

## Introduction

This lab will introduce you to JTAG hardware debugging of the AT91SAM7S device. Refer to your lecture notes for the principles of JTAG and how it applies to the AT91SAM7S development boards.

## Debugging Setup

1. You can install OpenOCD for Ubuntu through the Synaptic package manager (just search for “openocd”). NOTE: You will have to enable the “Community-maintained Open Source software (universe)” option in the Settings→Repositories dialog in Synaptic, as this is not an official Canonical-supported package.

If you are not using Ubuntu, you can download a pre-built binary of OpenOCD (Version 1.0) from the AT91SAM7S Wiki (built under Slackware 12.0), or you can build it from its source code. If you download the pre-built binary (recommended), it will be installed in `/usr/local/bin`. You may want to check its permissions and make sure everyone is allowed to read and execute it:

```
ls -al /usr/local/bin/openocd
chmod a+rx /usr/local/bin/openocd
```

To build from source, follow the installation instructions on the Wiki in the “Debugging” section.

2. OpenOCD’s behavior is entirely driven by a configuration file. This configuration file specifies the JTAG converter being used, the type of target hardware (ARM7, XScale, etc.), whether the target hardware does or does not have a separate reset line for the JTAG interface, etc.

The installation of OpenOCD for Ubuntu installs several sample configuration files in the `/usr/lib/openocd/interface` directory. For example, `olimex-arm-usb-ocd.cfg` file is appropriate for using the Olimex ARM-USB-OCD JTAG converter, and `parport.cfg` is appropriate for parallel-port-based converters.

These configuration files are very bare-bones as they do not make assumptions about the target hardware. More complete configuration files are available from the course Wiki (in the “Debugging” section).

3. Obtain a protocol converter from your lab instructor. Connect the JTAG protocol converter to the 20-pin JTAG connector on your evaluation board. The connector is polarized and will only connect in one orientation. Connect the other end to the PC (either parallel port using the extender cable, or plug it in to the USB port for the USB version).

The pinout for this 20-pin connector (if you are curious) is shown in the AT91SAM7S64-EK board schematic, which is included in the user’s manual for the board (available on the course Wiki). It is a de facto ARM standard pinout.

4. Power up the target board.
5. Become root and start the OpenOCD server using the `-f` flag to specify the configuration file:

```
openocd -f openocd_pp.cfg # For parallel port programmer
```

```
openocd -f openocd_usb.cfg # For USB programmer
```

Make sure there are no (serious) error messages. In the future, you do not have to be root if you properly set the permissions on the parallel port device or USB device.

## Kicking the Tires

The OpenOCD server is now connected to the JTAG interface on your target board and is waiting for connections to be made to it. The target board is running normally and should respond to interactions over the USB port. Type `'Sam_I_Am open , version'` to make sure.

To test out the “back door” JTAG port, open up another shell window and connect to the OpenOCD server:

```
telnet localhost 4444
```

Back in the first window (the one in which you started OpenOCD) you should see a log message indicating that a Telnet connection was accepted. Back in the Telnet window, let's send some commands: type `'poll'` to see the target state (running or halted). The program (SAM-BA) should be running.

Let's try halting the processor: type `'halt'`. You should see a status report that the processor has been halted at an address near 0x200000 (remember that SAM-BA copies itself from ROM to the lowest 8k of RAM at 0x200000 and runs there).

Let's look at the chip's registers: type `'armv4_5 reg'`. There they are...all of the banked registers.

Let's let SAM-BA go back to work: type `'resume'`.

To close the telnet session, press `Ctrl-]` then type `'quit'`. To make sure SAM-BA is truly running, use `Sam_I_Am` to query its version: `'Sam_I_Am open , version'`.

## Hardware Debugging With GDB

These low-level interactions with the OpenOCD server are OK for simple things like stopping the processor and inspecting registers, but we want to do more interesting things like setting breakpoints and single-stepping through a program. For this, we need GDB.

We also need a program to debug. Go back to your non-interrupt, RAM-based 'blink' program from a long time ago. Recompile the program but make sure that you specify the `'-g'` flag to the compiler. All of the tools in the toolchain (compiler, assembler, linker) recognize the `'-g'` flag as a request to include debugging information in the output of the tool to support debugging with GDB.

You should have an ELF file compiled with the `'-g'` flag. Convert this to HEX format and upload it into the evaluation board's RAM by using `Sam_I_Am`: `'Sam_I_Am open , send blinky.hex'`. Note the start address.

1. Start up the GDB debugger (we'll do it command-line for now) with the ELF file name as the parameter:

```
arm-elf-gdb blinky.elf
```

2. Connect to the OpenOCD server on your computer:

```
target remote localhost:3333
```

Note that you can replace `'localhost'` with any computer name or IP address indicating where the OpenOCD server is running. You could, in theory, have OpenOCD running on a computer in the lab, attached to hardware in the lab, and do all of your debugging from a Starbucks in Shanghai.

Connecting to the server automatically halts any running program so we can debug it.

3. You can send commands to the OpenOCD server using GDB's `'monitor'` command. These are the same commands we used while “Kicking The Tires” above. See the OpenOCD web page for a description of available commands. There aren't many commands you should have to send directly to the OpenOCD server. Some commands you may want to be aware of, however, are:

- `monitor poll`

See if the program is running or is halted.

- `monitor armv4_5 reg`

This command displays all of the ARM registers, including the banked registers from all the different operating modes.

- `monitor arm7_9 sw_bkpts [enable|disable]`

By enabling software breakpoints with this command you are reserving one of the EmbeddedICE watch-point units for this function. Remember that you can only set software breakpoints in RAM code.

- `monitor arm7_9 force_hw_bkpts [enable|disable]`

This command forces GDB to always set hardware breakpoints. You shouldn't have to do this as long as you can remember that GDB has an `'hbreak'` command which sets a hardware breakpoint (and `'thbreak'` to set temporary hardware breakpoints – these are breakpoints that are removed once they are hit).

Hardware breakpoints are the only option for FLASH code, and there are only 2 available hardware breakpoints, so use them wisely (`'thbreak'` is helpful here).

You can also send and receive binary files to RAM, as well as program FLASH memory (in theory) thus you may not need `Sam_I_Am` for these purposes. These functions, however, have not been tested and you are on your own.

4. Let's try running the program. Before doing so, we need to give GDB some method of regaining control over the debugging session else the program will return to SAM-BA and GDB will “hang” (not really, but it gets messy).

We'll run the program as usual by giving a `'go'` command from SAM-BA. Before doing so, let's set a hardware breakpoint at `_start`:

```
(gdb) hbreak _start
```

Now, let's let the program (still SAM-BA) continue so that we can send it the `'go'` command:

```
(gdb) cont
```

5. At this point, GDB is waiting for the program to stop at a breakpoint, SAM-BA is running, and all is well. Open up a third shell window and run the `'blinky'` program: `'Sam_I_Am open , go 202020'` (replace 202020 with the actual start address of your program).

6. Back in the GDB window, you should see that GDB stopped on the breakpoint that you set. The ARM core is now stopped and we can inspect registers, single-step, and so on.

Let's try single-stepping: type `'step'` in the GDB window. Press ENTER to repeat the `'step'` command several times. Cool, huh? We are actually single-stepping through real code on the real hardware, not in a simulator.

Keep single-stepping until GDB hangs. Where will it hang? In the long delay loop in between LED blinks. Why did it hang? Because OpenOCD keeps stopping the core, checking to see if the program hit a breakpoint, restarting the core, stopping the core, checking to see if the program hit a breakpoint, etc. Clearly, this isn't real-time, and you should keep this in mind – running your program in a hardware debugger is not necessarily the same as running it at full speed in real time!

7. Let's try to regain control of GDB by pressing Ctrl-c. You may have to press it more than once.
8. How about single-stepping through assembly-language instructions? Type `'stepi'` in the GDB window and press ENTER several times to single-step one assembly instruction at a time. Type `'disassemble'` to see the instructions surrounding the current program counter.
9. Type `'disassemble'` to list the assembly code of your function, and continue displaying code until you see the end of your function (mine ends at 0x2020FC). Note this address and set another hardware breakpoint at it:

```
(gdb) hbreak *0x2020FC
```

10. Resume the program and make sure the LED blinks:

```
(gdb) cont
```

11. To give control back to SAM-BA, let your program return to it:

```
(gdb) cont
```

Now, GDB is “hung” (it's waiting for a breakpoint to be hit) and we should be able to interact with SAM-BA in our third window. Try sending the `"Sam_I_Am open , version"` command to make sure SAM-BA is still responsive.

## Watchpoints

Recall that a watchpoint is like a breakpoint on a memory access, rather than a breakpoint on an instruction. One of the two hardware comparators in the EmbeddedICE module is used to watch the internal ARM address bus and halt the processor when a particular memory location is read or written. Using watchpoints to debug mysterious memory corruption problems can be much more effective than trying to set breakpoints all over the place.

1. Type in the following program as 'lab8.c'. This is a model of a “complicated” program that can modify a memory location in both obvious and less-than-obvious ways. If the global variable 'val' is ever found to have a surprising value, it would be difficult to track down which function was responsible.

```
int val;

void f(void) { val++; }
void g(void) { val--; }
void h(void) { f(); g(); }
void i(void (*func)(void))
{
    (*func)();
}

int main(void)
{
    i(h);

    return 0;
}
```

2. Compile the program for RAM-based execution. Make sure to use the '-g' flag to enable debugging.
3. Once again, start up the OpenOCD server in one window, send the HEX file to the board (note the start address), and then start up the GDB program.
4. This time, instead of using the 'hbreak' command to set an instruction breakpoint, use the 'watch' command to set a data breakpoint (i.e., a watchpoint):

```
(gdb) target remote localhost:3333
(gdb) watch val
(gdb) cont
```

5. Now issue a 'go' command using Sam\_I\_Am to start the program. Back in the GDB window you should see that the program has stopped upon an access to the 'val' global variable.
6. In the GDB window, issue a 'cont' command to continue the program and note that it stops again upon the variable access. Continue continuing the program until it is done and control returns to SAM-BA (at which point GDB will “hang” waiting for the program to stop on the watchpoint).