Ultimate Solutions, Inc.

Application Notes for Professional Developers of Embedded Systems # 02-001a

Using the Abatron BDI2000 to Debug a Linux Kernel

▶ *Introduction*

A common question we get asked about the Abatron BDI2000 is, "how can the tool be used with a Linux kernel on the target?" The short answer is that the BDI2000 can be used to debug any code that runs in kernel space. This AppNote goes into the details and takes you through all the steps you need to follow to debug a Linux kernel with the BDI2000.

You might be wondering why the BDI2000 only works with kernel-mode code and not user-land application code. The technical details are beyond the scope of this AppNote. The 10,000 foot answer is that while the BDI2000 can work with the MMU enabled, it can only work with a single continuous memory map. Kernel-mode code exists in a single continuous memory map, which is disjoint from all of the user-land process memory maps.

As an example of the usefulness of this tool in production environments, the PPC porting engineers at Monta Vista Software use the BDI2000 extensively for kernel bring-up and device driver debug.

The following areas are discussed in this AppNote:

- Preparing the Kernel to be Debugged
- Controlling the Hardware Target with the BDI2000
- Setting BDI2000 Breakpoints
- Downloading the Linux Kernel
- Connecting Host Debugger to BDI2000 Target
- Setting Host Debugger Breakpoints
- Regaining Control of the Debug Session
- Debugging Kernel Modules

❱ *Host / Target Setup*

Host system used is an x86 Desktop gray box, running RedHat Linux V7.3. Target system is an Embedded Planet RPXLite board running the PlanetCore Boot Loader, v1.02. The version information for the Abatron BDI2000 is as follows:

Component	Version
Bdisetup	1.09
Loader	1.04
Firmware	1.11 bdiGDB for MPC 8xx
Logic	1.02 MPC8xx/MPC5xx

The kernel used in this AppNote is Monta Vista Linux v2.1, available from Monta Vista Software.

Tech Tip:

The Abatron firmware can get out of sync with the Linux kernel. In order to support the functionality of a BDM or JTAG debugger, the kernel has to cooperate with the debug agent. If you have any trouble following the steps of this AppNote, consider the possibility that the kernel version you are working with is incompatible with the Abatron firmware version you have.

The following diagram shows the component connections: host, target, BDI2000.
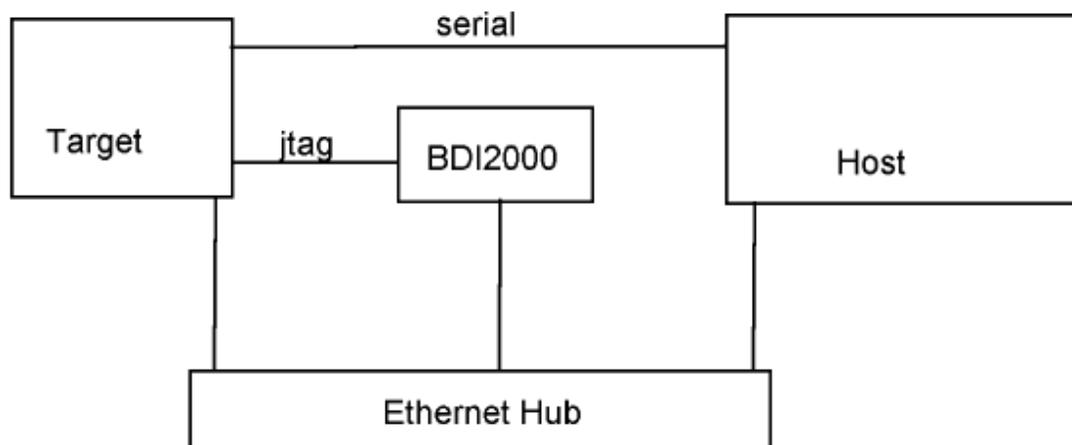
❱ *Preparing the Kernel to be Debugged*

This section describes building a Linux kernel with debug symbols enabled. The Makefile in the root of the kernel source tree needs to be modified. This is the same Makefile that controls the configuration and building of the kernel.

The CFLAGS macro needs to be modified. Do NOT modify the HOSTCFLAGS macro. Assuming the MontaVista Linux kernel is installed in the default location…

Step 1) **cd [root of kernel source tree]**

Step 2) **vi +/"CFLAGS :=" Makefile** (Please Note: there is a space between CFLAGS and :)

Step 3) add "-g" to the end of the CFLAGS macro definition

Step 4) **make clean dep**

Step 5) **make zImage**

Step 6) **make modules modules_install** (Please Note: last make target is: modules_install)

All of the make targets can be entered on one line if you prefer, I usually enter them as above. Assuming the zImage target completes successfully, a new kernel image can be found somewhere in the kernel source tree. This file should be copied to a location that is easy to reference for download purposes. The step below places the file in the '/tftpboot/ directory, renaming the file to 'vmlinuz.bdi'.

Step 7) **cp …/zImage.embedded /tftpboot/vmlinuz.bdi**

TechTip:
Do not configure KGDB support into the kernel. Using KGDB and the BDI2000 are mutually exclusive.

▶ *Controlling the Hardware Target with the BDI2000*

The BDI2000 requires a configuration file to initialize the hardware target. There are two options here. The configuration file can be very sophisticated and perform all of the hardware initialization. Or the file can be very simple, providing just enough initialization to allow the BDI2000 to jump to the board's firmware, allowing the firmware to provide the bulk of the configuration and initialization.

Writing a detailed configuration file for a specific hardware target is beyond the scope of this AppNote. For the purposes of the recipe provided here, the BDI2000 works in conjunction with the firmware, in this case the PlanetCore Boot Loader.

Step 8) Power up the target system and BDI2000. The Abatron documentation suggests the BDI2000 unit be powered first, and then apply power to the target.

The next step is going to establish the connection between the host and the BDI2000. The command assumes an entry exists in the host's '/etc/hosts' file identifying *bdi* as a network node with an IP address.

Step 9) **telnet bdi**

This telnet session can be thought of as the BDI2000 command or console window. A list of the emulator commands is displayed followed by a prompt, typically "BDI>".

When a recipe step refers to a command to type in the BDI2000 command window, the command will be prefaced with [BDI2000]. All other commands are Linux commands, unless stated otherwise. Gdb commands will be prefaced with [gdb] in the steps listed below. The **bold** portion of a recipe step refers to commands you type in exactly as written.

▶ *Setting BDI2000 Breakpoints*

I recommend being very familiar with the Abatron documentation that discusses debugging with gdb and embedded Linux MMU support. In my copy of the manual (v1.23), this is pp 32-36.

The Abatron documentation specifically states NOT to set a breakpoint where the MMU is disabled, and suggests the easiest way to get connected to the host debugger is to set a breakpoint after the kernel initialization code has enabled the MMU.

The implication of these warnings is that if you do have to start single-stepping from the first instruction, beware of what will happen to your debug session as you step-through code that is enabling the MMU. This AppNote assumes you do not have to debug the very first part of the kernel initialization and attaching to the kernel after the MMU is enabled is sufficient.

The kernel entry point *start_here* is a good early point in the kernel boot process to set a breakpoint. The next step is going to reference a file that you should be familiar with if you are doing kernel or device driver work. When the kernel is built, both an uncompressed kernel image [vmlinux] and a symbol map file [System.map] are generated in the root directory of the kernel source tree. The addresses in the map file are with the MMU turned on and this file is required for symbolic debug with gdb and the BDI2000.

Step 10) **grep start_here System.map**

Step 11) [BDI2000] **bi XXXXXXXX**
XXXXXXXX = address of start_here

TechTip:
> If the BDI2000 responds to the above command with the following message:
>> *# Target must be in debug mode for this action*
>
> The target is running and a breakpoint cannot be set. In the BDI2000 Command Window, type **halt**, and re-issue the above breakpoint command.

▶ *Downloading the Linux Kernel*

As noted above, the board's firmware is going to be used to initialize the hardware. In order to see what happens with the firmware, as well as to have a Linux root console when the target does finally boot, we will use a minicom session on the Linux host to communicate with the target, across a serial connection. The details of specifying baud rate, etc. to minicom is beyond the scope of this AppNote. It is recommended that you verify the firmware connection with minicom before you start the debug session.

Step 12) **minicom**

Step 13) [BDI2000] **go**

In the minicom window you should see some startup messages from the PlanetCore firmware and then see its prompt, which by default is '>'. Wait for the prompt.

Step 14) [BDI2000] **halt**

At this point the BDI2000 is in control of the target, which has been fully initialized by the onboard firmware. The target is now ready to receive a kernel download.

The [HOST] portion of my BDI2000 configuration file is as follows:

IP	192.168.0.11
FILE	vmlinuz.bdi
FORMAT	BIN 0x00400000
START	0x00410000
LOAD	MANUAL

These settings control how and what the BDI2000 will download when you issue the following command.

Step 15) [BDI2000] **load**

Step 16) [BDI2000] **go**

In the minicom window you should now see some activity. The system will sit for a couple of seconds displaying the kernel boot command line. This delay allows you to modify the command line, if need be. You can press CR in the minicom window or wait for the timeout. The last message displayed before the breakpoint is hit is:
> *Now booting the kernel*

In the BDI2000 Command Window a new message has appeared:
> *- TARGET : target has entered debug mode*

Step 17) [BDI2000] **ci**

TechTip:
> You can use the BDI2000 command "info" to determine why debug mode was entered. Typing **info** now yields the following information:

> | | |
> |---|---|
> | Target state | : debug mode |
> | Debug entry cause | : instruction breakpoint |
> | Current PC | : 0xC00020F0 |

My 'System.map' file lists this address as the 'start_here' entry point.

▶ *Connecting Host Debugger to BDI2000 Target*

At this point the target is in a halted state, partway through a Linux kernel boot sequence, under control of the BDI2000. The next step is to start up the debug session on the host and connect this debug session to the target. The Linux steps below assume you are in the root directory of the kernel source tree, see *Step 1.*

The next step is going to reference the uncompressed kernel image that was created in *Step 5.* *Step 5* created a compressed kernel (zImage.embedded). A side-effect of building the compressed kernel (or any kernel target) is that an uncompressed kernel image is always generated. This file [vmlinux] appears in the root of the kernel source tree, along with a symbol map file.

Step 18) ***ppc_8xx-gdb vmlinux***

Step 19) [gdb] ***target remote bdi:2001***

Warning:

As soon as the gdb remote command is issued, the following message is displayed twice in my BDI2000 Command Window:

*** *MMU : address translation for 0x00000018 failed*

This is a benign warning in this case. The entry point *start_here* is an assembly language function that does not have a full stack frame created for it. Gdb is probing the stack frame as it would for a C language function, which causes the warning messages to be displayed. If you want to stop in a C function, set your breakpoint on *identify_machine,* which is called by *start_here*. With your initial breakpoint set at *identify_machine*, there is no warning displayed in the BDI2000 Command Window.

TechTip:

If you prefer to use a graphical front-end to gdb, you can try using ddd. The command line for *Step 18* is changed as follows:

Step 18) **ddd –debugger ppc_8xx-gdb –gdb vmlinux**

If you have trouble connecting with just gdb, do not try ddd until you get your connection issues resolved. Please note the options above, debugger and gdb, are prefaced with two dashes, not one. Ddd is not recommended for use with KGDB, because of the serial connection required for KGDB and the overhead associated with supporting a graphical interface.

▶ *Setting Host Debugger Breakpoints*

At this point you are controlling the kernel boot through the BDI2000 with a gdb session on your host. One thing I like to do here is set a couple of breakpoints, one at 'panic' and one at 'sys_sync'. The 'panic' breakpoint should be self-explanatory. The 'sys_sync' breakpoint can be thought of as a backdoor into the debug session once the kernel boots and you have a login prompt from your target.

Step 20) [gdb] **b panic**

Step 21) [gdb] **b sys_sync**

Step 22) [gdb] **cont**

The 'cont' command in gdb allows the kernel to finish booting. If you are interested in stepping through specific boot sequences, you should set other breakpoints before continuing the boot process.

TechTip:

Steps 19-22 can be automated through the use of a gdb start-up file, [.gdbinit]. The start-up file can appear in your home directory or the current working directory. The CWD file takes precedence over the HD file. A sample [.gdbinit] file for these commands would be:

```
# .gdbinit
target remote bdi:2001
b panic
b sys_sync
cont
```

▶ *Regaining Control of the Debug Session*

Assuming a breakpoint has been set at 'sys_sync' (see *Step 21*), gdb on the host can be re-entered at any time by typing the Linux 'sync' command on the target. If the kernel boots successfully to a login prompt, login into the target system and type **sync** to verify this behavior.

▸ *Debugging Kernel Modules*

Where the BDI2000 truly shines as a tool for the Linux developer, is in its ability to debug kernel modules that are loaded after the kernel boots. This capability makes it extremely useful for device driver developers, who don't want to reboot the system for every change made during development cycles. We will introduce the user to the basics here, while saving the details for another AppNote.

One of the main benefits of using the BDI2000 as a Linux kernel debugger, is the ability to debug kernel modules. The recipe described below for debugging kernel modules, precludes the ability to debug the module's 'init' method.

Step 23) [target linux] **insmod –m** *module.o* **>** *module.map*

The filenames are not in bold in the above command because they will vary based on what module you are trying to debug. The important part of the command above is the '-m' flag to insmod. This instructs insmod to generate a load map for the module when it gets loaded into memory.

Step 24) [target linux] **grep ".text"** *module.map*

Step 25) [target linux] **sync**

Executing 'sync' on the target box allows gdb on the host to regain control.

Step 26) [gdb] **add-symbol-file** *module.o XXXXXXXX*
XXXXXXXX = address of '.text' section from grep command.

Step 27) [gdb] set breakpoints in your module code

Step 28) [gdb] **cont**

At this point, you should have a module loaded with breakpoints set for further debugging. In order to hit any breakpoints in your module you must initiate activity on your target that includes accessing the device controlled by this module/driver.

▸ *Summary*

In this AppNote we have tried to take you through a programmed sequence of steps, or recipe, for using the BDI2000 to debug a Linux kernel. Some of the presented steps will have to be modified, based on your specific setup. For example, if you have a different target board or are using a different embedded Linux kernel, or have different versions of the discussed tools, your mileage may vary.

The BDI2000 configuration file settings will also have an impact on some of the steps presented in this recipe.

You should be comfortable with the tool [BDI2000] and your hardware before trying to debug something as complex as the Linux kernel.

This recipe shows one way to use the BDI2000 in conjunction with gdb to debug the kernel boot process. The steps outlined here allow you to debug fairly early on in the boot process. As you become more comfortable with the process you can fine-tune this recipe to give you the control you need over your debug environment.

Follow this link http://www.ultsol.com/appnote.htm to the Ultimate Solutions website where additional information is presented regarding this AppNote. A BDI2000 configuration file is presented, as well as simple module examples that were both used in creating this AppNote.

Authored by: T.Michael Turney
                 Founder & CTO – Tools Made Tough

                 T. Mike has over 20 years of experience working on embedded software projects, from million dollar robotic systems to mass-produced handheld devices. Mike has contributed to the development and support of various products over the years to include VRTX from Hunter & Ready and SPOX, a Real-Time Operating System designed specifically for Texas Instruments' line of digital signal processors. Prior to founding Tools Made Tough in July 2002, T. Mike served as a Senior Field Application Engineer at MontaVista Software, Inc., an industry leader in providing embedded Linux solutions for a wide range of applications.

Ultimate Solutions, Inc.
10 Clever Lane
Tewksbury, MA 01876
Toll Free: 866.455.3383 Phone: 978.455.3383
Fax: 978.926.3091   Email: info@ultsol.com
Web: http://www.ultsol.com