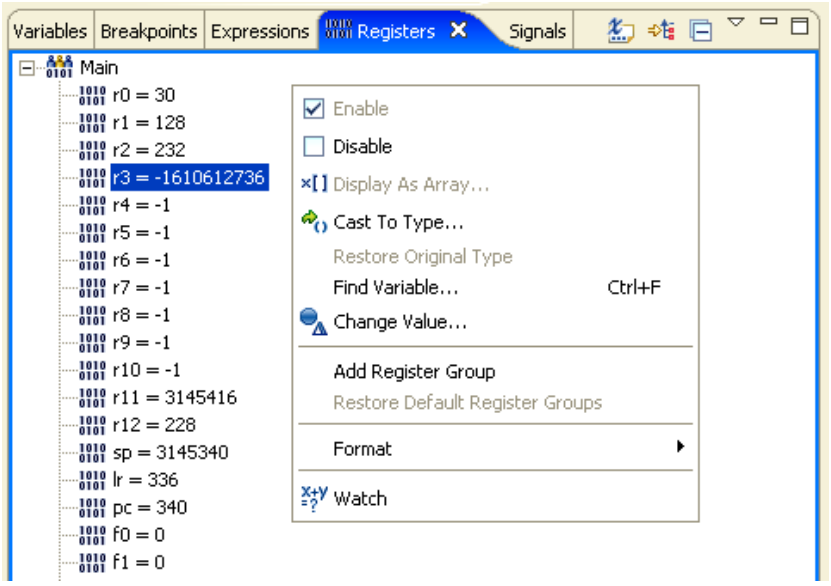
Inspecting Registers

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.

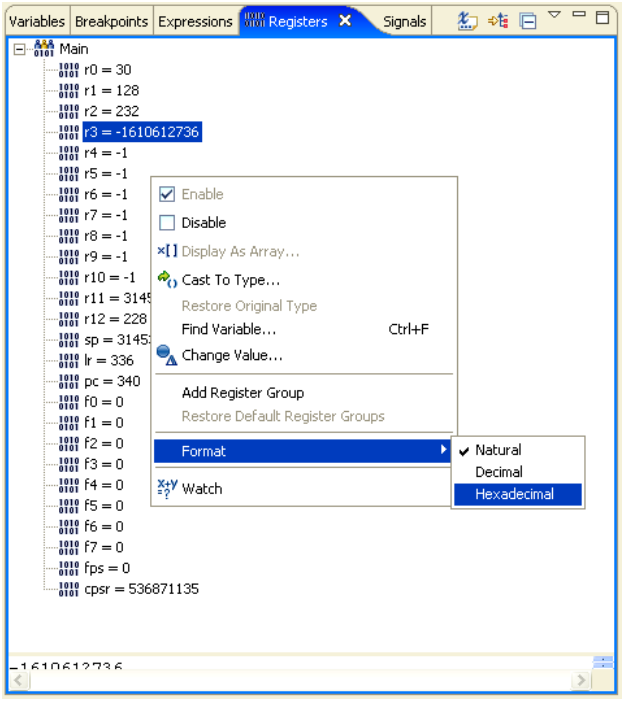
Click on the "**+**" symbol next to Main and the registers will appear. The Atmel AT91SAM7S256 doesn't have any floating point registers so registers F0 through FPS are not applicable.



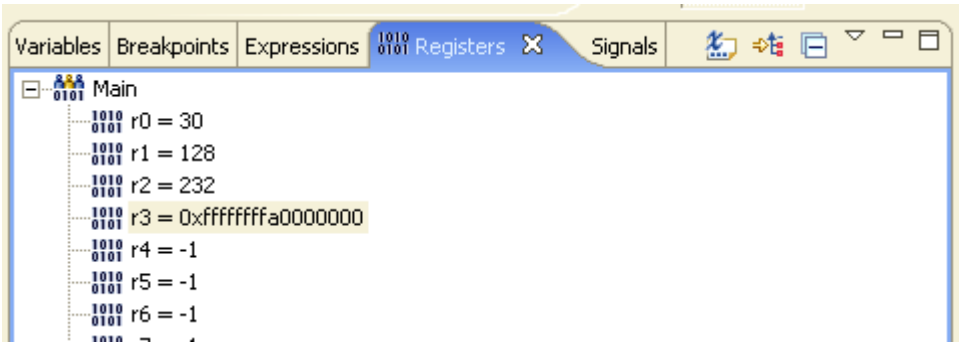
If you don't like a particular register's numeric format, you can click to highlight it and then bring up the right-click menu. You can, of course, highlight them all if desired.



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



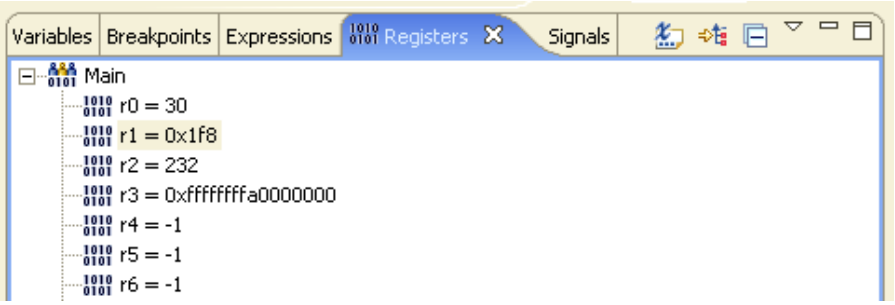
Now R3 is displayed in hexadecimal.



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.

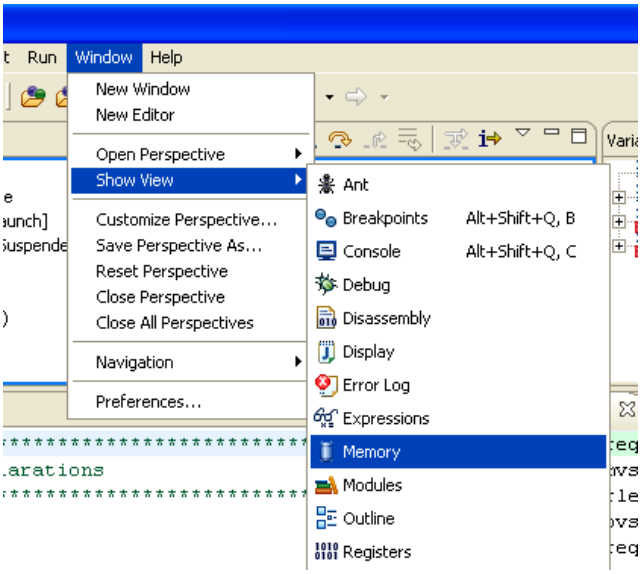
The register display now shows the new value for R1 (we also changed the display format to hexadecimal using the right-click menu).



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.

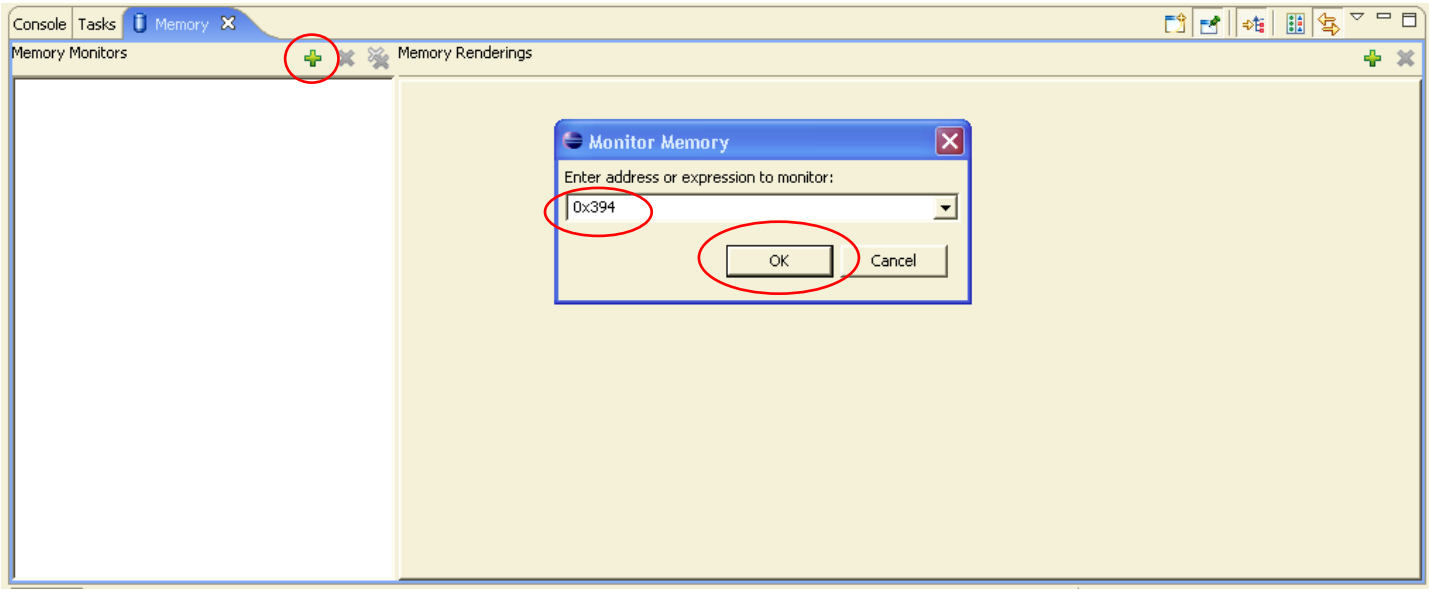
Inspecting Memory

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.



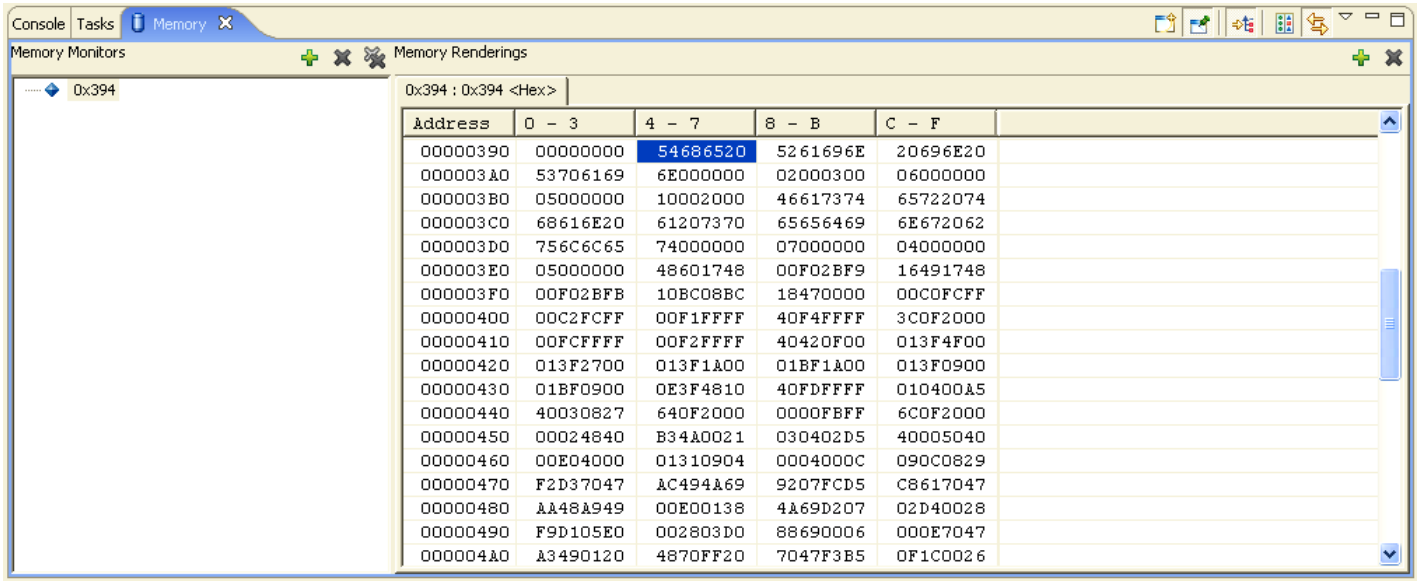
The memory view appears in the "**Console**" view at the bottom of the Debug perspective. At this point, nothing has been defined. Memory is displayed as one or more "**memory monitors**". You can create a memory monitor by clicking on the "+" symbol.

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



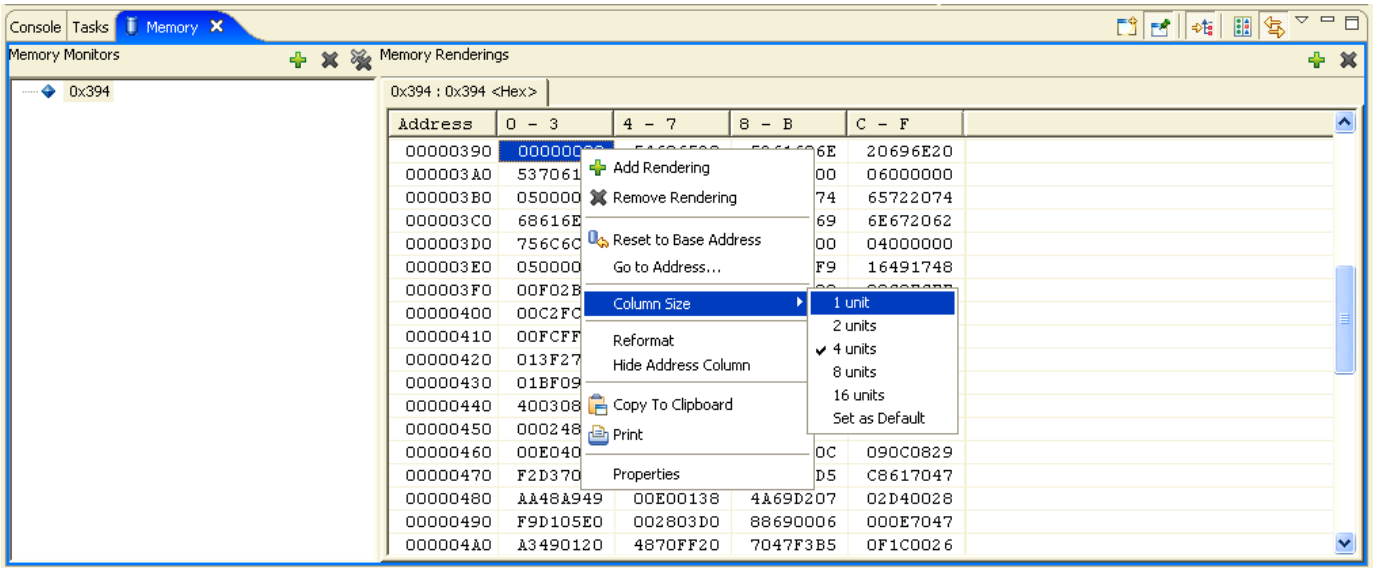
The memory monitor is created, although it defaults to 4-byte display mode. The display of the address columns and the associated memory contents is called a “**Rendering**”.

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.

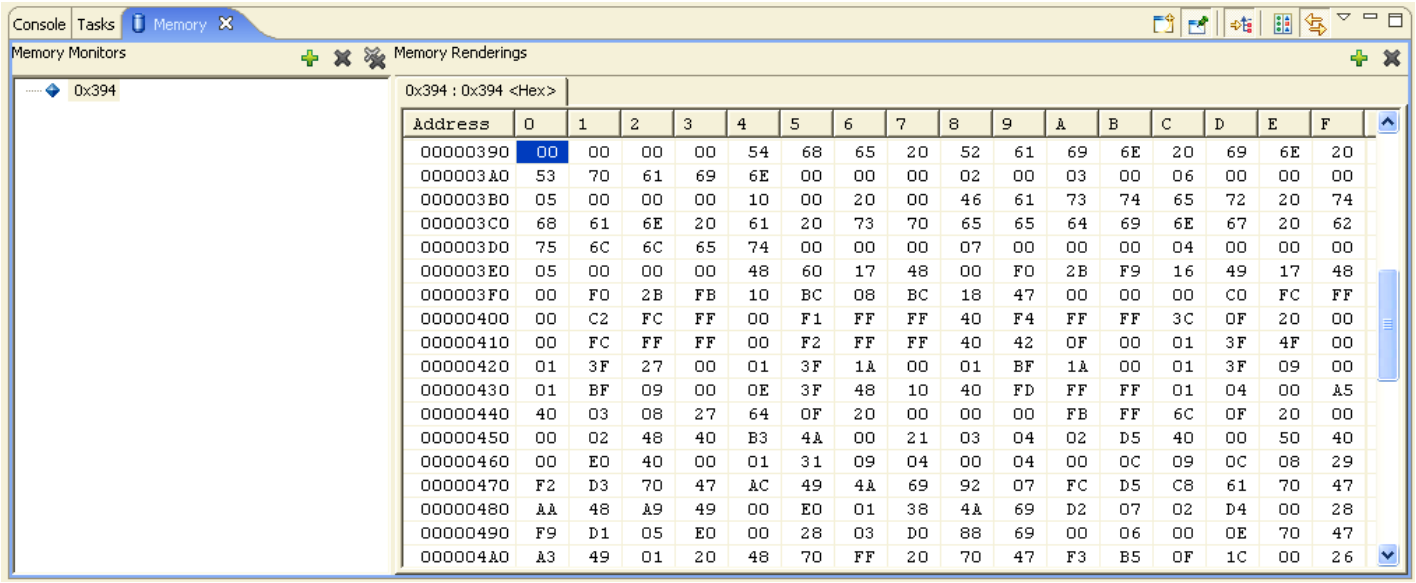


There’s also a “**Go to Address…**” right-click menu option that will jump all over memory for you.

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



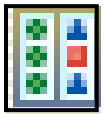
This will repaint the memory rendering in Byte format as shown below.



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.

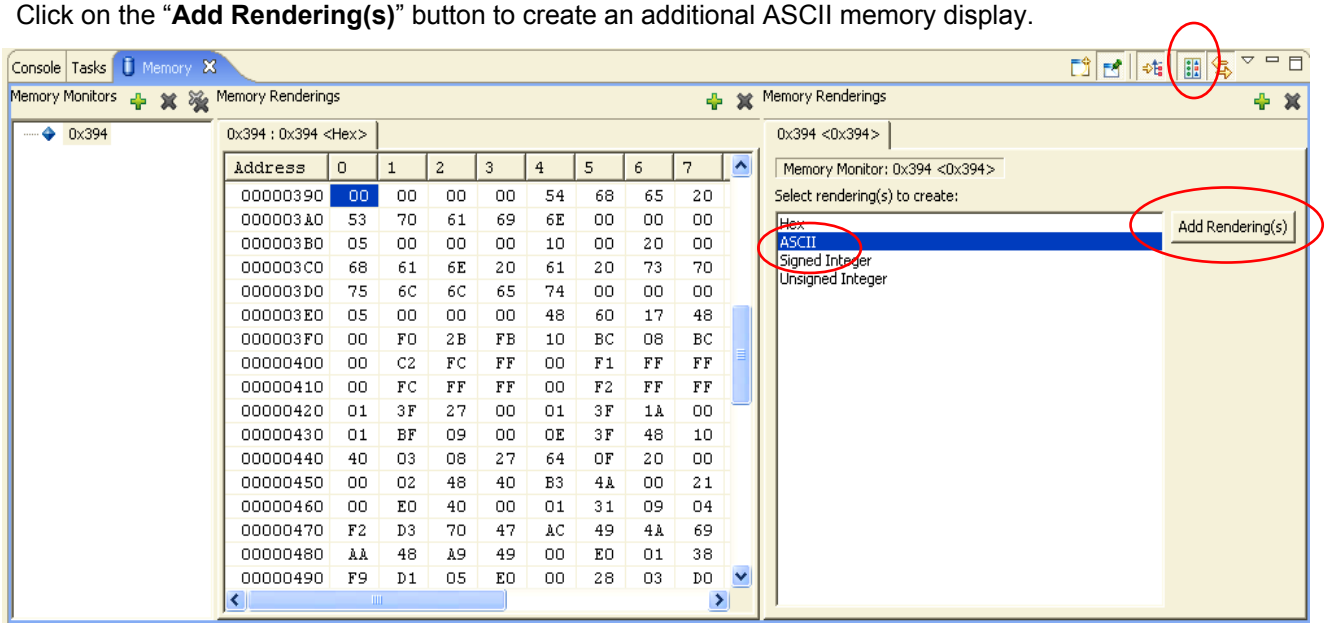
Now we will add a second rendering that will display the memory monitor in ASCII.

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.



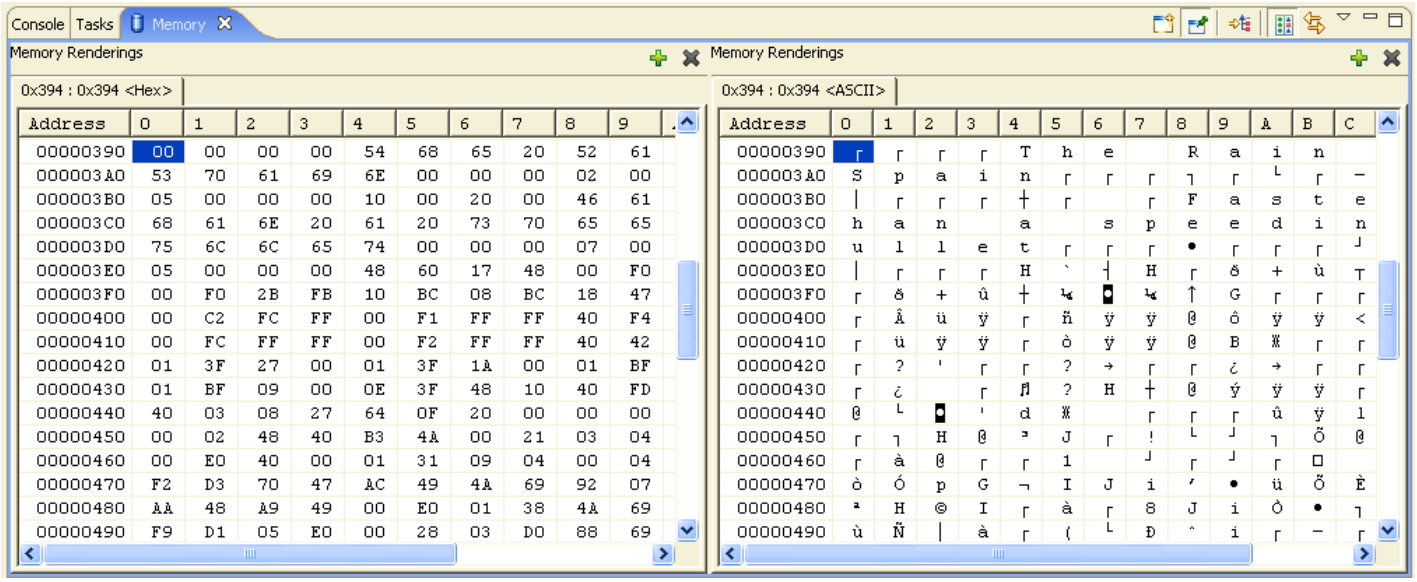
Now we have an additional display of the hex values and the corresponding ASCII characters.

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 not address this idea any further.

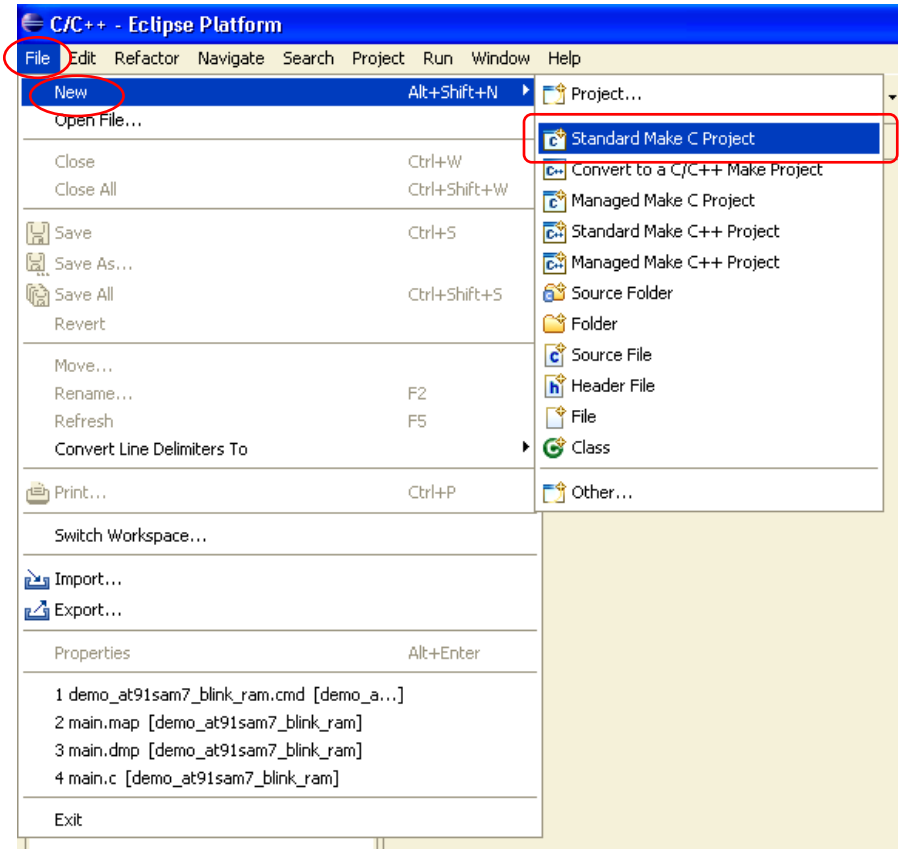
In the FLASH example shown previously, the OpenOCD utility permitted the Eclipse debugger to use the two on chip breakpoint units; allowing a breakpoint to be set in FLASH. This limits us to just two breakpoints. Note also that the OpenOCD setup converted every Eclipse breakpoint specification into a 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.

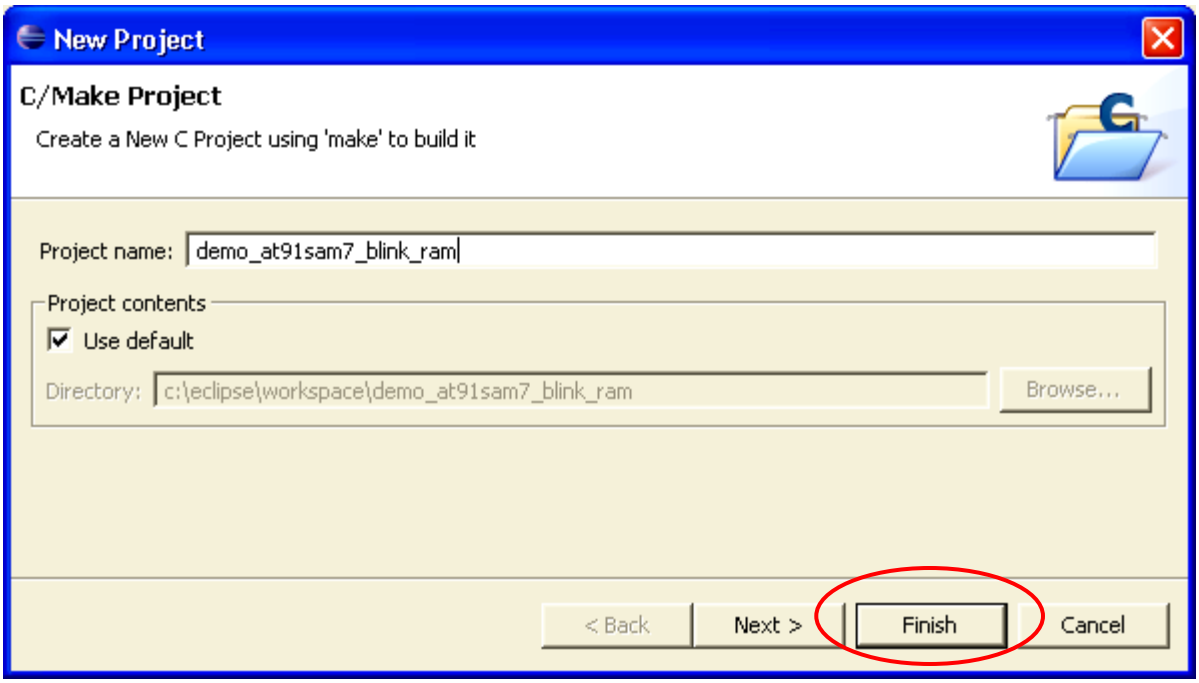
With this approach, we get an unlimited number of software breakpoints and can use the JTAG device to download the code (we don't have to use the OpenOCD FLASH programming facility). The disadvantage, of course, is that the application is limited to 64 Kbytes.

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

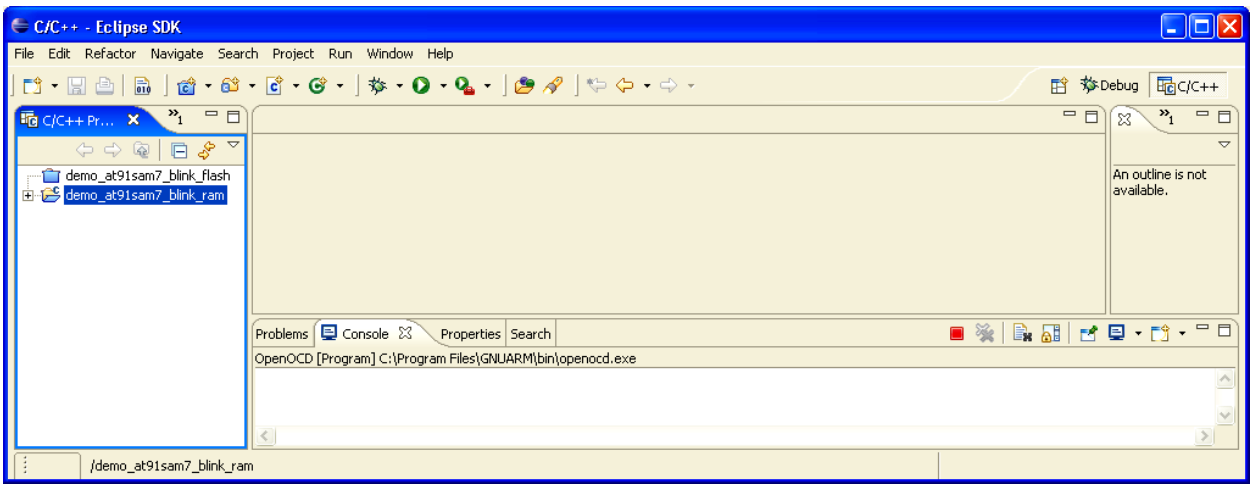
Click on "**File – New – Standard Make C Project**" as shown below.



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 **c:\atmel_tutorial_source** area for the project “**demo_at91sam7_blink_ram**” using the techniques learned earlier.

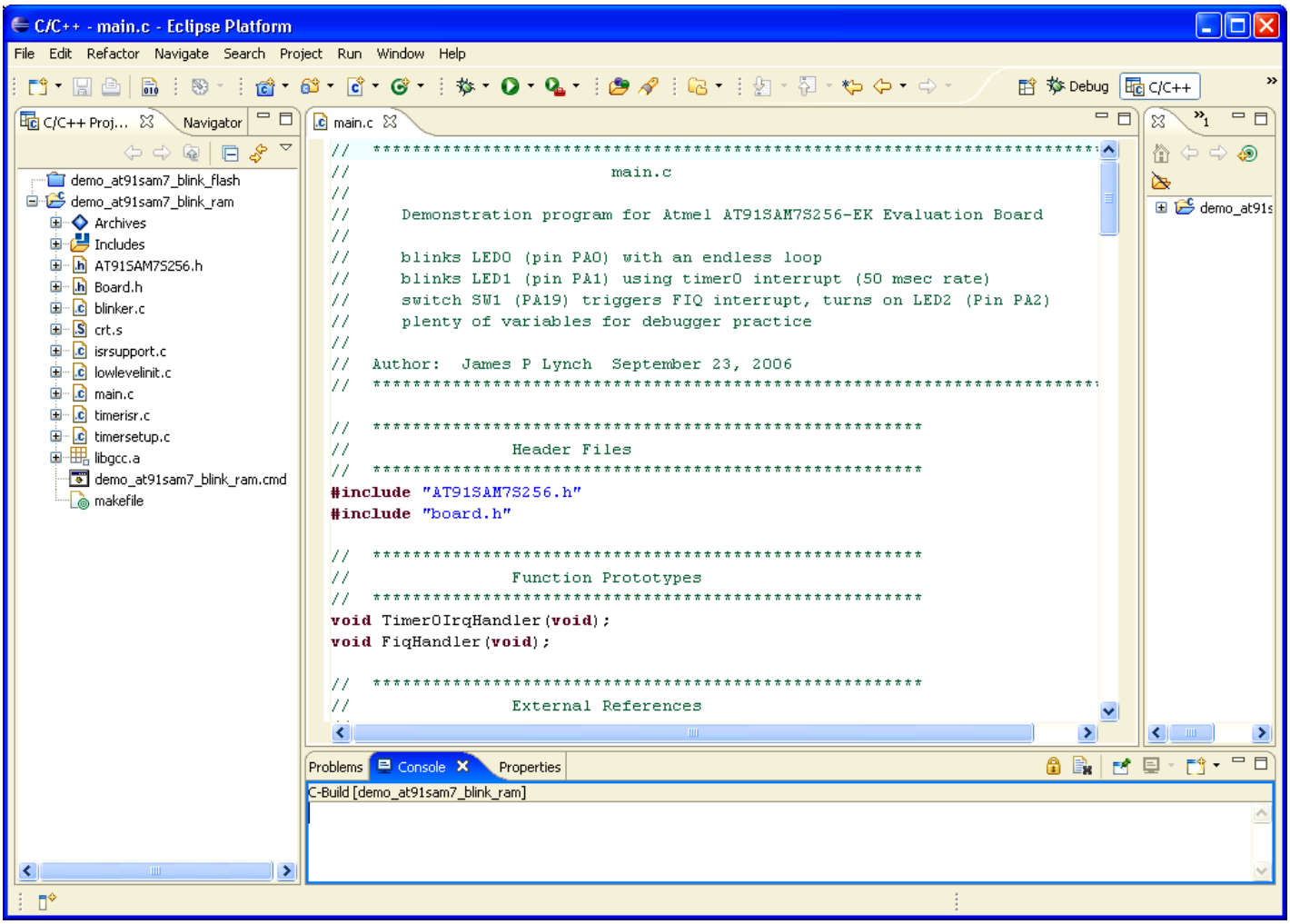
Only two files are different from the FLASH version:

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


     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 RAM, 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_ram.cmd
     directive to the linker flags in the makefile. For example,

                        LFLAGS  =  -Map main.map -nostartfiles -T demo_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 = (0x00000FFFF + 1) - 4  =  0x00001000 - 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         |
           .         |                                 |
           .         |                                 |
          RAM        |---------------------------------|0x0000FF5C
           .         |                                 |
           .         |                                 |
           .         |    IRQ Stack  128 bytes         |
           .         |                                 |
           .         |                                 |
           .         |---------------------------------|0x0000FEDC
           .         |                                 |
           .         |    SVC Stack  16 bytes          |
           .         |                                 |
           .         |---------------------------------|0x0000FECC
           .         |                                 |
           .         |     stack area for user program |
           .         |                                 |
           .         |                                 |
           .         |                                 |
           .         |          free ram               |
           .         |                                 |
           .         |.................................|0x000008F8 <---------- _bss_end
           .         |                                 |
           .         |  .bss   uninitialized variables |
           .         |.................................|0x000008E0 <---------- _bss_start, _edata
           .         |                                 |
           .         |                                 |
           .         |  .data  initialized variables   |
           .         |                                 |
           .         |---------------------------------|0x000008A4 <----------- _etext
           .         |                                 |
           .         |            C code               |
           .         |                                 |
           .         |                                 |
           .         |---------------------------------|0x0000015C  main()
           .         |                                 |
           .         |    Startup Code  (crt.s)        |
           .         |         (assembler)             |
           .         |                                 |
           .         |                                 |
           .         |                                 |
           .         |---------------------------------|0x00000020
           .         |                                 |
           .         | Interrupt Vector Table          |
           .         |          32 bytes               |
           .--------->|---------------------------------|0x00000000 _vec_reset


Author:  James P. Lynch
```

```
   ********************************************************************************************************* */




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

/* specify the AT91SAM7S256s  */
MEMORY
{
    flash : ORIGIN = 0, LENGTH = 256K  /* FLASH EPROM  (not used in this example) */
    ram   : ORIGIN = 0, LENGTH = 64K   /* static RAM area (after debugger re-maps RAM to address 0x00000000) */
}


/* define a global symbol _stack_end  (see analysis in annotation above) */
_stack_end = 0x00FFFC;

/* 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 */
    } >ram              /* put all the above into FLASH */

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

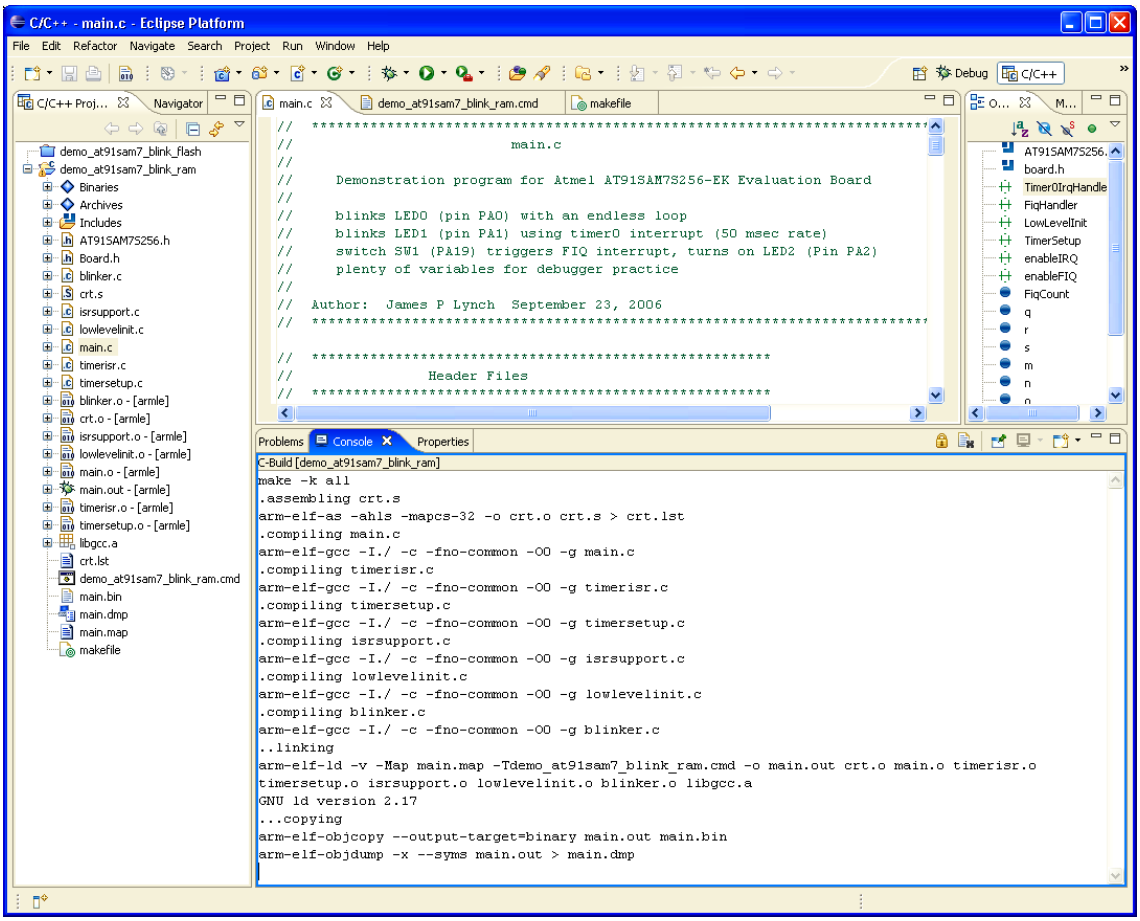
MAKEFILE.MAK

```
# ****************************************************************
# *     Makefile for Atmel AT91SAM7S256 - ram execution          *
# *                                                              *
# *                                                              *
# * James P Lynch  September 26, 2006                            *
# ****************************************************************

NAME   = demo_at91sam7_blink_ran

# variables
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

OBJECTS = crt.o  main.o timerisr.o timersetup.o isrsupport.o lowlevelinit.o blinker.o


# make target called by Eclipse (Project -> Clean ...)
clean:
        -rm $(OBJECTS) crt.lst main.lst main.out main.bin main.hex main.map main.dmp


#make target called by Eclipse  (Project -> Build Project)
all:  main.out
        @ echo "...copying"
        $(CP) $(CPFLAGS) main.out main.bin
        $(OD) $(ODFLAGS) main.out > main.dmp

main.out: $(OBJECTS) demo_at91sam7_blink_ram.cmd
        @ echo "..linking"
        $(LD) $(LFLAGS) -o main.out $(OBJECTS) libgcc.a

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

main.o: main.c  AT91SAM7S256.h board.h
        @ echo ".compiling main.c"
        $(CC) $(CFLAGS) main.c

timerisr.o: timerisr.c  AT91SAM7S256.h board.h
        @ echo ".compiling timerisr.c"
        $(CC) $(CFLAGS) timerisr.c

lowlevelinit.o: lowlevelinit.c AT91SAM7S256.h board.h
        @ echo ".compiling lowlevelinit.c"
        $(CC) $(CFLAGS) lowlevelinit.c

timersetup.o: timersetup.c AT91SAM7S256.h board.h
        @ echo ".compiling timersetup.c"
        $(CC) $(CFLAGS) timersetup.c

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

blinker.o: blinker.c AT91SAM7S256.h board.h
        @ echo ".compiling blinker.c"
        $(CC) $(CFLAGS) blinker.c
```

```
# ********************************************************************************************
#                            FLASH PROGRAMMING      (using OpenOCD and Amontec JTAGKey)
#
# Alternate make target for flash programming only
#
# You must create a special Eclipse make target (program) to run this part of the makefile
# (Project -> Create Make Target...  then set the Target Name and Make Target to "program")
#
# OpenOCD is run in "batch" mode with a special configuration file and a script file containing
# the flash commands. When flash programming completes, OpenOCD terminates.
#
# Note that the make file below creates the script file of flash commands "on the fly"
#
# Programmers: Martin Thomas, Joseph M Dupre, James P Lynch
# ********************************************************************************************

# specify output filename here (must be *.bin file)
TARGET = main.bin

# specify the directory where openocd executable resides (openocd-ftd2xx.exe or openocd-pp.exe)
OPENOCD_DIR = 'c:\Program Files\openocd-2006re93\bin\'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger)
#OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (pick the one for your device)
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-wiggler-flash-program.cfg
#OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-jtagkey-flash-program.cfg
OPENOCD_CFG = $(OPENOCD_DIR)at91sam7s256-armusbocd-flash-program.cfg

# specify the name and folder of the flash programming script file
OPENOCD_SCRIPT = c:\temp\temp.ocd

# program the AT91SAM7S256 internal flash memory
program: $(TARGET)
        @echo "Preparing OpenOCD script..."
        @cmd /c 'echo wait_halt > $(OPENOCD_SCRIPT)'
        @cmd /c 'echo armv4_5 core_state arm >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo flash write 0 $(TARGET) 0x0 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo mww 0xfffffd08 0xa5000401 >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo reset >> $(OPENOCD_SCRIPT)'
        @cmd /c 'echo shutdown >> $(OPENOCD_SCRIPT)'
        @echo "Flash Programming with OpenOCD..."
        $(OPENOCD) -f $(OPENOCD_CFG)
        @echo "Flash Programming Finished."
```

Build the RAM Project

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

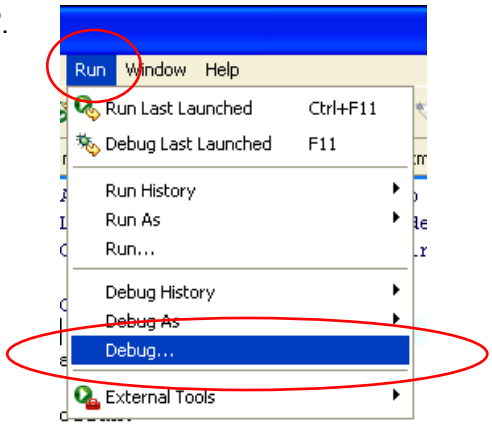


In this version, we will be using the “**main.out**” file to download the executable into RAM via the JTAG.

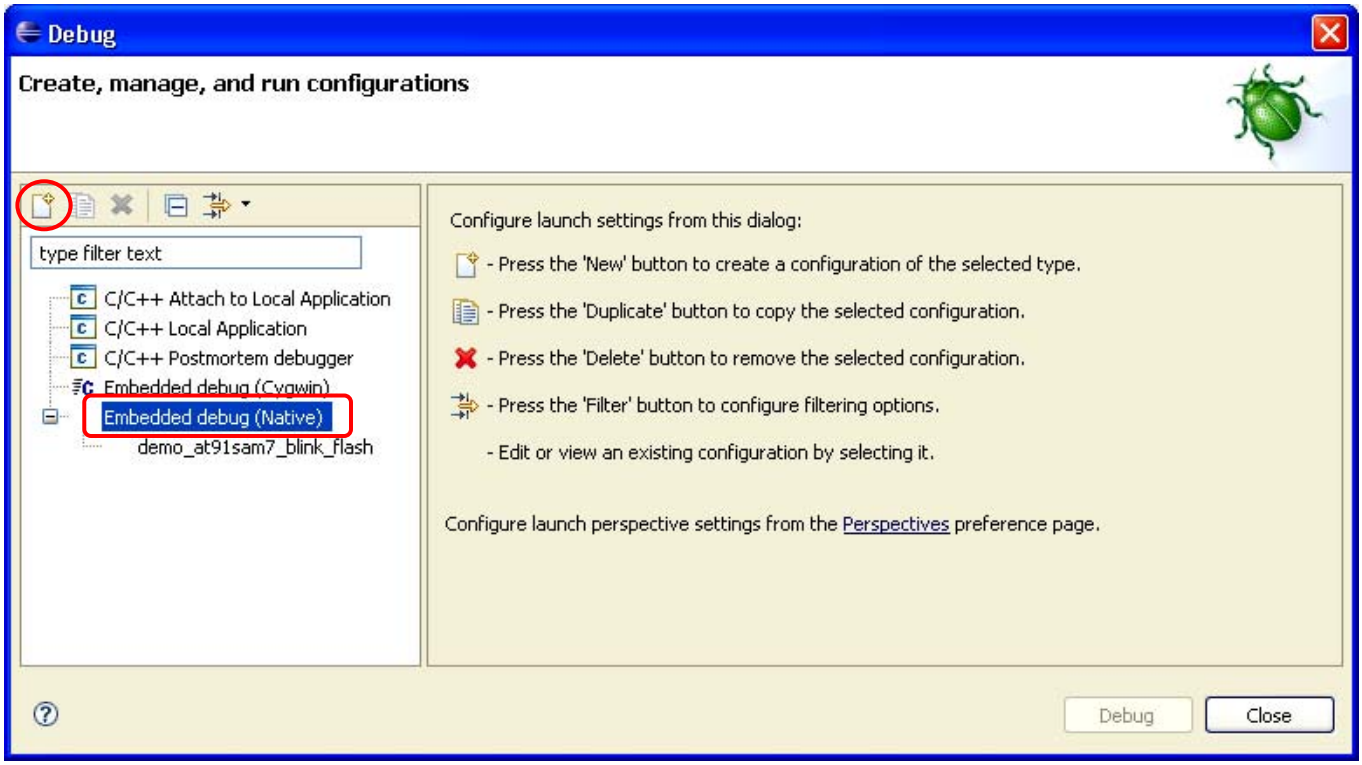
Create an Embedded Debug Launch Configuration

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

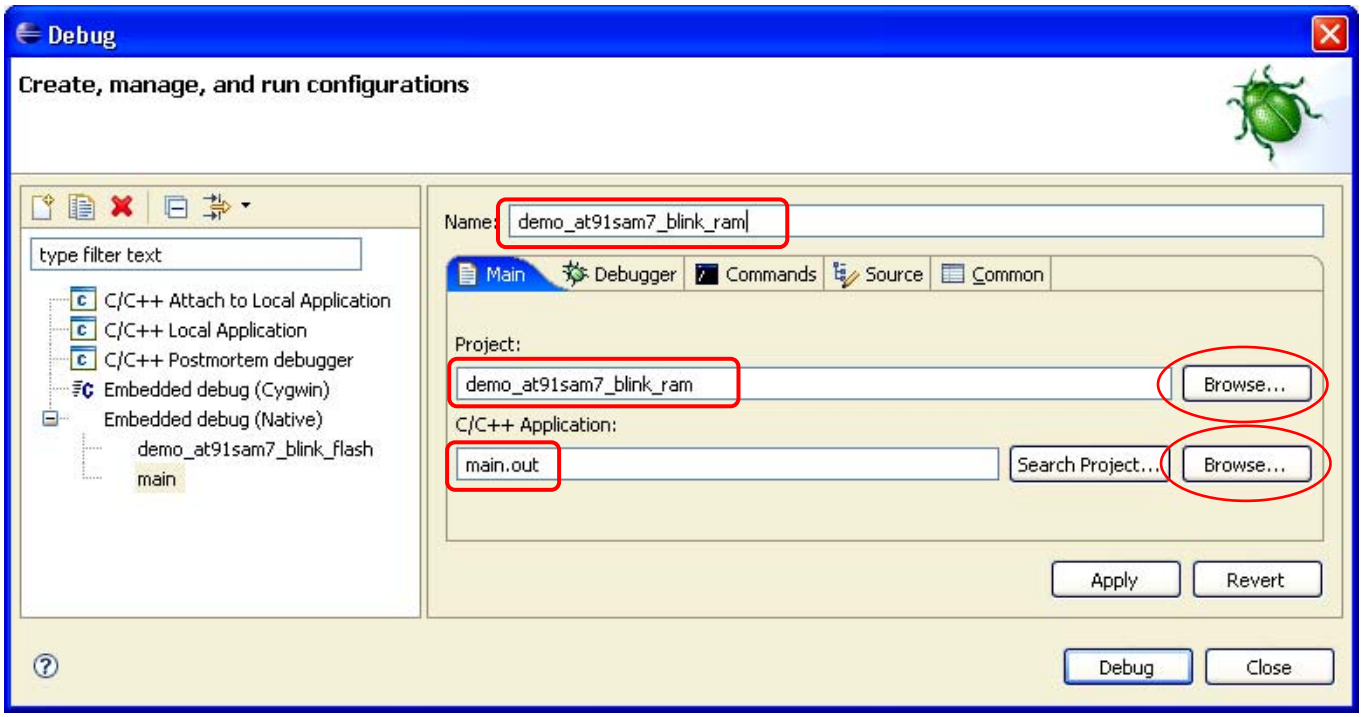
Click on “**Run**” followed by “**Debug…**”.



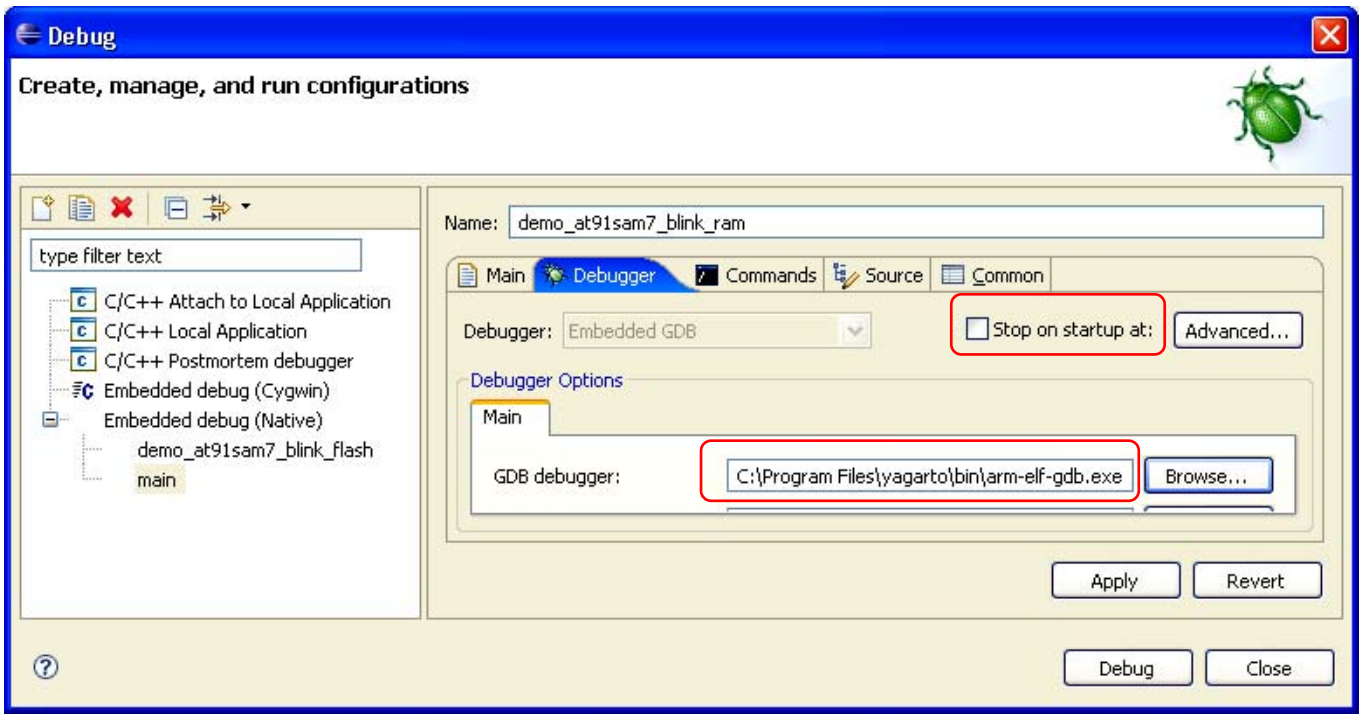
When the Debug “Create, manage, and run configurations” screen appears, click on “**Embedded debug (Native)**” followed by the “**New**” button.



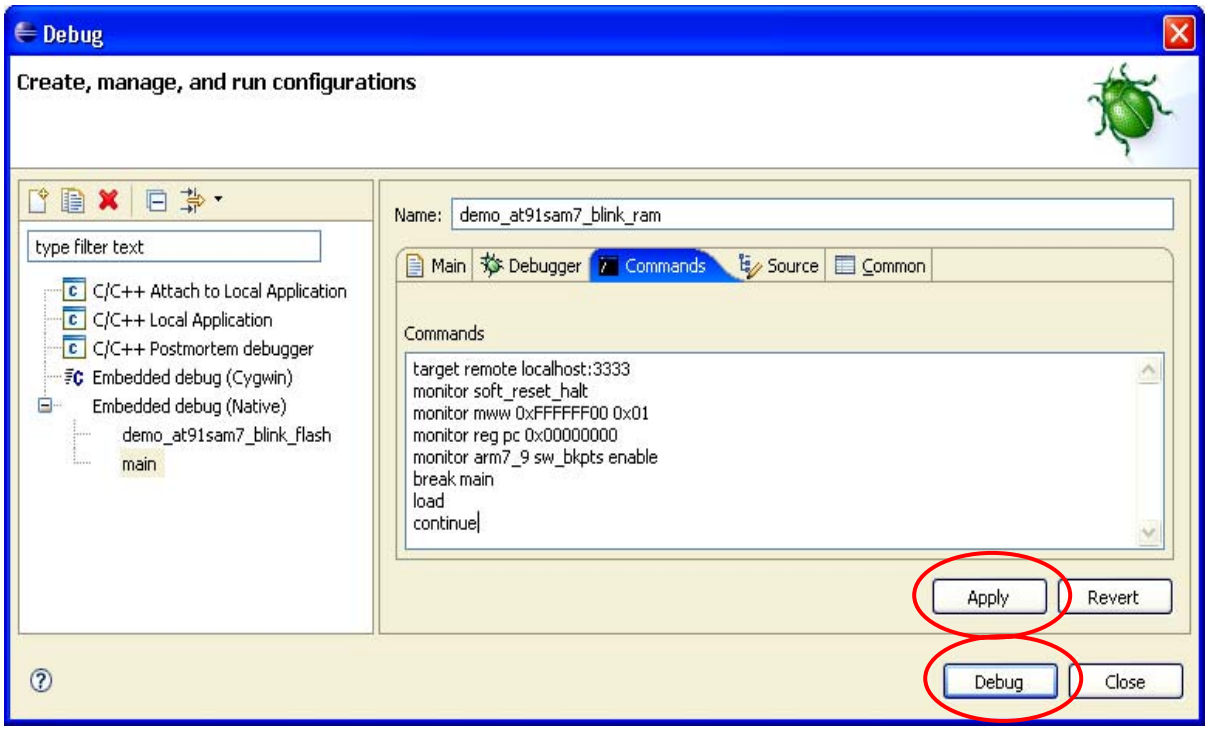
A new and empty “Embedded debug launch” configuration screen will appear. Under the “**Main**” tab, fill out the new configuration screen as shown below. Once again, I selected the project name “demo_at91sam7_blink_ram” as the debug launch configuration name. Use the “Browse” buttons to find the project and the C/C++ Application file as shown below.



Under the "**Debugger**" tab, fill out the screen as shown below. Note that we un-checked the "**Stop on startup at:**" check box since we will use a "break main" GDB command to do the same thing. Also use the "**Browse**" button to find the GDB debugger (it is the file: **c:\Program Files\yagarto\bin\arm-elf-gdb.exe**).



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 0x01

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 0x00000000

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 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().

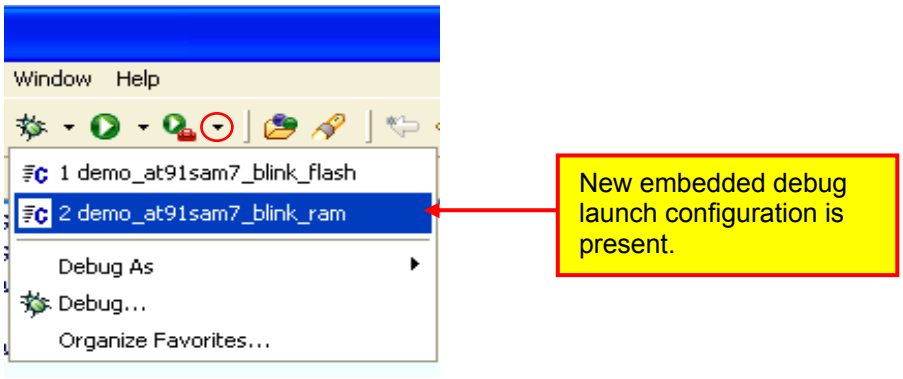
load main

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

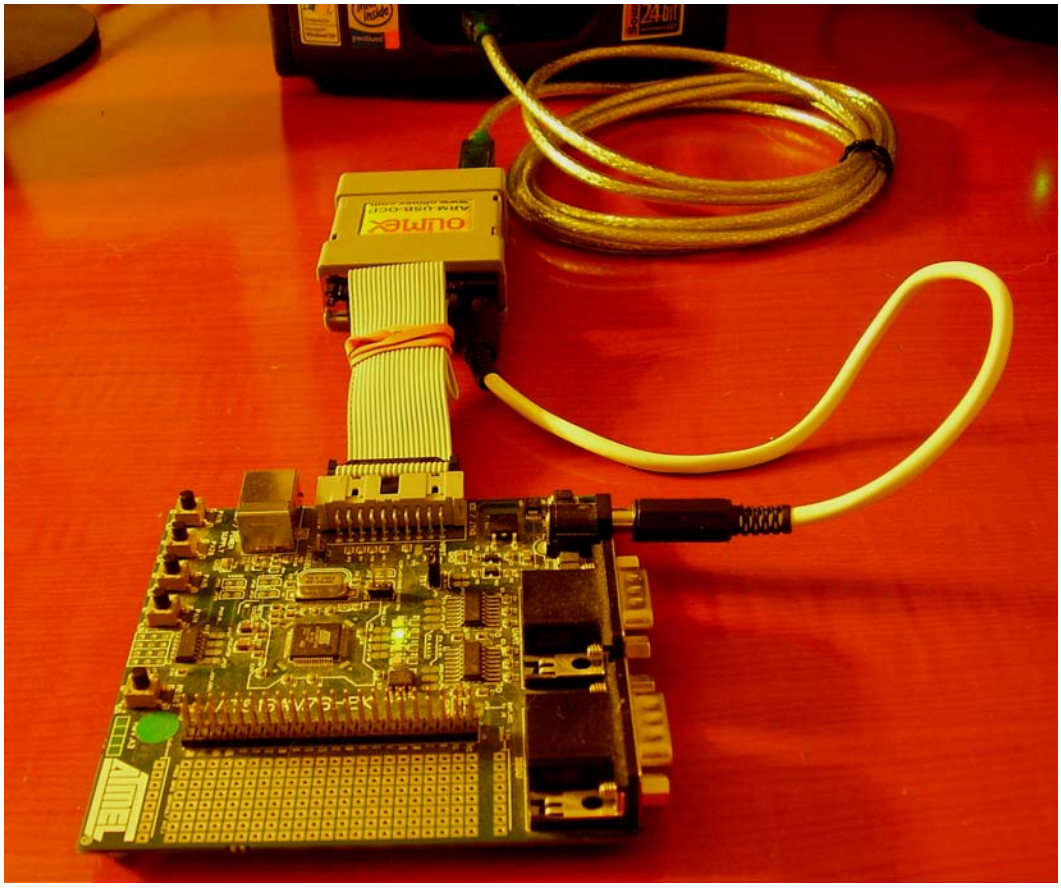
This is a GDB command. It forces the ARM processor out of halt state and resumes execution from the restart vector. Execution will halt on the breakpoint set at main( ).

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.



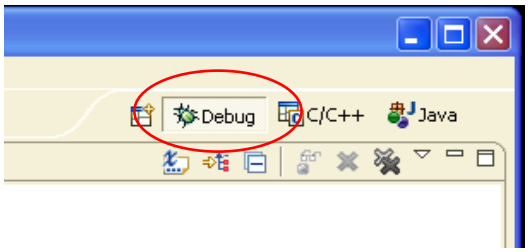
Set up the hardware

Whatever debugger you are using (wiggler, JTAGKey or ARM-USB-OCD), the same hardware setup used for FLASH programming and debugging will also apply to RAM-based applications. Shown below is the hardware setup for the Olimex ARM-USB-OCD JTAG debugger.

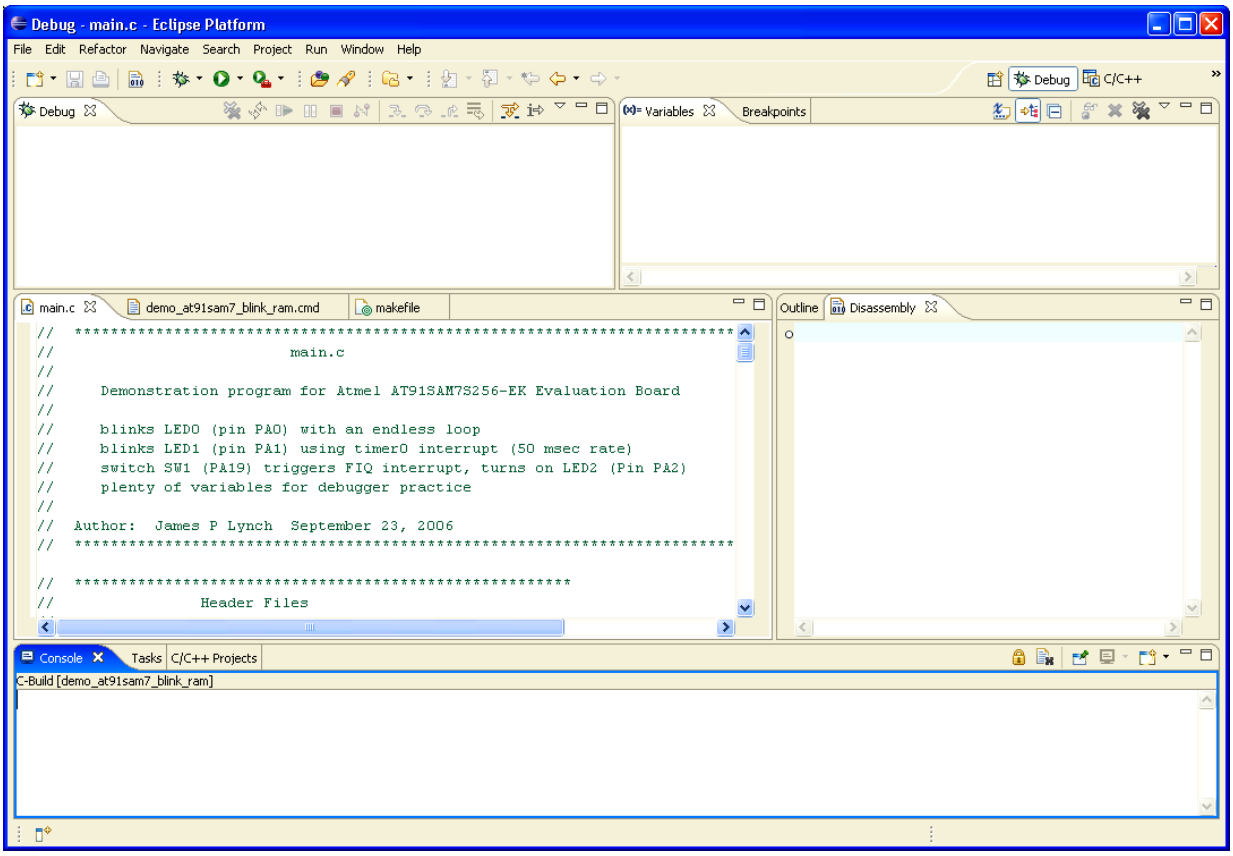


Open the Eclipse “Debug” Perspective

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



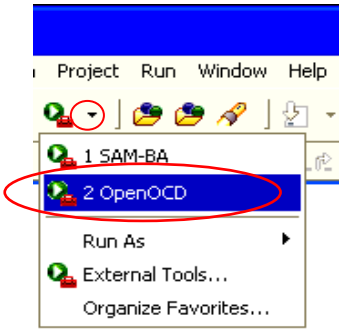
Now the **Debug** perspective will appear, as shown below.



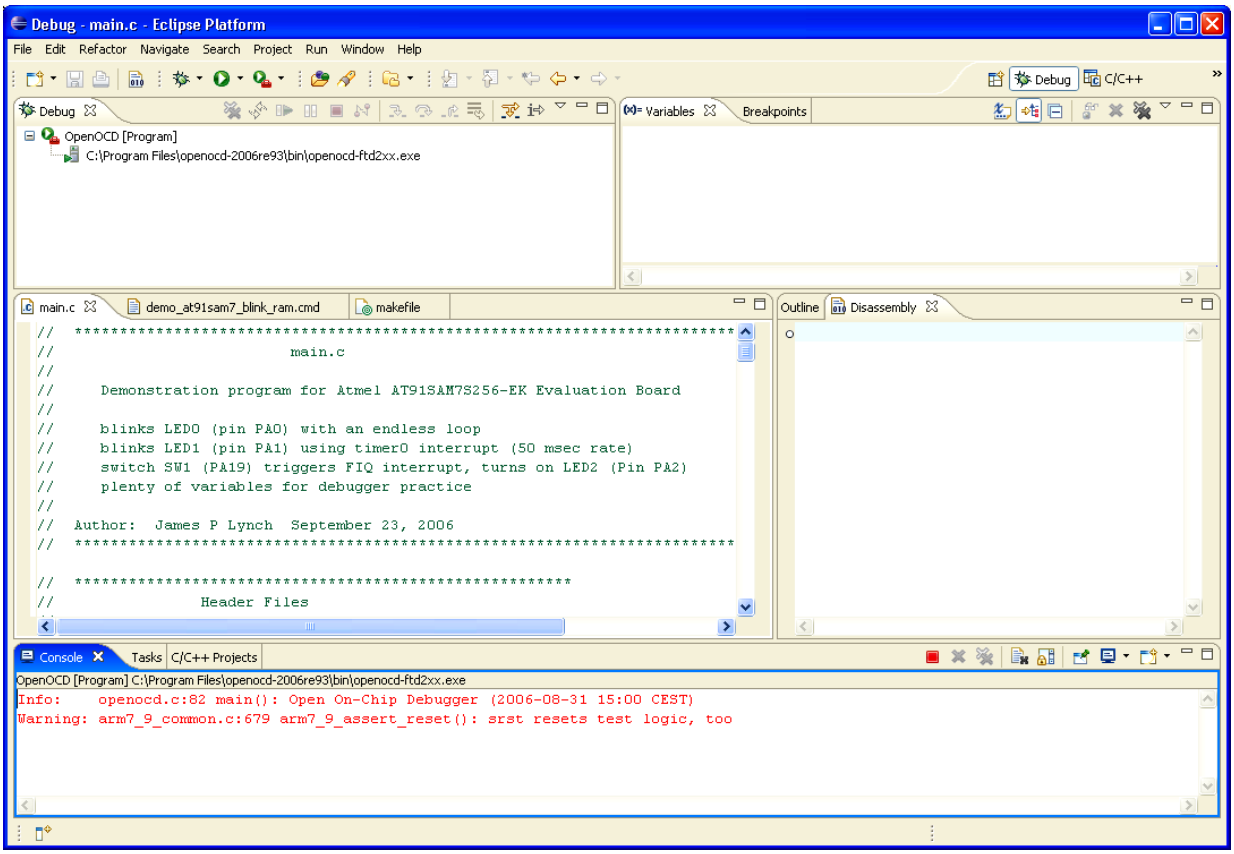
Start OpenOCD

Before starting **OpenOCD**, get into the habit of cycling the power of the Atmel **AT91SAM7-EK** board. Pulling and then re-inserting the power cable will accomplish this. 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**".



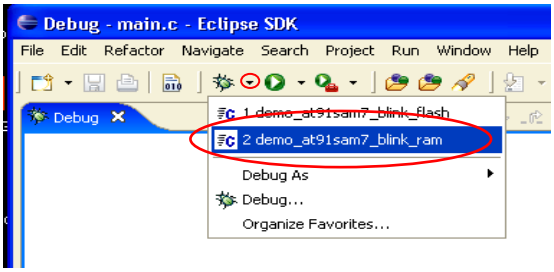
The debug view will show that **OpenOCD** is running and the console view shows no errors (warnings are OK).



Start the Eclipse Debugger

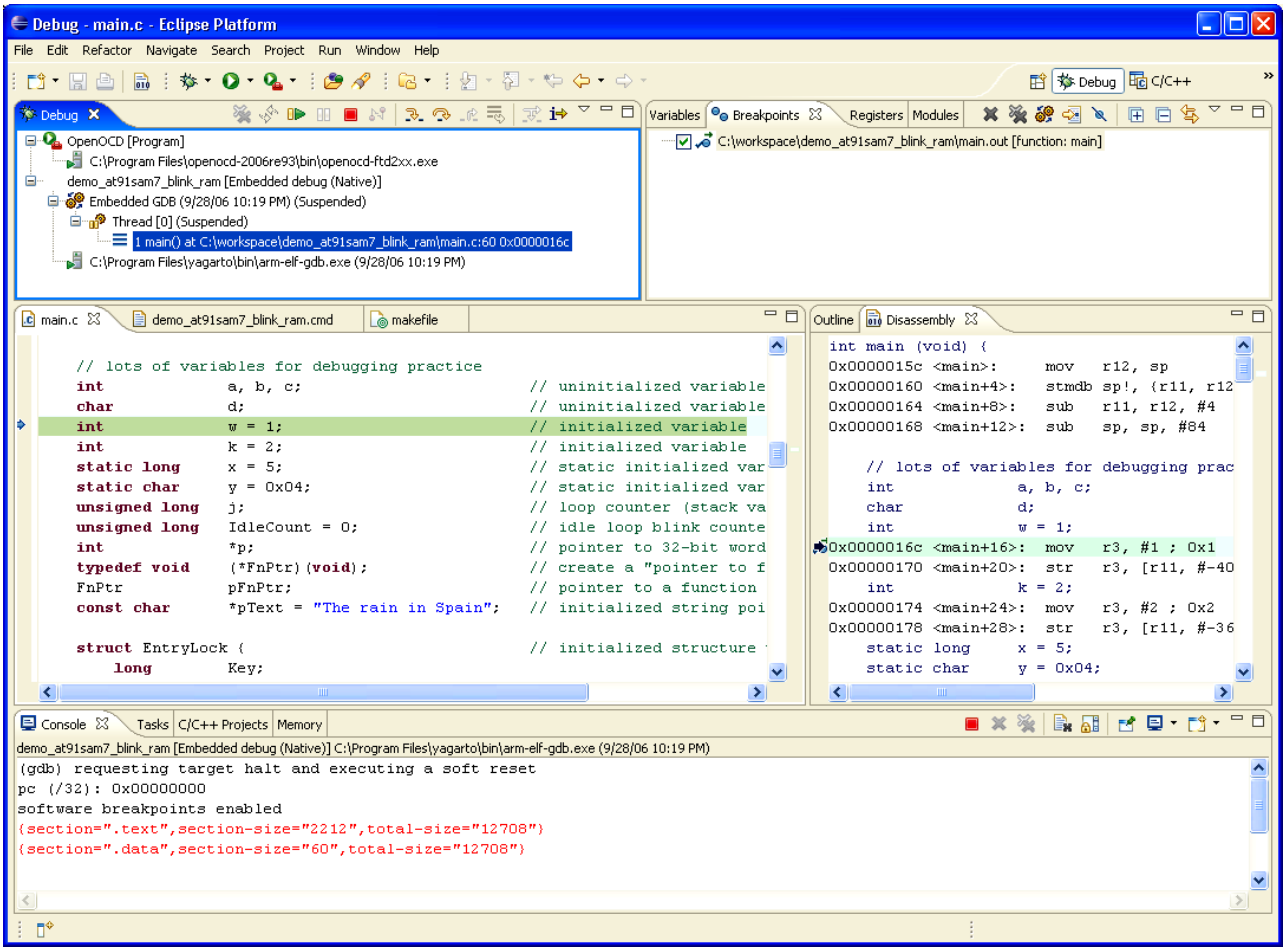
To start the Eclipse debugger, click on the "**Debug**" toolbar button's down arrowhead and select the debug launch configuration "**demo_at91sam7_blink_ram**" as shown below.

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.



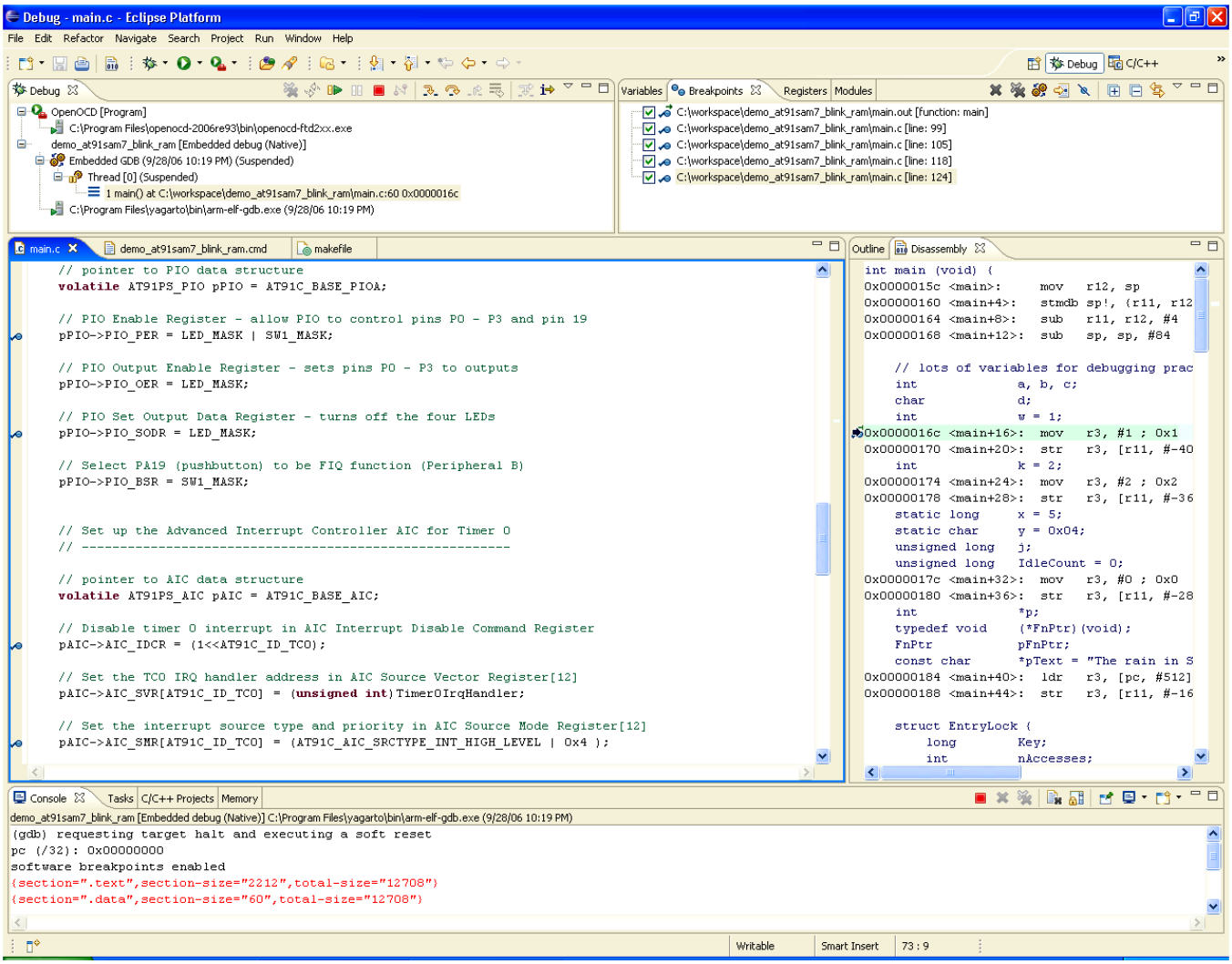
If the Eclipse debugger starts properly, the debug view (upper left) shows that the debugger has stopped at line 60 in main().

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.

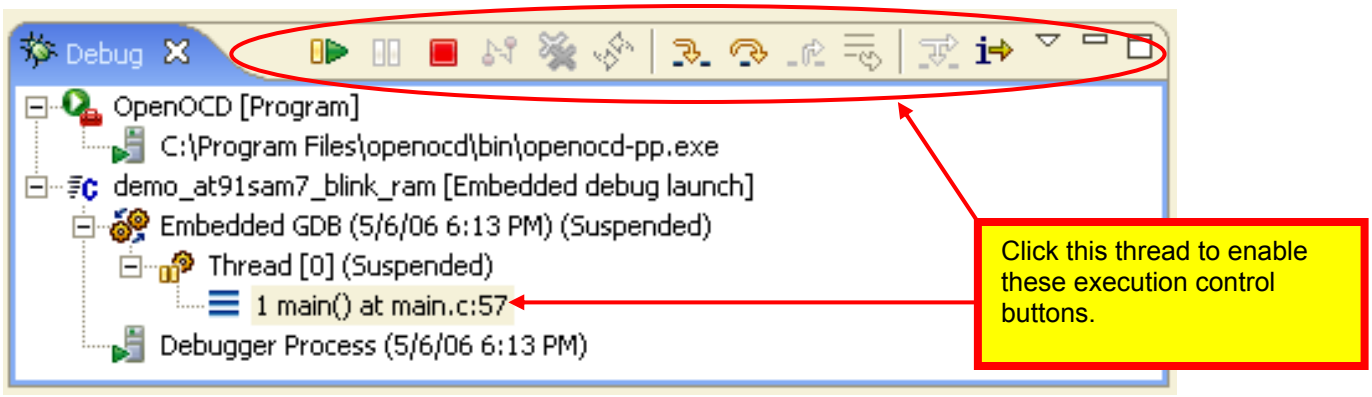


Setting Software Breakpoints

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.

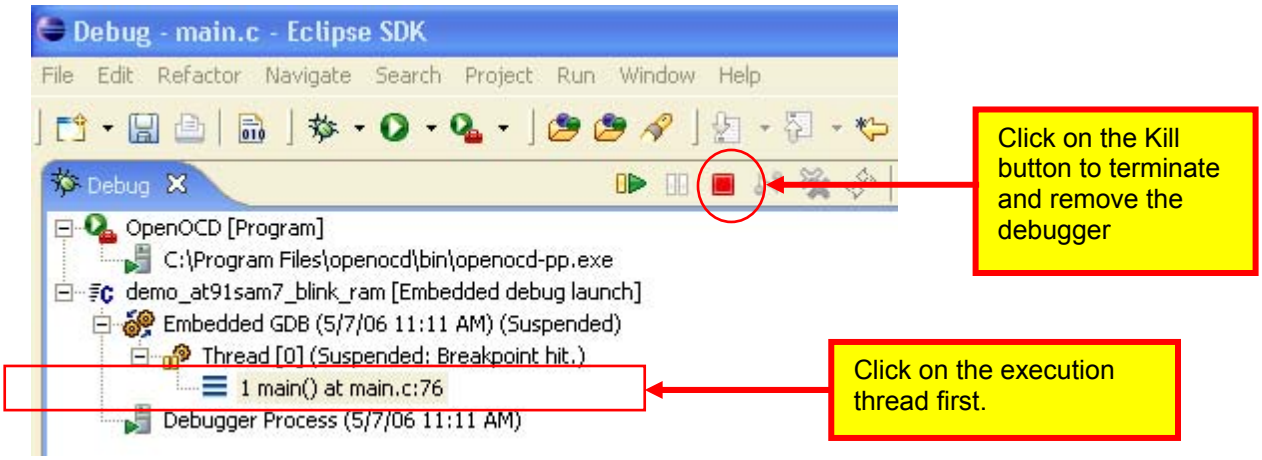
Now you can run through all the debugger operations covered earlier in this tutorial. Considering that modern desktop PCs and laptops are being manufactured without serial or parallel ports, a USB-based JTAG interface will soon be the only way to debug target boards.

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.

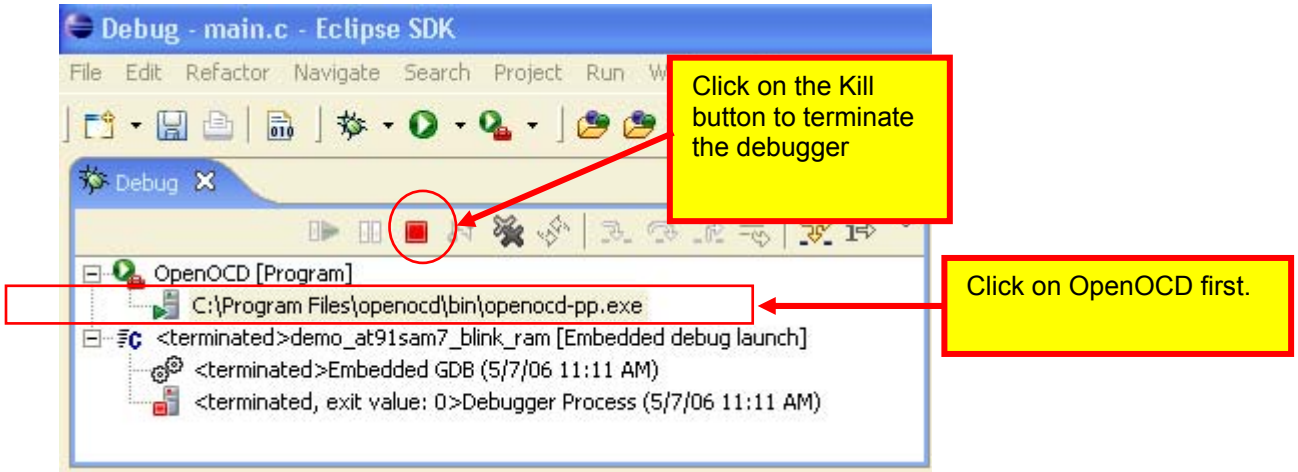
- **Stop the Debugger**

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



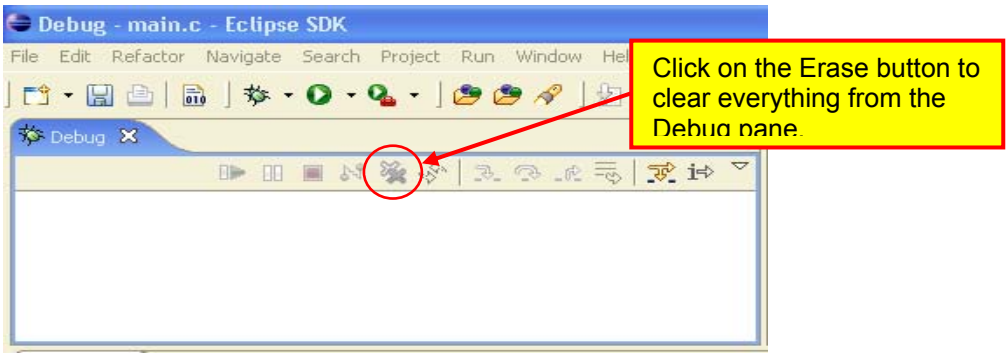
- **Stop OpenOCD**

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



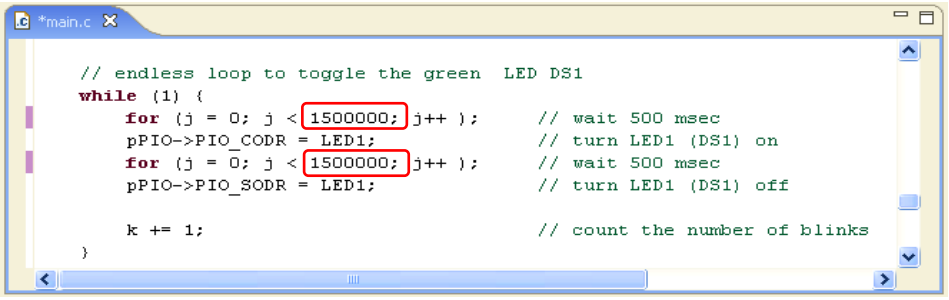
- **Erase the Debug Pane**

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



- **Modify the Source File**

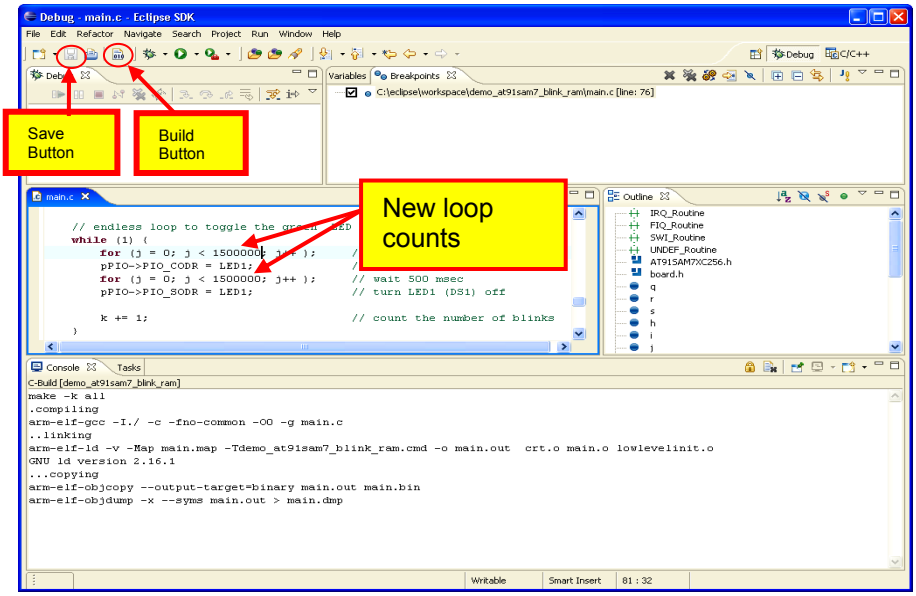
  Here we have changed the wait time by modifying the loop counts.



- **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.

- 

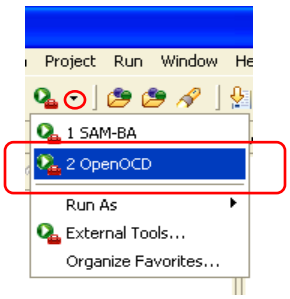
- **Cycle the Target Board Power**

  This can be done by detaching and re-inserting the power cable.

  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.

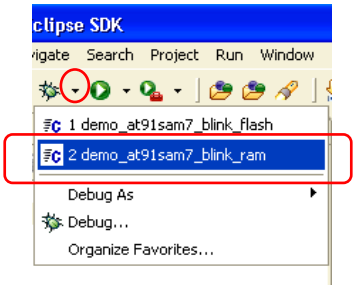
- **Start OpenOCD**

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



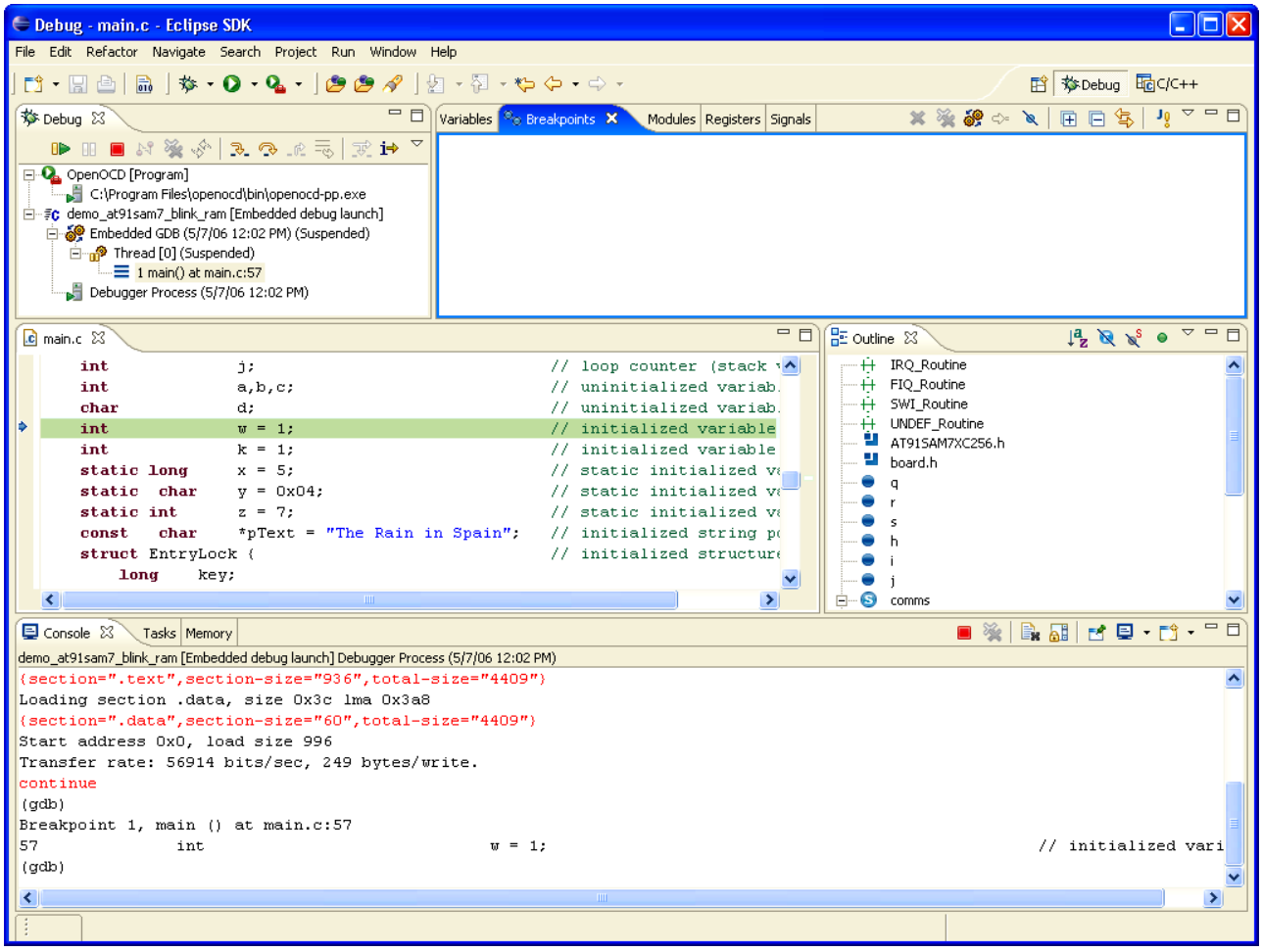
- **Start the Eclipse Debugger**

  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 run the OpenOCD flash programmer and program the flash before starting the OpenOCD and the Eclipse debugger. If you are using the JTAGKey or the ARM-USB-OCD debuggers that use the USB port, that operation is still quite speedy. Nonetheless, it's an extra step which explains the popularity of building and debugging entirely within RAM memory.

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. Thanks to the tireless contributions of open-source heroes such as Michael Fischer and Dominic Rath, the acquisition and installation of Open Source tools is becoming less complex and time-consuming. The reader needs a high speed internet connection to download the various components and a couple hours of 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

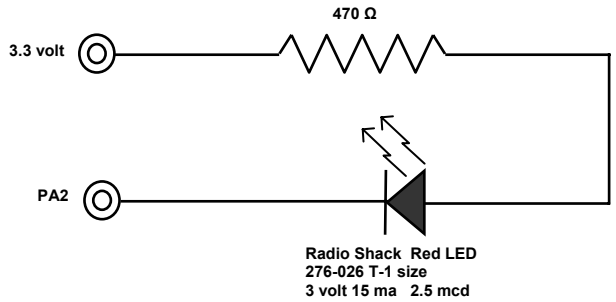
Jim Lynch lives in Grand Island, New York and is a software developer for Control Techniques, a subsidiary of Emerson Electric. He develops embedded software for the company’s industrial drives (high power motor controllers) which are sold all over the world.

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

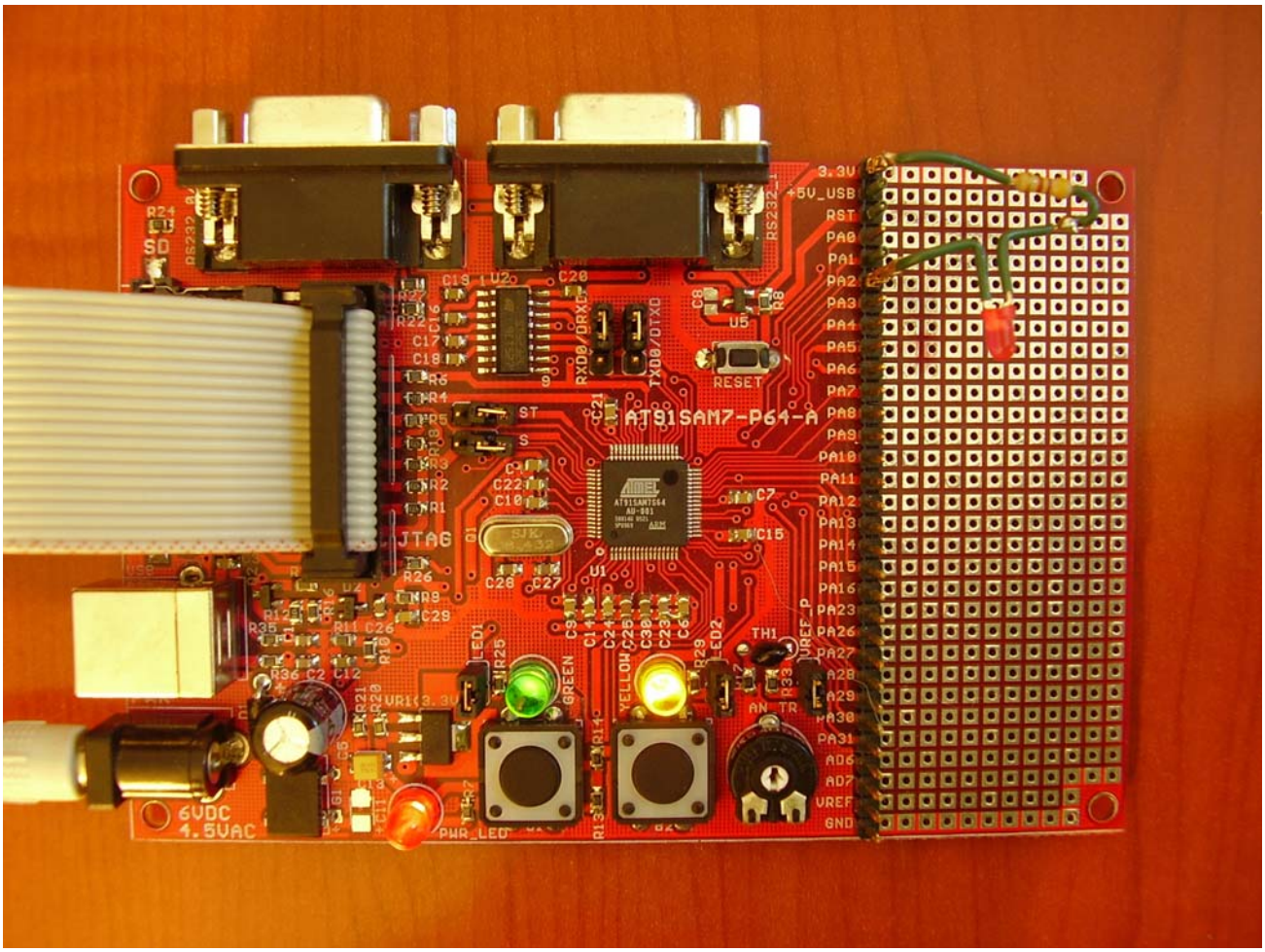
Appendix 1. Olimex AT91SAM7- P64 Board

The Olimex AT91SAM7-P64 board has two LED's and two pushbutton switches while the Atmel AT91SAM7S-EK board has four LEDs and four pushbutton switches. The application described in this tutorial uses one switch and three LEDs. Fortunately, the pushbutton switches use the same PIO ports as the Atmel AT91SAM7S-EK board, so no change is required for the single switch. The Olimex board uses different PIO ports for the LEDs, so we are required to do two things; add a LED to the board and adjust the board.h file to specify the correct ports.

Since LED3 was port PA2 in the Atmel evaluation board, the author added the following simple circuit to the Olimex board.



Below is a photograph showing the added LED3.



The board.h include file was modified to specify the correct ports for the LEDs and the switches.

```
/*----------------------------------------------------------------------------
*         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              : Olimex AT91SAM7-P64 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

//-----------------
// Leds Definition
//-----------------
//                                    PIO              PA    PB   PIN
#define LED1            (1<<18)      // PA18            RD    PCK2  10
#define LED2            (1<<17)      // PA17            TD    PCK1   9
#define LED3            (1<<2)       // PA2             PWM2  SCK0  44
#define NB_LEB           3

#define LED_MASK        (LED1|LED2|LED3)

//-------------------------
// Push Buttons Definition
//-------------------------
//                                    PIO              PA   PB     PIN
#define SW1_MASK        (1<<19)      // PA19            RK   FIQ     13
#define SW2_MASK        (1<<20)      // PA20            RF   IRQ0    16
#define SW_MASK         (SW1_MASK|SW2_MASK)


#define SW1     (1<<19)             // PA19
#define SW2     (1<<20)             // PA20

//------------------
// 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
```

The Olimex AT91SAM7-P64 board used the AT91SAM7S64 chip, which has 64K of FLASH and 16K of RAM. The linker command script, **demo_at91sam7_p64_blink_flash.cmd**, is modified to support these memory limits.

```
/* ********************************************************************************************************
     demo_at91sam7_p64_blink_flash.cmd           LINKER  SCRIPT    (for Olimex AT91SAM7-P64-A board)


     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_p64_blink_flash.cmd
     directive to the linker flags in the makefile. For example,

                       LFLAGS  =  -Map main.map -nostartfiles -T demo_at91sam7_p64_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 0x00000000


     The top of the stack (_stack_end) is (last_byte_of_ram +1) - 4

     Therefore:   _stack_end = (0x000203FFF + 1) - 4  =  0x00020400 - 4  =  0x00203FFC

     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
                            |                                 |
                 .--------->|---------------------------------|0x00204000
                 .          |                                 |0x00203FFC  <---------- _stack_end
                 .          |    UDF Stack  16 bytes          |
                 .          |                                 |
                 .          |---------------------------------|0x00203FEC
                 .          |                                 |
                 .          |    ABT Stack  16 bytes          |
                 .          |                                 |
                 .          |---------------------------------|0x00203FDC
                 .          |                                 |
                 .          |                                 |
                 .          |    FIQ Stack  128 bytes         |
                 .          |                                 |
                 .          |                                 |
                RAM         |---------------------------------|0x00203F5C
                 .          |                                 |
                 .          |                                 |
                 .          |    IRQ Stack  128 bytes         |
                 .          |                                 |
                 .          |                                 |
                 .          |---------------------------------|0x00203EDC
                 .          |                                 |
                 .          |    SVC Stack  16 bytes          |
                 .          |                                 |
                 .          |---------------------------------|0x00203ECC
                 .          |                                 |
                 .          |     stack area for user program |
                 .          |                                 |
                 .          |                                 |
                 .          |                                 |
                 .          |          free ram               |
                 .          |                                 |
```

```
             .         |..............................|0x002006D8 <---------- _bss_end
             .         |                              |
             .         |  .bss   uninitialized variables |
             .         |..............................|0x002006D0 <---------- _bss_start, _edata
             .         |                              |
             .         |  .data  initialized variables   |
             .         |                              |
             .-------->|______________________________|0x00200000


             .-------->|------------------------------|0x00010000
             .         |                              |
             .         |                              |
             .         |         free flash           |
             .         |                              |
             .         |                              |
             .         |..............................|0x000006D0 <---------- _bss_start, _edata
             .         |                              |
             .         |  .data  initialized variables   |
             .         |                              |
             .         |------------------------------|0x000006C4 <---------- _etext
             .         |                              |
             .         |            C code            |
             .         |                              |
             .         |                              |
             .         |------------------------------|0x00000118  main()
             .         |                              |
             .         |    Startup Code  (crt.s)     |
             .         |         (assembler)          |
             .         |                              |
             .         |------------------------------|0x00000020
             .         |                              |
             .         | Interrupt Vector Table       |
             .         |          32 bytes            |
             .-------->|------------------------------|0x00000000 _vec_reset


    Author:  James P. Lynch

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


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

/* specify the AT91SAM7S64  */
MEMORY
{
    flash : ORIGIN = 0,          LENGTH = 64K    /* FLASH EPROM          */
    ram   : ORIGIN = 0x00200000, LENGTH = 16K    /* static RAM area      */
}


/* define a global symbol _stack_end  (see analysis in annotation above) */
_stack_end = 0x203FFC;

/* 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 initializer copy into FLASH)  */

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

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

The download package containing the sample programs also includes two sample projects for the Olimex board, one for FLASH execution and one for RAM execution. You will have to define a new Debug Launch Configuration for the Olimex projects; just employ the methods shown earlier in this tutorial.

If you have a 256K version of the Olimex board (AT91SAM7-P256), you will need to adjust the memory limits and top-of-stack specification in the linker command file shown above.

For the 64K board, these specifications are:

```
/* specify the AT91SAM7S64  */
MEMORY
{
   flash   :  ORIGIN = 0,              LENGTH = 64K /* FLASH EPROM          */
   ram     :  ORIGIN = 0x00200000,  LENGTH = 16K          /* static RAM area         */
}

/* define a global symbol _stack_end  (see analysis in annotation above) */
_stack_end = 0x203FFC;
```

For the 256K board, these specifications are:

```
/* specify the AT91SAM7S256  */
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 = 0x20FFFC;
```

Warning: The author discovered that the Olimex ARM-USB-OCD JTAG interface's built-in power supply tends to latch up during OpenOCD FLASH programming. Possibly the Olimex board draws more current than the Atmel board; it does have a pot installed for A/D input. If you are planning to use OpenOCD FLASH programming with the Olimex board, it behooves you to use a separate "wall-wart" power supply instead.

Appendix 2. SOFTWARE COMPONENTS

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

Fortunately, the **YAGARTO** package prepared by Michael Fischer assuages any fears of incompatibility. Michael has chosen all the components of **YAGARTO** to be compatible with each other and has tested them. Yagarto is downloaded as three zip file installers hosted at this web address:

http://www.yagarto.de/

Everything the reader needs is in these three zip files. The included components are:

- **Eclipse IDE version 3.2**

  Eclipse CDT 3.1 Plug-in for C++/C Development (Zylin custom version)

- **Native GNU C++/C Compiler suite for ARM Targets**

- **OpenOCD version 93 for JTAG debugging**

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 CDT build may require that you utilize the latest Eclipse “stream build”.

A safe approach is to build the ARM software development system using the **YAGARTO** package above, get it to work and become familiar with it. Then you can monitor the Eclipse, Zylin, Yagarto and OpenOCD web sites for new versions and choose at a later time if you want to upgrade. So far, Michael has been very diligent in having the “latest and greatest” as part of YAGARTO.

A short zip file containing the tutorial and the sample Eclipse projects are hosted by Atmel at the following web address:

www.address_to_be_determined.com