



Debugging Linux Kernel/Drivers with the BDI2000 & GDB

BDI2000 CONFIGURATION

You have to use the MMU XLAT (Not needed for ARM or XSCALE) option in the config file or you will not be able to debug the Linux kernel once the virtual addressing is started. For the purposes of purely debugging Linux you don't need to initialize the memory controller or have any register initializations in the config file since Micromonitor bootloader will setup all the necessary registers. To let the Micromonitor bootloader perform all the initializations and use STARTUP RUN in the [TARGET] section of the config file.

In some cases you might need to write some register values before you will be able to debug or bootup your kernel. So start out by creating a copy of your config file and empty out the [INIT] list in the config file. This is important that you comment out the [INIT] settings because you may end up with conflicts with values of your bootloader initializes with. The effects of these conflicts are mostly not obvious until you start debugging. The config file should disable the watchdog timer if you have one and perform only the minimal register writes needed to enable proper booting of your kernel. Use STARTUP RESET to force the BDI2000 to process the [INIT] list if necessary.

HOST PC CONFIGURATION

On the host system, the BDI2000 should be able to read its config file from the TFTP server and you should have already installed GNU X-Tools. GDB can be run natively on Linux or you can run on a Windows based PC with the Microcross version of Cygwin.

LINUX CONFIGURATION

You will need to make two separate Linux kernel objects. GX-Linux already does this for the Cogent ARM and XScale targets, and the file names are 'zImage' (located in ../gxlinux/linux/arch/arm/boot directory) and 'vmlinux' (located in ../gxlinux/linux directory). On the Cogent MIPS targets, GX-Linux does not have a compressed image, but a zImage is created in the root ../gxlinux directory, and this file is the one that gets loaded onto the target. The 'vmlinux' file gets created in the standard ../gxlinux/linux directory. In all cases, the target kernel image is stripped of debug symbols (and in the case of ARM/XScale, the image is compressed). The kernel image size will be between 1.5 and 3 MB. The kernel image in ../gxlinux/linux directory is used for source level debugging and will have all of the debug symbols built in it once the kernel is configured for debug symbol generation. This debug kernel version, named 'vmlinux', will be much larger, 20 to 30 MB in size. Without the 'vmlinux' and debug symbols, source level debugging is not possible. To add symbols for debugging to the vmlinux file, you will need to run the kernel configuration utility to enable 'kernel hacking' and 'Compile the kernel with debug info'. Here is an overview of the steps using GX-Linux:

- Open an xterm/Bash shell on your Linux host.
- Change directory to ../gxlinux/linux.
- Issue 'make ARCH=arm/mips menuconfig' -- select either 'arm' (includes XScale) or 'mips', depending on your board architecture.
- Exit and save the configuration.
- Issue the kernel make commands: 'make clean' and then 'make'.
- When finished, load the zImage to the target board with 'ttftp' utility that Microcross provides (e.g., 'ttftp <target-ip> put zImage zImage') – of course you can only issue this command from the same directory where the image is located, and we assume the target board is configured with an Ethernet static IP address and the target is connected and powered on.



TARGET SPECIFIC ISSUES

In general writing to/clearing debug registers via software will cause the BDI2000 to lose any breakpoints it has already set. Hence, make sure your Linux Kernel does not modify the debug registers while you are debugging with the BDI2000. Also make sure the 'QACK' signal is low on the PowerPC, if debugging this kind of target. Also, if you keep hitting exceptions in your kernel or keep breaking at interrupts, you will need to enable 'VECTOR CATCH 0xAddress' in the BDI2000 config file. Moreover, you need to use 'STARTUP RESET' for this to work properly.

XScale (CSB625 config issues)

The BDI2000 Vector Table cannot be written to by software while there is a JTAG Emulator connected to an XScale target. You have to manually update the Vector Table with the BDI2000 through the config file to values expected by the kernel, otherwise the kernel will hang at bootup.

Here is a snapshot of the [TARGET] section of the config file for the CSB625. The [INIT] section is entirely commented out to allow bootup using Micromonitor initialization.

```
[TARGET]
CPUTYPE          PXA250          ;the target CPU type
JTAGCLOCK        1              ;use 16 MHz JTAG clock
DBGHANDLER       0xFFFF0800    ;debug handler base address
VTABLO 0         0xEAFFC00E
VTABLO 1         0xEA0000E9
VTABLO 2         0xEA000108
VTABLO 3         0xEA0000C7
VTABLO 4         0xEA0000A6
VTABLO 5         0xEA000169
VTABLO 6         0xEA000128
VTABLO 7         0xEA000147
ENDIAN           LITTLE         ;memory model (LITTLE | BIG)
VTABHI 0         0xEF9F0000
VTABHI 1         0xEA0000DD
VTABHI 2         0xE59FF410
VTABHI 3         0xEA0000BB
VTABHI 4         0xEA00009A
VTABHI 5         0xEA0000FA
VTABHI 6         0xEA000078
VTABHI 7         0xEA0000F7
ENDIAN           LITTLE         ;memory model (LITTLE | BIG)
BREAKMODE       SOFT           ;SOFT or HARD
STARTUP RUN
```

SELECTING A BREAKPOINT LOCATION

To be able to debug the Kernel after the MMU turns on, you have to set the initial breakpoint to a location where the MMU is already active. We suggest setting a breakpoint at 'start_kernel' as our symbol to break on. You should connect with GDB after you have stopped the Kernel at 'start_kernel'. If you connect to GDB before the MMU is enabled, the BDI2000 will give out address errors. The location of 'start_kernel' varies from each version of Linux kernel and each target board. To find all of your symbol addresses, look at your symbol map in ../gxlinux/linux/System.map file. You can see where 'start_kernel' is located. Alternatively, you can issue the command: 'grep start_kernel System.map'.



PROCEDURE FOR DEBUGGING LINUX KERNEL WITH BDI2000 & GDB

Assumptions

- BDI2000 is setup with a working config file and static IP address
- GNU X-Tools™ is installed on the host machine
- GX-Linux™ BSP has zImage and vmlinux (w/debug symbols) pre-built
- Target board is available with power supply, serial cable, and Ethernet cable

Steps

- 1) Connect the BDI2000 to the target board.
- 2) Start two shells and a serial terminal application:
 - Start an xterm/Bash or DOS (if on Windows) shell for creating a telnet session to the BDI2000 -- call this Shell 1.
 - Start an xterm/Bash or GNU X-Tools (if on Windows/Cygwin) for connecting GDB to the target -- call this Shell 2.
 - Start a serial terminal program (Minicom / Hyperterminal / uCon) to connect to the target board at 38,400 8N1 settings.
- 3) Power-up the BDI2000 first and start a telnet session in Shell 1; let the BDI2000 load the config file, and when it says waiting for target VCC, power-up the target board.
- 4) At this point the default behavior is reset halt, which means the target will not be running. Enter 'i' at the BDI2000 prompt, to make sure the target is not running then.
- 5) Type 'go' at the BDI2000 prompt and watch the target bootup through the serial monitor.
- 6) If not already loaded, then load the zImage into Micromonitor's TFS (Tiny File System) using the 'ttftp <target-ip> put zImage zImage' command, but do not start booting Linux.
- 7) Type in 'halt' at the BDI2000 prompt and set a breakpoint at 'start_kernel':
BDI> bi 0xCxxxxxxxx (substitute the actual address)
- 8) Issue a 'go' command to give control back to the target: BDI> go
- 9) Issue the command 'startlinux' at the Micromonitor prompt: uMON> startlinux
- 10) After a few seconds the BDI2000 prompt should read 'target has entered debug mode'; enter 'i' at the BDI2000 prompt and confirm that you have stopped at 'start_kernel' address. Issue a BDI2000 command 'ci' to clear the breakpoint:
BDI> ci
- 11) Issue a few 'ti' commands from the BDI2000 prompt to confirm that you can single step through the target. The BDI2000 prompt will tell you that it is stepping: BDI> ti
- 12) On the host platform in Shell 2, change directory to .../gxlinux/linux and start GDB. The command is 'arm-linux-gdb' or 'mips-linux-gdb', depending on your target architecture.



13) Issue a command to load the debug symbol table:

```
(gdb) file vmlinux
```

14) Connect the GDB to the target using the following commands in Shell 2:

```
(gdb) target remote 192.168.0.131:2001
```

Where '192.168.0.131' is a notional BDI2000 IP address and '2001' is the default port number.

15) Perform a few steps using 'stepi' command in the GDB prompt; you should see a step occur in the BDI2000 Prompt:

```
(gdb) stepi
```

```
(gdb) stepi
```

Also perform a 'list' command in the GDB prompt and make sure GDB knows where it is:

```
(gdb) list
```

or a disassemble command:

```
(gdb) disassemble
```

16) Finally, set your own break points in GDB and see if the BDI2000 hits the breakpoints:

```
(gdb) b <symbol_name>
```

```
(gdb) c (continue to run until a breakpoint is hit or to program end)
```

PROCEDURE FOR MODULE DEBUGGING

Before beginning, make the module and make sure it is accessible by the root file system on your target. Also be sure to compile the module with the '-g' option so that the paths to the source code is known. Follow the steps below:

1) Bootup Linux on your Target Machine with the BDI2000 connected then

2) Do issue an 'insmod' on the target:

```
[target]# insmod -f -m module.o > module.map
```

This creates a map file you'll use latter. Note the major number that is printed when you do 'insmod'.

3) Use the major number in this command:

```
[target]# mknod /dev/module c 254 0
```

Skip this step if the node already exists.

4) Determine the various addresses from of the loaded module:

```
[target]# grep '\.text' module.map
```

```
[target]# grep '\.rodata' module.map
```

```
[target]# grep '\.data' module.map
```

```
[target]# grep '\.sdata' module.map
```



```
[target]# grep '\.bss' module.map
```

```
[target]# grep '\.sbss' module.map
```

- 5) You can also just view the module.map file to find these addresses:

```
Halt the target with the BDI2000: BDI> halt
```

- 6) Open the module.map file and select a suitable symbol to break on. Then set an initial breakpoint in the BDI2000: BDI> bi <0xAddress>
- 7) Give control back to your target system: BDI> go
- 8) Initiate activity to the device: [target]# echo > /dev/module
- 9) This issues a write if the driver supports it. Skip this step if you use another mechanism to initiate activity to the module.
- 10) The BDI2000 will display 'entered debug mode', the current PC should be at the break point. At this point clear the breakpoint we just set:

```
BDI>ci
```

- 11) Start GDB using the vmlinux file that has debug symbols:

```
[host]# <prefix-name>-gdb vmlinux
```

- 12) Add symbols into GDB (addresses are notional):

```
(gdb) add-symbol-file <path-to-module-dir>/ex_sw.o 0x c2000060\-\n\ns .rodata 0xcf030354\-\ns .data 0xcf030488\-\ns .sdata 0xcf030488\-\ns .bss 0xcf030519\-\ns .sbss 0xcf03051c 0xc2000060=text address
```

- 13) Connect GDB to the BDI2000

```
(gdb) target remote 192.168.0.131:2001
```

where '192.168.0.131' is a notional IP address assigned to the BDI2000 and '2001' is the default port number.

- 14) Perform a list command,

```
(gdb) list
```

Current source code will be displayed.

Perform a few steps in GDB, you should see a step occur in the BDI2000:

```
(gdb) stepi
```

```
(gdb) stepi
```

- 15) Finally, set your own break points in GDB and see if it the BDI2000 hits the breakpoints:

```
(gdb) b (breakpoint-1)
```

```
(gdb) b (breakpoint-2)
```

```
(gdb) c
```

If upon doing the 'insmod' command the module gets loaded and runs instantly, then you will not get a chance to set a breakpoint with the BDI2000. For this case you will have to change the procedure for setting a breakpoint and setting up the environment for debugging. One alternative is to put a conditional loop in the module at main where the loop will send you if you modify a



register. Once you are in the loop you can halt the system through the BDI2000 and set the register to continue executing out of the loop. At this point you should proceed to issue the command `'insmod'` and make the `'module.map'` file, then restart the system and set a breakpoint in the BDI2000 before the `'insmod'` command is used again. This breakpoint is at a location in the `'module.map'` file that was made previously and should match the new `'insmod'`. Once this breakpoint is set, load the module as normal with `'insmod'` and the BDI2000 will indicate that the breakpoint has been hit.

APPLICATION DEBUGGING

The BDI2000 cannot be used to debug Linux applications; it is designed so that it can access kernel space memory, but it has no information about user space memory. Below is a diagram giving you a basic idea of how Linux divides the virtual address memory locations. Upon startup with `'startlinux'`, the Micromonitor boot loader moves the kernel from the Tiny File System (TFS) into memory and jumps to the kernel start routine. Linux then takes over and brings up the kernel. Once Linux is running, it enables the MMU and uses virtual addressing. The Linux kernel occupies a quarter of the virtual address space starting from address `'0xC0000000'`; this region is also known as "Kernel Space". Linux also creates an "Application Space" where it loads user-land programs. The virtual address for application space is different for different implementations of Linux; it can be at `'0x02'` or wherever the developer assigns it to be. Linux assigns the address to the program dynamically when it is loaded and has no fixed location where Linux will load a particular application. You will need to look at the mapfile under the PID entry for your program in the `/proc` file system to determine its memory map.

System Calls is the API to interface to IO and kernel space. Kernel Space is defined as all virtual addresses starting with `'0xC'`. Restricted access, applications must use system calls to access kernel Space. The application normally occupies a certain amount of memory and variables in the application do not get assigned their own memory space until an access is made to them; hence, there is no guarantee that a virtual address actually translates to a physical address. The BDI2000 is not designed to account for this type of behavior. Furthermore, data can get swapped out from application memory to save memory space, and gets loaded to a different location on the next access. This moves the breakpoint location, and this can confuse the BDI2000 when it tries to hit a breakpoint. In summary, the BDI2000 cannot be used for application debugging. You can, however, perform application debugging by using the gdbserver that resides on your target Linux machine. The connection steps are similar to what you do when you connect to the BDI2000. You have to make sure that the executable you use to debug on your host has debug symbols built-in. You cannot use the same executable that gets loaded on your target because it is not built with the proper debug headers. Read the steps describing application debugging below.

PROCEDURE FOR APPLICATION DEBUGGING

- 1.) On your target launch the application with the gdbserver specifying the host IP:

```
[target-session]# gdbserver 192.168.0.210:2001 <target-app>
```
- 2.) On your host start GDB with the application to debug on the host:

```
[host-session]# arm-linux-gdb <target-app> (or mips-linux-gdb)
```
- 3.) From GDB connect to the gdbserver on your target specifying the target IP:

```
(gdb) target remote 192.168.0.131:2001 (notional settings)
```



4.) You can now set breakpoints and begin debugging.

```
(gdb) b main
```

```
(gdb) cont
```

Debug as usual with GDB commands from this point forward.

The End