Ultimate Solutions, Inc.

Tool Talk #04-001

Application Notes for Professional Developers of Embedded Systems

Debugging Embedded NetBSD with the Abatron BDI2000

Introduction

Debugging embedded systems running a UNIX-like operating system is an arduous task. When a piece of the system goes awry, few assumptions can be made and only after ample testing to isolate the offending subsystem is progress made. Also complicating things is the fact that even the most popular evaluation boards have only a fraction of the runtime as a PC mainboard. Possessing a competent debugging solution is perhaps the first step to developing a solid embedded system.

The goal of this paper is to present a critical debugging solution for embedded systems running NetBSD. The solution centers on the BDI2000, a hardware JTAG/BDM debugger that is capable of performing address translations in a virtual memory environment. Although not applicable for all debugging needs, it is sufficient for many of them and even lends itself as the only solution for some debugging scenarios. For example, with this solution it is possible to step through source code of an application and into the kernel, examining memory and variables in both spaces. Both the underlying theory as well the practical applications will be presented for completeness.

The Memory Management Unit

Before we begin discussing the methodologies of debugging, let us digress for a moment and review a concept central to our purpose. The memory management unit (MMU) is a commonly found component in microprocessors. The MMU performs two primary tasks: address protection and address translation. The former allows read-write, read-only, or no access attributes to be associated with a region of memory. The latter takes place when the microprocessor core requests a read or write from memory, and is illustrated in Figure 1. The memory address referenced by the core (1) is sent to the MMU (2) and translated into a second address (3) that is then placed on the address bus (4). These memory addresses are coined “virtual address” and “physical address,” respectively.

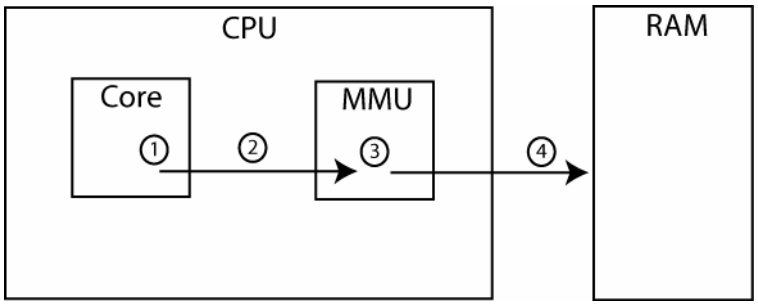

Figure 1

The most common implementation of address translation makes use of a series of tables in RAM called translation tables that contain mappings of virtual addresses to physical addresses. To improve translation efficiency, these translation tables are organized in a hierarchy of two or more levels. When programming the MMU, the physical address of the highest-level translation table is written to a table base register, the exact name and mnemonic differing between architectures. For example, in Freescale's PowerPC 8xx series this register is called M_TWB while in the i386 architecture it is called CR3.

In UNIX-like operating systems, each running process contains its own instance of translation tables describing the virtual space the process exists in. A corollary to this design is that multiple processes can exist within the same virtual addresses while the mappings to physical addresses differ. The operating system kernel also contains its own translation tables, differing from those of a process in one significant respect; the virtual addresses used by the kernel and those by a process do not overlap.

Three areas of debugging

Debugging firmware in an embedded system running NetBSD takes place in three distinct spaces as illustrated in Figure 2. *Bootloader space* begins when the system is powered on (1) and stops when the kernel enables the MMU (3). Between these two events the bootloader, sometimes called the standalone kernel, will copy the NetBSD kernel from persistent storage to RAM and begin executing it (2). The most of hardware initialization takes place in this space and includes but is not limited to external memory devices and the interrupt controller. *Kernel space* starts when the MMU is enabled and configured with the table base register pointing to the kernel's translation tables. It is the responsibility of the kernel to configure and enable the MMU, and few kernel duties are accomplished before this takes place. This space encompasses the debugging of kernel components including device drivers, process scheduling, network stacks, etc. *Application space* starts when the kernel schedules a process for execution and a switch to the process's context is performed (4). At some point during this switch the MMU's table base register will be configured to reflect the process's translation tables. Under

normal execution kernel and application space will toggle during a context switch (5) or an event requiring service by the kernel like a system call or a disk I/O event (6).

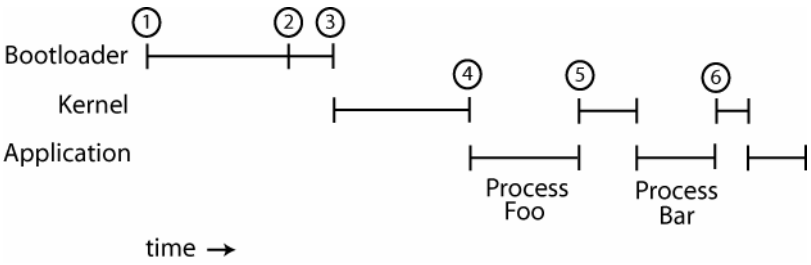

Figure 2

A variety of debugging techniques exist for debugging all three spaces. Due to the hardware centric nature of bootloader space, using a hardware tool to step through CPU instructions is perhaps the wisest decision. Since the MMU is disabled the tool need only concern itself with a flat address space; this drives down the tool's complexity and price. Debugging kernel space can be accomplished using a debugging agent compiled into the kernel or a more sophisticated hardware tool. The NetBSD kernel contains two options for agents, ddb and kgdb. The former is a lightweight yet powerful interface for examining the contents of memory, decoding selected data structures, and stepping through single CPU instructions. This agent is coupled with a command line interface also compiled in the kernel, making this solution stand-alone. The kgdb agent provides a remote debugging interface that communicates with the GNU debugger. This solution provides source level debugging but requires a host computer to run the GNU debugger as well as a free serial port on both systems. Debugging application space can be accomplished in a variety of ways, although limited to software solutions. The three most common solutions are to compile the GNU debugger for the target system, compile a small debugging stub into the application, or use a separate server application that understands a remote protocol as well as possessing the ability to control an application.

Debugging with the BDI2000

The BDI2000 provides a comprehensive solution for debugging an embedded system running NetBSD. Central to this functionality is its ability to control and query the state of the MMU; in particular the BDI2000 can perform address translation on addresses received from the GNU debugger. To set up a debugging session, the NetBSD kernel must be modified to support the BDI2000. Next, the kernel and applications to be debugged must be compiled with debugging information. Lastly, the BDI2000 configuration file must be modified to reflect the address of an expected symbol in the kernel.

Modifying the NetBSD Kernel

Modifications to the NetBSD kernel take the form of a pseudo-device that controls the translation tables to be used by the BDI2000's address translation mechanism. Per the BDI2000 manual, support for address translation is configured with the address of a pointer to an array of two pointers to translation tables. Although initially confusing, the purpose for this design will be evident shortly. The array of two pointers is configured by the pseudo-device and is exposed to an application using the `ioctl` system call. A pointer to this array must be stored at a location in the NetBSD kernel that remains constant in the presence of modified kernel options and other kernel development, i.e. device drivers. The most convenient place for this pointer is immediately following the entry point of the kernel. Figure 3 shows the layout of this series of pointers in RAM.

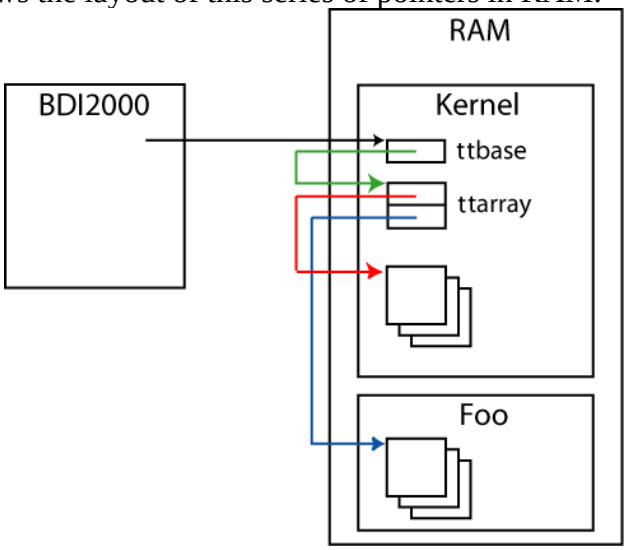

Figure 3

Due to the orderly and uniform structuring of the NetBSD source tree, the procedure for adding BDI2000 support to different architectures is nearly the same. Changes center around integrating a new pseudo-device, bdi2k, into the source tree. The following is a list of files that would require modification for a fictitious architecture Foo.

File	sys/arch/foo/conf/files.foo
Change	The name of the pseudo-device must be declared and the implementation files listed. Add these lines to the end of the file.
	`defpseudo  bdi2k` `file  arch/foo/dev/bdi2k.c  bdi2k`

File	sys/arch/foo/conf/FOO
Change	The pseudo-device must be added to the list of devices compiled into the kernel.Group the following line with the other pseudo-devices, usually found at the end of the kernel configuration file.

	`pseudo-device  bdi2k  # BDI2000 debugger support`

File	sys/arch/foo/conf/majors.foo
Change	A major number must be assigned to the pseudo-device. The next available number reserved for machine-dependent drivers will be used.
	`bdi2k  char 75  bdi2k`

File	sys/arch/foo/foo/locore.S
Change	Space must be reserved for the translation table pointer immediately following the __start symbol. It must be defined as a “C” global to be referenced by other object files. Variants of these assembler instructions should translate easily to other architectures.

	`__start:`
	`        b bdi2k_ttbase_end`
	`        .globl _C_LABEL(bdi2k_ttbase)`
	`_C_LABEL(bdi2k_ttbase):`
	`        .long 0`
	`bdi2k_ttbase_end:`

File	sys/arch/foo/include/bdi2000.h sys/arch/foo/dev/bdi2000.c
Change	These are the implementation files for the pseudo-device of which a full listing can be found for in Appendix A. The code requiring rework per architecture is limited to that which queries the addresses of the kernel and application translation tables. To simplify the task of porting, this code is isolated in two functions, “`init_bdi2k_ttptrs`” and “`bdi2kioctl`”.

Debugging Information

Debugging information must be compiled into applications and the NetBSD kernel in order to resolve symbols with addresses. The GNU compiler option “`-g`” will generate this information in the operating system's native format. Although the GNU debugger understands the most common formats of debugging information, using the “`-ggdb`” option will ensure GDB extensions are included. The quantity of information generated can also be specified. The GNU compiler supports three levels (1-3) with level 3 including the most information and level 2 being the default level. To compile the NetBSD kernel with maximum debugging information and guaranteed recognizable by the GNU debugger, add the following options to the kernel configuration file.

```
makeoptions DEBUG="-ggdb3"
```

Three sources of potential grief arise when the above kernel configuration line is added or modified. First, do not confuse the above kernel configuration line with the line “`options DEBUG`”. Second, “`config`” must be invoked on the kernel configuration to update the kernel configuration header files. Lastly, invoke a “`make clean`” to remove the object files generated by a previous kernel compilation.

Compiling applications with debugging information requires knowledge of how the application in compiled on the host platform. Applications taking advantage of the BSD make system or the GNU configure script can set an environment variable specifying a compiler command line option to be added for all target objects. The syntax for setting this variable in tcsh is "`setenv CFLAGS -ggdb3`" and in bash is "`export CFLAGS=-ggdb3`". If a different method of building an application is used, examining the make file and hand tuning it will probably be necessary.

Modifying the BDI2000 Configuration File

The BDI2000 must be made aware of the location of the symbol "`bdi2k_ttbase`" and the kernel base address. Execute the command "`nm netbsd | grep bdi2k_ttbase`" to find the address of the former. If the kernel was modified and compiled correctly, this command will display the symbol with an address beside it. The command "`grep –R TEXTBASE`" executed from the base directory of the port, i.e. "`src/sys/arch/foo`", should reveal the base address of the kernel. Using the two addresses gleaned from the above commands; add the following two lines to the BDI2000's configuration file in the [HOST] section.

```
MMU XLAT <kernel base address>
PTBASE <bdi2k_ttbase address>
```

In the [TARGET] section of the configuration file add the line "`BREAKMODE HARD`" to force the BDI2000 to use hardware breakpoints. Finally, reboot the BDI2000 by cycling power or typing "`boot`" at the telnet prompt to reload the configuration file and its changes.

Using the Pseudo-Device

The pseudo-device is initialized early in the kernel's booting process and will by default add the kernel's translation tables to the translation table array. However, when debugging the booting procedure of the kernel this may not be soon enough. In this case, the kernel's translation table should be added to the translation table array immediately after initialization of the MMU by calling the function "`bdi2k_init`".

The existence of the pseudo-device is mandated solely by our desire to debug applications; it gives a simple interface for applications to modify two pointers protected by the kernel. An application in which debugging support is desired can open the pseudo-device and have its translation tables added to the array by calling "`ioctl`". The file "`bdi2000.h`" contains examples of how to accomplish this in an application. Note that in order to open the device, a special file must be created in the "/dev" directory. Using the major number allocated in the file "`majors.foo`", execute the command "`mknod /dev/bdi2k c <major number> 0`".

To further simplify the use of the pseudo-device, a server application has been written to abstract any calls to "`ioctl`" and avoid recompilation of an application. The server is executed with the path and command line parameters of a desired application; a copy of the server process is then created using the "`fork`" system call. The new process opens the bdi2k pseudo-device, calls "`ioctl`" to add the newly created translation tables to the translation table array, and executes the "`exec`" system call to replace its current memory map with that of the desired application. In the mean time the original server process blocks on termination of the new process, at which time it will clean up the translation table array by removing the added address.

Two constraints exist when debugging applications with the BDI2000. Since hardware breakpoints completely freeze the system, some network applications may behave abnormally or cease working altogether. Compare this with a software solution; using kernel services, the application being debugged can be halted without interfering with other applications or the kernel. Both debugging solutions have merit when working on embedded systems, and discretion must be used to select the correct solution for each case.

The second constraint relates with the addresses used to trigger hardware breakpoints. Recall that disjointedness is the key difference between the virtual addresses used by the kernel and an application. Two applications do not share this luxury and are given the same base virtual addresses at link time. When a hardware breakpoint is set, it is impossible to distinguish between two instructions that come from different applications. An embedded system using a binary created by "crunchgen" solves this problem because all applications are linked together into a single binary and will naturally be given different virtual addresses. If more than one application binary exists on the system it may be necessary to recompile the offending application before being debugged. Using the linker option "`--image-base`" the base virtual address of an application can be changed to make it disjoint from other applications running on the system. If the GNU compiler being used does not support this option due to its version or configuration, a slightly modified linker script can be used. Executing "`ld --verbose > script_shift`" will dump the contents of the default linker script to a file. After removing extraneous lines from the beginning and end, on or about the sixth line should resemble something of the form "`. + 0x800000 :`". The number is the base virtual address and should be shifted down a value greater than the size of the program produced by "size <myprogram>". Finally, to use the new linker script pass the command line option "`-Wl,-Tscript_shift`" to the compiler.

Example 1: Stepping through the NetBSD kernel's main()

The target platform for this example is a Freescale MPC875 based single board computer and the NetBSD port is an unofficial one from Japan. The kernel was modified as described above including calling the function "`init_bdi2k_ttptrs`" early in the booting procedure. A raw binary image of the kernel was created by executing "objcopy –O binary netbsd netbsd.bin" and the resulting bits were loaded into RAM from the

BDI2000 telnet prompt by executing “load 0x10000 netbsd.bin bin”. On the host system, the following commands were issued to GDB to connect it with the BDI2000.

```
(gdb) sym netbsd
Reading symbols from netbsd...done.
(gdb) target remote mybdi:2001
Remote debugging using mybdi:2001
0x00010000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x8008370c: file ../../../../kern/init_main.c,
line 232.
(gdb) continue
Continuing.
```

As shown above, a breakpoint was added for “`main`” before starting execution of the kernel. Also note that the “`continue`” command was given to GDB to start execution as opposed to invoking the “run” command. The results of triggering the breakpoint as well as a few debugging commands are shown below.

```
Breakpoint 1, main () at ../../../../kern/init_main.c:232
232             l = &lwp0;
 (gdb) list 298
293     #endif
294
295             /*
296              * Create process 0 (the swapper).
297              */
298             p = &proc0;
299             proc0_insert(p, l, &pgrp0, &session0);
300
301             /*
302              * Set P_NOCLDWAIT so that kernel threads are
reparented to
(gdb) b 298
Breakpoint 2 at 0x80083800: file ../../../../kern/init_main.c,
line 298.
(gdb) c
Continuing.

Breakpoint 2, main () at ../../../../kern/init_main.c:298
298             p = &proc0;
(gdb)  frame
#0  main () at ../../../../kern/init_main.c:298
298             p = &proc0;
```

Example 2: Debugging a factorial application

A factorial program was written and cross-compiled for the target embedded system. The BSD-style makefile and source listing are shown below. After rebooting the BDI2000 and the target system, the NetBSD kernel was loaded into RAM and executed. On the

host system, the following commands were issued to GDB to connect it with the BDI2000.

```
(gdb) sym factorial
Reading symbols from factorial…done.
(gdb) target remote mybdi:2001
Remote debugging using mybdi:2001
0x800100f0 in ?? ()
(gdb) list factorial.c:1
1  #include <stdio.h>
2
3  int f(int x)
4  {
5      if (x == 2)
6          return 2;
7      else
8          return f(x-1);
9  }
10
(gdb) b 6
Breakpoint 1 at 0x1800580: file factorial.c, line 6.
(gdb) c
Continuing.
```

On the target system, the factorial application was executed with the BDI2000 server application using the command "`bdi2ksvr /tmp/factorial`". When prompted for a number, "`5`" was entered. The breakpoint was triggered and upon inspecting the stack a series of calls to our recursive function was produced.

```
Breakpoint 1, f(x=2) at factorial.c:6
6          return 2;
(gdb) bt
#0 f(x=2) at factorial.c:6
#1 0x0180059c in f(x=3) at factorial.c:8
#2 0x0180059c in f(x=4) at factorial.c:8
#3 0x0180059c in f(x=5) at factorial.c:8
#4 0x01800600 in main() at factorial.c:16
#5 0x018001b0 in _start()
#6 0x00000001 in ?? ()
```

```
NOMAN=

.include <bsd.own.mk>

PROG= factorial
SRCS= factorial.c

CFLAGS+= -ggdb3 -O0 –fno-inline
LDSTATIC= -static

.include <bsd.prog.mk>
```

```
1  #include <stdio.h>
2
3  int f(int x)
4  {
5      if (x == 2)
6          return 2;
7      else
8          return f(x-1);
9  }
10
11 int main(void)
12 {
13     int x;
14     printf("Enter a number: ");
```

```
15     scanf(“%d”, &x);
16     printf(“The factorial of %d
is %d\n”, x, f(x));
17     return 0;
18 }
```

Conclusion

We have seen how the BDI2000 can be used to debug the kernel and applications in NetBSD. Although the source code provided is specific to NetBSD, the concepts can be applied to other operating systems that fall under the umbrella of embedded UNIX. Also, the code provided is meant to be an example and is by no means industrial strength. However, with a few modifications and added features to the pseudo-device and server application, the BDI2000 may find itself being cleverly used in situations it was not intended for.

Appendix A - BDI2000 Pseudo-Device

Source Listing for “bdi2k.h”

```
#ifndef BDI2K_H
#define BDI2K_H

/*  Using the BDI2000 pseudo-device
 *  ---------------------------------------
 *  1. fd = open("/dev/bdi2k", 0, O_RDONLY);
 *  2. ioctl(fd, BDI2K_SETUSER, 0);
 *  3. Debug the program
 *  4. ioctl(fd, BDI2K_CLRUSER, 0);
 *  5. close(fd);
 */

#define BDI2K_SETUSER _IO('b', 3)
#define BDI2K_CLRUSER _IO('b', 5)

#ifdef _KERNEL

/*  This routine must be implemented per architecture and must
 *  be called after creating the kernels page tables.
 */
void init_bdi2k_ttptrs(void);

#endif /* _KERNEL */

#endif /* BDI2K_H */
```

Source Listing for “bdi2k.c”

```
/*-
 * Copyright (c) 2004 Jared Momose.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote products
 *    derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


#include <sys/param.h>
#include <sys/systm.h>
#include <sys/device.h>
```



```
#include <sys/proc.h>
#include <sys/conf.h>
#include <sys/ioctl.h>

#include <uvm/uvm.h>
#include <machine/pmap.h>
#include <machine/spr.h>

#include <machine/bdi2k.h>

struct bdi2k_softc {
    struct device sc_dev;
};

void bdi2kattach __P((int));

dev_type_open(bdi2kopen);
dev_type_close(bdi2kclose);
dev_type_ioctl(bdi2kioctl);

const struct cdevsw bdi2k_cdevsw = {
    bdi2kopen, bdi2kclose, noread, nowrite, bdi2kioctl,
    nostop, notty, nopoll, nommap, 0
};

/*  These are pointers to two page tables - one for the kernel
 *  and one for a user process.
 */
void *bdi2k_ttptrs[2];

void bdi2kattach(int n)
{
    /*  This needs to be called earlier on (possibly in locore.S)
     *  if kernel debugging is to take place before this device is
     *  attached.
     */
    init_bdi2k_ttptrs();
}

int bdi2kopen(dev_t dev, int oflags, int devtype, struct proc *p)
{
    return 0;
}

int bdi2kclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
    return 0;
}

int bdi2kioctl(dev_t dev, u_long cmd, caddr_t data, int fflag,
               struct proc *p)
{
    int error = 0;
    uint32_t utt;


    switch (cmd)
    {
    case BDI2K_SETUSER:
        /*  TODO: Architecture specific code to the user
         *        translation tables to the array should be added
         *        here.
         *        here.
         */

        /*  Since this is a system call, the M_TWB register will
         *  reflect the translation table of the calling process.
         */
        asm volatile ("mfspr %0,M_TWB" : "=r"(utt));
        bdi2k_ttptrs[1] = (void *)((utt & 0xfffff000) | 0x80000000);
        break;
```



```
    case BDI2K_CLRUSER:
        bdi2k_ttptrs[1] = (void *) 0;
        break;
    default:
        error = ENOTTY;
        break;
    }

    return error;
}

void init_bdi2k_ttptrs(void)
{
    extern void *bdi2k_ttbase;
    uint32_t ktt;

    bdi2k_ttbase = (void *) bdi2k_ttptrs;

    /*  TODO: Architecture specific code to add the kernel
     *        translation tables to the array should be added
     *        here.
     */
    asm volatile ("mfspr %0,M_TWB" : "=r"(ktt));
    bdi2k_ttptrs[0] = (void *)((ktt & 0xfffff000) | 0x80000000 );
    bdi2k_ttptrs[1] = (void *) 0;
}
```

Appendix B - BDI2000 Server Application

Source Listing for “bdi2ksvr.c”

```
/*-
 * Copyright (c) 2004 Jared Momose.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of the author may not be used to endorse or promote products
 *    derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <machine/bdi2k.h>

int main(int argc, char *argv[])
{
    int child, res, status, bdi2k;

    if (argc < 2) {
        printf("Usage: %s <prog> <prog_arg_1> <prog_arg_2> ...\n", argv[0]);
        return 0;
    }

    child = fork();
    if (child < 0) {
        printf("Cannot fork(), errno = %d\n", errno);
        exit(-1);
    }

    if (child == 0) {
        /*  Add this process's translation table to the bdi2k
         *  translation table array using ioctl().
         */
        bdi2k = open("/dev/bdi2k", 0, O_RDONLY);
        if (bdi2k < 0) {
            printf("Cannot open device /dev/bdi2k, errno = %d\n", errno);
            exit(-1);
        }
        res = ioctl(bdi2k, BDI2K_SETUSER, 0);
        if (res < 0) {
            printf("Cannot configure device /dev/bdi2k, errno = %d\n", errno);
            close(bdi2k);
```



```
            exit(-1);
        }

        /*  Now that the BDI2000 is aware of this process, turn ourselves
         *  into our target application to be debugged.
         */
        execv(argv[1], &argv[1]);

        /*  If exec() returns, we encountered a problem. We do not
         *  need to clean up the bdi2k device since the server
         *  will do it for us upon termination.
         */
        printf("Cannot exec() %s, errno = %d\n", argv[1], errno);
        exit(-1);

    } else if (child > 0) {
        while (1) {
            res = wait(&status);
            if (res != child)
                continue;
            if (WIFEXITED(status) || WIFSIGNALED(status)) {
                printf("Child with pid %d terminated\n", child);

                /*  Add this process's translation table to the bdi2k
                 *  translation table array using ioctl().
                 */
                bdi2k = open("/dev/bdi2k", 0, O_RDONLY);
                if (bdi2k < 0) {
                    printf("Cannot open device /dev/bdi2k, errno = %d\n", errno);
                    exit(-1);
                }
                res = ioctl(bdi2k, BDI2K_CLRUSER, 0);
                if (res < 0) {
                    printf("Cannot configure device /dev/bdi2k, errno = %d\n", errno);
                    close(bdi2k);
                    exit(-1);
                }
                return 0;
            }
        }
    }

    /* WE SHOULD NEVER GET HERE */
    return 0;
}
```


Short Biography

Jared Momose has been twiddling bits in embedded systems for about 6 years. He is currently a researcher at American Power Conversion investigating the latest trends in embedded communications. He holds a Bachelor of Science degree in Electrical Engineering from St.Cloud State University and a Master of Science degree in Computer Engineering from the University of Minnesota – Twin Cities.

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