Ultimate Solutions, Inc. Tool Talk #04-002

Application Notes for Professional Developers of Embedded Systems

Debugging Linux with the BDI2000 & bdiGDB

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 Linux 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 PPCBoot/Uboot (or your bootloader) will setup all the necessary registers. To let your bootloader perform all the initializations use STARTUP RUN in the [TARGET] section of the config file.

In some cases you will 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, It is important that you do this other wise you may end up with conflicts with values 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 BDI to process the [INIT] list.

HOSTPC CONFIGURATION

On host system is the BDI should be able to read its config file from the TFTP server and you should have already downloaded and configured GDB. GDB can be run natively on Linux or you can download cygwin from www.cygwin.com to run on a windows based PC. You need to make sure you are using cross-compiled GDB that is target and host specific otherwise you will not be able to connect the BDI that runs a target specific version of the GDB server. Download GDB from http://www.gnu.org/software/gdb/ and look at http://www.ultsol.com/pdfs/ToolTalk_03-001.pdf for instructions on how to compile it.

LINUX CONFIGURATION

You will need to make 2 separate vmlinux files. One gets loaded into the target, stripped of debug symbols; it will be 2-5 megs. The second one is used for source debugging and has the entire debug symbols built in. It will be much bigger, some 20 to 30 megs in size. Without this file source level debugging will not be possible. To add/remove symbols from the vmlinux file, edit the top level make file and add/remove the "–g" flag under the CFLAGS entry.

Please configure and use gdb and gcc that are cross-compiled for your target architecture, for instance, gdb for PPC8xx will be called something like ppc_8xx-gdb. You can find free versions of pre-configured and tested cross-compiled GNU tools for your target processor at http://www.denx.de/ELDK/.

© 2004 Ultimate Solutions, Inc.

There exists in some target architectures a special CONFIG_BDI_SWITCH, which can be set in the kernel configuration. Look for CONFIG_XSCALE_BDI2000 in XScale sources. Please enable this switch in your Linux kernel configuration file, it can be activated using “make menuconfig” and enabling “kernel hacking”, the CONFIG_BDI_SWITCH is under this section. This switch is used to setup pointers so that the BDI can find the page tables. Look at head.s, head_8xx.s, head4_xx.s to see if this option is used anywhere, if it isn’t you will probably have to add it in. Writing to page tables has to be enabled to use software breakpoints, see http://www.ultsol.com/faq_p305.htm.

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 BDI. Also make sure the QACK signal is low on the PowerPC.

Also if you keep hitting exceptions in your kernel or keep breaking at interrupts you will need to enable VECTOR CATCH in the BDI2000 config file. You need to use STARTUP RESET for this to work properly.

PPC4XX
In the case of PPC4xx the CONFIG_BDI_SWITCH prevents the kernel from modifying the debug registers (ppc4xx_setup.c). If this option does not exit in the ppc4xx_setup.c then please modify the code such that lines containing DBCR are not used when the switch is set. Also search your source code base for “Abatron” to find references to what it does.

PPC440
The PPC440 needs to have valid TLB entries before it can access memory. So you need a minimal init list that sets up the TLB’s in your BDI config file. You have to use STARTUP RESET.

PPC405
On the PPC405 we have to invalidate the kernel page tables on startup before using software breakpoint. Not doing so can result in a kernel panic when we hit the breakpoint. Use STARTUP RESET and add this line to the init list, “WM32 0x000000f0 0x00000000”.

MPC8XX
The debug registers cannot be written by software when a JTAG Emulator is connected; this can cause the Kernel to crash when it tried to modify the debug registers. It will also cause the kernel to break at exception entries since DER is not configured. You can use the BDI to initialize the debug registers to the state your Kernel expects them to be in, use the [INIT] list and STARTUP RESET. Also for this architecture, JTAG debug tools cannot debug the RFI instruction (or any exception routine), you have to set breakpoints around the RFI instruction or exception routine and make sure you run over them. See http://www.ultsol.com/faq_p302.htm for details.

PPC750FX
Make sure the QACK signal is low, when this signal is high the PowerPC cannot enter sleep mode, the result is that software breakpoints do not work.

MPC85xx
Some modifications to the Linux Kernel might be necessary to get breakpoints to work. Please read the corresponding FAQ at http://www.ultsol.com/faq-PPC_debug.htm.

 © 2004 Ultimate Solutions, Inc.

XSCALE
The Vector Table cannot be written to by software while there is a JTAG Emulator connected, you have to manually update the Vector Table with the BDI to values expected by your kernel otherwise you will get a kernel panic, http://www.ultsol.com/faq_x301.htm.

SELECTING A BREAKPOINT LOCATION

To be able to debug the Kernel after the MMU turns on we have to set the initial breakpoint to a location where the MMU is already active. We will select start_kernel as our symbol to break on; start_kernel is located a little bit after the MMU gets turned on. We should connect GDB after we have stopped the Kernel at start_kernel, if we connect GDB before the MMU is enabled and then try and break at start_kernel the BDI will give out address translation errors. The location of start_kernel varies from each version of Linux and each board, look at your system.map in the root of your kernel directory to see where start_kernel is located. Alternatively, you can use the command "objdump –t vmlinux | grep start_kernel" at the Linux prompt to get the address of start kernel, run this command from the root of the Linux kernel tree. Also check the GDB you are using and make sure it supports PowerPC or the target processor you are using.

PROCEDURE FOR LINUX KERNEL DEBUGGING

1.) Connect the BDI2000 to the Target
2.) Power up the BDI2000 first and start a telnet session, let it load the config file, when it says waiting for target VCC
3.) Power up the Target
4.) At this point the default behavior is Reset Halt, which means the target will not be running. Put 'i' at the BDI prompt, to make sure the target is not running then
5.) Type in "go" at the BDI prompt and watch your target boot up thru its serial monitor.
6.) If you use UBOOT to load the Linux Kernel then do so now, other wise skip this step. Load the smaller vmlinux (without the "-g" option) into PPCBoot/UBoot using the TFTP command but do not start booting Linux.
7.) Type in "halt" at the BDI prompt and set the Breakpoint at start_kernel, BDI> bi 0xCxxxxxxx
8.) If you haven't loaded the Linux kernel thru UBOOT, then you can load it now using the "load" command from the BDI, otherwise skip this step.
9.) Issue a "go" to give control back to the target, BDI> go
10.) If you loaded Linux from the BDI Linux is already booting up on your target otherwise you now have control back on PPCBoot/UBoot, use bootm and boot up Linux.
11.) After a few seconds the BDI prompt should read "target has entered debug mode", enter "i" at the BDI prompt and confirm that you have stopped at start_kernel. Use "ci" and clear the breakpoint,
BDI> ci
12.) Use a few "ti" commands from the BDI prompt to confirm that you can step thru the target, the BDI prompt will tell you that it is stepping, BDI> ti
BDI> ti
13.) On the host platform, start GDB with the larger vmlinux file located in the root of the Linux Kernel directory that has symbols (the "–g" option added to the make file), [host] gdb /path_to/vmlimux
14.) Connect to the target using,
(gdb) target remote 192.168.123.105:2001
Where 192.168.123.105 is the IP you configured the BDI with.
15.) Perform a few steps using "stepi" in the GDB prompt, you should see a step occur in the BDI Prompt,
(gdb) stepi
(gdb) stepi
Also perform a "list" command in the GDB prompt and make sure GDB knows where it is,
(gdb) list
16.) Finally, set your own break points in GDB and see if it the BDI2000 hits the breakpoints,
(gdb) b symbol_name

 © 2004 Ultimate Solutions, Inc.

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 BDI connected then
2.) Do an insmod on the target,
    [target]# insmod -f -m module.o > module.map
3.) This creates a map file we'll use latter. Note the major number that is printed when you do insmod.
4.) Use the major number in this command,
    [target]# mknod /dev/module c 254 0
    skip this step if the node already exists.
5.) 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
    You can also just view the module.map file to find these addresses.
6.) Halt the target with the BDI, "BDI> halt"
7.) Open the module.map file and select a suitable symbol to break on. Then set an initial breakpoint in the BDI,
    BDI> bi 0xBKPT0"
8.) Give control back to your target system,
    BDI> go
9.) Initiate activity to the device,
    [target]# echo > /dev/module
    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 BDI 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 larger vmlinux file,
    [host]# ppc_8xx-gdb vmlinux
12.) Add symbols into GDB,
    (gdb) add-symbol-file <path-to-module-dir>/ex_sw.o 0x c2000060\
    -s .rodata 0xcf030354\
    -s .data   0xcf030488\
    -s .sdata 0xcf030488\
    -s .bss   0xcf030519\
    -s .sbss   0xcf03051c
    0xc2000060=text address, and the rest of the addresses you will get from step 5
13.) Connect GDB to the BDI
    (gdb) target remote 192.168.123.105:2001
    where 192.168.123.105 is the IP you have assigned to the BDI.
14.) You will be taken to BKPT0 in GDB. Perform a list command,
    (gdb) list
    Current source code will be displayed.
15.) Perform a few steps in gdb, you should see a step occur in the BDI,
    (gdb) stepi
16.) Finally, set your own break points in GDB and see if it the BDI2000 hits the breakpoints,
    (gdb) b BKPT1
    (gdb) b BKPT2
    (gdb) c

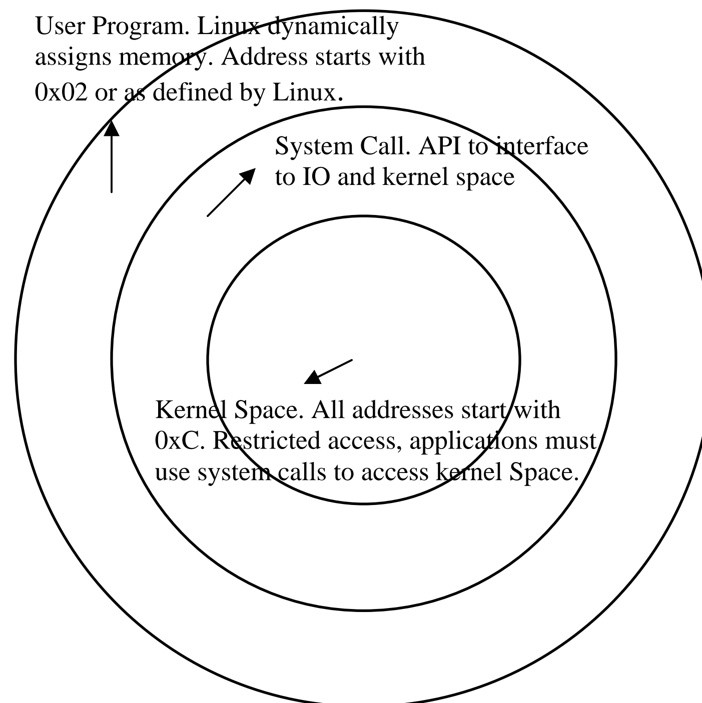
 © 2004 Ultimate Solutions, Inc.

If upon doing the insmod in step 3 the module gets loaded and runs instantly then you will not get a chance to set a breakpoint with the BDI. 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 if you modify a register. Once you are in the loop you can halt the system thru the BDI and set the register to continue executing out of the loop. At this point you should proceed directly from step 11.

Another method is to do an insmod and make the module.map file, then restart the system and set a breakpoint in the BDI before the insmod is done 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 BDI will indicate that the breakpoint has been hit. You can then proceed from step 11.

APPLICATION DEBUGGING

The BDI2000 cannot be used to debug Linux applications; it is designed so that it can access kernel space memory but has no information about user space memory. Below is a diagram giving a very basic idea of how Linux divides the virtual address memory locations suitable for our discussion.



Upon startup the boot loader moves the kernel from the root file system 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 where ever the developer assigns it to be. Linux assigns the address to the program dynamically when it is loaded and there is no fixed location where Linux will load a particular application. You will need to look at the maps file under the PID entry for your program in the /proc file system to determine its memory map.

 © 2004 Ultimate Solutions, Inc.

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 BDI is not designed to account for this type of behavior. Another point is that 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 BDI when it tries to hit a breakpoint. Hence 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 pretty similar to what you do when you connect to the BDI. You have to make sure that the executable you use to debug on your host has 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. See the steps described below.

PROCEDURE FOR APPLICATION DEBUGGING:

1.) On your target launch the application with the gdbserver specifying the host IP
[linuxtarget]# gdbserver 192.168.100.1:2001 targetapp
2.) On your host start GDB with the application to debug on the host.
[linuxhost]# target_specific-gdb targetapp
3.) From gdb connect to the gdbserver on your target specifying the target IP
(gdb) target remote 192.168.100.2:2001
4.) You can now set breakpoints and begin debugging.
(gdb) b main
(gdb) cont

Authored by: Fahd Abidi
Field Application Engineer with Ultimate Solutions, Inc.

10 Clever Lane
Tewksbury, MA 01876-1580 USA
Ph: 866.455.3383 Fx: 978.926.3091
Email: info@ultsol.com
Web: www.ultsol.com

 © 2004 Ultimate Solutions, Inc.