# Debugging ARM Embedded Applications using GNU Tools

Application Note

November, 2009

## Table of Contents

# 1.0 Introduction

## 1.1 Overview

The purpose of this document is to show developers how to debug ARM Embedded Applications using GNU Tools. In this document we will be discussing two different debugging procedures: GDB from the command line and from the Insight GUI. Both can be used to effectively debug your applications and use the same GDB commands. The main difference with Insight is that it provides a graphical interface for the debugging commands. The demonstration will show how to use both debugging procedures from the command line or from the Codeblocks IDE.

The major topics to be covered in this document are:

- OpenOCD setup
- Essential GDB commands and how to invoke it from Codeblocks
- Essential Insight commands and how to invoke it from Codeblocks

# 2.0 Debugging using GDB Commands

## 2.1 OpenOCD – Open On-Chip Debugger

The Open On-Chip Debugger (OpenOCD) provides debugging, in-system programming and boundary-scan testing for embedded target devices. It allows debuggers to execute JTAG functions on embedded targets either from a local system or across a network. The Micromint Eagle BSP installer in the Tools CD can be used to install a version of OpenOCD for Windows configured to use the LMI FTDI interface via USB. OpenOCD for Linux can also be downloaded from the web. Below are the most important files installed:

- openocd-ftd2xx.exe - OpenOCD application for Windows32
- eagle-50.cfg - OpenOCD configuration file for the Micromint Eagle using LMI FTDI
- program.script - Script file to transfer a binary image to the LMI controller
- FTD2xx.dll – Library to interface with FTDI-based devices (part of the Micromint Eagle USB driver)
- openocd.bat – batch file to start OpenOCD

By default OpenOCD reads the file configuration file “openocd.cfg” in the current directory. To specify a different configuration file, you need to use the “-f” option. The batch file included (openocd.bat) starts OpenOCD with the eagle-50.cfg configuration. Figure 2-1 shows how to invoke OpenOCD directly from the command line using the “-f” option to specify the configuration file.

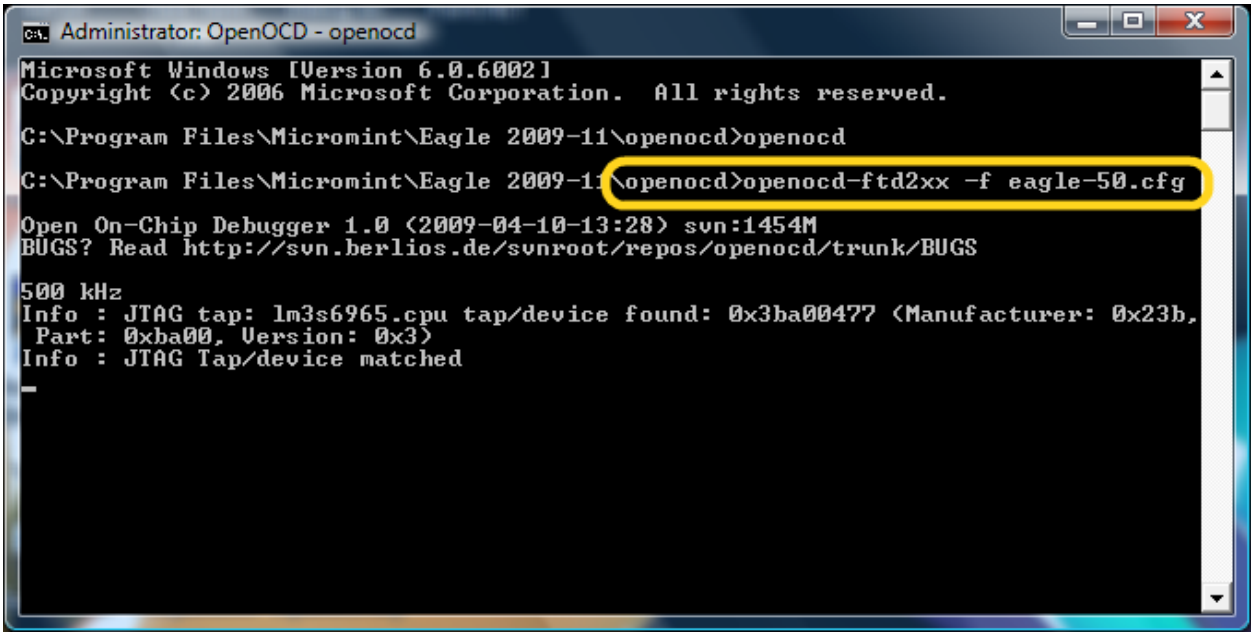


**Figure 2-1:** Example of OpenOCD starting session

Once started, OpenOCD connects to the JTAG device via the USB LMI FTDI interface and runs as a daemon, waiting for connections from clients (telnet, GDB or others). There are many ways you can configure OpenOCD and start it up. A simple way to organize them involves keeping a separate directory for each board with its processor and JTAG configurations. When you start OpenOCD it searches the current directory first for configuration files, scripts, and for code you upload to the target board. It is also the natural place to write log files and data to be download from the board upon startup. By pressing *ctrl –c*, the OpenOCD daemon is stopped, ending communication with the embedded device. You will need to stop OpenOCD when using the LMI Flash Programmer, to update firmware on the board.

OpenOCD uses a hardware interface to communicate with JTAG (IEEE 1149.1) compliant TAPs on your target board. A *TAP* is a "Test Access Port", a module which processes special instructions and data. In the Micromint Eagle 50 the USB debug port implements a TAP that is compatible with the FTDI interface in OpenOCD. TAPs can be daisy-chained within and between chips and boards.

OpenOCD currently supports many types of JTAG interfaces: USB based, parallel port based and even standalone boxes that run OpenOCD internally. Examples include USB FTDI FT2232**,** USB J-Link, USB RLINK, PC Parallel Printer Port and others. More information about OpenOCD is available from these references:

**Open On-Chip Debugger**
http://openocd.berlios.de/web/

**OpenOCD User's Guide**
http://openocd.berlios.de/doc/pdf/openocd.pdf

## 2.2 GDB – GNU Project Debugger

The purpose of a debugger is to allow you to monitor a program while it executes to identify and correct any undesired behavior. GDB (GNU Project Debugger) allows developers to debug applications by providing four important capabilities:

- Load the program in memory and start it with any parameters that may affect its behavior.
- Place breakpoints to stop the program on specified conditions.

- Watch variables and processor registers when the program stops to examine what has happened
- Change instructions and data in the program to test alternatives to correct program bugs before changing the source code and rebuilding the executable.

GDB is included with popular GNU ARM toolchains like devkitARM, Sourcery G++ Lite, WinARM and YAGARTO.

## 2.3 Introduction to GDB Commands

There are two important considerations when compiling an application to be debugged. To be able to view the source code when debugging , the executable must be compiled with debugging data (-g). To separate application bugs from compiler bugs it is a good practice to first compile without any compiler optimization (-O0 and no -Os). Once the application is working as expected, you can debug the final executable with the desired optimization level.

There are some essential commands needed to perform basic debugging tasks using GDB. The first thing you need to do is start GDB from the command line, usually from the directory containing the application binary to be debugged. The command used to invoke the GDB included with devkitARM is '`arm-eabi-gdb` *`project_name.out`*'. Figure 2-2 shows a sample debugging session using *`blinky.out`*.

```
C:\Projects\ARM\blinky\gcc> arm-eabi-gdb blinky.out
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-mingw32 --target=arm-
eabi"...
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x00000000 in ?? ()
(gdb) load
Loading section .text, size 0x288 lma 0x2000
Start address 0x2000, load size 648
Transfer rate: 2 KB/sec, 648 bytes/write.
(gdb) b blinky.c:78
Breakpoint 1 at 0x2198: file blinky.c, line 78.
(gdb) b blinky.c:90
Breakpoint 2 at 0x21ce: file blinky.c, line 90.
(gdb) j ResetISR
Continuing at 0x20f4.
Note: automatically using hardware breakpoints for read-only
addresses.

Breakpoint 1, main () at blinky.c:78
78              GPIO_PORTE_DATA_R |= 0x02;
(gdb)
```

**Figure 2-2:** Example of GDB debugging session

GDB can connect to a remote JTAG such as OpenOCD using the command '`target remote` *`host`*`:`*`port`*'. The *`host`* may be either a hostname or a numeric IP address; *`port`* must be a decimal number. In Figure 2-2, `localhost:3333` connects to the OpenOCD daemon running in port 3333 of the local machine. If the remote target is running on the same machine as your debugger session, you can omit the hostname.

The GDB '`load`' command as can be used to load the application binary to the device. GDB will use the memory addresses specified when the application was linked. OpenOCD allows GDB access to the JTAG functions required to flash the firmware to the current TAP device.

To jump to a specific location, you just need to invoke the GDB 'jump *location*' command where *location* specifies the address, label or source code location to start or continue execution. Commonly used GDB commands allow a shortcut. In the case of 'jump' you can also use 'j'. In the example on Figure 2-2 'j *ResetISR*' jumps to the location of ResetISR and continues execution until a breakpoint is found.

```
(gdb) b blinky.c:78
Breakpoint 1 at 0x2198: file blinky.c, line 78.
(gdb) b blinky.c:90
Breakpoint 2 at 0x21ce: file blinky.c, line 90.
(gdb) watch ulLoop
Hardware watchpoint 3: ulLoop
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00002198 in main at blinky.c:78
        breakpoint already hit 1 time
2       breakpoint     keep y   0x000021ce in main at blinky.c:90
3       hw watchpoint  keep y              ulLoop
(gdb) b blinky.c:83
Breakpoint 4 at 0x21b0: file blinky.c, line 83.
(gdb) cond 4 ulLoop=1000
(gdb) c
Continuing.
Breakpoint 4, main () at blinky.c:83
83              for(ulLoop = 0; ulLoop < 200000; ulLoop++)
(gdb) p ulLoop
$2 = 1000
(gdb)
```

**Figure 2-3:** Placing breakpoints, watching variables and conditional breakpoints

A breakpoint makes your program stop when a specified location in the program is reached. You can add conditions to the breakpoints to only stop when a specified behavior is found, allowing finer control when debugging. You can set breakpoints with the GDB 'break *location*' command or its shortcut 'b *location*'. In Figure 2-3 the ' b blinky.c:78' command sets a breakpoint on line 78 of blinky.c You can list the current breakpoints by using the 'info break' command.

A watchpoint is an expression defined with 'watch' command that can be monitored during execution of the application. The expression may be a value of a variable, or it could involve values of one or more variables combined by operators, such as 'a + b'. You can enable, disable, and delete both breakpoints and watchpoints using the same commands. Likes breakpoints, you can see all your declared watchpoints using the info watch command. Figure 2-3 also shows watchpoints declaration.

An expression can also be used as a condition for a breakpoint. When a condition is specified on a breakpoint it is evaluated each time the program reaches it, stopping execution only if the condition is true. When setting a conditional breakpoint you need to know breakpoint's number and declare the expression going to be evaluated to a specify value. This command is useful when you are working with instructions that are executed many times but need to be debugged only when specific values are reached. In Figure 2-3, ' cond 4 ulLoop=1000' sets breakpoint 4 to stop only when variable uLoop has a value of 1000 . That breakpoint was previously declared at line 83 of blinky.c.

Table 2-1 shows essential GDB commands. More information about GDB is available online from this reference:

**GDB Documentation**
http://www.gnu.org/software/gdb/documentation/

| Command | Description | Example |
|---|---|---|
| b *function* | set breakpoint at *function* | `b blinky.c:78` |
| bt | displays program stack | `bt` |
| c | continue running your program | `c` |
| cond *n [expr]* | new conditional expression on breakpoint *n*; make unconditional if no *expr* | `cond 4 ulLoop=1000` |
| info break or info watch | show defined breakpoints or watchpoints | `info break` |
| j *line* or j **address* | resume execution at specified *line* or *address* | `j ResetISR` |
| load | load program to your target | `load` |
| n | next line, stepping over function calls | `n` |
| p *expr* | display the value of an expression | `p ulLoop` |
| run | start your program with current argument list | `r` |
| s | next line, stepping into function calls | `s` |
| target remote *host:port* | debugging using a TCP connection to *port* on *host* | `target remote localhost:3333` |

**Table 2-1:** Common GDB Commands

## 2.4 Using GDB from the CodeBlocks IDE

GDB can be invoked from the CodeBlocks IDE using the proper settings. The GDB commands are the same as with the command line example in the previous section, but the IDE provides toolbar buttons to reduce typing and simplify its use. Figure 2-4 shows the location of the debugger command line in Codeblocks.

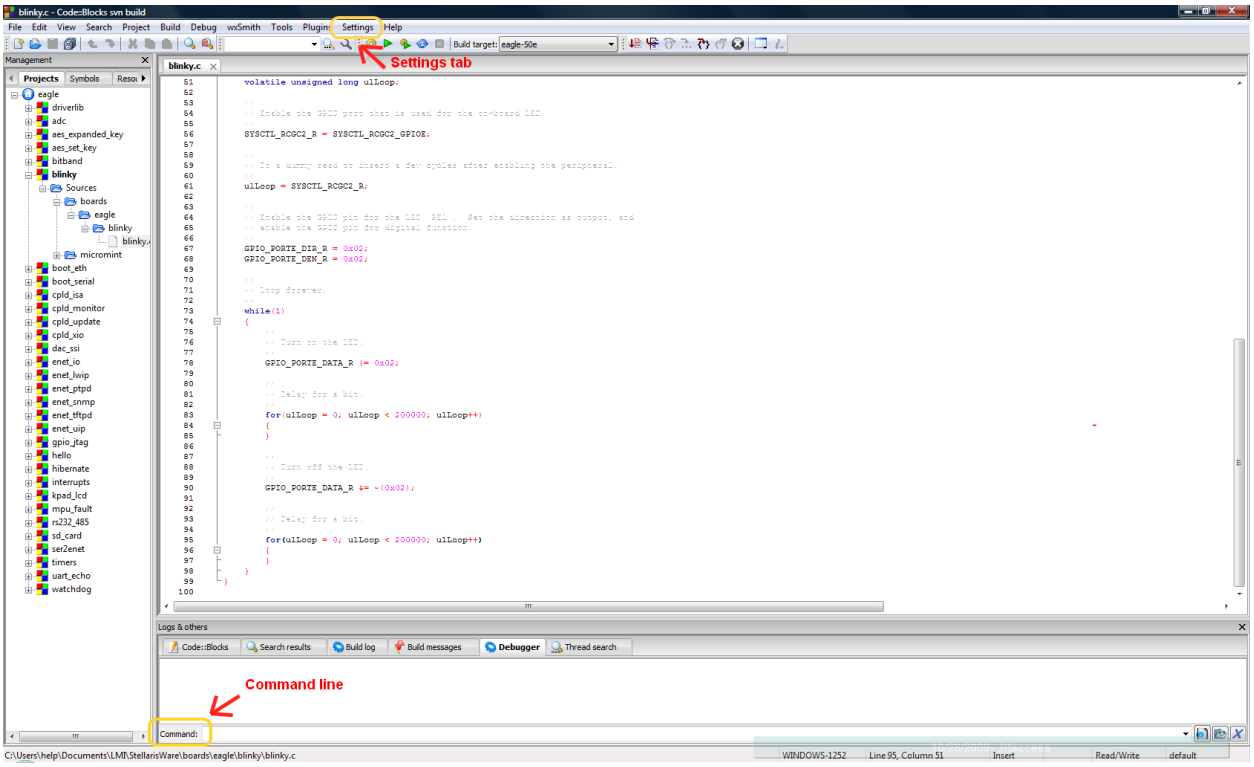


**Figure 2-4:** Debugger command line in Codeblocks

The following buttons can be accessed with the debugging toolbar in the IDE:

 Start/Continue – Start or continue the debugging process

 Next line – Step over function calls

 Next instruction – Step to the next instruction of a function

 Step into – Step into function calls

 Step out – Step out of function calls

 Run to cursor – Run program to cursor current position

 Stop debugger – Stop the debugging process

Breakpoints can be set by clicking the mouse next to the line number of the line in the source code where you want to stop the program. You can also set a conditional breakpoint by right clicking on the breakpoint location and specifying the desired condition. Two conditional options are available: ignoring the breakpoint for a specified number of counts before break or break when a specified expression is true. Both can be very useful to make the debugging process more efficient. Figure 2-5 shows where breakpoints are set and the window used to edit them.


**Figure 2-5:** Setting breakpoints

The debugger invoked by the Codeblocks IDE can be specified by selecting ‘Settings’, ‘Compiler and debugger’ on the menu, as shown on Figure 2-6.

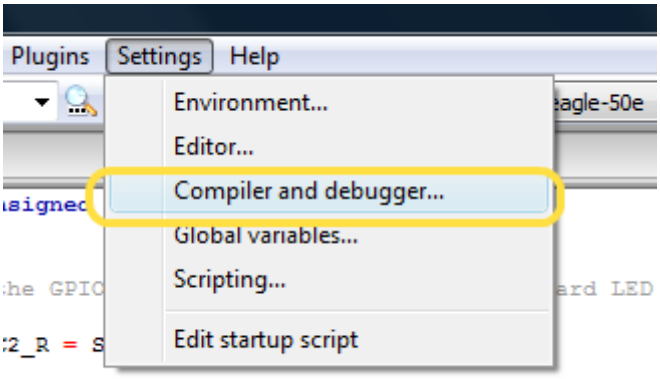


**Figure 2-6:** Compiler and debugger selection

On the 'Compiler and debugger' settings window, look for Global compiler settings and click on Toolchain executables. When using the devkitARM toolchain your Debugger should be set to arm-eabi-gdb.exe. This is the default value but it should be verified if your debugging sessions are not working as expected. Figure 2-7 shows an example of the proper configuration of toolchain executables.

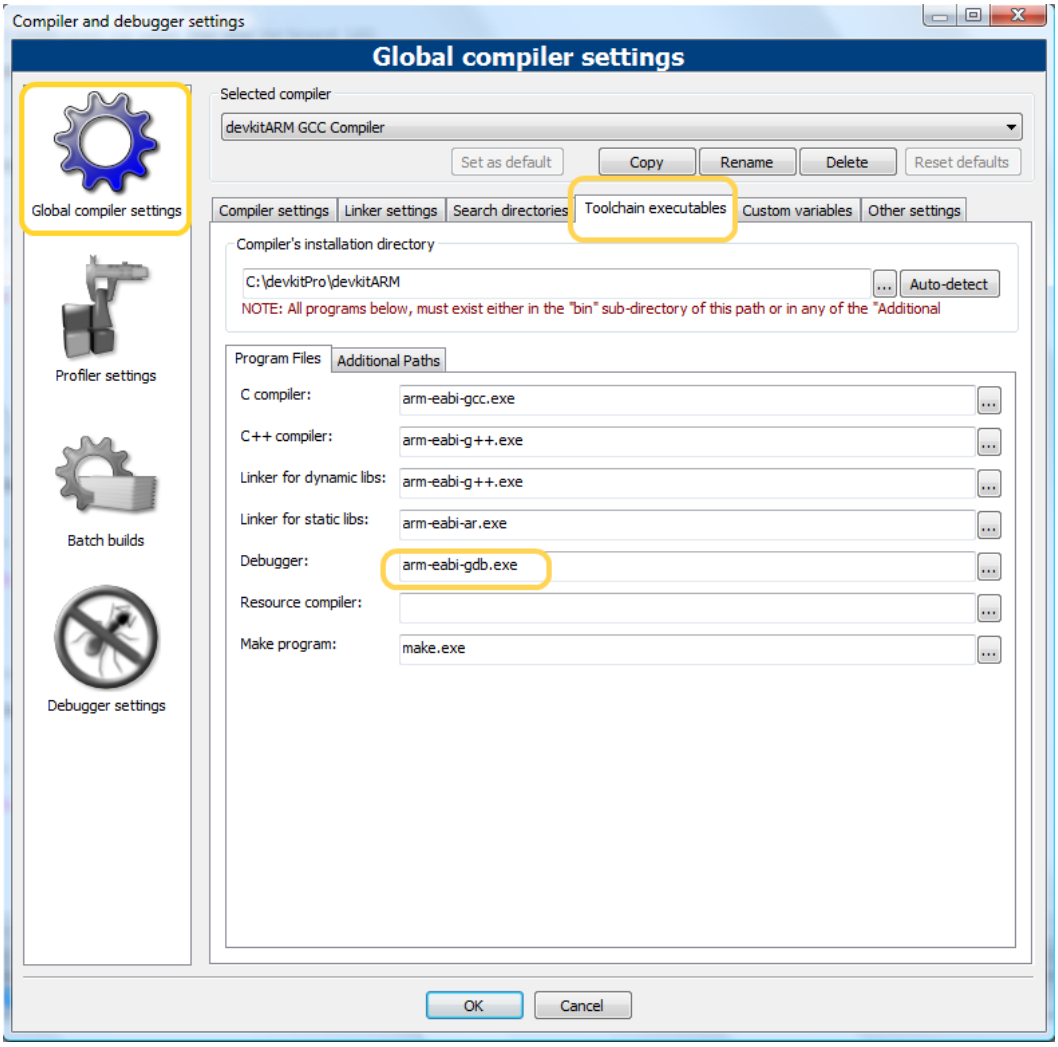


**Figure 2-7:** Setting the Debugger to GDB on Codeblocks

# 3 Debugging using Insight GUI

## 3.1 Insight – The GDB GUI

Insight is a graphical user interface to GDB written in Tcl/Tk. The main advantage of using the Insight GUI is that multiple debugging windows are displayed simultaneously. You can have a console window to type commands, a source window with the source code being executed, a watch window with the current contents of variables (including structures, linked lists, etc.), a stack window with the program's call stack, and so on. All your display windows are updated automatically as you step through your program.

Insight can be used with popular GNU ARM toolchains like devkitARM, Sourcery G++ Lite, WinARM and YAGARTO. Since it is not included with the toolchain distribution, you may need to install it separately. It is important to verify that the version of Insight in use supports the version of GDB in the toolchain.

## 3.2 Introduction to Insight Functions

The Insight console window allows access to the same GDB commands discussed in the previous chapter. Rules and procedures are the same, but Insight acts as a graphical interface to GDB to simplify its operation. Insight can be invoked from the command line, usually from the directory containing the application binary to be debugged. The devkitARM project includes build of Insight compatible with their toolchain. It can be invoked with '`arm-eabi-insight` *`project_name.out`*'. Figure 3-1 shows a sample debugging session of *`blinky.out`*. After Insight is loaded, a source window will be displayed similar to the one shown in Figure 3-2.

```
C:\Projects\ARM\blinky\gcc> arm-eabi-insight blinky.out
```

**Figure 3-1:** Invoking Insight from the command line

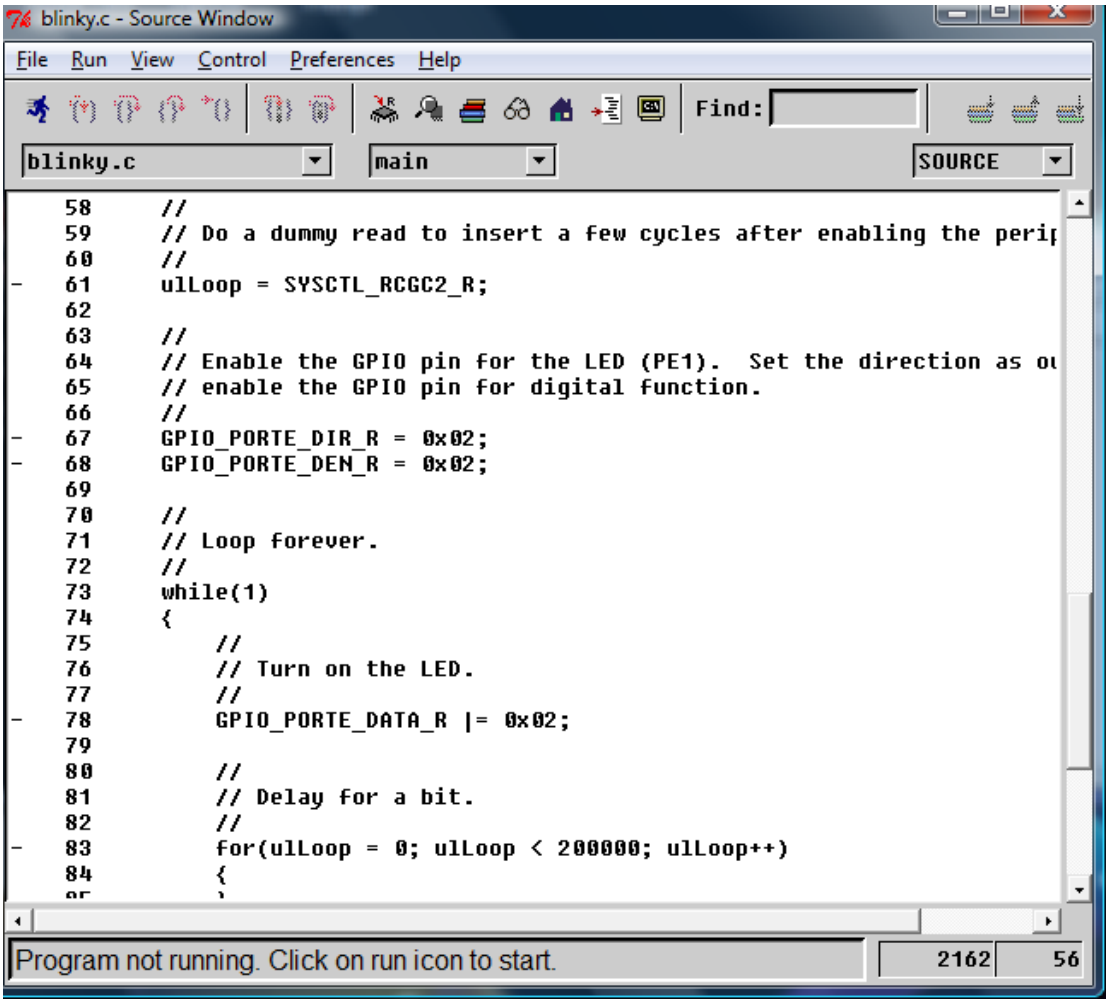


**Figure 3-2:** Insight source window

Insight has several windows useful for debugging, including:

- Console window
- Source window
- Register window
- Memory window
- Locals window
- Watch window
- Stack window
- Thread/process window
- Function browser window
- Debug window (for developers)

Insight can connect to a remote target via OpenOCD by selecting 'File', 'Target' in the menu and specifying the remote target parameters as shown in Figure 3-3. The `localhost:3333` in the example connects to the OpenOCD daemon running in port 3333 of the local machine. Once you have specified your target settings, you can click the Run button so Insight loads the application to your device. Another alternative is to enter the command `'target remote localhost:3333'` in the command window as shown in Figure 3-4. If you specified your target using the console window, you will need to manually load your program to your device typing *load* in the command line.

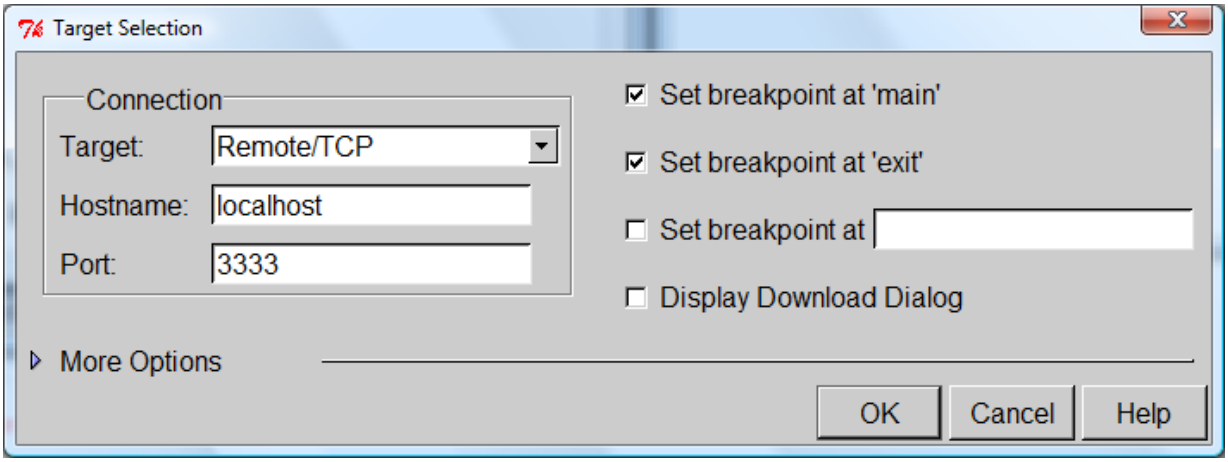


**Figure 3-3:** Target Settings window

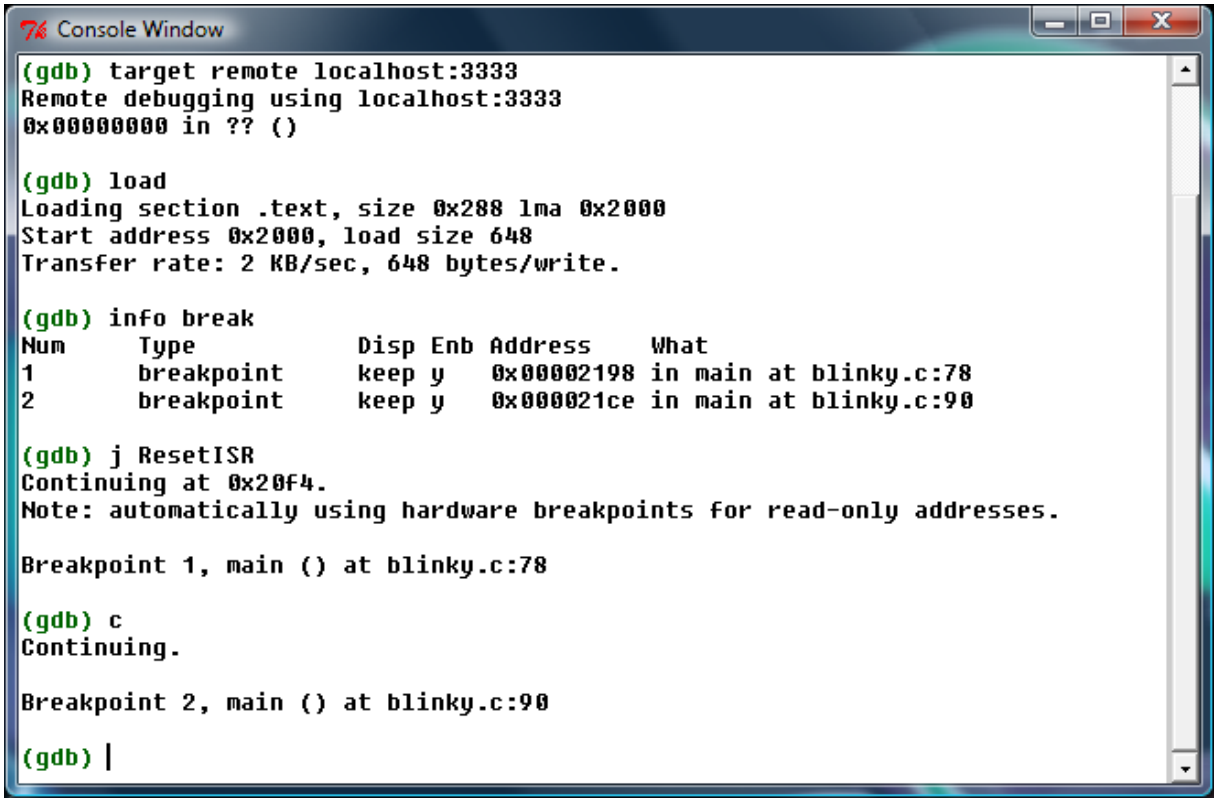


**Figure 3-4:** Console Window

To jump to the start of the application or to a specific location, you just need to invoke the jump command (j *line* or j **address*) as in GDB using the Console Window.

The GDB commands available from Insight are the same as with the command line example in the previous chapter. When placing breakpoints and watching variables you can do it using two methods: setting them manually using the console window as in GDB, or using Insight graphical interface. To set breakpoints in Insight graphical interface just click on left side of the line number where you want to stop the program. To add a variable to the watch window, right click on the variable and select add *variable* to watch. Figure 3-5 shows how to set breakpoints and watching variables. Conditional breakpoints are set as in GDB, using the console windows and entering the command: `cond n [expr]`.

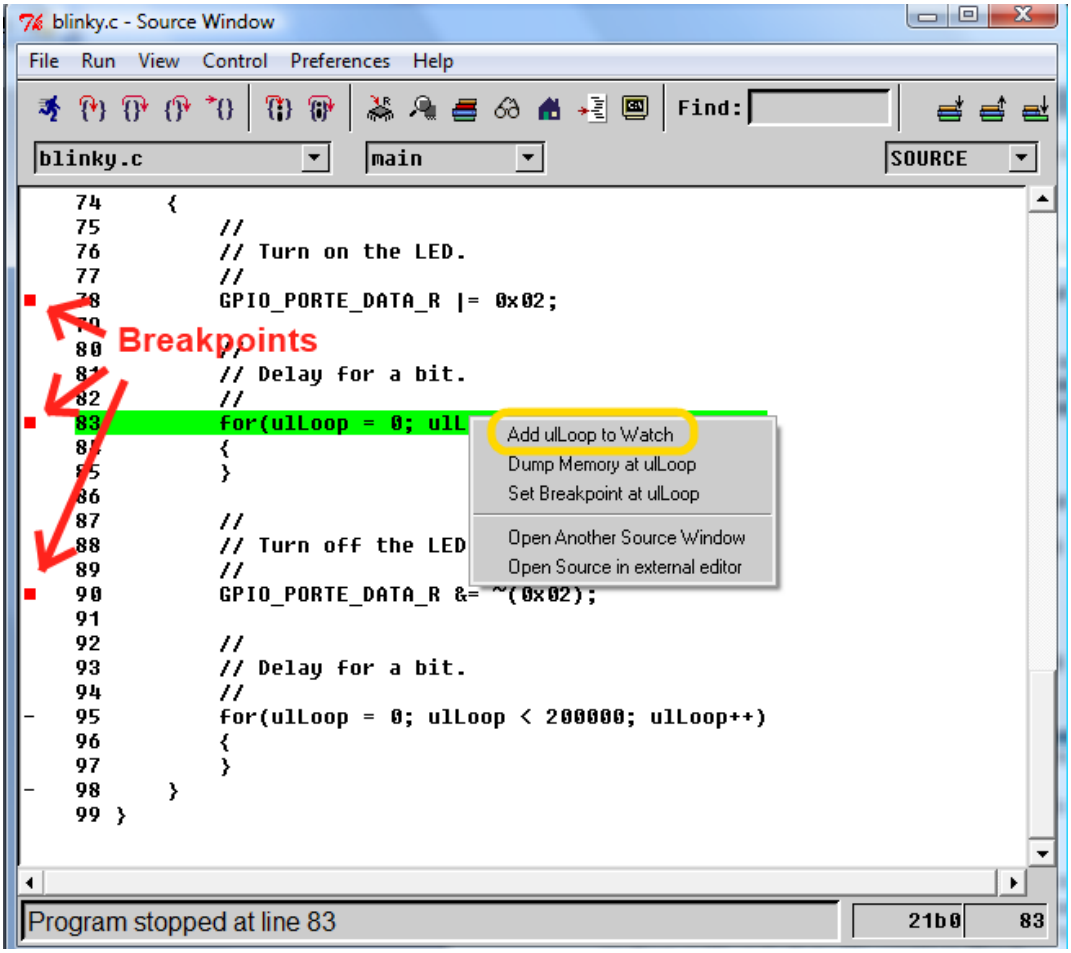


**Figure 3-5:** Setting breakpoints and watching variables

The following list describes the use of the debugger buttons:

 Run – Start the debugging process

 Step – Step into function calls

 Next – Step over function calls

 Finish – Step out of function calls

 Continue – Continue the debugging process

## 3.3 Using Insight from the CodeBlocks IDE

Insight can be invoked from the CodeBlocks IDE using the proper settings. The IDE acts as a launcher for Insight so all procedures discussed in the previous section also apply here.

To set Insight as your debugger within Codeblocks, the procedure is very similar as the one used for GDB. On the 'Compiler and debugger' settings window, look for Global compiler settings and click on Toolchain executables. When using the devkitARM toolchain your Debugger should be set to arm-eabi-insight.exe. Figure 3-6 shows an example of the proper configuration of toolchain executables.

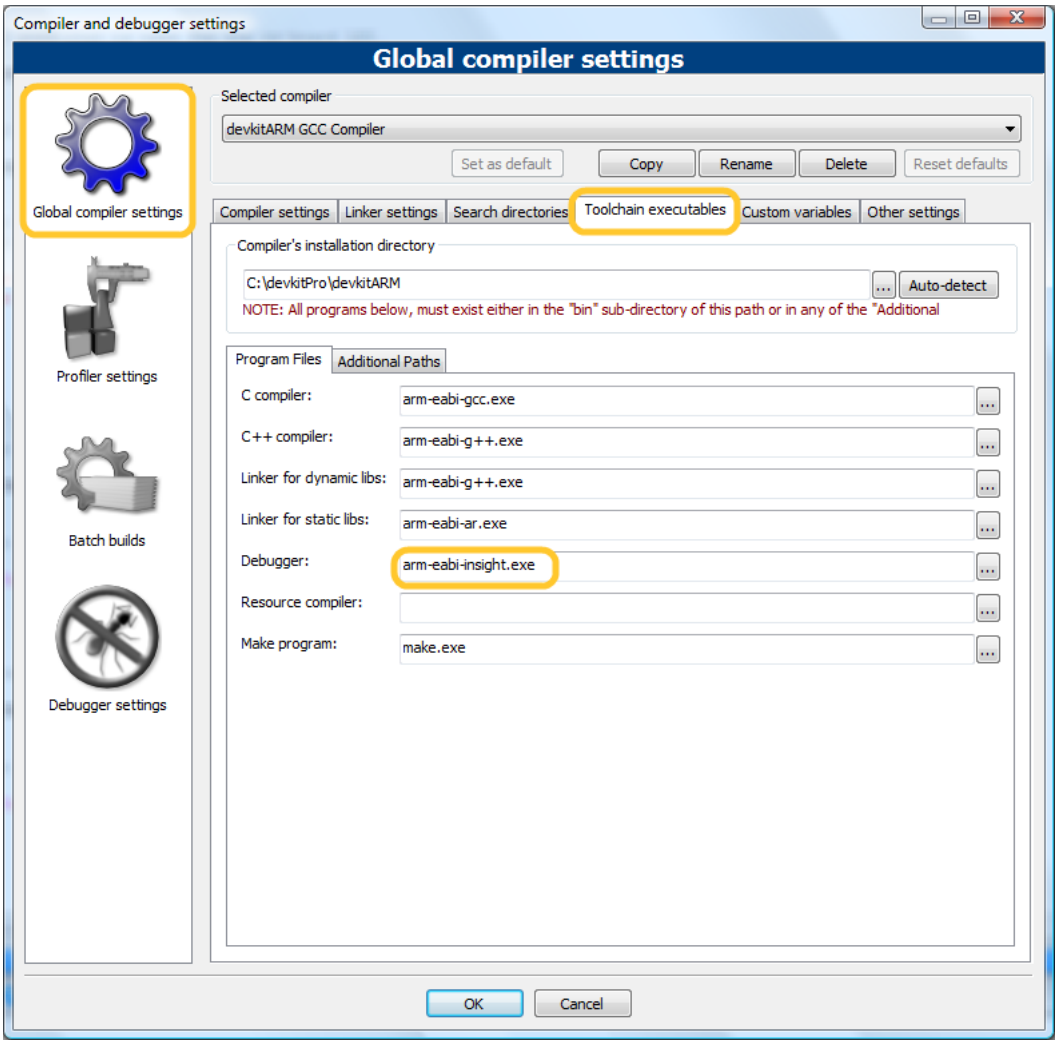


**Figure 3-6:** Setting the Debugger to Insight on Codeblocks

More information about Insight is available online from this reference:

**Insight Home Page**
http://sourceware.org/insight/