

Reference: http://processors.wiki.ti.com/index.php/Stellaris_Launchpad_with_OpenOCD_and_Linux

Motivation

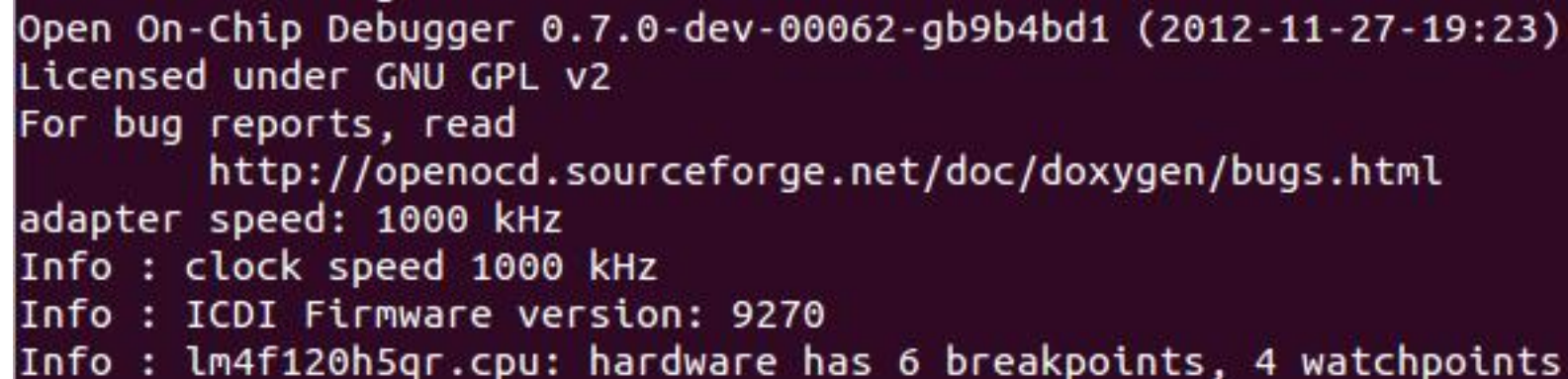
After installing the EABI toolchain for ARM development on Linux Mint (as VM on my Mac), I wanted to get the gcc compiler, gdb debugger, with OpenOCD. This brief set of notes provides a few examples on this and covers some extensions using an old favourite (of mine) a graphical debugger, **ddd** <see <https://www.gnu.org/software/ddd/>>

Run OpenOCD

Once this build of OpenOCD has completed installation, it is time to run OpenOCD. The main difference between this and installing the stable version with apt-get is that the files will be placed under /usr/local instead of /usr. After installing the toolchain, connect your target board, typically using USB, and start OpenOCD. Note that my demo uses a Tiva, so the OpenOCD configuration file is based on MCU on this board. A SAM4S will use a different cfg file:

```
$ sudo openocd --file /usr/share/openocd/scripts/board/ek-lm4f120xl.cfg
```

You should get a result that looks something like the following to indicate that OpenOCD has successfully connected:



```
Open On-Chip Debugger 0.7.0-dev-00062-gb9b4bd1 (2012-11-27-19:23)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
adapter speed: 1000 kHz
Info : clock speed 1000 kHz
Info : ICDI Firmware version: 9270
Info : lm4f120h5qr.cpu: hardware has 6 breakpoints, 4 watchpoints
```

This terminal window will need to stay open but you will no longer be entering commands in it. From here, open up another terminal window for your debugger, gdb.

For this tutorial, we will use the `project0` example (see the URL at the top of this page), which is essentially a **CCS tutorial** but using Linux and open source tools from the command line.

This means changing into the project directory (not `/gcc` just yet), and execute a “make clean” and then “make”. You should see the following files, where “`gcc/project0.axf`” is the one you’re going to use next.

```
sweddell@sweddell-Parallels-Virtual-Platform ~/Documents/Courses/ENCE464_2017/arm_src/tivaware2/boards/ek-lm4f120xl/project0 $ make
CC      project0.c
CC      startup_gcc.c
LD      gcc/project0.axf
```

Next, change into the `/gcc` directory. To get a debugging window we need to start gdb with the compiled project, *project0*, and a successful compilation will produce a binary file, *project0.axf*. To ensure that gdb communicates with your target board, and not simply your PC in simulation mode, enter the following commands in your terminal window. The first command will get gdb to run on your PC, and second and ongoing (gdb) commands will then communicate commands to OpenOCD to reset your target MCU and load your binary file, *project0.axf*:

```
$ arm-none-eabi-gdb project0.axf
(gdb) target extended-remote :3333
(gdb) monitor reset halt
(gdb) load
(gdb) monitor reset init
```

At this point you should have loaded the binary file, *main.axf*, but the chip is currently halted in startup routine. We can then set up a breakpoint at “main” and run the program to get to “main” simply by executing “continue”, as shown in the following gdb commands:

```
(gdb) break main
```

```
(gdb) continue
```

You should now be at the “main” routine. Running another “continue” will run the program.

Once you see that the LED is blinking, you can stop the program with:

```
(gdb) kill or Ctrl-C
```

Once openOCD is running, you can also connect to a local host by opening another terminal window and typing in: `$telnet localhost 4444`

Summary

Overall, the whole development process was quite straightforward. It was quite easy to use OpenOCD. I’d installed the Flash utility but I couldn’t get this to work. The next thing I want to try was DDD, the redhat unix graphical debugger. It works by laying over the top, and interacting with, gdb commands. I’ve had some experience with DDD when I was using the SAM7 a few years ago and I really liked its versatility. In the next section I explain how you can add a graphical debugger, like DDD, to the mix of tools.

Adding a Graphical Debugger - DDD

Run OpenOCD as outlined earlier but don’t run the gdb debugger just yet.

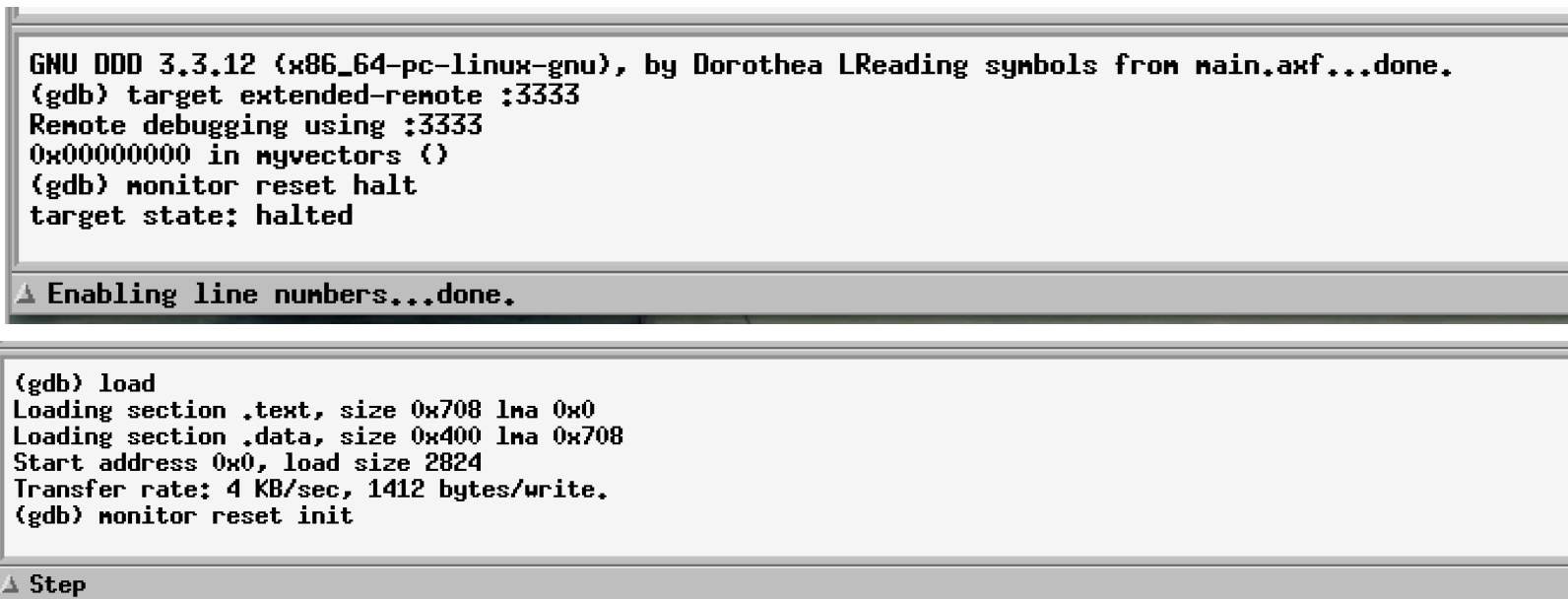
If not installed, install DDD. The following website provides details on this under “How to Install”:

http://shukra.cedt.iisc.ernet.in/edwiki/EmSys:Debugging_with_DDD

Next, before invoking gdb in a separate window, as outlined on Page 2, open a separate terminal window and type the command (note: 2 dashes before “debugger”):

```
ddd --debugger arm-none-eabi-gdb main.axf
```

In other words, you need to run the arm-none-eabi-gdb debugger **as a part of** ddd. Then, you should see (gdb) appear in the debug window of your ddd gui tool. To allow gdb to connect and reset the target board, go through the same initialisation procedure as outlined on Page 2 of this guide, but in the ddd gdb window. Make sure you have symbol information after building your program. If you don't, gdb will tell you on the first line. If you have symbols, enter the following:



The image shows two screenshots of the GNU DDD GUI. The top screenshot shows the main window with the text: GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea L. The bottom screenshot shows the gdb window with the text: (gdb) target extended-remote :3333, Remote debugging using :3333, 0x00000000 in myvectors (), (gdb) monitor reset halt, target state: halted. Below the gdb window is a menu bar with the text: Enabling line numbers...done. The bottom screenshot shows the gdb window with the text: (gdb) load, Loading section .text, size 0x708 lma 0x0, Loading section .data, size 0x400 lma 0x708, Start address 0x0, load size 2824, Transfer rate: 4 KB/sec, 1412 bytes/write, (gdb) monitor reset init. Below the gdb window is a menu bar with the text: Step.

```
GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea L
(gdb) target extended-remote :3333
Remote debugging using :3333
0x00000000 in myvectors ()
(gdb) monitor reset halt
target state: halted

▲ Enabling line numbers...done.
```

```
(gdb) load
Loading section .text, size 0x708 lma 0x0
Loading section .data, size 0x400 lma 0x708
Start address 0x0, load size 2824
Transfer rate: 4 KB/sec, 1412 bytes/write.
(gdb) monitor reset init

▲ Step
```

A couple of examples, showing basic (left) and floating (right) windows and menus are shown on the following page.

DDD: /home/sweddell/Documents/Courses/ENCE464_2017/arm_src/tiware2/boards/ek-lm4f120xl/project0/project0.c

```

81  {
82      //
83      // Turn on the LED
84      //
85      GPIOWrite(GPIO_PORTF_BASE, RED_LED|BLUE_LED|GREEN_LED, RED_LED);
86      //
87      // Delay for a bit
88      //
89      SysCtlDelay(1000000);
90      //
91      // Turn on the LED
92      //
93      // Delay for a bit
94      //
95      GPIOWrite(GPIO_PORTF_BASE, RED_LED|BLUE_LED|GREEN_LED, BLUE_LED);
96      //
97      // Delay for a bit
98      //
99      //
100     SysCtlDelay(2000000);

```

```

0x0000027c <main+16>:      movs    r1, #14
0x0000027e <main+18>:      bl      0x40e <GPIOWrite>
0x00000282 <main+22>:      movs    r1, #14
0x00000284 <main+24>:      movs    r2, #2
0x00000286 <main+26>:      ldr     r0, [pc, #36] ; (0x2ac <main+64>)
0x00000288 <main+28>:      bl      0x408 <GPIOWrite>
0x0000028c <main+32>:      ldr     r0, [pc, #32] ; (0x2b0 <main+68>)
0x0000028e <main+34>:      bl      0x488 <SysCtlDelay>

```

```

(gdb) step
(gdb) step
(gdb) break project0.c:95
Breakpoint 2 at 0x292: file project0.c, line 95.
(gdb)

```

DDD: /home/sweddell/Documents/Courses/ENCE464_2017/arm_src/tiware-template/LM4F_startup.c

```

243  /*
244  void rst_handler(void){
245      // Copy the .data section pointers to ram from flash.
246      // Look at LD manual (Optional Section Attributes).
247      //
248      // source and destination pointers
249      unsigned long *src;
250      unsigned long *dest;
251      //this should be good!
252      src = &_end_text;
253      dest = &_start_data;
254      //this too
255      while(dest < &_end_data)
256      {
257          *dest++ = *src++;
258      }
259      // now set the .bss segment to 0!
260      dest = &_start_bss;
261      while(dest < &_end_bss){
262          *dest++ = 0;
263      }
264      // after setting copying .data to ram
265      // to start the main() method!
266      // There you go!
267      main();
268  }

```

```

0x000002c8 <+24>:      str     r3, [r7, #0]
0x000002ca <+26>:      b.n     0x2dc <rst_handler+44>
0x000002cc <+28>:      ldr     r3, [r7, #0]

```

```

stepi ignored. GDB will now fetch the register state from the target.
Program received signal SIGINT, Interrupt.
rst_handler () at LM4F_startup.c:245
(gdb) step
(gdb) step

```

After which, you now have control of gdb through a graphical version of gdb, i.e, ddd. Here are a few screen shots... The figure above shows how the ddd gui can be used to debug a program. For example, setting a breakpoint in a GPIO Pin Write function is shown using a stop sign in the gdb source window. Either the console window or pull-down

menus will allow you to set a breakpoint. From there, running the code using “continue” will determine if the function does indeed get executed, periodically. The other nice thing about a gui is that you see the updates via the debugger, and more importantly, several screens are updated at the same time. Since stepping is essentially the same as selecting “s” or “Step” in the gdb command window, you can also simply use the Commands pull-down menu as shown on the next page. Note, as mentioned earlier, it is always a good idea to place a breakpoint on the `main` function symbol after you do a load. This is so you can recover to this point if the program counter gets out of synch.

DDD is nothing fancy, but provides basic functionality that CCS supports. In short, some developers like using a GUI and gdb provides users with the flexibility of employing a GUI. The ddd GUI however, is a little slow. gdb commands executed on the command line are a lot faster. It may help to keep a gdb command guide handy!

Note that when using gdb commands in the ddd debugger console window, there is no need to precede commands with “monitor”. For example, (gdb) `monitor break main`, should be simply (gdb) `break main` in ddd.

Appendix A : Debugging with the SAM7S

Using OpenOCD with Olimex ARM-USB-OCD JTAG debugger on the SAM7s, took some time to run. This was due to two existing scripts that had to be altered. After compiling a few old application programs, like *led-blink-pj*, and making sure that I had “-g3” in the compiler flags for debugging information, I completed a make clean and then make.

The default directories where openocd configuration files are stored on Linux Mint are: **usr\share\openocd\scripts**. This is different from the Ubuntu Linux configuration.

Next, the syntax needed to run OpenOCD with two, slightly modified scripts, was:

```
openocd -f openocd.cfg -f interface/ftdi/olimex-arm-usb-ocd.cfg -f target/at91sam7x256.cfg
```

The “openocd.cfg” configuration file simply establishes a telnet connection with the target and host...

```
telnet_port 4444
gdb_port 3333
gdb_memory_map disable
adapter_khz 3000
```

The second configuration file, “arm-usb-ocd.cfg” is the same as the olimex-arm-usb-ocd.cfg file that contained in the /scripts/interface directory of openOCD.

```
interface ft2232
ft2232_device_desc "Olimex OpenOCD JTAG"
```

```
ft2232_layout "olimex-jtag"  
ft2232_vid_pid 0x15BA 0x0003
```

Next, run openOCD as described earlier. Open another terminal and make the nightrider source by just typing “make”. Then, enter the following:

OpenOCD to reset your MCU and rather than use the file on Page-2, load your binary file, *main.elf*, using:

```
$ arm-none-eabi-gdb main.elf
```

note that you can also use the DDD debugger, but typing:

```
$ ddd --debugger arm-none-eabi-gdb main.elf
```

```
(gdb) target extended-remote :3333
```

When you want to communicate with the monitor, i.e, openocd, precede commands with “monitor”:

```
(gdb) monitor sleep 500  
(gdb) monitor poll  
(gdb) monitor soft_reset_halt  
(gdb) monitor arm7_9 fast_memory_access enable
```

When you want to communicate with the debugger, just use the debugger commands:

```
(gdb) load  
(gdb) break main  
(gdb) continue
```


References

- [1] Microchip development tool discussions, “debugging problems using arm-usb-ocd, eclipse on the at9`sam...”, accessed 1 July 2019, <http://www.at91.com/viewtopic.php?t=5185>
- [2] Debugging with DDD, accessed on 1 July 2019, https://www.gnu.org/software/ddd/manual/html_mono/ddd.html
- [3] ddd, Linux Mint, accessed on 1 July 2019, <https://community.linuxmint.com/software/view/ddd>
- [4] A Brief Introduction to DDD, accessed on 1 July 2019, <http://knuth.luther.edu/~leekent/tutorials/ddd.html>
- [5] OpenOCD - Open On-Chip Debugger, accessed 1 July 2019, <https://sourceforge.net/p/openocd/mailman/openocd-user/>
- [6] OpenOCD as a Flash Programmer, Olimex tutorial, Accessed 1 July 2019, https://www.olimex.com/Products/ARM/JTAG/resources/Manual_PROGRAMMER.pdf
- [7] Olimex discussion forum, “SAM7-P256 with ARM-USB-TINY Problem”, which contains useful programming information and scripts, Accessed 1 July 2019, <https://www.olimex.com/forum/index.php?topic=33.0>
- [8] Olimex SAM7-H256 ARM7TDMI-S Microcontroller (used in ECE courses between 2003 to 2007), Accessed 1 July 2019, <https://www.olimex.com/Products/ARM/Atmel/SAM7-H256/>

[9] “Insight debugger” (an alternative to ddd), supported on Ubuntu, Accessed 1 July 2019, <http://www.dalfonso.co/2016/04/23/setting-insight-debugger-on-ubuntu-16-04-lts/>

[10] Micromint USA, “Debugging ARM Embedded Applications using GNU Tools”, accessed 1 July 2019, http://www.micromint.com/component/docman/doc_view/42-debugging-arm-embedded-applications-using-gnu-tools.html?Itemid=144

[11] AT91SAM7S Quick Start Guide for Beginners, 6 March 2013, Adam Goodwin (ECE postgrad, 2013), http://ecewiki.elec.canterbury.ac.nz/mediawiki/images/b/b6/AT91SAM7S_Quick_Start_Guide_for_Beginners.pdf

[12] Olimex, ARM-USB-OCD User’s Manual, accessed 1 July 2019, http://ecewiki.elec.canterbury.ac.nz/mediawiki/images/b/b6/AT91SAM7S_Quick_Start_Guide_for_Beginners.pdf