Using AVR-GDB and AVaRICE Together

Using the JTAG ICE without AVR Studio

Colin O'Flynn

Introduction

This guide is intended to get you to use the GNU De-Bugger (GDB) for Atmel AVR microcontrollers. It will walk you through what it is and how to use it.

What is AVR-GDB

AVR-GDB is the GDB for the Atmel AVR line of microcontrollers. A de-bugger is a great tool, it lets you view the source code as it executes, view variables, change around information, and a whole wack of other things.

Perhaps your wondering why not just use AVR Studio, a free product from Atmel. However there are a number of reasons why AVR Studio may not be of preference to you. First, AVR-GDB can be run under Linux perfectly, which the newer versions of Studio have trouble doing at all.

Secondly, the GDB is a huge project that was originally design for use in computers. The GDB is used by many people for debugging computer programs that would run on PC's. This means it has a massive feature list as it has been developed by many people over a great deal of time, a few of these features will be shown in this document. Of course not every feature will work on the AVR platform that you have on a normal computer, but there is still a nice set of them.

Finally, AVR-GDB has a very large support group; most questions asked on the avr-gdb mail list will get answered quickly.

In case your wondering, yes AVR-GDB is free.

What is AVaRICE?

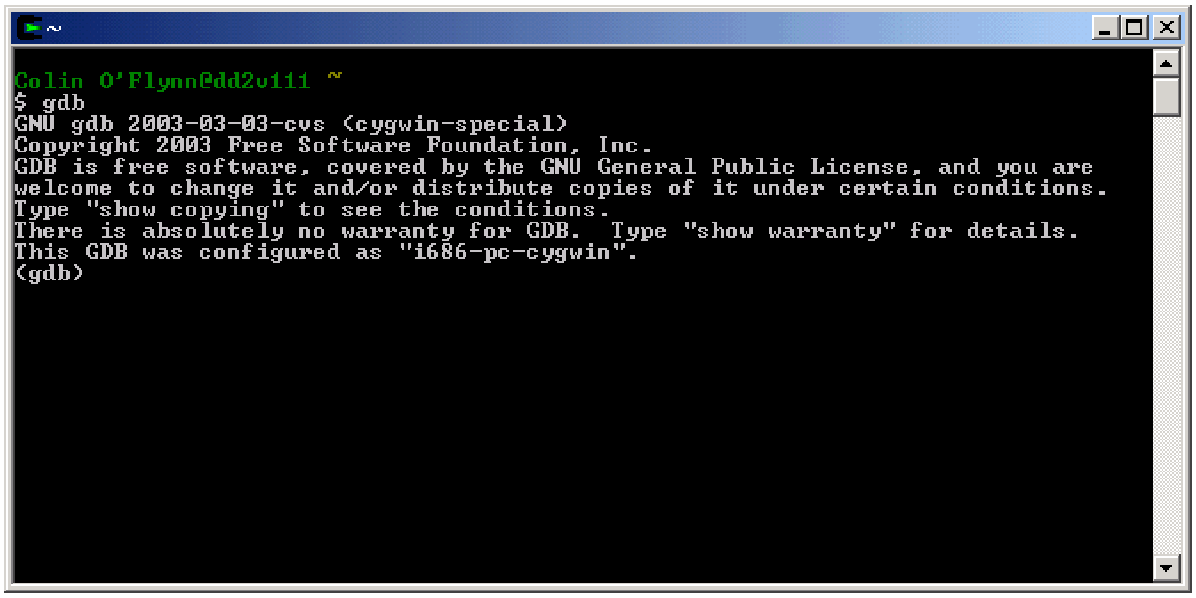
AVaRICE is the tool that talks to the JTAG ICE from Atmel, and passes this information onto AVR-GDB. This is done using GDB's serial debug protocol, so the de-bugger is running on a different target than the program being de-bugged. Also its worth nothing that this is done with a TCP socket, so in theory if you wanted you could have the JTAG ICE connected to one computer with AVaRICE on it, then another computer at some distant point on the network could connect to AVaRICE and de-bug the program from there!

A Quick GDB Tutorial

For now just pretend you already have the GDB set up and running, as it will be better to first understand about GDB and how it will work.

Natively, GDB has no graphical user interface (GUI). It is purely text based, but is very well designed so you don't even need a graphical interface. You can add on a graphical interface is you so desire, but its important to understand the console interface to GDB. Later on this document will shown you how to use a GUI. So if you are a bit lost for this beginning part – don't worry about it yet. It will be easier to understand when you are using the debugger.

Kick off the GDB by typing `gdb` at the console, and the screen then looks something like figure 1 (this is using Cygwin, so unless you have installed that you probably won't have normal GDB on a Windows machine). If you want to follow along, you can type avr-gdb instead at a MS-DOS prompt. If you can't get to one, goto `Start...Run` and type `command.com` then click OK.

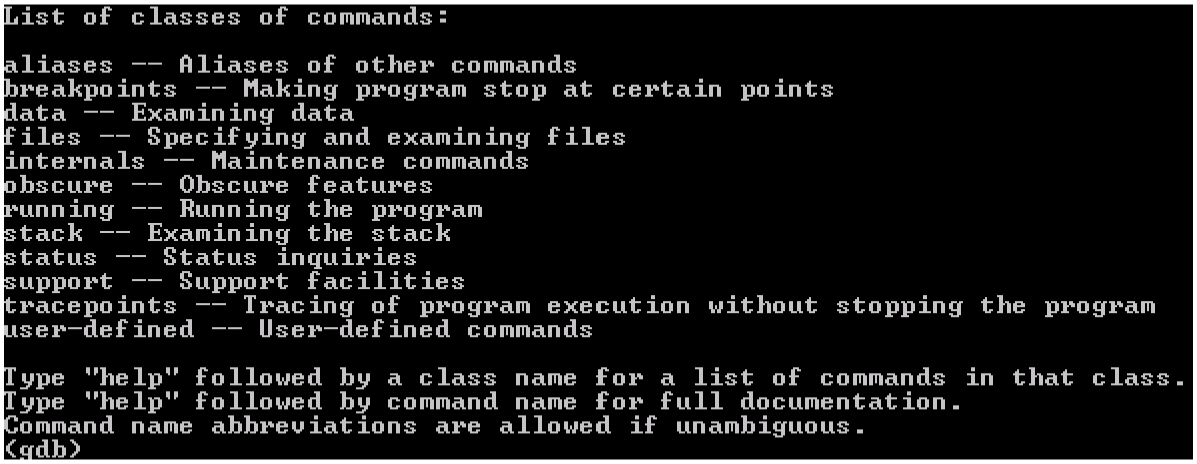


```
Colin O'Flynn@dd2v111 ~
$ gdb
GNU gdb 2003-03-03-cvs (cygwin-special)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-cygwin".
(gdb)
```

Figure 1 Main screen of GDB

The line that says `(gdb)` is the GDB prompt, and this is where you type commands. GDB has a very useful built-in help. To access it, just type `help` at the GDB prompt and hit enter. With the GDB prompt there is a number of speed-savers you can use. First, to repeat the last command just hit `<Enter>`, there is no need to re-type the command again. Also you don't have to type out the entire command name. For example instead of typing out `help` you can just enter `hel` which will work exactly the same. As long as GDB knows what you are going to type you can shorten the command as much as you want.

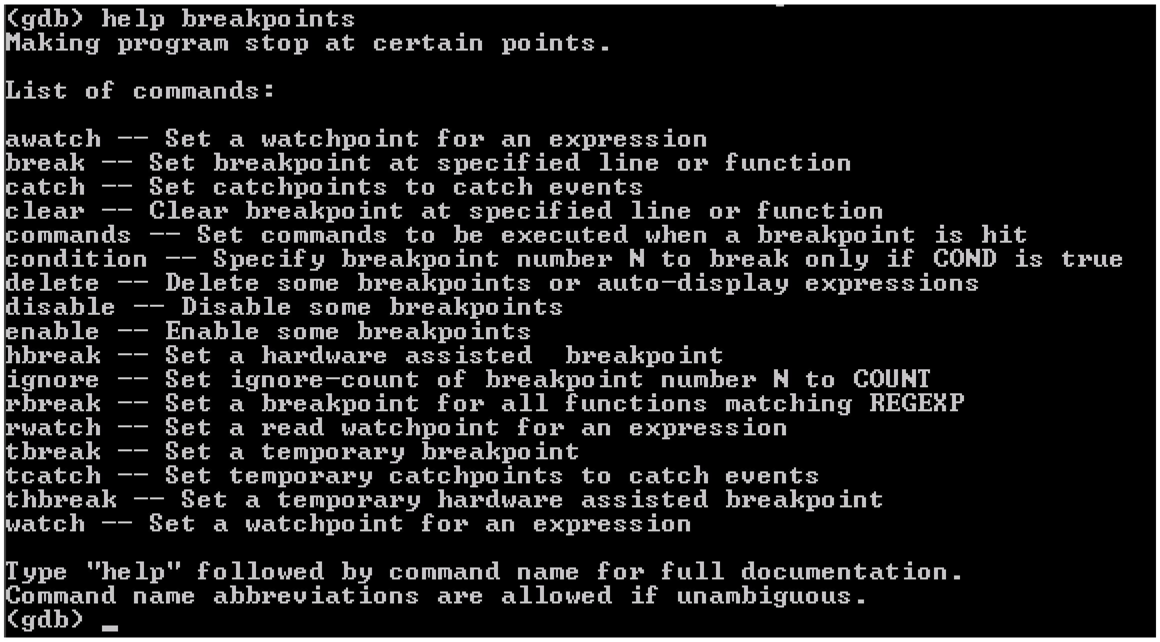
Using help you can get a list of all the commands. The first time you used help it should come up with a screen like that in figure 2.



```
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Figure 2 The GDB help sub-menu options

Now select what topic you want more help on. So for example type `help breakpoints` to see a list of the commands that relate to breakpoints. Now your screen should look like figure 3.



```
(gdb) help breakpoints
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified line or function
catch -- Set catchpoints to catch events
clear -- Clear breakpoint at specified line or function
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
delete -- Delete some breakpoints or auto-display expressions
disable -- Disable some breakpoints
enable -- Enable some breakpoints
hbreak -- Set a hardware assisted  breakpoint
ignore -- Set ignore-count of breakpoint number N to COUNT
rbreak -- Set a breakpoint for all functions matching REGEXP
rwatch -- Set a read watchpoint for an expression
tbreak -- Set a temporary breakpoint
tcatch -- Set temporary catchpoints to catch events
thbreak -- Set a temporary hardware assisted breakpoint
watch -- Set a watchpoint for an expression

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) _
```

Figure 3 The Breakpoints sub-menu

Finally, get the documentation on the command you are interested for. So in this case try `help break` and you will get a blurb on what this command is and what it does.

From now on some of the commands will be discussed, but you won't be able to try them out yet as there is no program loaded on GDB. This will have to wait for later.

Viewing Variables

Viewing the values of variables is an essential part of debugging. So it stands that GDB has some nice facilites to view and access variables. Listing XX shows some examples of this (bolded text is where the user has entered information):

```
(gdb) print count
$1 = 34
(gdb) print $1-10
$2 = 24
(gdb) print my_array[0]@2
$3 = {12, 123, -19, 3, 20}
(gdb) whatis count
type = int
(gdb) whatis my_struct
type = struct my_struct_def *
(gdb) ptype my_struct
type = struct my_struct_def {
    int member1;
    char * address_char;
    } *
```

The first command prints the value of a variable in the program. Also note that it is assigned an identifier, $1. This can be used in other expressions as shown when 10 is subtracted from the value.

For printing arrays, you specify the array, followed be the array index (in [ ] brackets just line in C), followed by @*x* where *x* is how many elements to print.

Finally the `whatis` command can tell you the type of variable you are looking at. The `ptype` command is used to examine structures, as otherwise you are just told it is a structure.

Changing Variables

Changing variables in a running program (of course you will have to pause it before changing the value) is very easy.

To do this, use the GDB `set` command:

```
(gdb) set variable my_var = 10
```

This will set the variable my_var to the value 10.

Changing and Viewing Program Flow

Viewing and adjusting the program flow is an essential part of debugging, and like other funtions GDB gives you a variety of commands.

The `step` and `next` commands are the equivalent of 'step into' and 'step over', respectively. So the `step` command will step one instruction at a time, and if it encounters a subroutine will step 'into' it, calling it and allowing you to step through it. The `next` command on the other case will execute the entire sub-routine when it comes across it.

If you are in a subroutine and want to just finish it without single-stepping, use the `finish` command. The `finish` command finishes the current routine and prints its return value, stopping execution when it returns as well.

If at any time you want to call another subroutine, that can be done using the `call` command. It is just like calling it from a C routine, except for no trailing semi-colon. For example to call the routine 'send_string("Test of UART", counter_var1)' you would do this:

```
(gdb) call send_string("Test of UART", counter_var1)
```

So this will call the routine send_string with the string "Test of UART" as the first argument and the variable counter_var1 as the second.

At this point you may be wondering one thing: sure I know how to single-step through the code, and view its variables, and call routines, but how do I see what the code is!? That is where the `list` command comes in. The `list` command shows you a few lines of code around the currently executing block.

If you want to let the program run full-speed, and continue until it hits a break-point, you would use the `run` command normally. However this is NOT the case with avr-gdb, instead the program will already be running, but paused, when you start it up. For this reason you use the `continue` command. The `continue` command starts program execution from where it is currently paused.

Now the other problem is how do you stop the program flow? This is where breakpoints come in. You can set them on any line of code in any file, and before that line executes the program execution is stopped. Note however that the current version of avarice only supports hardware breakpoints, of which there are only three. So you cannot set more than three breakpoints yet, although perhaps by the time you read this avarice will support software breakpoints as well.

To set a breakpoint, use the `break` command (what a suprise). Here are some examples of using the command:

(gdb) **break 100**

(gdb) **break putchar**

(gdb) **break uart.c:45**

```
(gdb) break uart.c:outch
```

You can specify to break at a certain line number (the case with break 100), at a certain function (the case with putchar), and you can specify to put the breakpoints in another file than the current one being executed.

To get information on what breakpoints are set, use the `info breakpoints` command. Then you can delete or disable them with the `delete` or `disable` command, as such:

```
(gdb) info breakpoints
(gdb) delete 2
(gdb) disable 1
(gdb) info breakpoints
```

Running AVaRICE and AVR-GDB Together

Now that you know how to use GDB, its time to get it running with AVaRICE for your JTAG use.

First connect your JTAG ICE to the serial port, and open a terminal. In Windows you can do this by selecting "MS-DOS Shell" from somewhere on the Start Menu, or lacking that open the "Run" dialog on the Start Menu and type in command.com and hit OK as shown in figure 4.

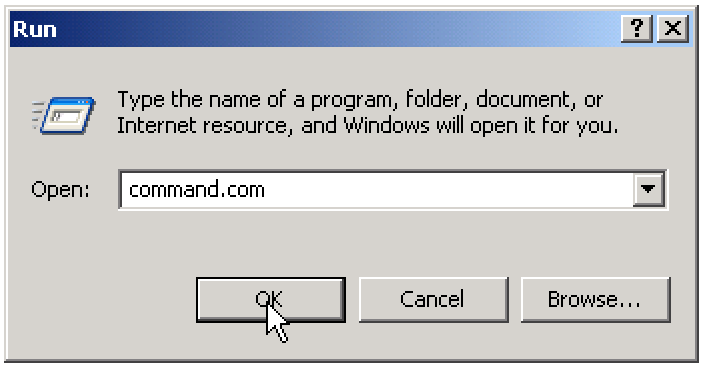
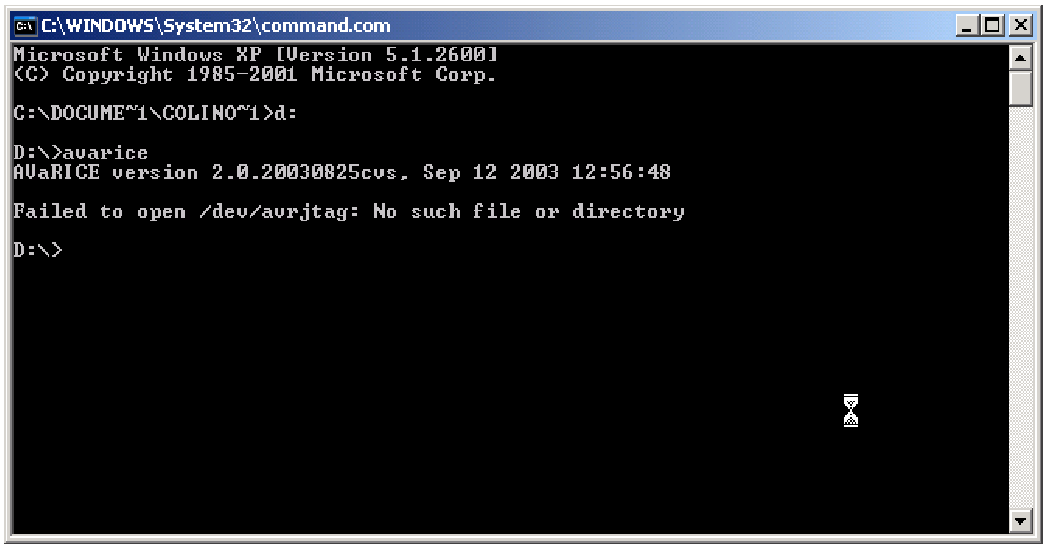

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
command.com
OK
Cancel
Browse...

Figure 4 - Run Dialog

Now you will want to make sure you actually have avarice installed correctly, and your path set up as well. If you have not installed WinAVR already, now would be a good time to do that! Type `avarice` at the command prompt, and hit enter. Hopefully you get some sort of response indicating avarice is working. Pretty much anything besides "File Not Found" is a good sign.

Now you can get more information by running `avarice --help` which will bring up a huge list of all the options for avarice, you should read through this list to get an idea of what to do.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\DOCUME~1\COLINO~1>d:

D:\>avarice
AVaRICE version 2.0.20030825cvs, Sep 12 2003 12:56:48

Failed to open /dev/avrjtag: No such file or directory

D:\>
```

Figure 5 - Running avarice for the first time

Go to the directory with your project file in it, and make sure you have compiled it recently. In this example the project is called main, with the elf file being main.elf. At the command prompt type:

```
avarice --program --file main.elf --part atmega128 --jtag com1 :4242
```

There may be a newer version of avarice with more options by the time you read this, which is why you should examine the --help output to see what is new. This example will use the file main.elf, program the flash, with the JTAG ICE connected to com1, then listen for a connection on port 4242.

This port is how avr-gdb communicates with avarice – and since it is the TCP/IP protocol this means you could actually run avarice on one computer, then connect to that computer with another computer running avr-gdb.

You may want to get the latest info on where avarice and avr-gdb stand, so see what documents come with your version. For example if you are using WinAVR check the WinAVR\doc\avarice directory, and read the files to see what they say.

Open up a new terminal, go to the directory with your project in it, and start up avr-gdb specifing your elf file on the command line. For example I would type:

```
avr-gdb main.elf
```

And you should be greeted with a message from avr-gdb, and the gdb prompt should come up. Now you have to connect to avarice, to do so type this at the gdb prompt:

```
(gdb)  target remote localhost:4242
Remote debugging using localhost:4242
0x00000000 in __vectors ()
```

Now try a few commands, first try `list`. Then start playing around with avr-gdb. Note that if you type `continue` without breakpoints set, the program will never stop executing. To break it hit `<Ctrl> + C` (hold down Ctrl and tap the C key, then release both). To exit avr-gdb type quit, and not that this will close the connection to avarice so to restart avr-gdb you will first have to restart avarice.

A Graphical Interface to GDB

At this point you might not be thinking to kindly of GDB, with its text interface. Luckily WinAVR comes with Insight, which is a graphical front-end for GDB that is very easy to use.

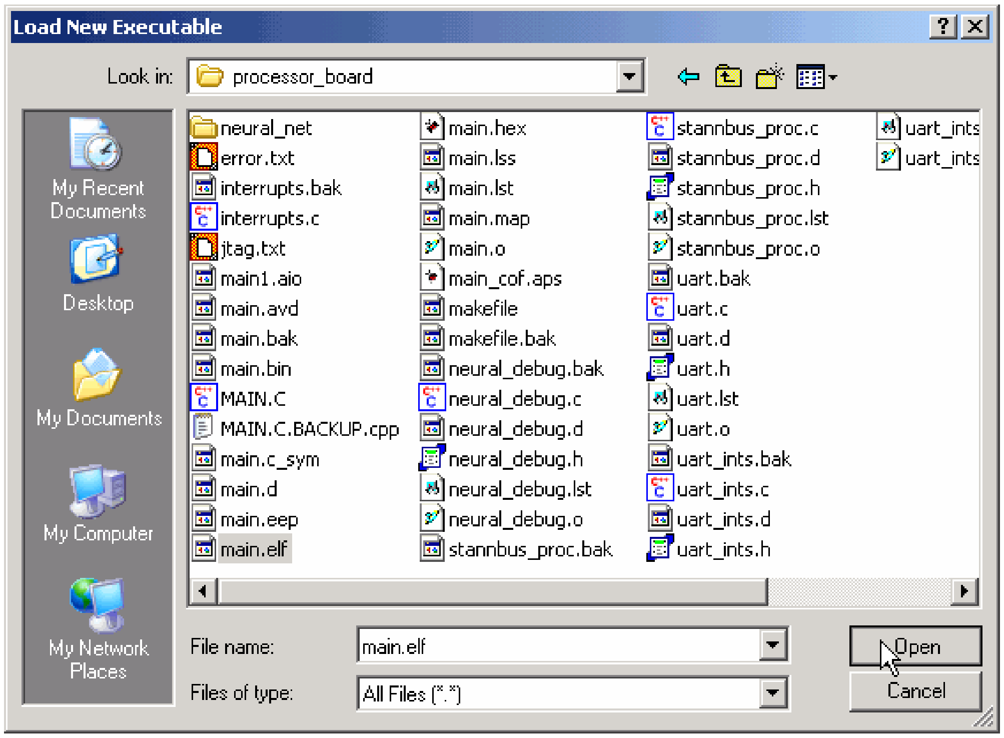
First you have to run avarice as described before, for example type:

```
avarice --program --file main.elf --part atmega128 --jtag com1 :4242
```

Now you have to start up Insight. WinAVR may have put an icon on your desktop, it will look something like figure 6. If that is not present, then directly run the program, its default location being: C:\WinAVR\bin\avr-insight.exe.

Figure 6

When AVR Insight loads, go to the "File" menu and select "Open", and find your elf file as shown in figure 7.



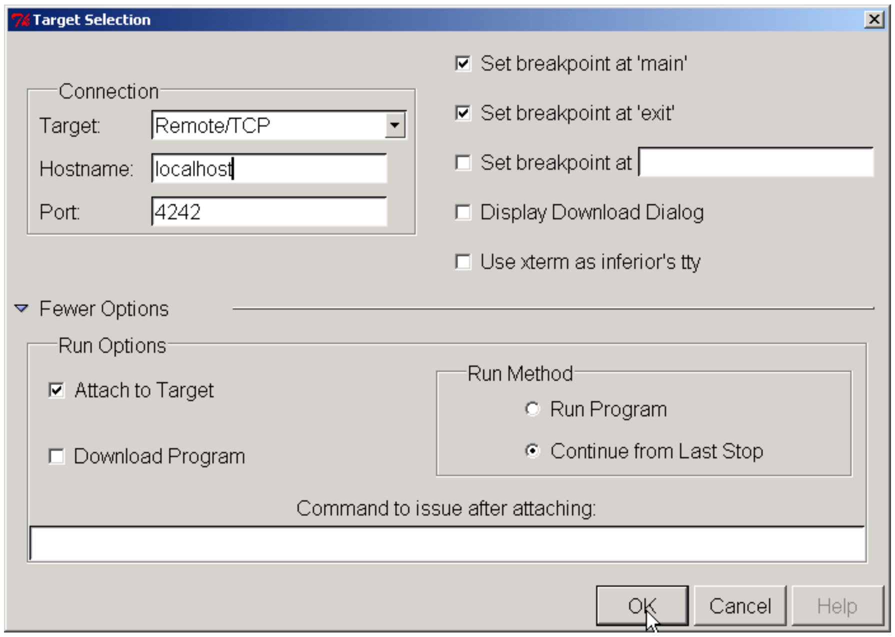
neural_net
error.txt
interrupts.bak
interrupts.c
jtag.txt
main1.aio
main.avd
main.bak
main.bin
MAIN.C
MAIN.C.BACKUP.cpp
main.c_sym
main.d
main.eep
main.elf
main.hex
main.lss
main.lst
main.map
main.o
main_cof.aps
makefile
makefile.bak
neural_debug.bak
neural_debug.c
neural_debug.d
neural_debug.h
neural_debug.lst
neural_debug.o
stannbus_proc.bak
stannbus_proc.c
stannbus_proc.d
stannbus_proc.h
stannbus_proc.lst
stannbus_proc.o
uart.bak
uart.c
uart.d
uart.h
uart.lst
uart.o
uart_ints.bak
uart_ints.c
uart_ints.d
uart_ints.h
uart_ints
uart_ints

File name: main.elf

Files of type: All Files (*.*)

Figure 7 - Loading the elf file in Insight

Now the Insight Window should be looking like a debugger! However you can't run anything yet, you still have to connect to avarice. Go to the “File” menu and select “Target Settings”, and a dialog like figure 8 should appear.



Target: Remote/TCP
Hostname: localhost
Port: 4242

[✓] Set breakpoint at 'main'
[✓] Set breakpoint at 'exit'
[ ] Set breakpoint at
[ ] Display Download Dialog
[ ] Use xterm as inferior's tty

[✓] Attach to Target
[ ] Download Program

Run Program [✓ Continue from Last Stop]

Command to issue after attaching:

Figure 8 - Target Settings

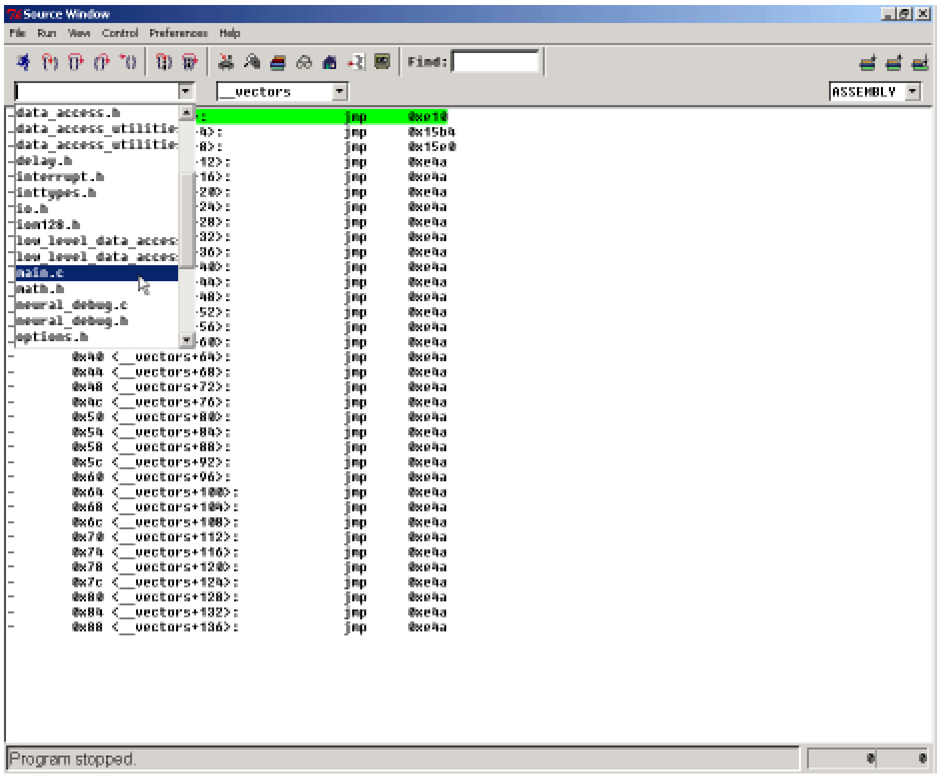
It is important you set this dialog up properly. First, select “Remote/TCP” as the “Target:” type. The “Hostname” will normally be “localhost” (as long as the JTAG ICE is connected to the same computer Insight is running on), and the “Port” should be what it is set to in avarice, which is “4242” for this example.

Then move to the check-boxs, make sure “Set breakpoint at 'main'” is checked. Then hit the “More Options” arrow, and more options should appear (if they weren't already there).

Make sure “Attach to Target” is checked, and “Download Program” is NOT checked. The “Run Method” should be “Continue from Last Stop”. Then hit the OK button.

Now go to the “Run” menu and hit “Connect to Target”. A message should come up saying the connection was made OK, and your Insight screen should change. Likely it will be displaying the first line of code, which will just be interrupt vectors.

Now you should be able to hit the run button, and the code will run until it hits the first breakpoint, which should be set at main because you checked the box in the “Target Settings” window. This doesn't always seem to work though, so you should manually check it. First select the source file with the main() routine in it, as shown in figure 9.

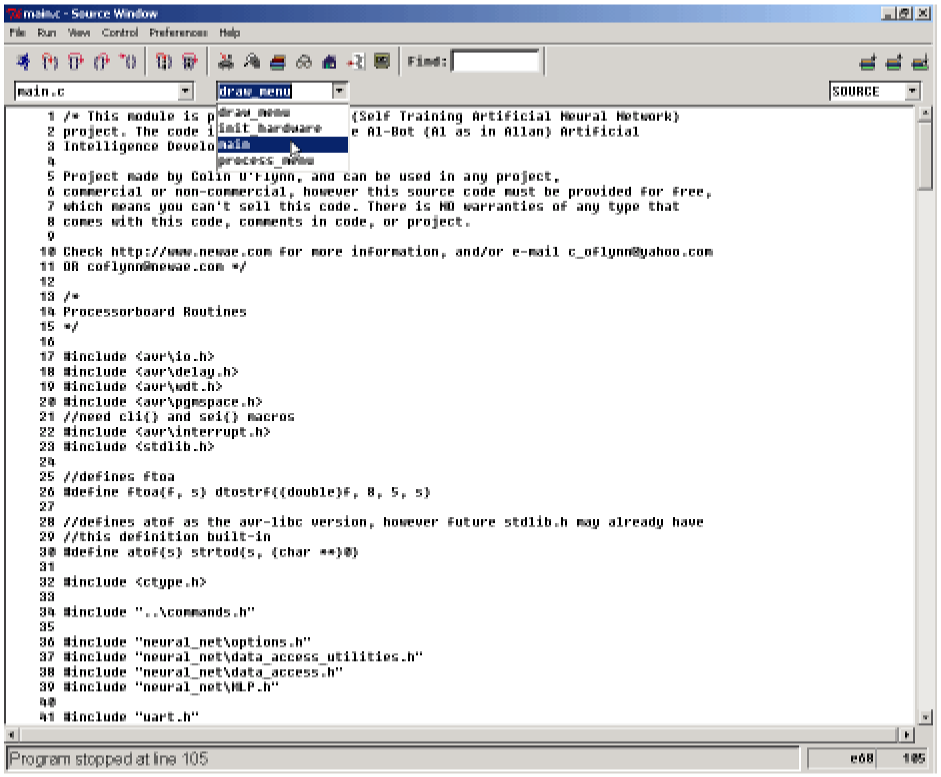


```
:                        jmp     0xe10
4>:                      jmp     0x15b4
8>:                      jmp     0x15e0
12>:                     jmp     0xe4a
16>:                     jmp     0xe4a
20>:                     jmp     0xe4a
24>:                     jmp     0xe4a
28>:                     jmp     0xe4a
32>:                     jmp     0xe4a
36>:                     jmp     0xe4a
40>:                     jmp     0xe4a
44>:                     jmp     0xe4a
48>:                     jmp     0xe4a
52>:                     jmp     0xe4a
56>:                     jmp     0xe4a
60>:                     jmp     0xe4a
-        0x40 <__vectors+64>:     jmp     0xe4a
-        0x44 <__vectors+68>:     jmp     0xe4a
-        0x48 <__vectors+72>:     jmp     0xe4a
-        0x4c <__vectors+76>:     jmp     0xe4a
-        0x50 <__vectors+80>:     jmp     0xe4a
-        0x54 <__vectors+84>:     jmp     0xe4a
-        0x58 <__vectors+88>:     jmp     0xe4a
-        0x5c <__vectors+92>:     jmp     0xe4a
-        0x60 <__vectors+96>:     jmp     0xe4a
-        0x64 <__vectors+100>:    jmp     0xe4a
-        0x68 <__vectors+104>:    jmp     0xe4a
-        0x6c <__vectors+108>:    jmp     0xe4a
-        0x70 <__vectors+112>:    jmp     0xe4a
-        0x74 <__vectors+116>:    jmp     0xe4a
-        0x78 <__vectors+120>:    jmp     0xe4a
-        0x7c <__vectors+124>:    jmp     0xe4a
-        0x80 <__vectors+128>:    jmp     0xe4a
-        0x84 <__vectors+132>:    jmp     0xe4a
-        0x88 <__vectors+136>:    jmp     0xe4a
```

```
Program stopped.                                0       0
```

Figure 9 - Selectin the main source file

Next scroll down to your main routine, or select it from the routine list as shown in figure 10.

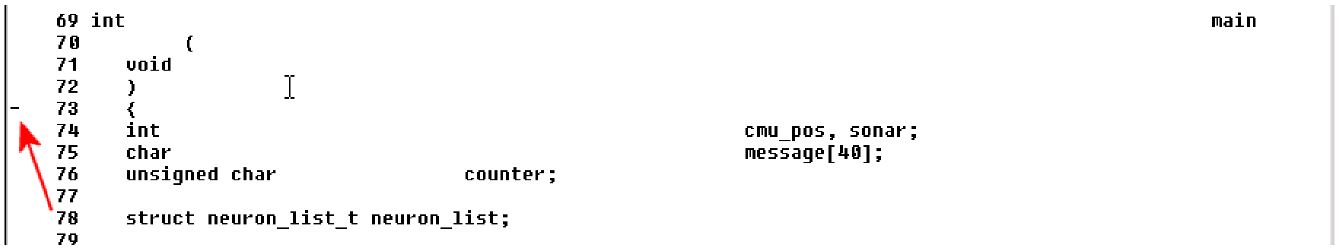


```
/* This module is p                    (Self Training Artificial Neural Network)
project. The code i                    e A1-Bot (A1 as in Allan) Artificial
Intelligence Develo

Project made by Colin O'Flynn, and can be used in any project,
commercial or non-commercial, however this source code must be provided for free,
which means you can't sell this code. There is NO warranties of any type that
comes with this code, comments in code, or project.

Check http://www.newae.com for more information, and/or e-mail c_oflynn@yahoo.com
OR coflynn@newae.com */

/*
Processorboard Routines
*/

#include <avr\io.h>
#include <avr\delay.h>
#include <avr\wdt.h>
#include <avr\pgmspace.h>
//need cli() and sei() macros
#include <avr\interrupt.h>
#include <stdlib.h>

//defines ftoa
#define ftoa(f, s) dtostrf((double)f, 8, 5, s)

//defines atof as the avr-libc version, however future stdlib.h may already have
//this definition built-in
#define atof(s) strtod(s, (char **)0)

#include <ctype.h>

#include "..\commands.h"

#include "neural_net\options.h"
#include "neural_net\data_access_utilities.h"
#include "neural_net\data_access.h"
#include "neural_net\MLP.h"

#include "uart.h"
```

```
Program stopped at line 105                     e68     105
```

Figure 10 - Selecting the main() routine

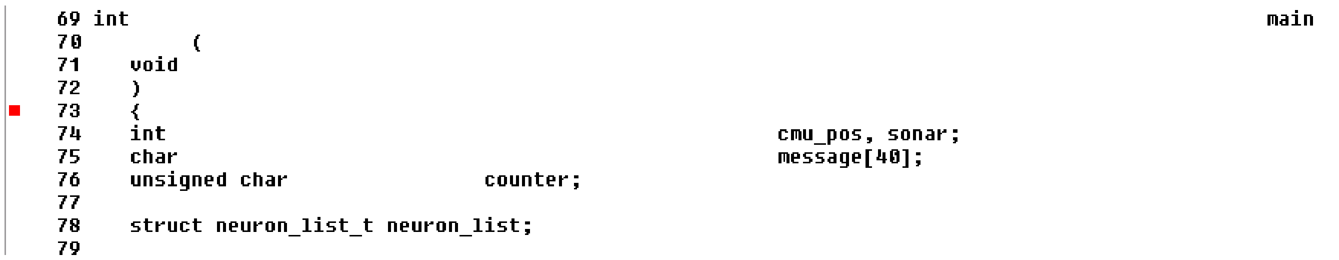
Now you should see a '–' on the left of the line numbers, right around your main() decleration. Anywhere there is a dash you can click to set a breakpoint. See figure 11 for an example of what this looks like, the red arrow points to the dash. If you have a red box instead of a dash, the main() breakpoint was properly set.



```
   69 int                                                                                    main
   70          (
   71    void
   72    )
-  73    {
   74    int                                                cmu_pos, sonar;
   75    char                                               message[40];
   76    unsigned char                counter;
   77
   78    struct neuron_list_t neuron_list;
   79
```

Figure 11 - This dash means a breakpoint could be set, but currently isn't

Then click on it with your cursor, when your cursor is over the dash it should change shape. After you click a red box will be there instead, this means a breakpoint is set. This is shown in figure 12.



```
   69 int                                                                                    main
   70          (
   71    void
   72    )
■  73    {
   74    int                                                cmu_pos, sonar;
   75    char                                               message[40];
   76    unsigned char                counter;
   77
   78    struct neuron_list_t neuron_list;
   79
```

Figure 12 - A breakpoint is set at this location

Now you can go ahead and hit the “run” button, which is on the top toolbar. If you hover the cursor over a button it will give its name as well as the keyboard shortcut (ie: for Run it is the 'R' key).

When the code hits that breakpoint, execution will be stopped. You may then want to remove the breakpoint by clicking on it again, as it will not be hit again and just takes up one of the available breakpoints.

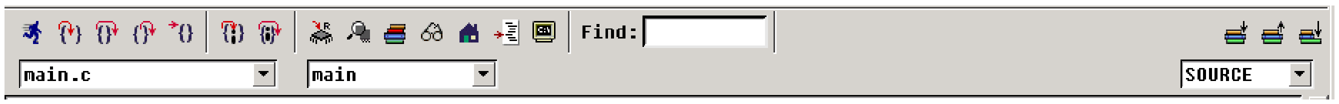


Figure 13 - The Toolbar

With the top toolbar (which is shown in figure 13) you can run, stop, step into ('step'), step over ('next'), step out ('finish'), and even just step assembly-level instructions.

Also you can view register contents, memory contents, view all local variables, and access the GDB prompt (for executing commands that aren't in the graphical interface) as well as a few other things.

A final note of interest is the pull-down menu in the right-corner that will likely say “SOURCE” lets you change what is being viewed. You can view the high-level source code, the low-level assembly code, or even view the two mixed together in the same or separate windows.

Closing Notes

This document will hopefully have shown you how to use AVaRICE and AVR-GDB together. By the nature of these tools they are always evolving, and new features will be added.

You should always expore what it looks like the current tools can do, read the documentation that comes with them, and even read some of the latest postings on the various mailing lists to see if there is talk of new features.

If you have trouble related to AVaRICE, you should contact the avarice mailing list, look for it on its homepage at http://avarice.sourceforge.net.

If you have other trouble you should post either to the AVRFreaks.net avr-gcc forum or the avr-gcc mailing list, see http://www.openavr.org for a link to the mailing list.