

An Introduction to the GNU Debugger

The GNU Debugger, part of the GNU Compiler Tools software suite, is the source-level debugger used in the Digital Systems Laboratory. This powerful debugger allows you to run your programs under controlled conditions. Doing this lets you see exactly what is going on in your program, helping you to remove any problems (“bugs”) that might be present.

The GNU Debugger is extensively documented in the *GNU Debugger Manual*, which can be found on your CD-ROM in the *gnutools/doc* directory. This Introduction is a summary of that manual specifically for the Laboratory.

The *examples* directory and its subdirectories on your CD-ROM contain many examples of assembly language and C programs that you can use to build up your debugging skills. And you are encouraged to do so, as part of your laboratory preparation! Ideally, you should read this document in front of a computer with the GNU Debugger, so that you can try out the various commands. Please see the end of this document for details.

Invoking the Debugger

The GNU Debugger can be run in two different modes: with a *graphical interface* called Insight, and with a traditional *command-line interface*. The graphical interface makes basic debugging tasks much easier, while the command-line interface gives you considerable power for more complicated tasks. You will most likely want to use the graphical interface for most of your debugging, using the command line only when necessary.

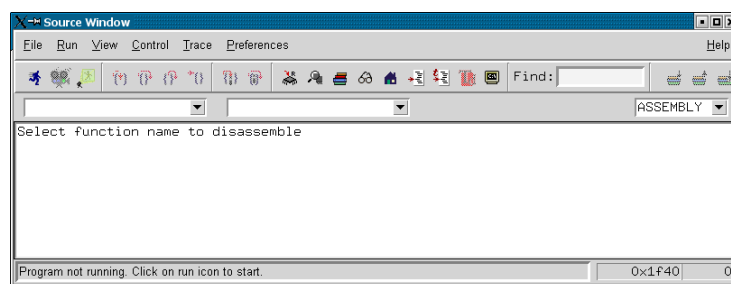
Starting the Graphical Interface

You can start the debugger in its graphical interface mode by entering the following command line in the Unix shell:


```
arm-elf-insight filename.elf &
```

Naturally, replace *filename.elf* with whatever is appropriate for your executable file. Notice the ampersand “&” at the end of the command line: this makes the **arm-elf-insight** program execute as a shell background task.

The following Source window should appear:



Other windows may also appear; you may ignore them for now.

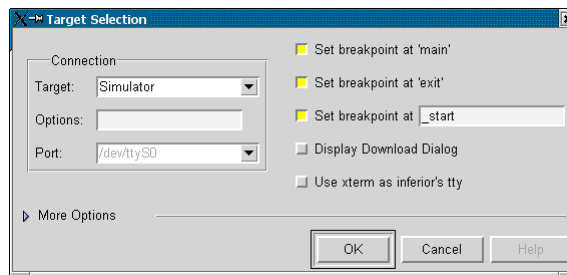
One of the first things you should do is open the debugger’s Console window: this gives you the full GNU Debugger command-line interface, which you will need for more advanced tasks. You can do this by selecting **View » Console** from the main menu, or by pressing the  button in the toolbar. In this document, “the main menu” always means “the main menu in the Source window”; the “toolbar” is the toolbar in the same window.

Before you can begin the actual task of debugging your ARM executable, you need to download that executable to a hardware board or to a simulator. This is called *connecting and downloading to the target*.


To do this, first select **File » Target Settings** from the main menu. A Target Selection dialog box will appear, allowing you to choose your target.

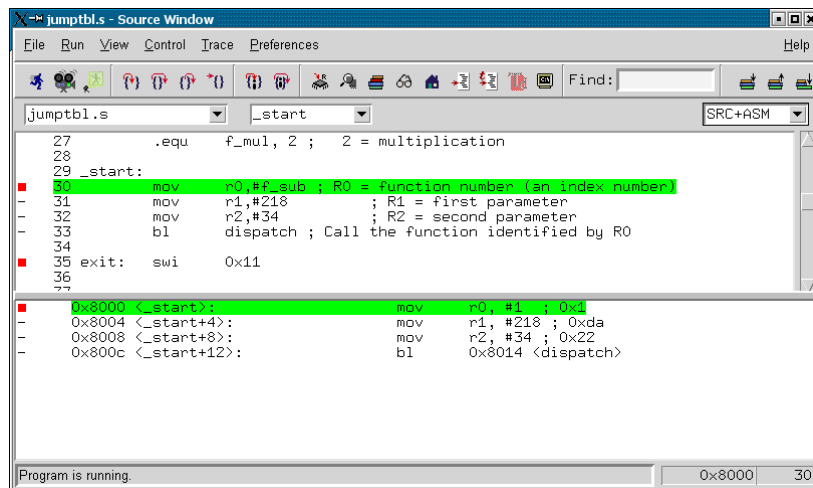
If you have a real hardware board in front of you, select the appropriate target, baud rate and port. Make sure that **Set breakpoint at main**, **Set breakpoint at exit** and **Set breakpoint at _start** are all selected. Click **OK** to close the dialog box.

If you do not have a hardware board in front of you, or don't need to use it, select **Simulator** as the target. The *simulator* is a piece of software on your computer that pretends to be a real hardware board—it simulates the ARM microcontroller and some of the hardware that is present on the real board. Once again, make sure that the three breakpoint choices are all selected, as shown below. Click **OK** to close the dialog box.



Now you are almost ready to run the program. Select **Run » Connect to Target** from the main menu to connect to the hardware or simulator, then **Run » Download** to download the program to the board's memory.

Finally, select **Run » Run** from the main menu to run the program. You can also click on the  button in the toolbar. You should see something like the following (make sure that you select **SRC+ASM** mode, as shown):



You are finally ready to start debugging!

Starting the Command-Line Interface

You can start the command-line mode of the GNU Debugger by entering the following Unix shell command line:

```
arm-elf-gdb filename.elf
```

You will obviously need to replace *filename.elf* with whatever is appropriate for your ARM executable. After printing some introductory remarks, the debugger will issue the prompt “(gdb)”.

As in the graphical interface mode, you need to connect to the target and download your program to it. If you want to connect to real hardware, type:

```
set remotebaud baudrate
target targetname port
load
```

where *targetname* is the name of the target for your hardware board (eg, *rdi*), *baudrate* is the appropriate baud rate (eg, 19200) and *port* is the appropriate port name (eg, */dev/ttyS1*).

If you want to use the built-in simulator, type:

```
target sim
load
```

You should set a few breakpoints, for your own convenience, before you start running the program (ignore any messages that might occur):

```
b _start
b main
b exit
```

Now you can start running the program until it hits the first breakpoint:

```
run
```

Getting Help

One of the first things you should learn about the GNU Debugger is how to get help when you need it. The command to do this is called, appropriately enough, **help**. You can type this at the command line by itself, or with optional arguments (parameters). For example, **help b** gives you a brief explanation of the **b** command used previously.

The GNU Debugger allows you to abbreviate commands; the most commonly used commands can be shortened to a single letter. For example, you can reduce **help** to **h**.

The command-line interface has a number of short-cuts that will make your life a little easier. Pressing the UP ARROW and DOWN ARROW keys will retrieve command lines that you typed in previously. Pressing ENTER will usually repeat the last command you typed.¹ Pressing the TAB key once or twice will complete a command or list all possible completions. For example, typing in **h TAB TAB** will make the GNU Debugger list all commands starting with “h”.

The Insight graphical interface comes with additional help: selecting **Help » Help Topics** from the main menu gives you an explanation of each of the windows that can be opened.

Quitting the Debugger

The command to terminate the debugger is **quit**, or **q** for short. In the graphical interface, you can also select **File » Exit** from the main menu.


By the way, there is no need to quit the debugger just because you have made some changes to your program. For example, imagine you have found an error in the program being debugged. Simply modify the program source code in your editor

¹ The GNU Debugger actually tries to implement a DWIM (Do What I Mean) interface, and this usually works reasonably well... By the way, see <http://www.catb.org/~esr/jargon/html/> for a useful resource that explains acronyms like this one.

window and save it, then type **make** in the Unix shell.² You now need to download the new executable (select **Run » Download** from the main menu, or type **Load**) and restart the program (select **Run » Run** from the main menu, or type **run**).

Starting Your Program


You have already met the command to start running your program from the beginning: **run** (it can be abbreviated to **r**). Essentially the only effect of this command is to reset the program counter (register R15 on the ARM microcontroller, also called PC) to the start of your program and to start executing it. In particular, it does *not* reset the stack pointer register R13 (SP)!

The graphical interface equivalent of the **run** command is **Run » Run** from the main menu. You can also press the  button in the toolbar; place the mouse cursor over a button (“hover” the mouse) to see what that button does.

Section 4.2 of the *GNU Debugger Manual* has considerably more information about the **run** command; however, most of it does not apply to embedded systems debugging, which is what you are interested in doing.

Stopping Your Program

The whole purpose of a debugger is to control the execution of your program. In particular, a debugger allows you to stop your program at any point and then to proceed in a way most convenient to you. There are two main ways of stopping: the “abnormal” or emergency way, and the normal way.

The abnormal or emergency way to stop your program is by sending it an *interrupt*; this is especially useful if your program has “run away” from you! Under the graphical interface, simply press the  button in the toolbar. In the command-line interface,³ press the CTRL and C keys together (CTRL+C).

The normal way to stop a program is by setting breakpoints in it. A *breakpoint*, as its name implies, breaks the program’s execution at that point. The GNU Debugger has a wealth of commands for setting different types of breakpoints, as described in Section 5.1 of the *GNU Debugger Reference*; however, you really only need to know the basics.

The main command to remember is **break** (abbreviated to **b**). This command takes an argument: where the breakpoint is to be set. Some examples are:

```
b dispatch
b 211
b copy.s:14
b *0x004A4E5A
```

The first example sets a breakpoint at the label or function name **dispatch**, the next example sets one at line 211 in the current source code file (you should have the source files opened in your text editor), the third at line 14 of the file *copy.s* and the last at address 0x004A4E5A.

Setting a breakpoint under the graphical interface is even easier: simply click the left mouse button on the “-” symbol of the line at which you wish to stop. You should see a red square replace the “-” symbol.

² This assumes that you are using Makefiles to manage your project. This is highly recommended! If you are using the command-line interface mode, you can type **make** from within the debugger itself.

³ This is one of the rare times that the Console window in the graphical interface does *not* behave in the same way as the command-line interface. In other words, you **do** have to use the **Stop** button, even if you are using a Console window.

Every breakpoint is given a number when it is set. You can delete a breakpoint by using the **delete** command (which can be shortened to **d**) with its number. For example, **d 5** deletes breakpoint number 5. Under the graphical interface, you can delete a breakpoint by clicking on it with the left mouse button.

You can see a list of breakpoints by using the **info breakpoints** command (**i b** for short). This gives you information about the breakpoint number, whether it is enabled or disabled, where that breakpoint is, any conditions associated with it and how many times the program has already stopped at that point.

The phrase “any conditions associated with it” in the previous sentence implies some of the power of the GNU Debugger. For example, if you only want to stop at the label `loop_start` once register R3 is less than or equal to one (so that you skip the first 4000-odd iterations of the loop, say), you can type:


```
b loop_start if ($r3 <= 1)
```


Notice that the condition is a full-blown C expression⁴ that can be as complicated as you like. Note also that the GNU Debugger requires you to put “\$” in front of all register names—this is so that you can still have a variable of the same name in your program. In other words, **\$r0** is register R0, **r0** is a variable or label named `r0`. Having variables or labels of the same name as registers is not recommended in assembly language programs, but is quite all right in C.


The current version of the graphical interface does not allow you to set conditional breakpoints using a mouse: you need to use the **break** command in the console window if you want to do this.

You might also want to know about the **disable** and **enable** commands; issue **help disable** and **help enable**, or see Section 5.1.5 of the *GNU Debugger Reference*, for more information.

Stepping Through Your Program

Once your program has stopped at a breakpoint, you can let it continue running (until it reaches another breakpoint or the end of the program). You do this with the **continue** command (**c** for short). In the graphical interface, select **Control » Continue** from the main menu or click on the  button in the toolbar.

If you want to execute just the next line of source code, use the **step** (**s**) command. The graphical interface equivalent is **Control » Step** from the main menu or the  toolbar button. If your program is written in C, be aware that the “next line of source code” most likely corresponds to many assembly language instructions; using **step** will execute all of them.

If you only want to execute a single assembly language instruction, use the **stepi** (**si**) command. The equivalent in the graphical interface is **Control » Step Asm Inst** from the main menu or the  toolbar button. Be careful, however, if you are programming in C: the compiler might rearrange your source code lines when optimising, and this leads to rather unusual behaviour when using the **stepi** command!⁵


If you want to step more than one line or instruction in one go, simply specify that number as a parameter to the **step** or **stepi** command. For example, **s 10** will step through ten lines of your code (unless a breakpoint causes it to stop earlier). There is no equivalent in the graphical interface.

Another useful command is **until** (**u**), which executes your program until it reaches a specified location. You can specify that location as a parameter in the same way

⁴ In other words, make sure you use “==” for “equals to”, not the “=” variable assignment symbol!

⁵ See the file *optimise.c* in the *examples/intro* directory on your CD-ROM for an example of this in action.


that you do for the **break** command. In the graphical interface, using the right mouse button above a line number will display a context menu; simply choose the **Continue to Here** option.

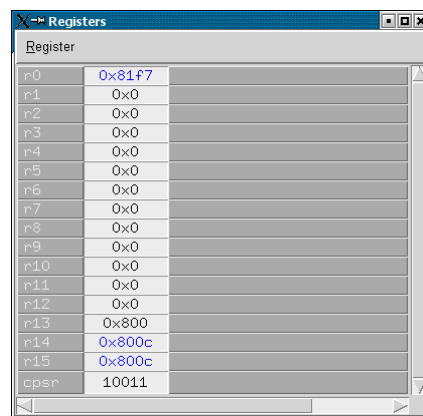
There is one other type of “stepping” command that you should know about: the **next** command (or, in the graphical interface, **Control » Next** from the main menu or the  toolbar button). This command is similar to **step**, except that it treats function calls as a single line. In other words, **step** will jump *into* functions, while **next** will jump *over* functions. This distinction is a little subtle and can cause no end of grief if you get it wrong! The best way to remember it is by practising using both.

As with the **step** command, the **next** command (or **n** for short) has an assembly language statement equivalent: **nexti (ni)** executes a single instruction, but treats function calls (the **bl** instruction) as a single instruction. It also has some rather unusual behaviour when used with backward branches, so be warned... A good rule of thumb is to always use **stepi** unless there is a **bl** instruction that you do not wish to trace through.

Examining the Registers

A debugger would be pretty useless if all you could do was stop and start your program. Its power lies in the fact that you can *see* a program’s state while it is stopped: in other words, you can actually see the program’s registers and variables.

Under the graphical interface, select **View » Registers** from the main menu (or the  toolbar button) to examine the ARM processor’s registers. This will bring up a window similar to the following:



Remember that, on the ARM microcontroller, register R15 is also known as PC, the Program Counter; register R14 is used as LR, the Link Register; and register R13 is almost always used as SP, the Stack Pointer.⁶

If you right-click on any register, you can change the way that that register is displayed. For example, you can choose to display the CPSR (Current Program Status Register) in binary, making it easier to interpret its contents. By the way, you will probably want to keep the *ARM Architecture Reference Manual* open to page A2-9 (page 41 of the PDF document) while interpreting this register; this document can be found on your CD-ROM in the *reference* directory.


⁶ If you are very keen, you could read the *ARM-Thumb Procedure Call Standard*, which specifies how the ARM registers should be used in a program. This document can be found on your CD-ROM in the *reference* directory. The GNU Debugger supports this standard; you can change the way registers are displayed by changing the setting of **Disassembly Flavour** in the **Preferences » Source** dialog box. The default setting is **Raw**.

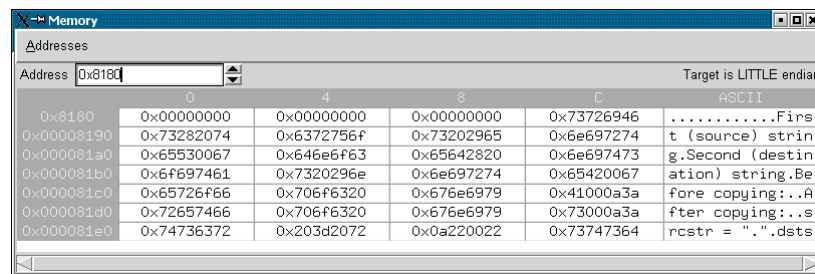
The equivalent command under the command-line interface is **info registers** (or **ir** for short), which displays all registers in hexadecimal and decimal. You can also use the **print** command, which is described later.

Examining Memory

The GNU Debugger gives you considerable flexibility in displaying the contents of memory. As usual, the graphical interface gives you ease of use with some restrictions; you will probably find yourself resorting to the command line once you realise the power available to you.

Using the Graphical Interface

Under the graphical interface, select **View » Memory** from the main menu (or the  toolbar button) to display a section of your program's memory. You should see a window similar to the following:



Simply modify the **Address** value to display that address; you can use ordinary numerical addresses (most likely expressed in hexadecimal, such as **0x8000**), register names with a leading “\$” (such as **\$r13**), or labels with a leading “&” (such as **&src**).⁷

You can change the way the Memory window displays its contents: select **Addresses » Preferences** to do so. Opening a new Memory window is somewhat tricky: select any memory location with the left mouse button (that cell will then be coloured grey), then use the right mouse button to select **Open New Window at n** (for some address *n*).

By default, the GNU Debugger automatically updates the memory display after each step in your program's execution. However, this can be very slow, so you should close any unnecessary Memory windows, or else disable the Auto-update feature (**Addresses » Auto Update** in the Memory window) until you need it.

Using the Command-line Interface

There are two main commands that display the contents of memory: **x** and **print**. These two commands are very powerful and only basic examples are given in this document; you should read Chapter 8 of the *GNU Debugger Reference* if you want a more in-depth treatment.

The **x** command examines an area of memory at a particular address and displays it in a particular format. This command has the following form:

x/nfu address

The first three parameters, *n*, *f* and *u*, are all optional (if you omit all of them, you do not need to specify the “/”, either). These parameters specify the repeat count *n*, the format *f* and the unit size *u*.

⁷ In actual fact, you can use any expression that is valid for the **x** command, described later; this gives you vast capabilities, most of which you will never exploit.

The most useful formats are **x** for hexadecimal, **d** for signed decimal, **u** for unsigned decimal, **t** for binary (mnemonic: **t** for “two”) and **i** for an ARM instruction. The most useful unit sizes are **b** for bytes, **h** for half-words (16-bit quantities) and **w** for words (32-bit quantities). The parameter *address* has the same format as that used in the Memory window in the graphical interface. All this sounds very complicated, but some examples should help:

x/4ub 0x08	Display four bytes as unsigned integers, starting from address 0x08. Pressing ENTER on an empty line after this command will display the <i>next</i> 4 bytes. ⁸
x/16xw \$r13	Display 16 hexadecimal words (32-bit quantities) starting from the address contained in register R13; in other words, display the top 16 words on the processor stack (since register R13 is the Stack Pointer register SP).
x/10i &_amp;start	Display 10 ARM instructions starting from the label <code>_start</code> . This is also known as disassembling your program.

The other major command to display memory is **print** (or **p** for short). This command operates at a much higher level than the **x** command: it understands things like full-blown C expressions, the types of variables and can even call functions in your program. It is also the command used to change the contents of variables and registers.

One major drawback of using the **print** command is that at times it requires a good understanding of C pointers. Pointers are a notoriously difficult subject for the beginner—an understanding of the difference between a variable, the *address* of that variable and a *pointer* to that variable does not come easily!

Examples are probably the best way to illustrate some of the power of the **print** command. Some of these examples use the files that can be found on your CD-ROM in the *examples/intro* directory.⁹ And do remember that **help print** (or **h p** for short) gives you a brief summary:

p 102 * (22 + 23) + 0x1234 - 9208	A simple calculator!
p (char) 0x4A	Print the hexadecimal number 0x4A as the character “J”. This is an example of a C type-cast, in this case to char.
p/x &_amp;start	Print the address of the <code>_start</code> label, in hexadecimal.
p/t \$cpsr	Print the value of the ARM processor register CPSR, in binary.
p/d (int) *0x10	Print the 32-bit integer (ie, a C type of <code>int</code>) stored at address 0x10 (since <code>*0x10</code> essentially tells the debugger to “treat 0x10 as an address and look at what is stored there”).

The next four examples use the program *wordcopy.elf*; note that the `src` array is defined in the `.data` section, *not* in the `.text` section:

p/x &src	Print the address of the <code>src</code> array as a hexadecimal number.
p src	Print the contents of the word stored at <code>src</code> . The value “1” is printed.

⁸ This is where the GNU Debugger’s DWIM (Do What I Mean) interface comes into play... Try it and see!

⁹ You would need to start the debugger on the appropriate file, connect to the simulator, download the program and run it to the `_start` label, to see what the examples do.

- p (int) src** The same as the above, except that a C-style type-cast to `int` is used.
- p (int[10]) src** Treat `src` as a pointer to an array of 10 `ints`, and print out that array. This is another example of a type-cast.

The next seven examples use the program *strcpy-c.elf*. You need to run the program at least to line 31 (the first `printf` statement) to get correct results:¹⁰

- p srcstr** Print the string pointed to by the `srcstr` variable.
- p dststr** Print the array of characters contained in the `dststr` variable.
- p dststr[2]** Print the character at index 2 in the `dststr` array (ie, the character “c”). Remember that arrays in C start at index 0.
- p (int *) srcstr** Treat `srcstr` as a pointer to `int` (a C type-cast) and print out its value (ie, the value stored in the variable `srcstr`, which is an address).
- p/x *((int *) srcstr)**
Print the value *pointed to* by `srcstr` (when it is treated as a pointer to `int`), in hexadecimal. You should see the value “0x73726946” (the first four bytes of what `srcstr` points to).
- p/x (int *) (*((int *) srcstr) - 1936877878)**
Print that value subtracted by 1936877878, and treat the result as a pointer to `int`. You should see the value “0x10”.
- p/x (int) *((int *) (*((int *) srcstr) - 1936877878))**
Finally, print the 32-bit integer that is stored at *that* address (ie, the address 0x10). All this is a rather esoteric way of printing the hexadecimal representation of the instruction at address 0x10!

The last four examples use the program *jumpbl.elf*:

- p dispatch(1, 218, 34)**
Call the `dispatch` function with three parameters and print the result. This can be done even if you are in the middle of debugging that function!¹¹
- p dispatch(&f_mul, 1234, 4321)**
Call the same function with different parameters. Note that `&f_mul` is used instead of `f_mul`: this is due to the fact that labels assigned via the `.set`, `.equ` or “=” assembler directives are treated as *addresses*, not integers.
- p/t do_add(-1, 26, 16)**
Print the result of the function call to `do_add` in binary. Can you figure out why the first parameter is -1? Hint: the first parameter is always passed in register R0.
- p do_add(-1, \$r0, \$r0)**
Call the function `do_add` with the current value of the register R0 and print the result. Note that, even though that function returns the result in R0, your real R0 (ie,

¹⁰ These examples rely on the fact that the GNU Debugger treats `srcstr` and `dststr` as pointers to characters (ie, as strings). The GNU C Compiler inserts appropriate directives into the output file to make sure that the GNU Debugger does this.

¹¹ The function must conform to the *ARM-Thumb Procedure Call Standard* for this to work. A simple summary of that Standard is that R0-R3 contain the parameters on entry, R0 must contain the result on exit, and all other registers (R4-R14) must remain unchanged.

the register R0 in the program being debugged) is not modified. Issue **p \$r0** to check this, if you like.

The best way to become proficient in using the **x** and **print** commands is by constant practice! And do remember the **help x** and **help print** commands...

Modifying the Registers

The GNU Debugger allows you not only to examine registers and memory, but to modify them as well. You do need to be careful, however: it is often very easy to crash your own programs when they are presented with unexpected values!

In the graphical interface, you change a register's value by double-clicking on that register in the Register window. An edit cursor will appear; simply enter the new value (using BACKSPACE to delete the old one) and press ENTER.

The command-line interface uses the **print (p)** command to modify the processor's registers. You simply specify the register name (with a leading "\$", of course), the C assignment operator "=" and the new value.

Some examples:

```
p $r0 = 0x4A4E5A00
Set register R0 to 0x4A4E5A00.

p $r3 = $r3 - 5
Set register R3 to its current value less 5.

p $cpsr = ($cpsr & 0xFF) | (0xE << 28)
Set the N, Z and C flags, and reset the V flag, in the
Current Processor Status Register. See page A2-9 of
the ARM Architecture Reference Manual (page 41 of the
PDF document) for more information about CPSR.
```

Modifying Memory

Modifying the contents of memory is as easy as modifying the processor's registers. Under the graphical interface, find the appropriate address in the Memory window, then double-click on the value you wish to modify. Simply change the value to whatever you want (using the BACKSPACE key as necessary) and press ENTER. Voila!

Under the command-line interface, you use the **print** command with the C assignment operator "=". Some examples (using the program *wordcopy.elf*):

```
p *((char *) 0x100)
Print the character (ie, a C type of char) stored at
address 0x100. You should always print the current
value before making modifications, just to check that
everything is OK!

p *((char *) 0x100) = 'J'
Store the character "J" at address 0x100.

p (((int *) &dst) + 2)
Print element number 2 of the array dst. Unfortu-
nately, you cannot specify p dst[2] (as you would if
you were debugging a program written in C) as your
program does not contain sufficient type information.12
```

¹² If you are keen enough, you can add the necessary debugging statements to your assembly language program. Write the equivalent program in C, then use **arm-elf-gcc -g -S** to compile it to assembly language code to see how the C compiler does it.

```
p *((int *) ((int *) &dst) + 2) = 7
```




Set element number 2 of the array `dst` to 7.

Chapter 11 of the *GNU Debugger Reference* has more information about modifying your program's state of execution, if you are interested. Not all of it, however, is relevant to debugging programs on embedded systems.

Example Files

As already mentioned, the *examples* directory and its subdirectories on your CD-ROM contain many examples of assembly language and C programs. You can use these example files to practise the debugging commands discussed in this document. And you are encouraged to do so, as "practice makes perfect"!

In particular, the *examples/intro* directory contains the following example files (amongst others); run the debugger on each (as explained in the section Invoking the Debugger) and follow the suggestions given:

- | | |
|---------------------|--|
| <i>pseudo.elf</i> | Compare the source code (in <i>pseudo.s</i>) and the disassembled code in each of the subroutines using the graphical interface (with the Source window set to SRC+ASM instead of just SOURCE). You can use commands like <code>x/16i &_start</code> under the command-line interface, if you wish. |
| <i>subr.elf</i> | Try stepping through this program using both the <code>stepi</code> and <code>nexti</code> commands (or the  and  toolbar buttons in the graphical interface). Note how <code>nexti</code> executes the whole of a function call (the <code>b1</code> instruction) as if it were a single step. |
| <i>optimise.elf</i> | Try stepping through this program using the  toolbar button under the graphical interface. Note how the C compiler rearranged the instructions and how the "current address" highlight jumps around from place to place. |
| <i>jumptbl.elf</i> | Try using the <code>print</code> command examples, as suggested in this document. Stepping through this program will also help you understand jump tables. |
| <i>wordcopy.elf</i> | Again, try the various <code>print</code> command examples. Try modifying the contents of the <code>src</code> array and rerunning the program. |
| <i>strcpy-c.elf</i> | Tracing through a compiled C program can be an interesting experience! Hint: if you do not want to wade through hundreds of lines of start-up code, use the <code>c</code> (<code>continue</code>) command to run without tracing until the program reaches <code>main</code> . You will also want to use the <code>next</code> or <code>nexti</code> commands to avoid tracing calls to <code>printf</code> or <code>puts</code> . ¹³ Notice how the GNU Debugger handles multiple source code files when you trace into the <code>strcpy</code> function. |

¹³ Can you figure out why the GNU C compiler replaced some `printf` function calls with calls to `puts`? It's all in the name of optimisation...

Summary

The GNU Debugger is a powerful source-level debugger, and learning how to use it can be quite complicated. You can use the following table as a “quick reference” to the most useful commands:

h	Give you help with commands.
q	Quit the debugger.
r	Run your program from the beginning.
b <i>location</i>	Set a breakpoint at <i>location</i> , which can be a label, a line number in the source code or *address (eg, *0x8000).
i b	Show information about breakpoints in your program.
d <i>num</i>	Delete breakpoint <i>num</i> .
en <i>num</i>	Enable breakpoint <i>num</i> .
dis <i>num</i>	Disable breakpoint <i>num</i> .
c	Continue running your program until it reaches a breakpoint or the end
u <i>location</i>	Run until the program reaches <i>location</i> (<i>location</i> has the same syntax as the b command).
s	Step through the next source line of code.
n	Step through to the next line, treating function calls as if they were a single line.
si	Step through the next assembly language instruction.
ni	As with si , but do not trace through functions or backward jumps.
i r	Display all ARM processor registers.
x/nfu <i>address</i>	Examine memory at <i>address</i> . The most useful formats <i>f</i> are x for hexadecimal, d for signed decimal, u for unsigned decimal, t for binary and i for ARM instructions. The most useful unit sizes <i>u</i> are b for bytes, h for half-words and w for words. Examples of <i>address</i> : 0x1234 , \$r0 , &dst .
p <i>expression</i>	Print the value of <i>expression</i> (which can be any complicated C expression that you like).
p/f <i>expression</i>	Print the value of <i>expression</i> using format <i>f</i> (see the x command for a list of these).

Happy debugging!