Using a JTAG in Linux
Driver Debugging

Supporting New Hardware

Mike Anderson
Chief Scientist
The PTR Group, Inc.
http://www.theptrgroup.com
What We Will Talk About

- What are we trying to do?
- Hardware debuggers
- What is JTAG?
- How does it work?
- Board bring up
- The Linux boot sequence
- Debugging the kernel and device drivers
What are we trying to do?

The board bring-up process is loaded with potential gotchas
- Obtaining data sheets may be near impossible
- The hardware may or may not be working
- The boot firmware may have restrictive licensing issues

There are two phases of device driver development that we’ll need to address
- Getting the board to work at all
- Adding features for peripherals
Porting Linux

Bringing Linux up on a new board will require some knowledge of assembly language for your processor

- There are several transitions from assembly to “C” and back if we’re using zImages

Debugging at this level will require the use of JTAGs, or other hardware assistance

- Never underestimate the power of an LED
Device Drivers in Linux

Linux has several driver types
- Character, block, network, etc.

Linux uses a formal driver model
- Drivers present a common API such as `open()`, `release()`, `read()`, `write()`, etc.

User-mode device drivers are also possible
- Via `/dev/mem`, `/dev/ioports`, etc.
- Easier to debug using standard GDB
Statically Linked – Dynamically Loaded

The typical kernel-mode driver can be statically linked into the kernel at kernel build time

- Must be GPL
- Initialized in `start_kernel()` sequence

Dynamically-loaded drivers, a.k.a. kernel modules are loaded after the kernel is booted and init is running

- Can be loaded from initramfs/initrd
- Can have proprietary licenses
Driver Initialization Sequence

- Drivers must register themselves with the kernel
  - `register_chrdev()`, `register_blkdev()`, `register_netdev()`, etc.

- For block and character drivers you’ll need to assign major/minor numbers
  - Can be done statically or dynamically
  - Coordinate with `<linux>/Documentation/devices.txt`

- You’ll need to create device nodes as well
  - Statically or via UDEV
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

#define MODULE_NAME "celf"

int __init celf_init_module(void) {
    printk("celf_init_module() called, ");
    return 0;
}

void __exit celf_cleanup_module(void) {
    printk("celf_cleanup_module() called\n");
}

module_init(celf_init_module);
module_exit(celf_cleanup_module);
Old School Driver Registration

Kernel is made aware of a character device driver when the driver registers itself

- Typically in the `__init` function

Registration makes the association between the major number and device driver

```c
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)
```
Likewise, when a device driver removes itself from the system, it should unregister itself from the kernel to free up that major number.

Typically in the \_\_exit function:

```c
int unregister_chrdev(unsigned int major, const char *name);
```
New–School Driver Registration

If you need to get beyond the 256 major limit, you’ll need to use a different approach

- This uses a different API, dev_t, cdev structures and a much more involved registration approach

All of this is beyond scope for the current discussion, however
Giving Your Driver Something to do

Character device driver exports services in file_operations structure
  - There are 26 supported operations in the 2.6 kernel
    - Up from 17 in the 2.4. kernel

You only supply those calls that make sense for your device

You can explicitly return error codes for unsupported functions or have the system return the default ENOTSUPP error

Typically, the file_operations structure is statically initialized
  - Using C99 tagged initializer format
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t,
                     loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *,
                          size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
                  unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);}
struct file_operations #2 of 2

    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
                     lloff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
                     lloff_t *);
    ssize_t (*sendfile) (struct file *, lloff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                         lloff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                         unsigned long, unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*dir_notify) (struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
};
Which File Operations do I Need?

Typically, a driver will implement:

- open()
- release()
  - a.k.a., the user-space close()
- read()
- write()
- ioctl()

Additional features like mmap(), poll(), fasync(), and flush() are nice to have:
  - You can add them at any time during development

Some methods like llseek() and readv()/writev() may not apply to your device:
  - You decide what to support and errors to return
C99 tagged initialization of the structures allows you to initialize the fields by name

- No worry about the structure layout (which may change between kernel revisions)

Un-initialized function entries in the structure shown below will be initialized to NULL

```
struct file_operations fops = {
    .read       = my_read,
    .write      = my_write,
    .ioctl      = my_ioctl,
    .open       = my_open,
    .release    = my_release
};
```
Debugging Device Drivers

- Statically-linked device drivers are notoriously difficult to debug
  - An error can cause a panic or oops before you can even get `printk()` to work
  - These will typically require a JTAG to debug them easily

- Dynamically-linked drivers are marginally easier because you can get more debugging infrastructure into place before loading them
  - The use of `read_proc()`/`write_proc()` functions and `printk()` are typical
  - JTAGs can help here too
Hardware Debugging Tools

The traditional hardware debug tool was the In–Circuit Emulator (ICE)
  - A device that plugged into the CPU socket and emulated the CPU itself

These were rather expensive
  - $30K+ for the good ones

Today, most devices that call themselves an ICE are actually JTAGs
Why the Traditional ICE has Faded Away

- The biggest problem faced by the ICE concept was the increasing pin counts of processors
  - E.g., 939 pins for the Athlon-64
- Each pin required a wire to the ICE
  - Each wire started to become an antenna as frequencies increased
- Processors also started to move to Ball Grid Array (BGA) packages
  - No way to get to the pins in the center of the part because the part is soldered to the motherboard
Enter the JTAG Port

The Joint Test Action Group (JTAG) is the name associated with the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture

- Originally introduced in 1990 as a means to test printed circuit boards
- An alternative to the bed of nails
How JTAG Works

JTAG is a boundary-scan device that allows the developer to sample the values of lines on the device

- Allows you to change those values as well

JTAG is built to allow chaining of multiple devices

- Works for multi-core processors, too
JTAG Details

JTAG is a simple serial protocol

Configuration is done by manipulating the state machine of the device via the TMS line

1. TDI (Test Data In)
2. TDO (Test Data Out)
3. TCK (Test Clock)
4. TMS (Test Mode Select)
5. TRST (Test ReSet) optional.
JTAG–Aware Processors

- Most embedded processors today support JTAG or one of its relatives like BDM
  - E.g., ARM/XScale, PPC, MIPS
- Even the x86 has a JTAG port although it is rarely wired out
  - Grandma can barely send e-mail, let alone know what to do with a JTAG port
- Some processors like MIPS come in different versions
  - Some with JTAG ports for development, some without in order to save $$$
JTAG Vendors

Several different vendors sell JTAG port interface hardware

- JTAG is also referred to as On-Chip Debugging (OCD)

Here are a few of the vendors:

- Wind River Systems (http://www.windriver.com)
- Abatron AG (http://www.abatron.ch)
- American Arium (http://www.arium.com)
- Mentor Graphics (http://www.epitools.com)

Some vendors do certain processors better than others

- MIPS will usually have a more custom EJTAG interface
JTAG Connections

The maximum speed of JTAG is 100 MHz
- A ribbon cable is usually sufficient to connect to the target

Connection to the development host is accomplished via
- Parallel port
- USB
- Serial port
- Ethernet
Some JTAG interfaces use a GDB–style software interface

- Any GDB–aware front end will work

Others have Eclipse plug-ins to access the JTAG via an IDE

Some still use a command line interface
What can you do with a JTAG?

- Typical JTAG usage includes reflashing boot firmware
  - Even the really cheap JTAG units can do this
- However, it is in the use as a debugging aid that JTAG comes into its own
  - You can set hardware or software breakpoints and debug in source code
  - Sophisticated breakpoint strategies and multi-core debugging usually require the more expensive units
- JTAG units can also be used to exercise the address bus and peripherals
  - This is what JTAG was originally designed for
Most JTAG units require you to describe the hardware registers in a configuration file
  - This is also how you describe what processor architecture you are using

All of that information about register maps that you collected earlier now goes into the configuration file

Unfortunately, there is no standard format for these configuration files
  - Each JTAG vendor uses different syntax
Example Configuration Files

Many JTAG units split the configuration files into a CPU register file and a board configuration file.

```
; ; SDRAM Controller (SDRAMC)
;
sdram_mr MM 0xFFFFFFFF90 32 ;SDRAMC Mode Register
sdram_tr MM 0xFFFFFFFF94 32 ;SDRAMC Refresh Timer Register
sdram_cr MM 0xFFFFFFFF98 32 ;SDRAMC Configuration Register
sdram_srr MM 0xFFFFFFFF9C 32 ;SDRAMC Self Refresh Register
sdram_lpr MM 0xFFFFFFFFA0 32 ;SDRAMC Low Power Register
sdram_ierr MM 0xFFFFFFFFA4 32 ;SDRAMC Interrupt Enable Register

; bdigDB configuration file for AT91RM9200-DK
; ------------------------------------------
;
[INIT]
WREG CPSR 0x0000000D3 ;select supervisor mode
WM32 0xFFFFFFFF00 0x00000001 ;Cancel reset remapping
WM32 0xFFFFFFFFC0 0x0000FF01 ;PMC_MCR : Enable main oscillator ; OSCOUNT = 0xFF
;
; Init Flash
WM32 0xFFFFFFFF10 0x00000000 ;MC_PUIA[0]
WM32 0xFFFFFFFF50 0x00000000 ;MC_PUP
WM32 0xFFFFFFFF54 0x00000000 ;MC_PUE: Memory controller protection unit disable
;WM32 0xFFFFFFFF04 0x00000000 ;MC_ASR
;WM32 0xFFFFFFFF08 0x00000000 ;MC_AASR
WM32 0xFFFFFFFF64 0x00000000 ;EBI_CFGR
WM32 0xFFFFFFFF70 0x00003284 ;SMC2_CSR[0]: 16bit, 2 TDF, 4 WS
;
; Init Clocks
WM32 0xFFFFFFFFC8 0x20263E04 ;PLLAR: 179,712000 MHz for PCK
DELAY 100
WM32 0xFFFFFFFFC2 0x10483E0E ;PLLBR: 48,054857 MHz (divider by 2 for USB)

Source: Abatron
```
Developing the Configuration File

❖ The JTAG vendor will likely already have a register file for the processor
  ▸ ARM920, PPC8241, etc.

❖ Your task will be to develop the board configuration file
  ▸ There may be a configuration file for the reference board that you can use as a starting point

❖ The configuration file is essentially a script of commands to initialize the target board
  ▸ You keep working on it until you can initialize memory
  ▸ Once memory is on-line, you should then be able to write values into memory via the JTAG that can be read back
  ▸ Then, enhance the configuration to initialize other peripherals
Linux-Aware JTAGs

There are several rather tricky transitions during the Linux booting process:
- Transitioning from flash to RAM
- Transitioning from physical addresses to kernel virtual addresses
- These transitions require the use of hardware breakpoints

Make sure that your JTAG is “Linux aware”:
- It must understand Linux’s use of the MMU to be of much use for driver debugging
The Linux Boot Sequence

Like the boot firmware, the Linux kernel starts in assembly language
- Sets up the caches, initializes some MMU page table entries, configures a “C” stack and jumps to a C entry point called `start_kernel()` (init/main.c)

`start_kernel()` is then responsible for:
- Architecture and machine-specific hardware initialization
- Initializing virtual memory
- Starting the system clock tick
- Initializing kernel subsystems and device drivers

Finally, a system console is started and the init process is created
- The init process (PID 1) is then the start of all user-space processing
JTAG and Early Kernel Debug

- An odd thing happens when the MMU is enabled
  - All of the physical addresses suddenly get translated into virtual addresses
- The kernel’s debug symbols are all built assuming a virtual address space
  - You’ll need to turn debugging symbols on in the kernel
- Consequently, while you can step through the early code by using a hardware breakpoint address, software breakpoint on symbols will only work after the MMU is enabled
  - Fortunately, this happens fairly early in the kernel initialization
- You can typically tell the JTAG to step so many instructions and then stop again
  - Step past the MMU initialization, stop and then set additional breakpoints
Configure Kernel for Debugging

Enable debugging info and rebuild the kernel

Linux Kernel v2.6.14.7-selinux1 Configuration

Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus -->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < >

[*] Spinlock debugging
[*] Sleep-inside-spinlock checking
[*] kobject debugging
[*] Compile the kernel with debug info
[*] Debug Filesystem
[*] KGDB: kernel debugging with remote gdb
[*] Verbose user fault messages
[*] Wait queue debugging
[*] Verbose kernel error messages
[*] Kernel low-level debugging functions
[*] Kernel low-level debugging via EmbeddedICE DCC channel

<Select>  < Exit >  < Help >
Loading Symbols into the JTAG UI

- Depending on the JTAG UI, you may simply have to load the kernel’s vmlinux image to be able to access the symbols by name
  - The techniques for doing this vary by JTAG vendor
- Attach the JTAG to the hardware
  - Reset the board via JTAG and hold in reset
  - Set H/W breakpoint using the JTAG
  - Load the vmlinux via the JTAG (this loads the symbols)
  - Command the JTAG to tell the hardware to “go”
- Once you encounter the hardware breakpoint, you can step in assembly until the MMU is enabled
  - The MMU will translate physical addresses to virtual addresses
  - Once virtual addressing is on, set breakpoints as normal
Using JTAG to Dump printk Buffer

If you kernel hangs right after displaying “Uncompressing Kernel Image ... OK” message...

> You probably have printk() output, but the serial console isn’t initialized yet

We can dump the printk buffer using the JTAG!

> Look in the kernel’s System.map file for something like “__log_buf”

```
$ grep __log_buf /boot/System.map
 c0445980 b __log_buf
```
Dumping printk Buffer #2

The address of the buffer is a translated kernel address
- Strip off the 0xC0000000 portion of the address to get (typically) the physical address on processors like the X86
- i.e., 0xc0445980 would typically be at physical address 0x445980
- You must understand your processor to do the translations correctly

Now, use the JTAG to dump that address
- Raw printk output, but you can get an idea of what it was doing when it crashed
- Data is still there even after reset (but not power-off)
GDB–Aware JTAGs

If the JTAG is GDB–aware, then you will be able to control it using normal GDB commands

- Attach to the JTAG via “target remote xx” command where “xx” is via Ethernet, serial or other connection between your JTAG and the host

- Use the GDB “mon” command to pass commands directly to the JTAG
**Invoked from command line with vmlinux compiled for debugging**

*Then attach to JTAG using “target remote” command*
Debugging Device Drivers

- Statically linked driver symbols are already built into the kernel’s symbol table
  - Simply set break points on the driver methods themselves
- Dynamically loaded drivers require additional steps
  - We need to find the addresses used by the driver
- The next few charts assume a GDB–aware JTAG
Debugging Loadable Modules

In order to debug a loaded module, we need to tell the debugger where the module is in memory

- The module’s information is not in the vmlinux image because that shows only statically linked drivers

How we proceed depends on where we need to debug

- If we need to debug the __init code, we need to set a breakpoint in the sys_init_module() function
Debugging Loadable Modules #2

- We’ll need to breakpoint just before the control is transferred to the module’s `__init`
  - Somewhere around line 1907 of module.c
- Once the breakpoint is encountered, we can walk the module address list to find the assigned address for the module
  - We then use the `add-symbol-file` GDB command to add the debug symbols for the driver at the address for the loaded module
  - E.g.,
    ```
    add-symbol-file ./mydriver.ko 0x<addr> -e .init.text
    ```
Now, you can set breakpoints via the GDB commands to the JTAG and tell the system to continue until a breakpoint in encountered.
What if the __init is Working?

If you do not need to debug the __init code, then load the driver and look in the /sys/modules/<module name>/sections/.text for the address of the text segment

Next, use the add-symbol-file command again, but use the .text address and omit the “-e .init.text”

- Set your breakpoints and continue
User–Space Addresses

Within Linux, each user–space application occupy the same virtual address space

- The address spaces are physically different, but the addresses overlap
JTAG Confusion

- JTAGs normally run in what is called halt mode debugging
  - The entire processor is stopped when a given breakpoint address is accessed
- This works reasonably well in kernel space
  - Only one kernel address space
- While it is possible to debug user applications with the JTAG, the JTAG can get confused by seeing the same virtual address in different applications due to context switches
  - This requires run mode support for the JTAG
Run–Mode Support

Using a debugging agent in user space and register support like the ARM’s Debug Communications Channel (DCC) we can associate a virtual address to a particular context

- This allows the breakpoint to only stop the one application instead of any application that matches the address

Only a few JTAGs support this run mode debugging mechanism

- Otherwise, we are left with normal GDB process trace (ptrace) debugging control via an application like gdbserver

Naturally, GDB already does a reasonable job for user–space debugging

- The need to use JTAG for user–space debug is rare
Summary

🌟 Hardware debuggers such as JTAG are invaluable for exercising new hardware
  - They let us test address lines and registers
🌟 Once we can configure the board via the JTAG, we then take that info and use it to port the boot firmware
  - We can usually burn the boot firmware into flash via the JTAG as well
🌟 Once the boot firmware is loading Linux, the JTAG can then help again in early kernel debugging and device driver debugging
🌟 Don’t start your next bring-up project without one!
🌟 Demo time...