

Using OpenOCD JTAG in Android Kernel Debugging

Making Android Drivers Work

Mike Anderson

Chief Scientist

The PTR Group, Inc.

<http://www.theptrgroup.com>

THE LINUX FOUNDATION
ANDROID™
BUILDERS
SUMMIT



What We Will Talk About

- ✚ Device drivers in Android
- ✚ The JTAG interface
- ✚ Types of JTAG interfaces
- ✚ How to use a JTAG to debug the kernel
- ✚ OpenOCD project
- ✚ Getting and installing OpenOCD
- ✚ Starting OpenOCD
- ✚ Connecting GDB
- ✚ Debugging Kernel Code w/ OpenOCD

Device Drivers in Android

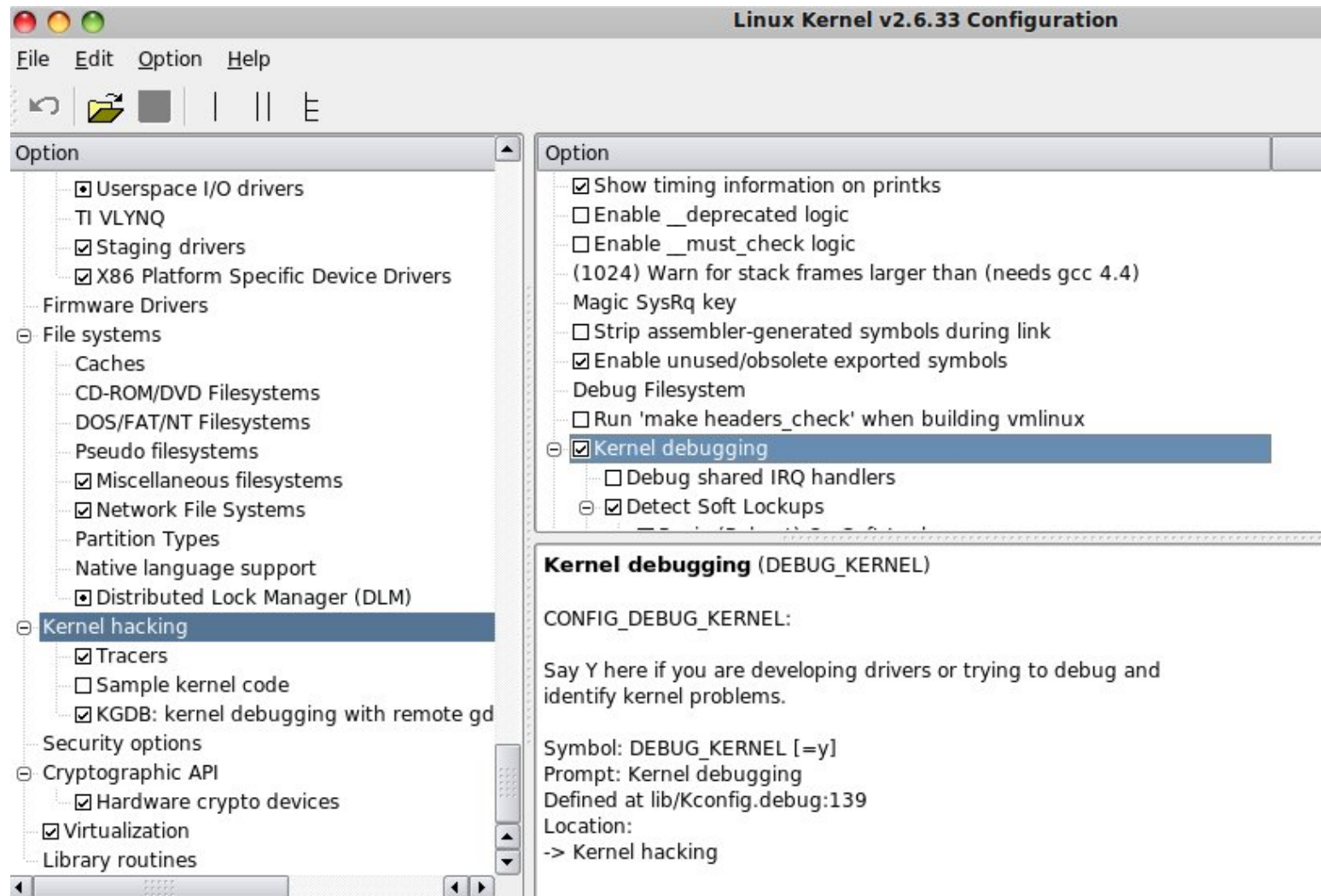
- ✦ Thanks to the use of the Linux kernel, Android has several driver types
 - ▶ Character, block, network, etc.
- ✦ Android uses a formal driver model
 - ▶ Drivers present a common API such as `open()`, `release()`, `read()`, `write()`, etc.
- ✦ User-mode device drivers are also possible
 - ▶ Via `/dev/mem`, etc.
 - ▶ Easier to debug using standard GDB

Basic Driver Debugging Requirements

- ✦ In order to be able to debug a driver under Android, you'll need to add some support to your platform
 - ▶ You must have it rooted
 - You have to be able to load/unload the drivers
 - Can be done via adb
 - ▶ You'll need to have busybox installed
 - You need several utilities provided by busybox
 - ▶ You'll need to compile the kernel with debugging enabled

Configure Kernel for Debugging

✦ Enable debugging info and rebuild the kernel



The Kernel Boot Sequence

- ✦ Like the boot firmware, the 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
- ✦ Kernel modules can now be dynamically loaded

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

Example Loadable Module

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL and additional rights");
MODULE_AUTHOR( "Driver_Author@someplace.org" );
MODULE_DESCRIPTION( "My first driver!" );

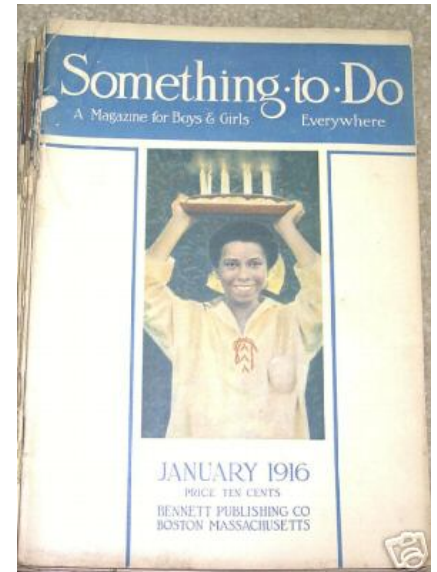
int __init mymodule_init_module(void) {
    printk(KERN_DEBUG "mymodule_init_module() called, ");
    return 0;
}

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

module_init(mymodule_init_module);
module_exit(mymodule_cleanup_module);
```


Giving Your Driver Something to do

- ✦ Character device driver exports services in `file_operations` structure
 - ▶ There are 25 supported operations in the 2.6/3.x kernel
 - The function list has changed since early 2.6 kernels
- ✦ 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



Source: tiger.uic.edu

struct file_operations #1 of 2

```
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 (*write) (struct file *, const char __user *,
                     size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                        unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                          unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    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 *, fl_owner_t id);
```

struct file_operations #2 of 2

```
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t, loff_t, 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 (*sendpage) (struct file *, struct page *, int,
                    size_t, loff_t *, int);
unsigned long (*get_unmapped_area) (struct file *,
                                   unsigned long, unsigned long, unsigned long,
                                   unsigned long);
int (*check_flags) (int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write) (struct pipe_inode_info *,
                        struct file *, loff_t *, size_t, unsigned int);
ssize_t (*splice_read) (struct file *, loff_t *,
                       struct pipe_inode_info *, size_t, unsigned int);
int (*setlease) (struct file *, long, struct file_lock **);
long (*fallocate) (struct file *file, int mode, loff_t offset,
                 loff_t len);
};
```

Initializing the file_operations

- ✖ 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,  
    .compat_ioctl  = my_ioctl,  
    .open           = my_open,  
    .release        = my_release  
};
```

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

```
int register_chrdev(unsigned int major,  
    const char *name, struct file_operations *fops)
```



Source: tecbd.asu.edu

Old School Driver Registration #2

✦ 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:

```
int unregister_chrdev(unsigned
    int major, const char *name);
```



Source: midnight-ride.com

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



Source: flickr.com

Statically Linked – Dynamically Loaded

- ✦ The typical kernel-mode driver can be statically linked into the kernel at kernel build time
 - ▶ Must be licensed under GPL
 - ▶ Initialized in `start_kernel()` sequence
- ✦ Dynamically-loaded drivers, a.k.a. kernel modules, are loaded after the kernel is booted
 - ▶ Typically during the `init.rc` script
 - ▶ Can have proprietary licenses

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

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



Source: Test Electronics

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

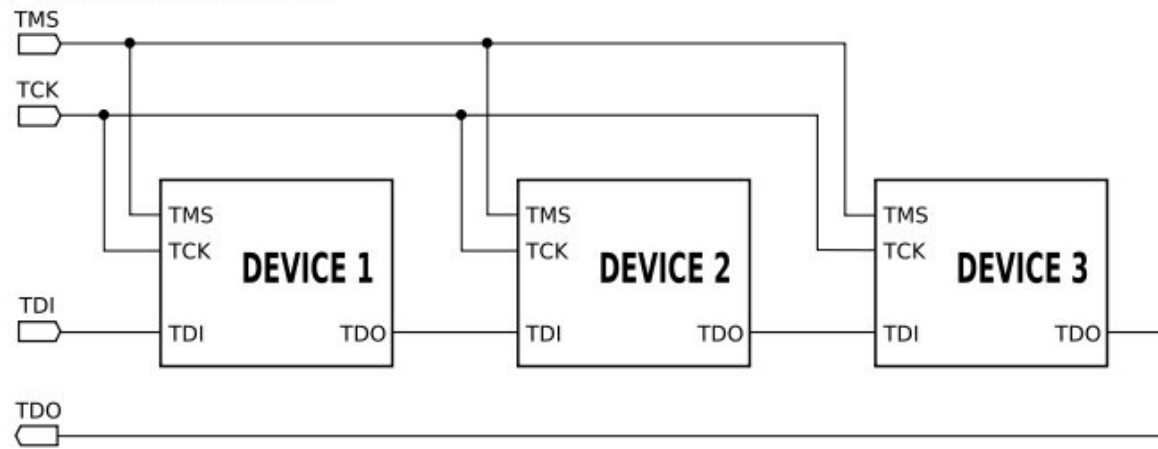
JTAG Details

✦ JTAG is a simple serial protocol

▶ Enables the use of “wiggler”–style interfaces

✦ 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
 - ▶ Intel Atom-based processors typically support the XDB port
 - ▶ Other x86-based processors may need ITP-700 connectors or interposer boards
- ✦ Some processors like MIPS come in different versions
 - ▶ Some with JTAG ports for development, some without in order to save \$\$\$

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

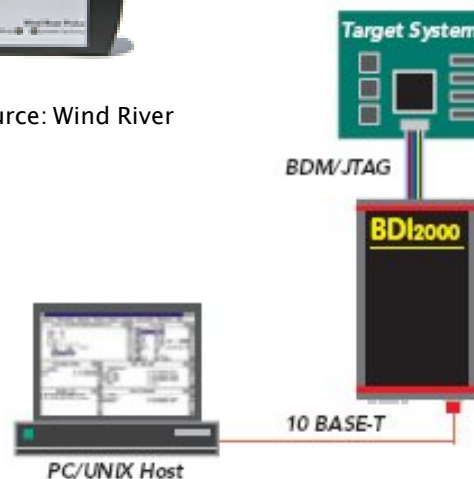
- ▶ Parallel port
- ▶ USB
- ▶ Serial port
- ▶ Ethernet



Source: Wind River



Source: Macraigor



Source: Abatron

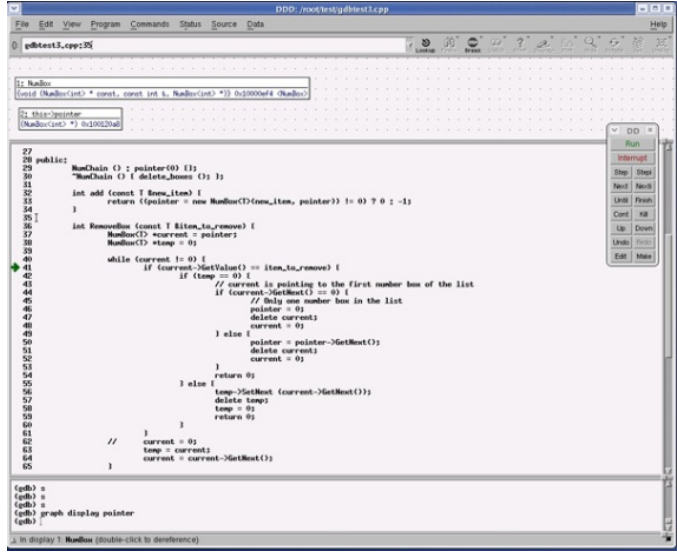
JTAG User Interface

✚ 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

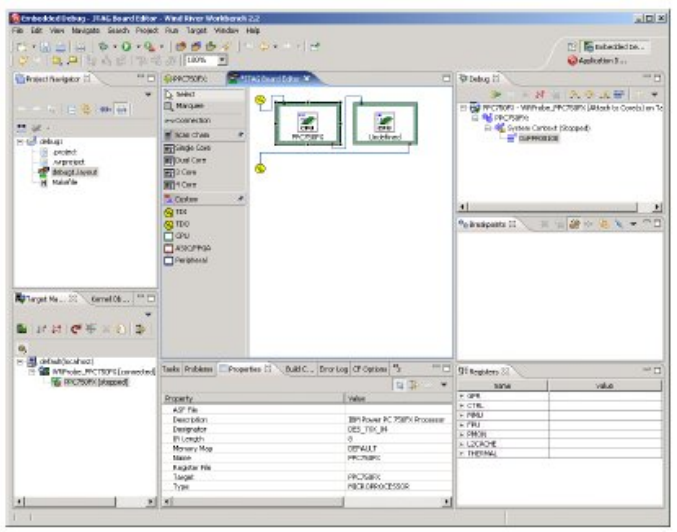


```
1: Node
const Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}

2: Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}

27 public:
28 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
29 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
30 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
31 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
32 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
33 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
34 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
35 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
36 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
37 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
38 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
39 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
40 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
41 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
42 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
43 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
44 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
45 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
46 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
47 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
48 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
49 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
50 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
51 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
52 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
53 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
54 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
55 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
56 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
57 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
58 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
59 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
60 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
61 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
62 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
63 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
64 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
65 Node(int * const, const int &, Node(int *) = 0) : mNext(0) {}
```

Source: ibm.com



The screenshot shows the Eclipse IDE interface. On the left, there's a Project Navigator showing a project named 'JTAG'. The main editor area displays a hardware diagram with components like 'PC104', 'PC104', and 'PC104'. On the right, there's a Debug Console showing system output. At the bottom, there's a Properties window showing details for the 'PC104' component, including fields like 'Name', 'Type', and 'Value'.

Source: WindRiver.com



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

Hardware Configuration Files

- ✦ 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)
;/
sdramc_mr      MM      0xFFFFFFFF90    32      ;SDRAMC Mode Register
sdramc_tr      MM      0xFFFFFFFF94    32      ;SDRAMC Refresh Timer Register
sdramc_cr      MM      0xFFFFFFFF98    32      ;SDRAMC Configuration Register
sdramc_srr     MM      0xFFFFFFFF9C    32      ;SDRAMC Self Refresh Register
sdramc_lpr     MM      0xFFFFFFFFA0    32      ;SDRAMC Low Power Register
sdramc_ier     MM      0xFFFFFFFFA4    32      ;SDRAMC Interrupt Enable Register

*****
#
# The following are defines for the System Control register addresses.
#
*****

set SYSCTL_DID0      0x400FE000 ;# Device Identification 0
set SYSCTL_DID1      0x400FE004 ;# Device Identification 1
set SYSCTL_DC0       0x400FE008 ;# Device Capabilities 0
set SYSCTL_DC1       0x400FE010 ;# Device Capabilities 1
set SYSCTL_DC2       0x400FE014 ;# Device Capabilities 2
set SYSCTL_DC3       0x400FE018 ;# Device Capabilities 3
set SYSCTL_DC4       0x400FE01C ;# Device Capabilities 4
set SYSCTL_DC5       0x400FE020 ;# Device Capabilities 5
set SYSCTL_DC6       0x400FE024 ;# Device Capabilities 6
set SYSCTL_DC7       0x400FE028 ;# Device Capabilities 7
set SYSCTL_DC8       0x400FE02C ;# Device Capabilities 8 ADC
                        ;# Channels
set SYSCTL_PBORCTL   0x400FE030 ;# Brown-Out Reset Control
set SYSCTL_LDOPCTL   0x400FE034 ;# LDO Power Control
set SYSCTL_SRCR0     0x400FE040 ;# Software Reset Control 0
set SYSCTL_SRCR1     0x400FE044 ;# Software Reset Control 1
```

Android-Aware JTAGs

- ✦ There are several rather tricky transitions during the Android booting process
 - ▶ Transitioning from flash to RAM
 - ▶ Transitioning from physical addresses to kernel virtual addresses
- ✦ These transitions require the use of hardware breakpoints
 - ▶ Both your processor and the JTAG unit need to support hardware breakpoints to debug these transitions
- ✦ Make sure that your JTAG is “MMU aware”
 - ▶ It must understand Android/Linux’s use of the MMU to be of much use for driver debugging

OpenOCD Project

- ✖ This project was started in 2008 to create a software interface for the inexpensive wiggler-style interfaces
 - ▶ Based on a graduate thesis paper
- ✖ Original targets where lower-end ARM MCUs
 - ▶ ARM7TDMI/ARM9TDMI
- ✖ Now supports many high-end ARM processors such as TI Davinci and Cortex A8
 - ▶ No working multi-core yet
- ✖ Currently hosted as a GIT repository at <http://sourceforge.net/projects/openocd>

Supported Operating Systems

✦ OpenOCD is available for the three major O/S platforms

- ▶ Linux, Windows, OS/X

✦ OS/X installation can be found here:

- ▶ <http://www.ethernut.de/elektor/tools/unix/openocdosx.html>

✦ Windows installation can be found here:

- ▶ <http://www.ethernut.de/elektor/tools/win/openocdwin.html>

- ▶ Based on the YAGARTO project

- Windows ARM tool chain
- <http://www.yagarto.de/>
- Includes Eclipse support

✦ We'll show you the basics of Linux installation next

Getting/Installing OpenOCD – Linux

✚ OpenOCD is available as an anonymous git clone request:

```
git clone git://openocd.git.sourceforge.net/gitroot/openocd/openocd
```

✚ Once retrieved, configure and build (OpenOCD 5.0)

```
cd openocd  
./bootstrap  
./configure --enable-maintainer-mode  
make  
sudo make install
```

✚ You'll need to enable the interface that your JTAG uses

Example Supported Interfaces

OpenOCD supports over 50 different JTAG interfaces

```
altera-usb-blaster.cfg      hilscher_nxhx50_etm.cfg    olimex-jtag-tiny.cfg
arm-jtag-ew.cfg            hilscher_nxhx50_re.cfg     oocdlink.cfg
arm-usb-ocd.cfg           hitex_str9-comstick.cfg    openocd-usb.cfg
at91rm9200.cfg            icebear.cfg                openrd.cfg
axm0432.cfg               jlink.cfg                  parport.cfg
buspirate.cfg             jtagkey2.cfg               parport_dlc5.cfg
calao-usb-a9260-c01.cfg   jtagkey2p.cfg              rlink.cfg
calao-usb-a9260-c02.cfg   jtagkey.cfg                sheevaplug.cfg
calao-usb-a9260.cfg       jtagkey-tiny.cfg           signalyzer.cfg
chameleon.cfg             kt-link.cfg                 signalyzer-h2.cfg
cortino.cfg               lisa-1.cfg                  signalyzer-h4.cfg
dummy.cfg                 luminary.cfg                signalyzer-lite.cfg
flashlink.cfg             luminary-icdi.cfg           stm32-stick.cfg
flossjtag.cfg             luminary-lm3s811.cfg        turtelizer2.cfg
flossjtag-noeeprom.cfg   neodb.cfg                   usb-jtag.cfg
flyswatter.cfg            ngxtech.cfg                 usbprog.cfg
hilscher_nxhx10_etm.cfg   olimex-arm-usb-ocd.cfg      vpaclink.cfg
hilscher_nxhx500_etm.cfg  olimex-arm-usb-ocd-h.cfg    vsllink.cfg
hilscher_nxhx500_re.cfg   olimex-arm-usb-tiny-h.cfg   xds100v2.cfg
```


Supported Target CPUs

 The ARM CPU support is also quite good

aduc702x.cfg	at91sam9261.cfg	hilscher_netx10.cfg	lpc2478.cfg	samsung_s3c6410.cfg
amd37x.cfg	at91sam9263.cfg	hilscher_netx500.cfg	lpc2900.cfg	sharp_lh79532.cfg
ar71xx.cfg	at91sam9.cfg	hilscher_netx50.cfg	lpc2xxx.cfg	smdk6410.cfg
at32ap7000.cfg	at91sam9g10.cfg	icepick.cfg	lpc3131.cfg	smp8634.cfg
at91r40008.cfg	at91sam9g20.cfg	imx21.cfg	lpc3250.cfg	spear3xx.cfg
at91rm9200.cfg	at91sam9g45.cfg	imx25.cfg	mc13224v.cfg	stellaris.cfg
at91sam3sXX.cfg	at91sam9rl.cfg	imx27.cfg	nuc910.cfg	stm32.cfg
at91sam3u1c.cfg	atmega128.cfg	imx31.cfg	omap2420.cfg	stm32xl.cfg
at91sam3u1e.cfg	avr32.cfg	imx35.cfg	omap3530.cfg	str710.cfg
at91sam3u2c.cfg	c100.cfg	imx51.cfg	omap4430.cfg	str730.cfg
at91sam3u2e.cfg	c100config.tcl	imx.cfg	omap5912.cfg	str750.cfg
at91sam3u4c.cfg	c100helper.tcl	is5114.cfg	omapl138.cfg	str912.cfg
at91sam3u4e.cfg	c100regs.tcl	ixp42x.cfg	pic32mx.cfg	swj-dp.tcl
at91sam3uXX.cfg	cs351x.cfg	lpc1768.cfg	pxa255.cfg	test_reset_syntax_error.cfg
at91sam3XXX.cfg	davinci.cfg	lpc2103.cfg	pxa270.cfg	test_syntax_error.cfg
at91sam7se512.cfg	dragonite.cfg	lpc2124.cfg	pxa3xx.cfg	ti_dm355.cfg
at91sam7sx.cfg	dsp56321.cfg	lpc2129.cfg	samsung_s3c2410.cfg	ti_dm365.cfg
at91sam7x256.cfg	epc9301.cfg	lpc2148.cfg	samsung_s3c2440.cfg	ti_dm6446.cfg
at91sam9260.cfg	faux.cfg	lpc2294.cfg	samsung_s3c2450.cfg	tmpa900.cfg
at91sam9260_ext_RAM_ext_flash.cfg	feroceon.cfg	lpc2378.cfg	samsung_s3c4510.cfg	tmpa910.cfg

What can OpenOCD do?

- ✦ In spite of the fact that it's free, OpenOCD is really quite full featured
- ✦ It supports erasing/programming NOR/NAND flash segments
- ✦ Loading and debugging code on the supported targets
- ✦ Accessing registers on the target processor
- ✦ Working with PLD/FPGA devices
- ✦ Adding extensions via Tcl interface

Running OpenOCD

- ✦ OpenOCD uses a stripped version of Tcl (JIM-Tcl) at its core
 - ▶ Knowing Tcl isn't required, but it sure helps
- ✦ OpenOCD uses a daemon to front end the JTAG hardware interface
 - ▶ You can access it via telnet at port 4444
- ✦ The OpenOCD command line allows you to pass configuration files
 - ▶ E.g., `openocd -f interface/flyswatter.cfg \`
`-f board/ti_beagleboard_xm.cfg`

The Telnet Interface

- ✦ From the telnet session you can exercise a lot of control over the target
 - ▶ reset, halt, load code, access registers, dump/change memory, set breakpoints/watchpoints, single-step instructions and much more
- ✦ The telnet session also supports file I/O for loading and dumping memory and Tcl commands for scripting

GDB and OpenOCD

- ✦ GDB can connect to OpenOCD daemon via “target remote” command to port 3333
 - ▶ Another option is to use Linux pipes
- ✦ Supports the use of various GDB front-ends such as DDD, Eclipse, SlickEdit, Nemiver and others
- ✦ Use the GDB “mon” command to pass a command to the OpenOCD daemon
 - ▶ E.g., **mon mdw 0x2100000** to dump memory at 0x2100000

DDD GUI Front-End Example

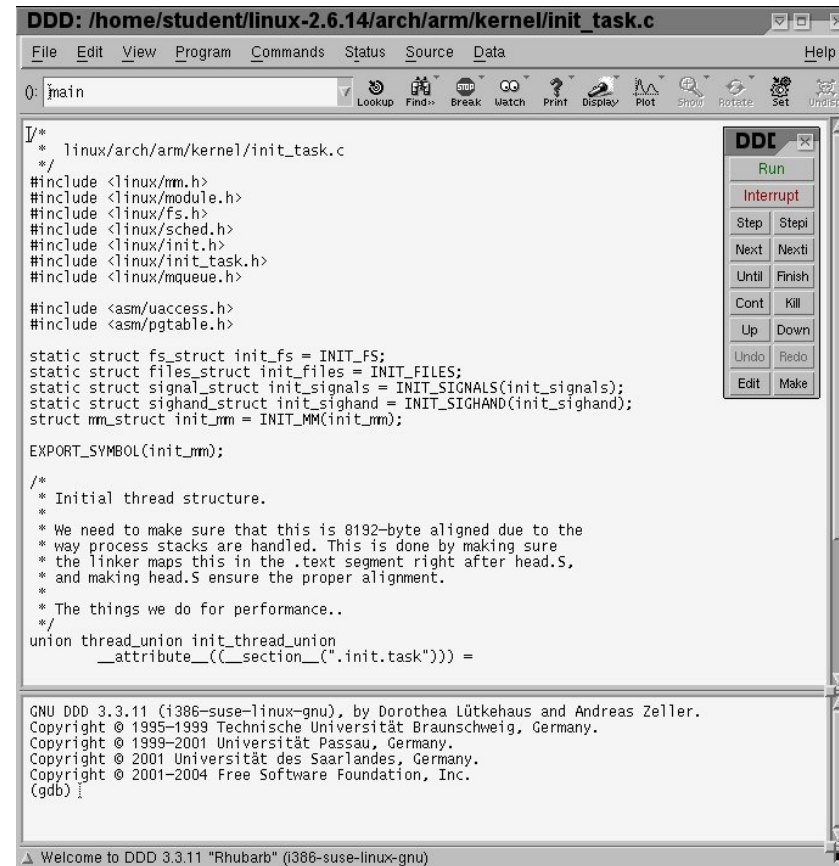
 Invoked from command line with kernel compiled for debugging

- ▶ Use the `-debugger` command line option to load the cross debugger back end:

```
ddd -debugger  
arm-linux-gdb vmlinux
```

 Then attach to JTAG using “target remote” command:

```
(gdb) target remote 127.0.0.1:3333
```



```
DDD: /home/student/linux-2.6.14/arch/arm/kernel/init_task.c
File Edit View Program Commands Status Source Data Help
0: |main
/*
 * linux/arch/arm/kernel/init_task.c
 */
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/init_task.h>
#include <linux/mqueue.h>

#include <asm/uaccess.h>
#include <asm/pgtable.h>

static struct fs_struct init_fs = INIT_FS;
static struct files_struct init_files = INIT_FILES;
static struct signal_struct init_signals = INIT_SIGNALS(init_signals);
static struct sighand_struct init_sighand = INIT_SIGHAND(init_sighand);
struct mm_struct init_mm = INIT_MM(init_mm);

EXPORT_SYMBOL(init_mm);

/*
 * Initial thread structure.
 * We need to make sure that this is 8192-byte aligned due to the
 * way process stacks are handled. This is done by making sure
 * the linker maps this in the .text segment right after head.S,
 * and making head.S ensure the proper alignment.
 * The things we do for performance..
 */
union thread_union init_thread_union
__attribute__((section("__init.task"))) =

GNU DDD 3.3.11 (i386-suse-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(gdb) |
```

Adding Device Driver Symbols

- ✦ 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 you're using OpenOCD

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 kernel image because that shows only statically-linked drivers
- ✦ This information can typically be found in
 - `/proc/modules` OR
 - `/sys/module/<modulename>/sections/.text`
- ✦ We then use the `add-symbol-file` GDB command to add the debug symbols for the driver at the address for the loaded module
 - ▶ `(gdb) add-symbol-file ./mydriver.o 0x<addr>`
- ✦ How we proceed depends on where we need to debug

Debugging Loadable Modules #2

- ✖ If we need to debug the `__init` code, we need to set a breakpoint in the `load_module()` function
- ✖ We'll need to breakpoint just before the control is transferred to the module's `__init`
 - ▶ Somewhere around line 2981 of `module.c`:

```
/* Start the module */  
if (mod->init != NULL)  
    ret = do_one_initcall(mod->init);
```
- ✖ Once the breakpoint is encountered, we can walk the module address list to find the assigned address for the module

Adding Additional Breakpoints

✖ Once you've added the module's symbols, you can set breakpoints at the various entry points of the driver

```
(gdb) b mydriver_read
```

✖ Other good breakpoint locations include:

- ▶ `sys_sync`

- ▶ `panic`

- ▶ `oops_enter`

✖ When you hit the breakpoint, the debugger will drop to the source code and you can start single stepping code

Summary

- ✦ Despite its low cost, OpenOCD and simple wiggler-style JTAG interfaces make a powerful combination
 - ▶ Support for a wide-variety of processors
 - ▶ Support for Flash erasing/programming
 - ▶ Supports debugging Linux kernel code using standard GDB interface and techniques
- ✦ Unfortunately, there is no multi-core support in OpenOCD yet
- ✦ However, for the developer on a budget, OpenOCD is a great option