

Debugging uClinux on Coldfire

By David Braendler
davidb@emsea-systems.com

What is uClinux?

[uClinux](#) is a version of Linux for CPUs without virtual memory or an MMU (Memory Management Unit) and is typically targeted at deeply embedded systems with very little memory or permanent storage.

One of the two most fundamental changes to Linux in 2.6 comes through the acceptance and merging of much of the uClinux project into the mainstream kernel. This variant of Linux has already been a major driver of support for Linux in the embedded market, and its inclusion in the official release should encourage further development in this space.

Although uClinux is a true multitasking Linux systems, it is missing memory protection and other related features. (Without memory protection, it is possible for a wayward process to read the data of, or even crash, other processes on the system.) This may make them undesirable for a multi-user system, but an excellent (cheap!) choice for a dedicated devices.

There are several embedded processors supported by Linux 2.6. [Motorola](#) and [ARM](#) processors are probably the most widely supported of MMU-less processors at this time, although there are a host of other ones which are supported including Hitachi's [H8/300](#) series, the [NEC v850](#) processor, Analog Devices [Blackfin](#). uClinux has even started appearing on 'soft' processor cores on FPGAs – see the following for details on Xilinx's [Microblaze](#) and Altera's [Nios](#) processor .

How do you get started??

Probably the best way to get started is with emulation software. David McCullough (one of the maintainers of uClinux) has written a great article on this entitled [Getting Started with uClinux](#).

What about real hardware??

Eventually your going to want to try real hardware. There are a number of nice development boards out there. No bias (of course) but check out the development board on offer from [emsea-systems](#). There are also a lot of boards on offer from other vendors. The [uCdot](#) website has a list of development boards, or ask on the [uClinux mailing list](#) – there are a lot of very helpful people who will be more than happy to

answer your questions. Chances are if you are using a specific MMU less processor it is supported under uClinux!

How do you debug on the Coldfire??

The whole point of this talk was to describe how to debug uClinux *on the Coldfire* using the BDM port. once it was up and running. So here us what you need.

- A board running uClinux on a coldfire.
- Parallel port cable.
- [BDM Debug Board](#)

You then need to download the uClinux kernel, and compile it. At this point I'll pretty much just be reproducing the steps described by David McCulloch in his article on [Using m68k-bdm-elf-gdb with uClinux/Coldfire](#). This article is reproduced below (with a few minor modifications)

Using m68k-bdm-elf-gdb with uClinux/Coldfire

By David McCullough
davidm@snapgear.com

Compiling the tree

The first thing to do if you want to get serious about debugging with GDB is to turn on the kernel/user options for symbolic debug. Just run:

- make config, or make xconfig
- be sure to enable "Customize kernel Settings"
- be sure to enable "Customize Vendor/User Settings"
- In the kernel settings, turn on "Full Symbolic/Source debugging support" under "Kernel hacking".
- In the Vendor/User settings, turn on "build debugable libraries" and "build debugable applications" under the "Debug Builds" menu.

- make clean; make dep; make

This builds all the source in the tree with full symbolic debug support and stack frames. You need the stack frames so that local variables and stack backtraces work.

Debugging the Kernel

To debug the kernel with gdb you will need a BDM cable, and a unix machine with the BDM driver installed. Once you have these you are ready to run gdb.

Firstly you need to boot the kernel you are going to debug. This may mean updating the flash before doing the following.

```
cd linux-2.0.x or linux-2.4.x
m68k-bdm-elf-gdb ./linux
gdb> target bdm /dev/bdmcfd
gdb> cont
```

If you are using a dev board just follow the steps your would normally use to get the kernel running on the board. If you are running from flash then the kernel will have started.

At this point you have kicked the kernel into life and it will boot. This is all fine and good if you can boot the kernel, you can now break into the debugger and add some break points to debug your code. If, however, you are crashing early in the boot, you will need to do some extra things before you can debug. The reason for this is that you cannot set break points until the kernel has been loaded into RAM.

The best way to break into the debugger early, before any runs, but after the code is loaded into ram is to put a "halt" instruction into the code at the appropriate point. A good spot for this is is just before we start the kernel. For example in the file:

```
linux-2.X.Y/arch/m68knommu/platform/5307/NETtel/crt0_ram.S
Add the line marked with '***' to stop the CPU and drop into the debugger just before
the 'C' code starts up.
```

```
/*
 * Assembler start up done, start code proper.
 */
***    halt
        jsr start_kernel          /* Start Linux kernel */

_exit:
        jmp _exit                /* Should never get here */
```

When the code hits this point, gdb will drop back to the prompt and you can set breakpoints or continue with whatever debugging you need. Once finished type "cont"

to continue running and hit your breakpoints. **NOTE** once you put the "halt" in you will have to use the debugger to start the image, otherwise it will just halt and do nothing :-)

Debugging an Application

Debugging applications is fairly simple, you will need "gdbserver" in the romfs (or accessible via NFS) and a network.

To network debug an application you just need to start the application using gdbserver, instead of running it as you usually would. For example, on your uClinux target system run:

```
/> gdbserver :3000 /bin/stty
```

This will load the application and stop it at the first instruction. You can then connect to it from your host system by running:

```
m68k-bdm-elf-gdb user/stty/stty.gdb
gdb> target remote 10.0.0.182:3000
```

Notice that on the host side you use the stty.gdb file. This file has the correct format for gdb to relocate it to match the binary that is already started in the target system. I usually set a break point on main and continue to there to get some 'C' code to look at.

```
gdb> b main
Breakpoint 1 main .....
gdb> cont
Hit breakpoint 1 ...
gdb>
```

To debug an application using the BDM debugger is a little special. Check the [Debugging Kernel and Applications together](#) and [Notes](#) sections for more information on what can/can't be done.

Debugging Kernel and Applications together

If your system is crashing, and you are not sure if it is kernel or application, then you can setup to debug both at the same time. First recompile the whole tree with the symbolic debugging turned on as described above, then follow these steps.

- Start the debugger as given above in [Debugging the Kernel](#).
- Let the kernel run (cont), then reproduce the crash/hang.

- If the target system crashes it usually hangs. Go to your debugger and break into it with ^C (or your interrupt key). This will get you a debugger prompt.
- Run "bt" to get a backtrace of where the code was when it died. If this gives you a list of function names then there is a good chance that you crashed in the kernel. Use this backtrace to begin finding the problem.
- If the backtrace only contains numbers, then you probably died in an application. First lets find out which process is running. Do this by printing the name of the current process with:

| Kernel Version | Command |
|----------------|------------------------|
| 2.0.x | p current_set[0]->comm |
| 2.4.x | p _current_task->comm |

- Lets say that this returned "sh". Then we know that process that crashed was the "sash" shell (if that is the shell you are using ;-).
- The next step is to load the symbols for that process so we can find out where it died. For "sh" we would use:
- ```
add-symbol-file user/sash/sh.gdb current_set[0]->mm-
>start_code
```

If the above command asks you if you want to load the symbols at .text-addr = 0, answer no. You will have to find out the start address and type it in manually -

```
(gdb) p _current_task->mm->start_code
1234566
(gdb) add-symbol-file ../user/sash/sh.gdb 1234566
```

This will automatically load the applications symbols at the correct location.

- Re-run "bt" to get a new backtrace with symbols. Make sure it looks ok. If you have loaded the symbols for the wrong application it will look fairly obvious. You should now have a good idea what the app was doing when it died and can start tracking it from there.

There is all kinds of good information in the current\_set structure to find out things about the current process. Try some of these to learn more:

| Kernel Version | Command |
|----------------|---------|
|----------------|---------|

|       |                                                       |
|-------|-------------------------------------------------------|
| 2.0.x | <pre>p *current_set[0] p *current_set[0]-&gt;mm</pre> |
| 2.4.x | <pre>p *_current_task p *_current_task-&gt;mm</pre>   |

---

## Cool GDB commands to remember

### **p** or **print**

Print the value of something (or some expression). This is about the most used command. You can print any 'C' expression and it formats the output to suit that types of the fields. Some examples:

```
gdb> p 1 << 10
1024
gdb> p /x 1 << 10
0x400
gdb> p my_array[0].name
my_array[0].name = 0xNNNNNNNNNN "my name is fred"
gdb> p *my_name[0].name
*my_array[0].name = 0xNNNNNNNNNN 'm'
```

So you can see that the print command can tell you just about anything about anything. You can even cast values to help save time finding the offsets of fields.

```
gdb> p * (struct stat *) 0x10000000
....
```

Remember you can change the format it prints with the '/' options, for example 'p /x' prints the values all in hex.

### **x**

This is similar to 'hexdump' and is good for general memory dumping.

### **bt**

Display a back trace (each function call gives you a frame number)

### **frame *N***

This allows you to select a frame from the back trace. You may then look at the local variables for that frame to see their values or set breakpoints in that function.

### **list [*line number* | *function name*]**

list some source code. Good for looking at the source around the current line (ie list), checking other functions (ie., list main) or listing source from a line number in the current file (ie., list 20).

### **b (function name | line number)**

set a break point on a function or a line number

### **step [*count*] or *s***

step one line of 'C' code, will step into function calls.

**next [count] or n**

step one line of 'C' code, do not step into functions.

**stepi [count]**

step one assembler instruction (best used with "display /i \$pc").

**display**

Just like the "print" command, it allows you to have the value of an expression displayed every time you do something. As above, "display /i \$pc" display the current assembler instruction, every time you stepi, the next instruction will be displayed.

**delete [number]**

Delete all breakpoints (or breakpoint number)

**disable [number]**

Disable all (or numbered) breakpoints.

**enable [number]**

Enable all (or numbered) breakpoints.

**info breakpoints**

List all your breakpoints and their status.

**info reg**

print the HW registers and their values. If this fails with a no-frame error then try running "frame 0" first, then doing it again.

**sym filename**

If you have loaded a new program or kernel then get the new symbols by running this command.

**NOTE:** make sure you remove or disable all break points before running this command or your system will crash.

**set**

Set allows you to change the value of variables or memory. For example, to set the program counter to address 0x400:

```
set $pc = 0x400
```

To set the variable "i" to some new value (it can be quite complex):

```
set i = 10 * 1024 + j * 2
```

**help**

I learn't everything I know about GDB through this command, you can use it too :-)

Don't forget the man page and the info pages as well.

---

## Notes

The latest m68k-bdm-elf-gdb versions have hardware watchpoints. This is invaluable as you can break on a value being changed without affecting the speed of execution. If you know that a single 32 bit value is changed by someone at address 0x123456, but do not know who, try the following:

```
(gdb) watch * (int *) 0x123456
(gdb) cont
```

Once the debugger breaks, use the Debugging Kernel and Applications together method for finding out who it was, and better still, exactly which line it happened on ;-).

The only catch with HW watch points is that you only have one. After that you get SW watch points which are impossibly slow (step one instruction, check value, repeat).

You cannot set breakpoints in user programs from the BDM debugger. This is because it inserts "halt" instructions and these can only be executed by kernel code. You can set HW breakpoints in user code though using "hb". You can also "stepi" in user code. Both "step" and "next" should be avoided when running in user mode. It is possible to modify the BDM driver to allow the debugging of user applications via the BDM interface. You can see the original email on it [here](#). Here is what you need to do:

- This will only work if your app runs in RAM vs read only memory (ROM/Flash).
- You need to rebuild the bdm driver for linux after modifying the bdm.c file that comes with Cybertec's latest linux BDM source archive, gdb-bdm-20020210.tar.gz.

You can download it from [here](#).

- Before following the excellent instructions provided in the archive you need to mod the drivers/bdm.c file as follows. These mods were based on an old post by Mark Abbate:

- In cf\_pe\_read\_sysreg:
  - //self->cf\_csr = ioc->value = (self->cf\_csr & 0x0f000000) | ioc->value;
  - self->cf\_csr = ioc->value = (self->cf\_csr & 0x0f000400) | ioc->value;
  -
- In cf\_pe\_run\_chip:
  - // csr\_ioc.value = 0x00000000;
  - csr\_ioc.value = 0x00000400;
  -

- In `cf_pe_step_chip`:
- `// csr_ioc.value = 0x00000030;`
- `csr_ioc.value = 0x00000430;`
- 
- and
- 
- `// self->cf_csr = 0x01000000;`
- `self->cf_csr = 0x01000400;`

Sometimes when your system locks up and you break into the debugger with ^C the "bt" command may give complete garbage. I have found that a "stepi" may pull you back to reality. It's worth a try at least ;-)