Using a JTAG in Linux Bring-up and Kernel Debugging

Porting Linux to new Hardware

Mike Anderson
Chief Scientist
The PTR Group, Inc.
http://www.theptrgroup.com

What We Will Talk About

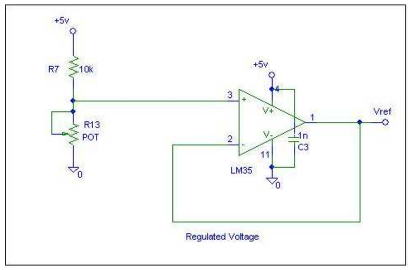
- What are we trying to do?
- Hardware debuggers
- What is JTAG?
- How does it work?
- Board bring up
- The Linux boot sequence
- Debugging the kernel

The Board Bring-up Process

- You have your shiny new board from manufacturing
  - They say the board works
- They claim that they've done the hard part, now you just need to bring up Linux on the board ☺
- Where do you start?
  - First, get the data sheets for all of the programmable parts
    - This may be harder than you think because of NDAs and paperwork
    - You should start the data sheet collection process even before the board is ready

Board Bring up #2

- Talk to the hardware folks
  - Get the schematics for the board
  - Ask about the chip selects
    - Determine how the board is wired up
  - Figure out how fast the SDRAM is supposed to be
    - You'll figure out later how fast it *really* is
- Collect a listing of the key registers on the board and where they are mapped
  - Are they I/O or memory mapped?
  - Register width, read/write capabilities, etc.



Source: cornell.edu

Board Bring-up #3

- Produce a memory map diagram from the registers you found earlier
  - This is needed to describe the register maps to the hardware debugger configuration files
  - Pictures always help put it into perspective
- Create a block diagram for the board that outlines the connectivity
  - Get the hardware folks to help on this
- Dust off that assembly language book
  - You'll need it

Board Bring-up #4

- Choose your boot firmware
  - Try to pick one that already supports your processor architecture
- Port the boot firmware
- Learn from the boot firmware porting effort what needs to be done, then apply that to the Linux boot sequence as needed
  - Many Linux boot issues are similar to that of the boot firmware
  - Also depends on how much board set up is done by your boot firmware and how much is left to Linux

Board Bring-up #5

- Get Linux to a bash prompt
  - If possible, run the Linux Standard Base and POSIX test suites to make sure the Linux is functional
- Package the boot firmware and Linux BSP so it can be turned over to the applications developers
- Ship it
- Miller Time ☺

Harsh Realities

- When the hardware folks say the board works, what does that mean?
  - Frequently, it simply means that the *magic blue smoke* doesn't escape the chips when they powered it up
- The assertion that the board works is often based on simulating the board
  - Errors in the manufacturing process, board layout bugs, bad solder joints, etc. will come into play as you start testing

Exercising the Board

- How do you test that the board is working?
  - Hardware debuggers such as an In-Circuit Emulator (ICE), JTAG, logic analyzer, oscilloscope, LEDs, etc.
- Essentially, you have got to drive signals on the board to make sure the hardware responds correctly
  - Some of this may have been done by the hardware folks
    - Ask them what, if any, tests they performed and how they hooked the hardware up to do these tests
    - You'll need to duplicate their setup and tests to verify that your test board is also working

Value of a Reference Board

- Often times, the hardware designers will have based their design on a manufacturer's reference design
  - The chip vendors will often give you their design layout if you commit to buy enough chips
- If there is a reference board, obtain a copy of it and any software available for it from the manufacturer
  - They may have already ported boot firmware and Linux to it
  - Having the reference board allows you to test how the board *should* work for comparison to yours

Example Reference Board

- Find out just how close the target board is to the reference board from the hardware designers
  - Leverage as much information as you can

Reference Board

Source: atmel.com

Source: kwikbyte.com

Target Board

Hardware Debugging Tools

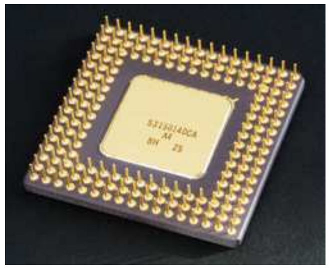
- The traditional hardware debug tool was the In-Circuit Emulator (ICE)
  - A device that plugged into the CPU socket and emulated the CPU itself
- These were rather expensive
  - $30K+ for the good ones
- Today, most devices that call themselves an ICE are actually JTAGs

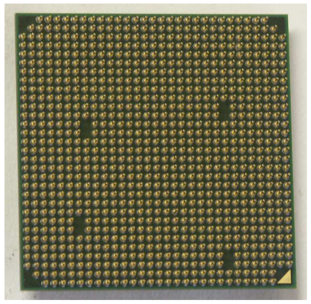
Source: Avocet Systems

Source: Hitex Devel Tools

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


Source: Intel


Source: AMD

Source: ESA

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

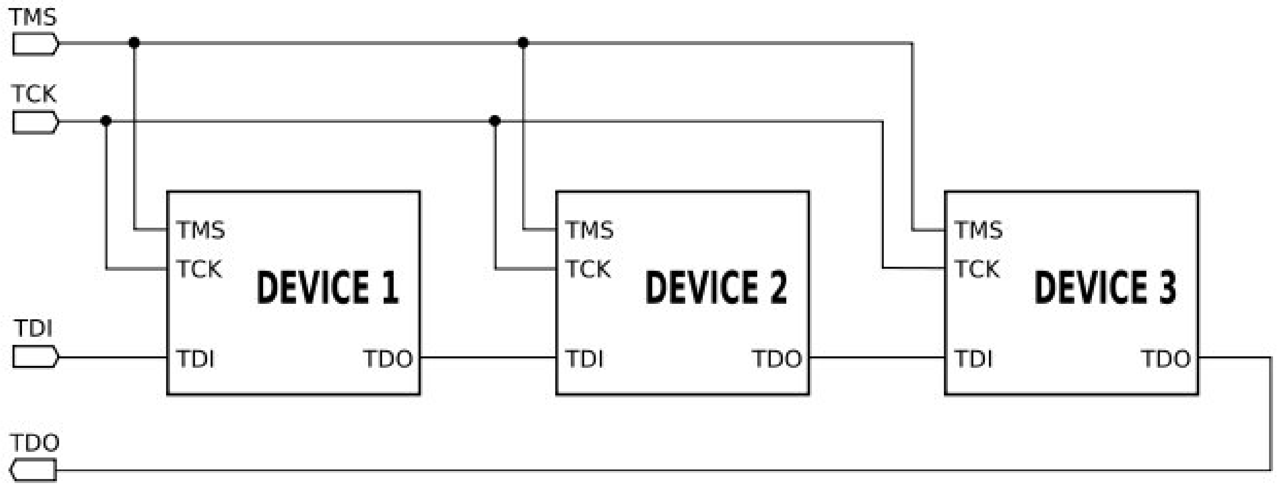
How JTAG Works

- 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
- 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

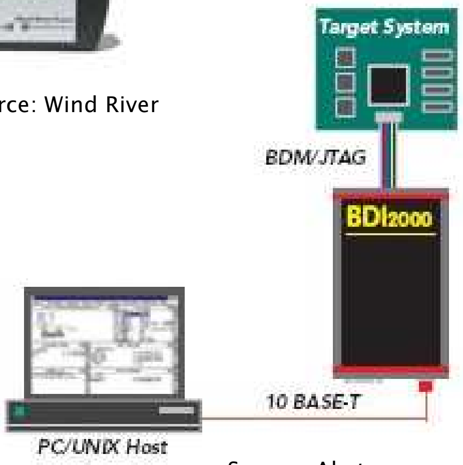
- Several different vendors sell JTAG port interface hardware
  - JTAG is also referred to as On-Chip Debugging (OCD)
- Here are a few of the vendors:
  - Wind River Systems (http://www.windriver.com)
  - Abatron AG (http://www.abatron.ch)
  - American Arium (http://www.arium.com)
  - Mentor Graphics (http://www.epitools.com)

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

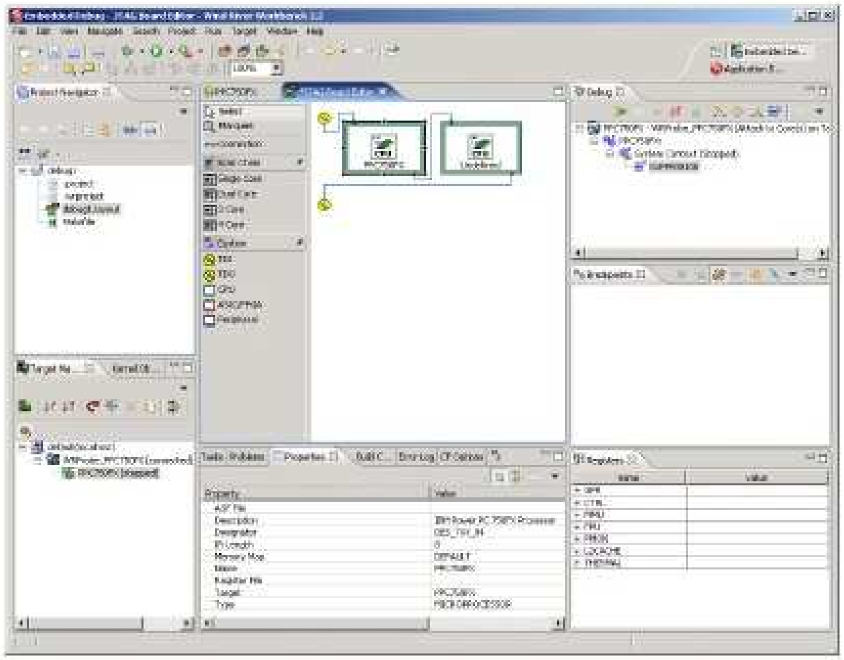
Source: Olimex



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



Source: Wind River

What can you do with a JTAG?

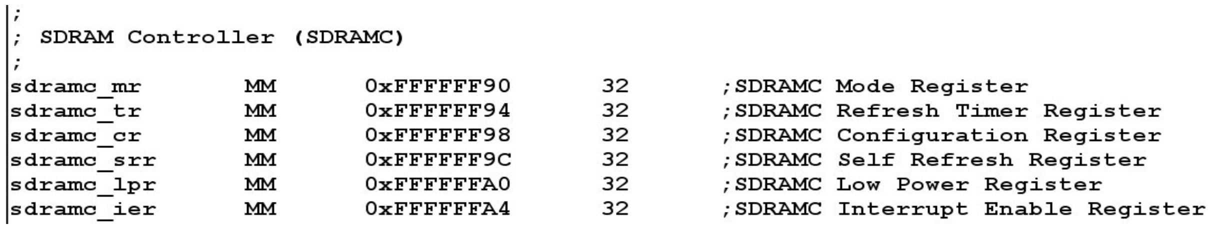
- Typical JTAG usage includes reflashing boot firmware
  - Even the really cheap JTAG units can do this
- However, it is in the use as a debugging aid that JTAG comes into its own
  - You can set hardware or software breakpoints and debug in source code
  - Sophisticated breakpoint strategies and multi-core 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

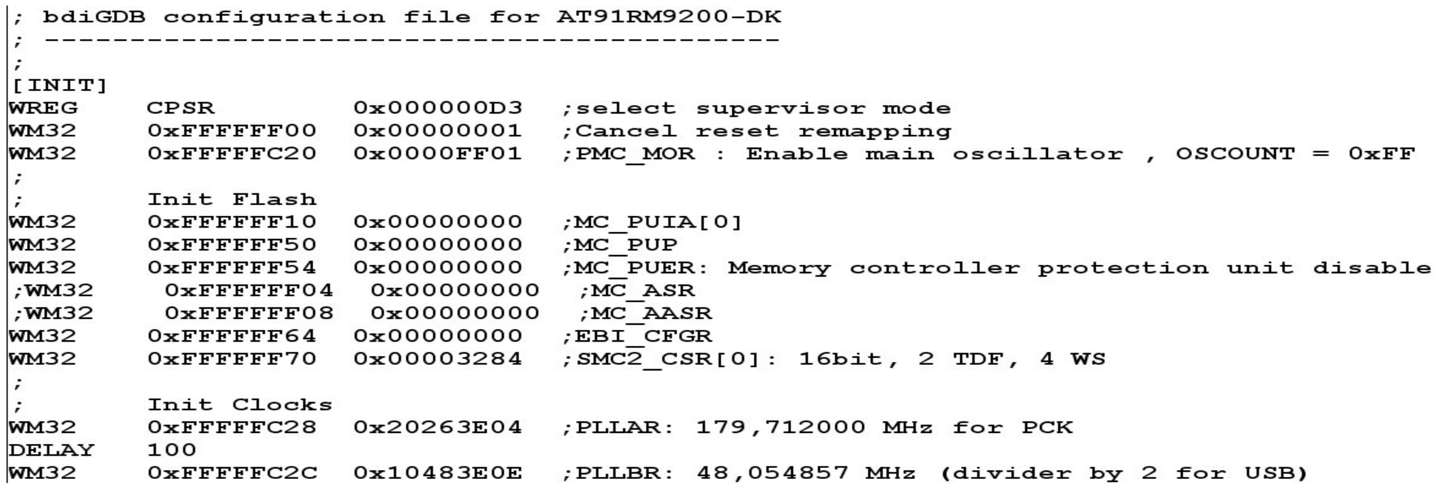
- Most JTAG units require you to describe the hardware registers in a configuration file
  - This is also how you describe what processor architecture you are using
- All of that information about register maps that you collected earlier now goes into the configuration file
- Unfortunately, there is 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      0xFFFFFF90      32      ;SDRAMC Mode Register
sdramc_tr       MM      0xFFFFFF94      32      ;SDRAMC Refresh Timer Register
sdramc_cr       MM      0xFFFFFF98      32      ;SDRAMC Configuration Register
sdramc_srr      MM      0xFFFFFF9C      32      ;SDRAMC Self Refresh Register
sdramc_lpr      MM      0xFFFFFFA0      32      ;SDRAMC Low Power Register
sdramc_ier      MM      0xFFFFFFA4      32      ;SDRAMC Interrupt Enable Register
```



```
; bdiGDB configuration file for AT91RM9200-DK
; --------------------------------------------
;
[INIT]
WREG     CPSR          0x000000D3   ;select supervisor mode
WM32     0xFFFFFF00    0x00000001   ;Cancel reset remapping
WM32     0xFFFFFC20    0x0000FF01   ;PMC_MOR : Enable main oscillator , OSCOUNT = 0xFF
;
;        Init Flash
WM32     0xFFFFFF10    0x00000000   ;MC_PUIA[0]
WM32     0xFFFFFF50    0x00000000   ;MC_PUP
WM32     0xFFFFFF54    0x00000000   ;MC_PUER: Memory controller protection unit disable
;WM32     0xFFFFFF04    0x00000000   ;MC_ASR
;WM32     0xFFFFFF08    0x00000000   ;MC_AASR
WM32     0xFFFFFF64    0x00000000   ;EBI_CFGR
WM32     0xFFFFFF70    0x00003284   ;SMC2_CSR[0]: 16bit, 2 TDF, 4 WS
;
;        Init Clocks
WM32     0xFFFFFC28    0x20263E04   ;PLLAR: 179,712000 MHz for PCK
DELAY    100
WM32     0xFFFFFC2C    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
- 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

Translating JTAG Configuration to Code

- Next, you'll take the settings from the configuration file and start translating them into the boot firmware
  - Must of this starts in assembly language for the memory initialization and then transitions to C for the rest of the peripherals
- Make sure you test the boot firmware without using the JTAG
  - Avoids hidden JTAG dependencies

Once the Configuration File is Done

- Once you have a working configuration file, you should be able to begin exercising the board
  - Memory tests, addressing tests, reading/writing registers, blink LEDs, etc.
- You exercise the board using the same type of configuration files that you used to put the board into a known state
  - You can treat the configuration file like a script in most cases

Picking your boot firmware

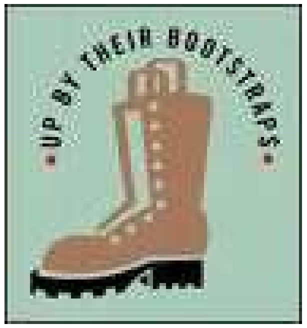
- The x86 has the BIOS
    - This is the boot firmware responsible for loading the operating system
- Non-x86 processors don’t have this luxury
    - You’ll need to port some boot firmware to the target board
- There are several examples of boot firmware
    - Pick one that best supports your hardware

Boot Firmware Environment

- The boot firmware is the code located at the processor's power-on jump (POJ) address
  - 0xFFF00100 for many PPCs
  - 0x0 for most ARM/XScale CPUs
  - The POJ address is typically in your flash segment
- The boot firmware starts in a very primitive state
  - Physical address mode of the processor
    - No MMU
  - Initial code must be in assembly language
    - No memory available at boot time, just CPU registers

Job of the Boot Firmware

- The boot firmware must place the hardware into a known state
  - Essentially, do what the JTAG configuration does
- Enable memory, disable processor caches, disable MMU, enable clocks/PLLs, set up chip selects, disable interrupts
- The last thing the firmware typically does in assembly is to establish a “C” calling stack and then jump to C code



Source: agriculture.com

Job of the Boot Firmware #2

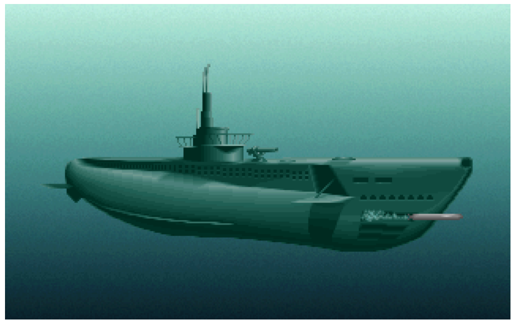
- Once executing C, the firmware typically copies itself into RAM and jumps to the RAM copy
  - Improves performance
- Firmware will then typically initialize a serial console, initialize boot devices and load Linux from someplace
  - This will require device drivers be written for those devices
    - Those device drivers are unique to the firmware
- The firmware is usually single threaded
  - No multi-tasking at this point

Commonly-Used Boot Firmware

- U-Boot
  - http://sourceforge.net/projects/u-boot
- RedBoot
  - http://sources.redhat.com/redboot/
- PMON2000
  - http://www.opsycon.se/pmonmain
- GRUB and Lilo (x86 PC w/ BIOS)
  - http://www.gnu.org/software/grub
  - http://www.tldp.org/HOWTO/LILO.html
- LinuxBIOS
  - http://www.linuxbios.org

U-Boot

- Arguably, the most used boot firmware in the Linux community for embedded applications
  - Supports PPC, ARM/XScale, MIPS
- Released under GPL
- Over 50 different boards already supported
- Supports boot from network, flash and disk
  - Provides environment variables and passing command lines to Linux


Source: U.S. Navy

RedBoot

- Derived from eCOS, now distributed by FSF
  - Source under modified GPL
- Supports boot from flash and network
- Supports
  - ARM/XScale, SH, SPARC, 68K, MIPS, PPC

Source: Dr. Martens USA

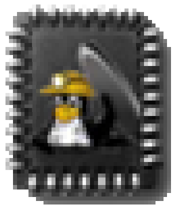
PMON2000

- MIPS-centric boot firmware released as open source
  - Also supports PPC
- Released under BSD license
- Supports boot from flash, mass storage and network
  - Understands the FAT-32 file system
- Can also perform Power-on Self Test (POST) operations

Source: Opsycon AB

x86 Boot loaders: GRUB/LILO/LinuxBIOS

- Linux Loader (LILO)
  - Uses x86 BIOS to load LILO from disk
  - LILO then allows choice of O/S in dual boot systems
- GRand Universal Bootloader (GRUB)
  - Like LILO, but allows booting from network too
- LinuxBIOS
  - Actual replacement for x86 BIOS
    - Runs on some PPC as well
  - Used in One Laptop Per Child project


Source: LinuxBIOS

Using JTAG to Bring Up Firmware

- A problem typically encountered in firmware bring up is that the code starts and runs for a while in flash
- You cannot use typical debugging approaches because flash cannot be easily rewritten to set breakpoints
  - Use the JTAG to set *hardware* breakpoints
    - Hardware breakpoints stop the processor when the address on the bus matches the hardware breakpoint register value
    - There are a limited number of these hardware breakpoint registers available in the processor
      - In fact, there may only be one

Debugging After the Copy

- Once the firmware copies itself from flash into RAM, normal software breakpoints can be used
  - Some firmware supports GDB debugging via the serial port after the copy
- The source debugging capability of some JTAG units is invaluable at this point

Firmware Device Drivers

- At a minimum, the firmware will typically support the initialization of a serial console
  - Used to interact with the firmware and set default behaviors
- Network booting requires drivers for an Ethernet and PHY
  - Also, a small IP stack implementation to support potential DHCP requests, TFTP, and/or FTP
- The IP stack portion should "just work" once you have the network driver working
- Other drivers for erasing and writing flash will also be needed if your board supports this capability
- All of these drivers need to be debugged to support loading Linux

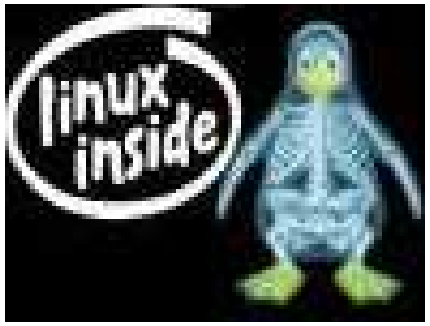
Source: team-xecutor.com

Loading Linux

- The boot firmware will already know how to copy the Linux image into RAM
  - It might need to handle decompression and relocation
- You will need to tell the boot firmware where the image is coming from and store this information in non-volatile storage
  - Typically, some reserved flash sectors
- After the boot firmware moves into RAM, the non-volatile storage is examined to discover what image to load and where it's coming from

Jumping to Linux

- Once the firmware loads Linux into RAM, we can jump to the entry point of Linux
  - Some firmware can pass parameters such as processor info, memory size and speed and a kernel boot line to Linux
- After the handoff to Linux is made, the RAM occupied by the firmware is reclaimed
  - Linux does not refer back to the firmware at run time like Windows® does



Source: gbax.com

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

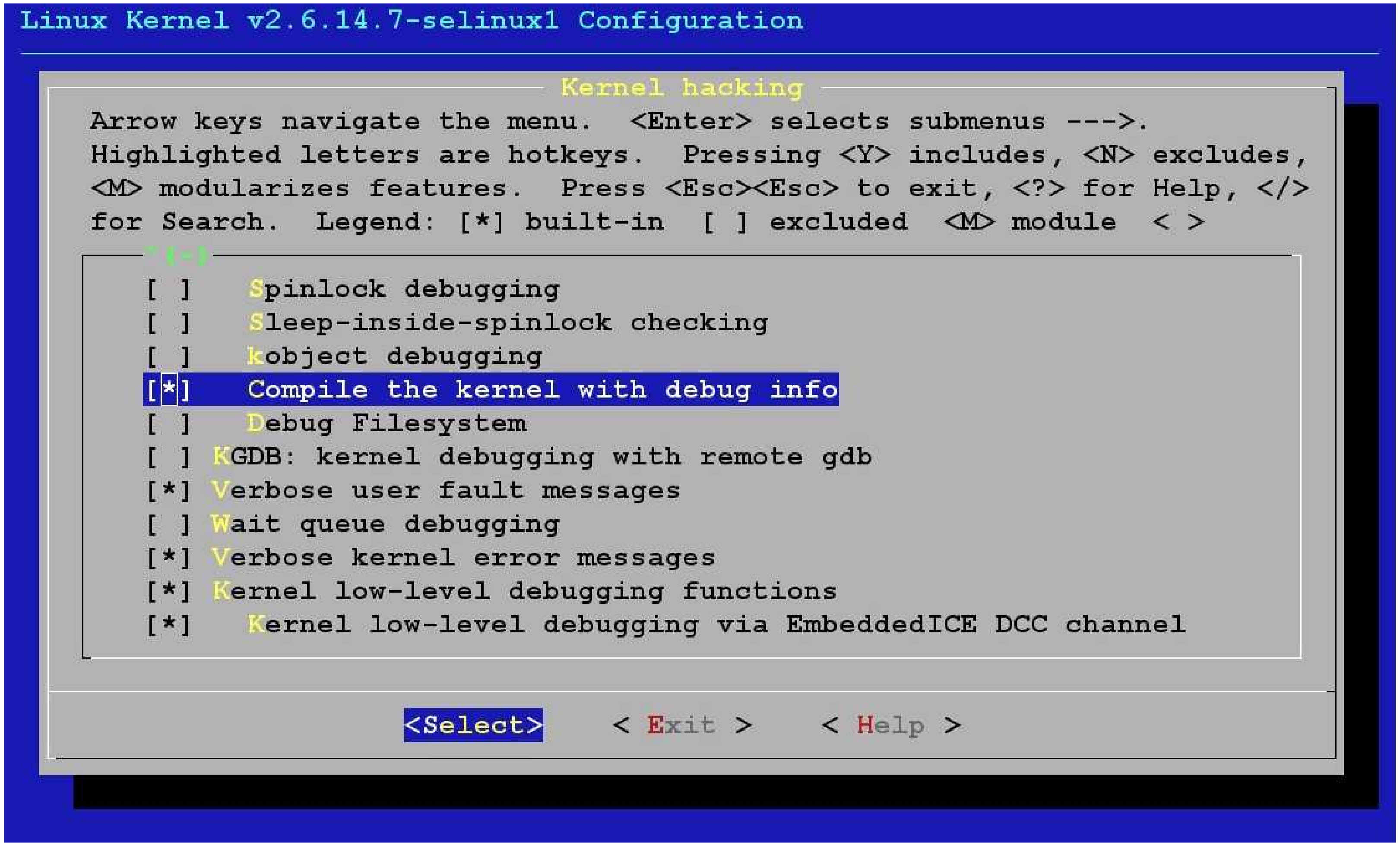
JTAG use in Linux Debug

- Because the Linux kernel was loaded into RAM via the boot firmware, only software breakpoints should be required after the initial breakpoint
  - A hardware breakpoint can be used to stop the kernel on entry, then software breakpoints can be set
- Make sure to compile the kernel with debugging symbols so you can set breakpoints on symbol names rather than addresses

Source: EMAC, Inc.

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

JTAG and Early Kernel Debug

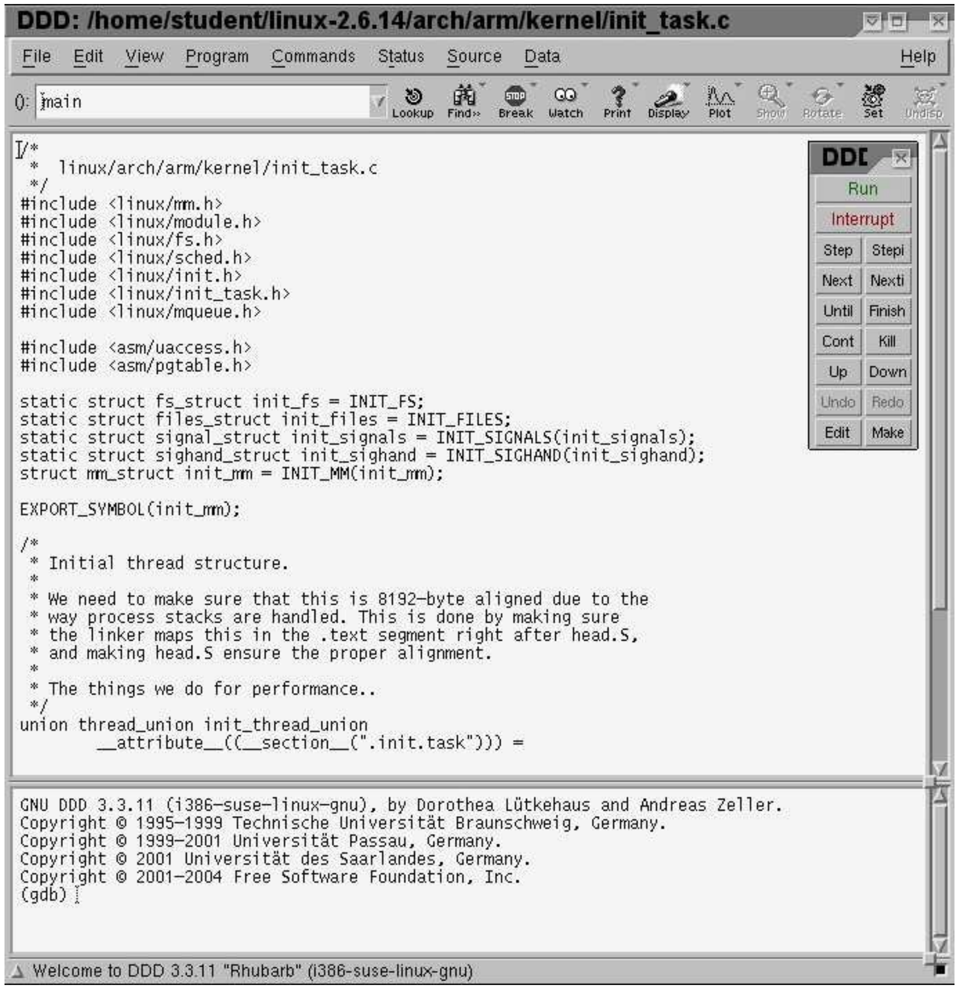
- An odd thing happens when the MMU is enabled
  - All of the physical addresses suddenly get translated into virtual addresses
- The kernel's debug symbols are all built assuming a virtual address space
- Consequently, while you can step through the early code by using a hardware breakpoint address, 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

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

DDD GUI Front-End Example

- Invoked from command line with vmlinux compiled for debugging
- Then attach to JTAG using “target remote” command


DDD: /home/student/linux-2.6.14/arch/arm/kernel/init_task.c
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
/*
 *  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)
```

Welcome to DDD 3.3.11 "Rhubarb" (i386-suse-linux-gnu)

Debugging Device Drivers

- Device driver debugging can be split into those drivers that are statically linked into the kernel and those that are dynamically loaded
- Statically linked drivers are already built into the kernel's symbol table
  - Simply set break points on the driver methods themselves
- Dynamically loaded drivers require additional steps
- 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.,

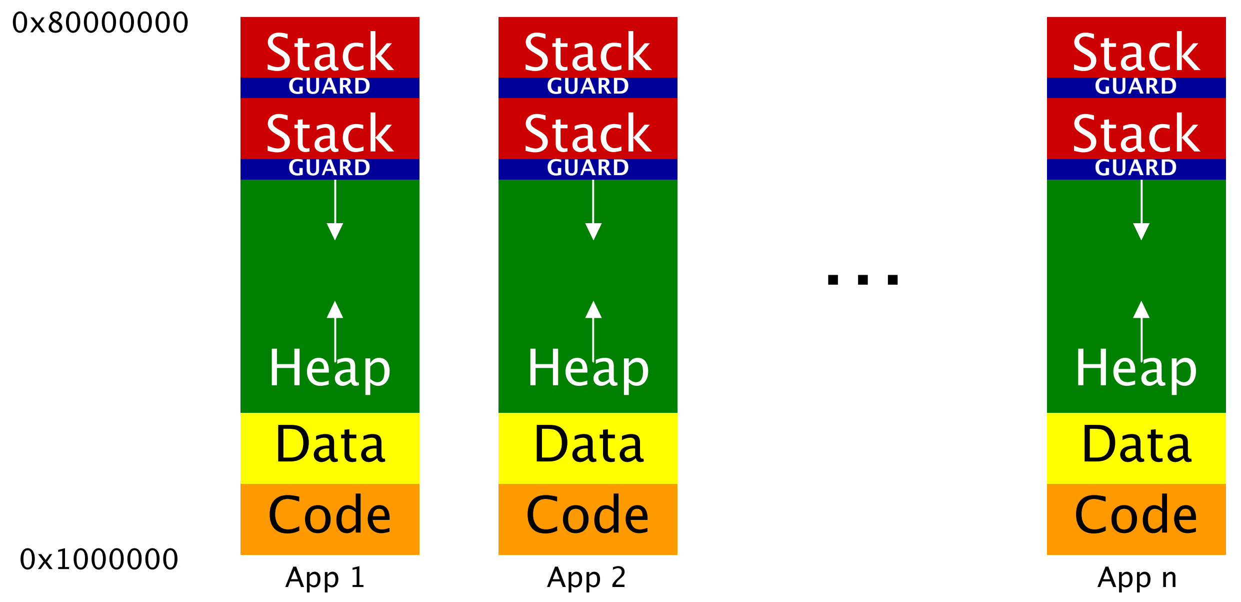
```
add-symbol-file ./mydriver.ko 0x<addr> -e .init.text
```

Debugging Loadable Modules #3

- Now, you can set breakpoints via the GDB commands to the JTAG and tell the system to continue until a breakpoint in encountered
- 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 *h*alt 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…