WS3 – Developing Drivers for Altera SoC

Introduction to writing linux device drivers for the Altera SoC FPGA.

Altera SW SoC Workshop Series

- SW Workshop #1 – Altera SoC SW Development Overview
- SW Workshop #2 – Introduction to Linux on Altera SoC
- SW Workshop #3 – Developing Drivers for Altera SoC Linux

Agenda

- Essential Information Resources
- SoC Device Overview
- Developing Drivers
  - Detailed agenda later…
- Take Home Lab

Welcome. Here's What You Can Expect Today

Experienced Linux Developers

- Find a familiar embedded Linux development flow
- Standard linux device driver model
- Guide to SoCFPGA resources

New Linux Developers

- An exposure to the linux driver model
- The fundamental APIs that are leveraged by most device drivers
- Essential Linux learning and documentation resources

Hardware Developers

- HW handoff to Linux driver development flows
- SW driver implications of HW architecture

Everyone

- SoC FPGA architecture-specific information
- SoC FPGA recommendations and best practices

Essential Information Resources

Where to learn more…

…a non-exhaustive list

Linux Foundation Training

Linux Developer classes are designed to help participants:

- Learn how to develop an embedded Linux product
- Become familiar with and learn to write device drivers
- Get practical experience with the Linux kernel
- Learn how to work with the Linux developer community

Developer Courses

- *LFD331* – Developing Linux Device Drivers
- *LFD405* – Building Embedded Linux with the Yocto Project
- *LFD411* – Embedded Linux Development
- *LFD414* – Introduction to Embedded Android Development
- *LFD205* – How to Participate with the Linux Community
- *LFD211* – Introduction to Linux for Developers
- *LFD262* – Developing with Git
- *LFD312* – Developing Applications for Linux
- *LFD320* – Linux Kernel Internals & Debugging
- *LFD415* – Inside Android: An Intro to Android Internals
- *LFD432* – Optimizing Linux Device Drivers for Power Efficiency

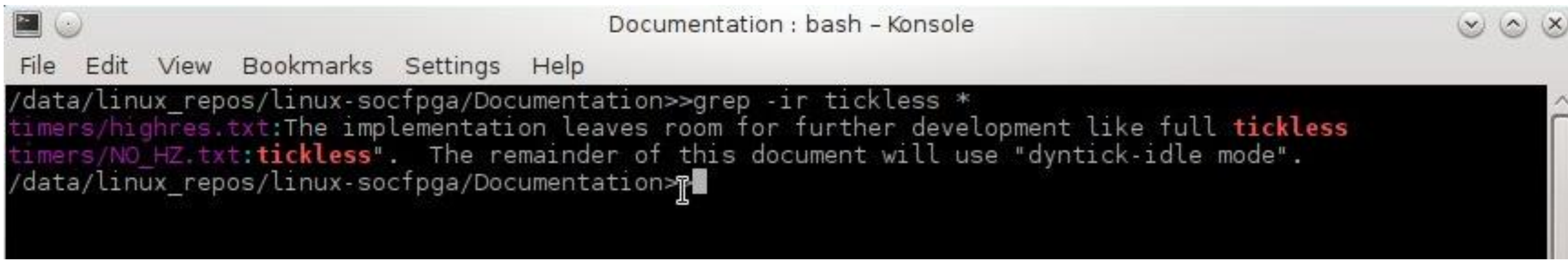
http://training.linuxfoundation.org/linux-courses/development-training

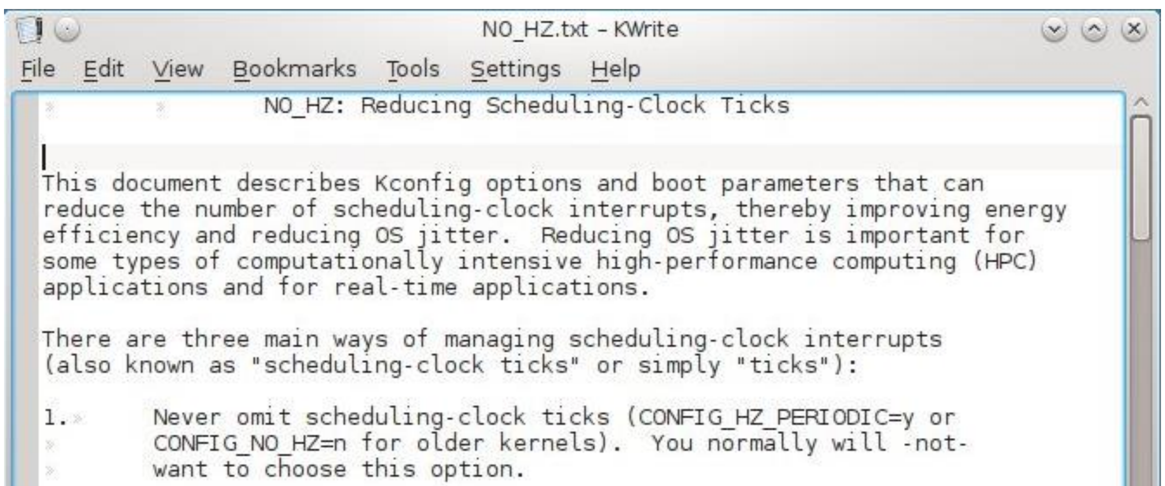
Linux Documentation Resources

- GIT
  - Distributed revision control system to enable distributed collaboration
  - On-line documentation & training:
    - http://git-scm.com/doc
    - https://training.github.com
- Denx U-Boot Manual
  - Complete documentation from the folks who wrote Das U-Boot
    - http://www.denx.de/wiki/U-Boot/Documentation
- Free-Electrons:
  - Complete training materials posted free
    - http://free-electrons.com/docs/
- Device Tree for Dummies
  - http://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf

The Two Best Sources for Linux Development Information

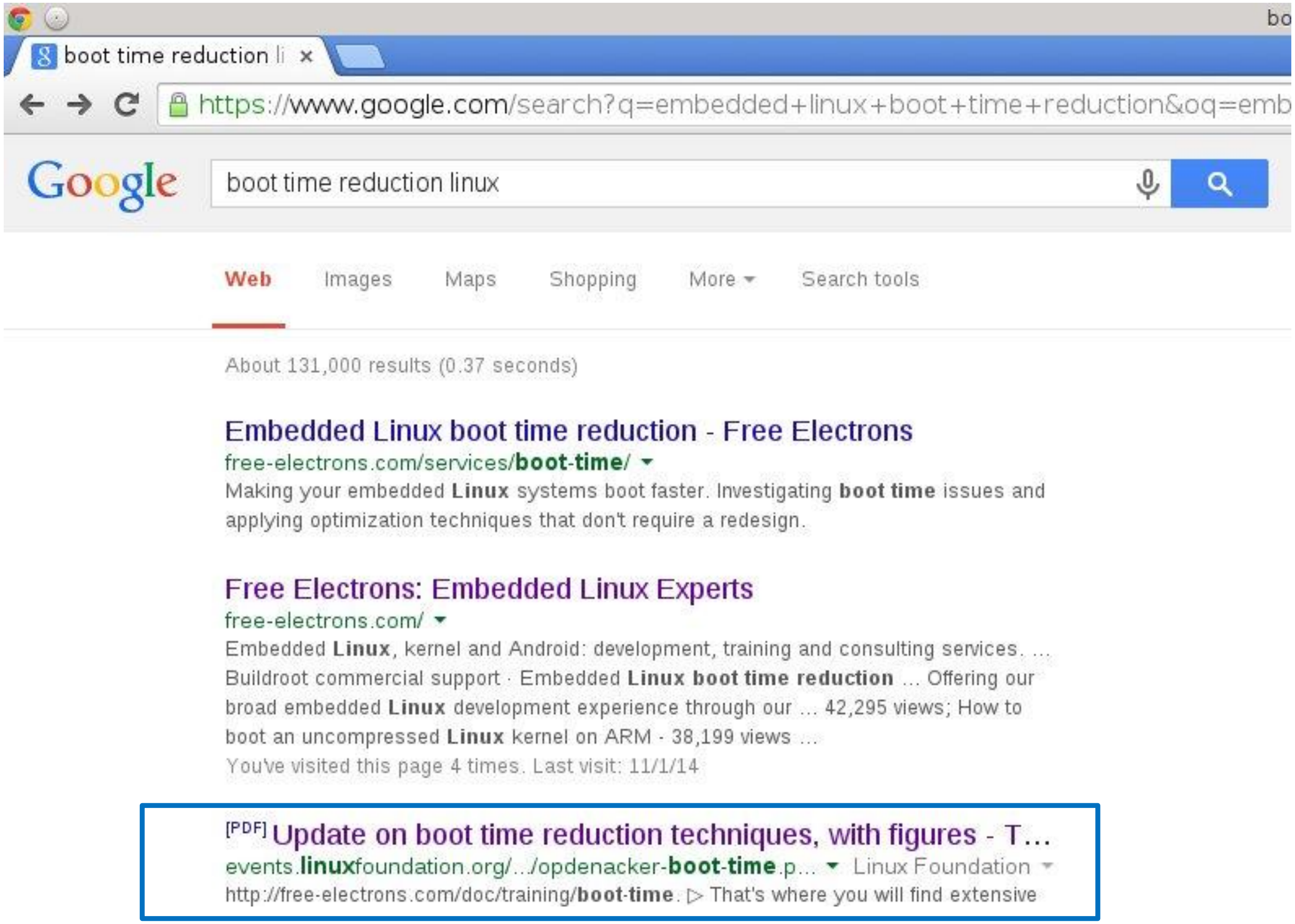
- Linux Kernel Documentation
  - The most complete and most essential Linux kernel documentation
  - Included with the Linux kernel source code
    - <local GIT repo>/Documentation





The Two Best Sources for Linux Development Information

- An open source OS breeds open source information



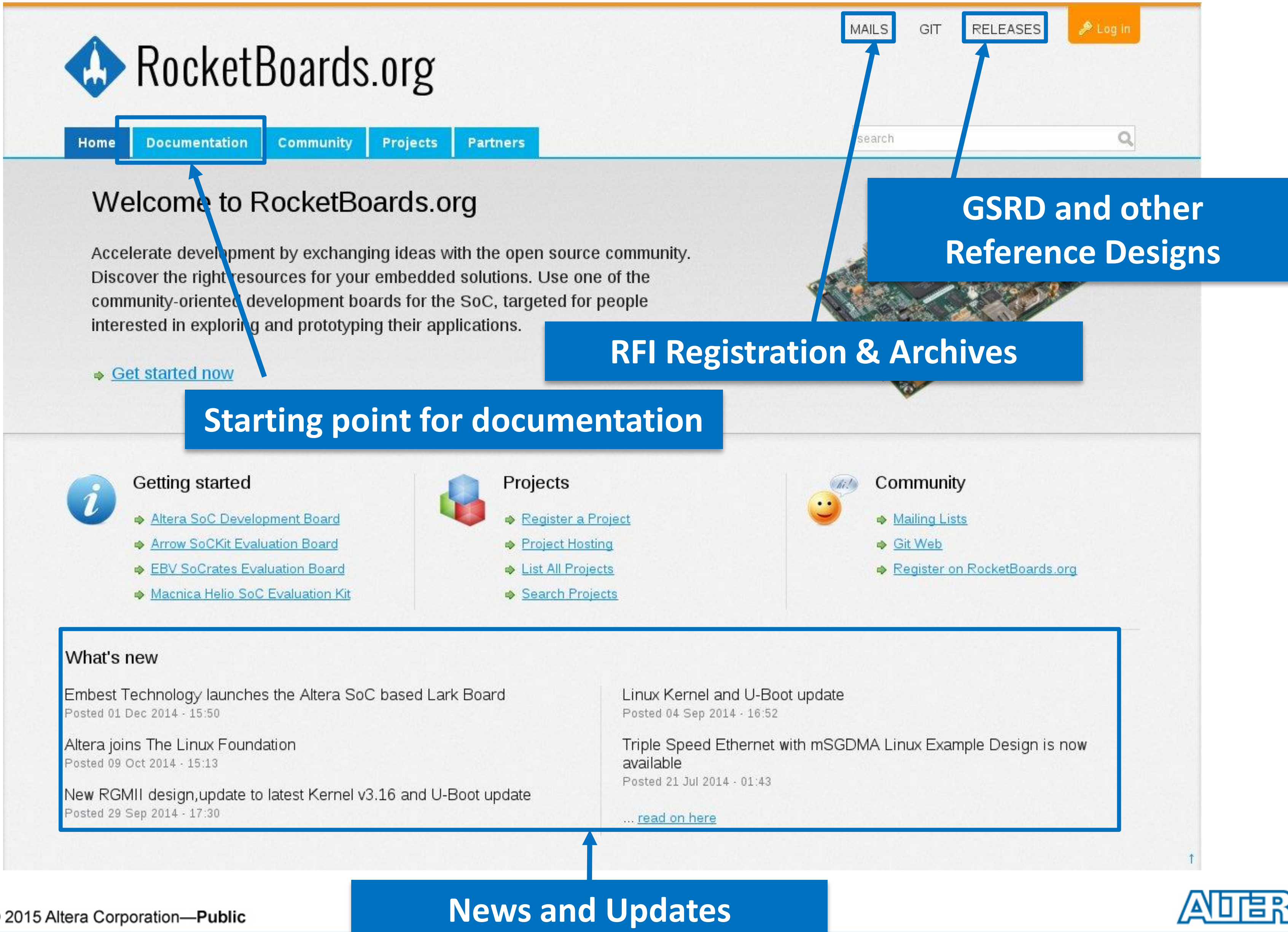
RocketBoards.org – Altera SoC Linux Community Portal

- The source for SoC FPGA Linux info
  - Golden System Reference Design (GSRD)
  - Updates on latest releases
  - Step-by-step getting started guides
- SoC FPGA Mailing List - RFI
  - Active community participation in answering SoC FPGA and Linux questions
- Example Projects, Applications, and Designs
  - From Altera and the SoC community
- Public GIT server
- Enables the SoC community to support Linux

RocketBoards.org

ALTERA

RocketBoards.org Resources



ALTERA

RocketBoards.org – Useful Links

- GSRD User Manual
  - The best starting point for Linux development
  - http://www.rocketboards.org/foswiki/Documentation/GSRD

Getting Started Guides

- Booting Linux Using Prebuilt SD Card Image
- Connecting to Board Web Server
- Connecting to Board Using SSH
- Running Sample Linux Applications
- Compiling the Hardware Design
- Generating and Compiling the Preloader
- Generating the Device Tree
- Compiling Angstrom Linux Distribution
- Creating and Updating SD Card
- GSRD FPGA Programming
- FPGA Programming with Quartus II Programmer
- Using System Console
- Angstrom Getting Started

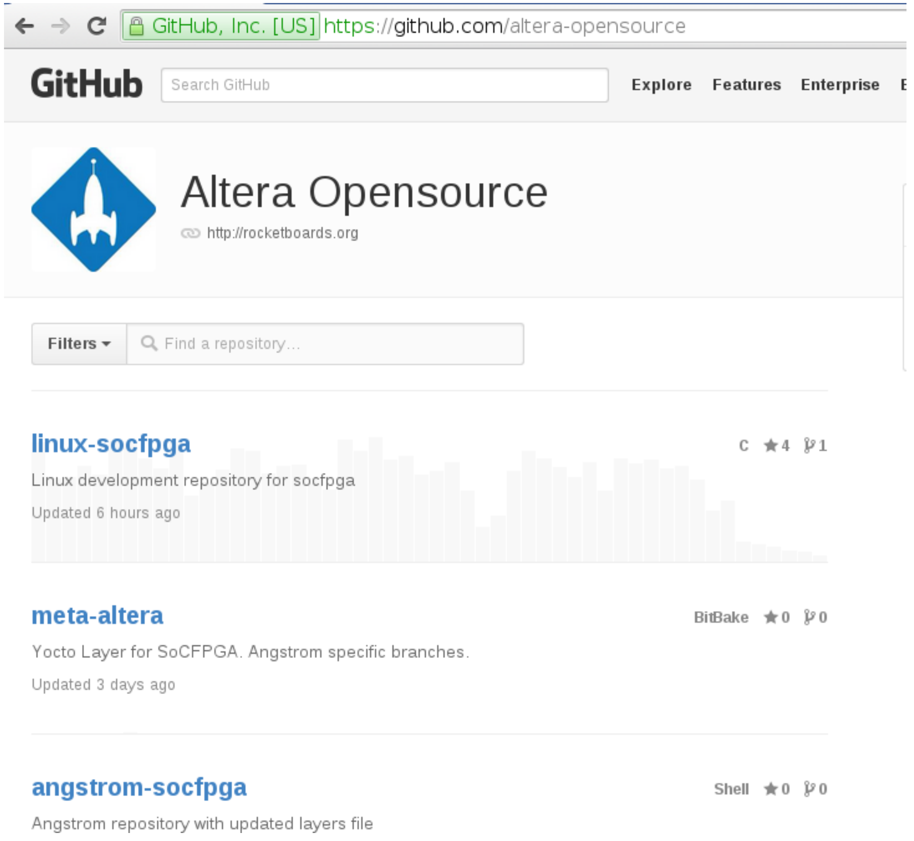
- Device Tree Generator User Guide
  - http://www.rocketboards.org/foswiki/Documentation/GSRD141DeviceTreeGenerator
- Programming FPGA from HPS
  - http://www.rocketboards.org/foswiki/Documentation/GSRD131ProgrammingFPGA

SoCFPGA Linux Code Repositories on GitHub

- Public git repos for SoCFPGA

https://github.com/altera-opensource

- Migrated from RocketBoards
  - Rocketboards.org git repos no longer updated



Several Ways to Learn!

- **Instructor-led training**
  – Face to face with an Altera expert Training Engineer
  – 20+ courses to choose from (8 hour classes)

- **Virtual classes (taught via WebEX)**
  – Can ask questions to Altera expert Training Engineer
  – Course content same as instructor-led classes
    (1/2 day sessions)

- **Online training (free and always available)**
  – 200+ topics available (~30 minutes in length)

- **Videos (free and always available)**
  – YouTube videos (~4 minutes each)

ALTERA

SoC Classes Available

- Instructor-led or virtual classes
  - Designing with an ARM-based SoC
  - Developing Software for an ARM-based SoC

- Online classes
  - Hardware Design Flow for an ARM-based SoC
  - Software Design Flow for an ARM-based SoC
  - SoC Hardware Overview: the Microprocessor Unit
  - SoC Hardware Overview: Interconnect and Memory
  - SoC Hardware Overview: System Management, Debug, and General Purpose Peripherals
  - SoC Hardware Overview: Flash Controllers and Interface Protocols
  - SoC Bare-metal Programming and Hardware Libraries
  - Getting Started with Linux for Altera SoCs

Essential SoC Hardware Documentation Resources

- Hard Processor System Technical Reference Manuals
  - Available in Device Handbooks:
    - https://www.altera.com/products/soc/portfolio/cyclone-v-soc/support.html
    - https://www.altera.com/products/soc/portfolio/arria-v-soc/support.html
    - https://www.altera.com/products/soc/portfolio/arria-10-soc/support.html
  - Contain Functional Descriptions Peripheral
  - Contain Control Register Address Map and Definitions
    - These are also available online at the links above in HTML and PDF formats
- HPS SoC Boot Guide
  - Cyclone V SoC & Arria V SoC: AN709 - HPS SoC Boot Guide
  - Arria 10 SoC: included in HPS TRM in Arria 10 Device Handbook
- ARM Documentation Site
  - Documentation available for all ARM IP
    - Cortex-A9 & A53 MP Cores, FPU, NEON, GIC, ARM Peripherals, etc.
  - Requires free registration
  - Refer to HPS TRM for IP core names and revision information
  - http://infocenter.arm.com/help/index.jsp

Essential SoC Software Documentation Resources

- Altera SoC Embedded Design Software (SoC EDS) Tools
  - User Guide:
    https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_soc_eds.pdf
    - Linux & Baremetal Software Development Tools Overview
    - HPS Preloader User Guide
    - HPS Flash Programmer User Guide
    - SD Card Boot Utility
  - Getting Started Guides: Preloader, Linux, Bare Metal, Debug, HW Library
    http://www.alterawiki.com/wiki/SoCEDSGettingStarted
  - SoC HPS Release Notes
  - SoC Abstraction Layer (SoCAL) API Reference
    - <SoC EDS install dir>/ip/altera/hps/altera_hps/doc/socal/html/index.html
  - Hardware Manager API Reference
    - <SoC EDS install dir>/ip/altera/hps/altera_hps/doc/hwmgr/html/index.html
  - GCC Documentation
    - <SoC EDS install dir>/ds-5/documents/gcc/getting_started.html
  - Bare Metal Compiler
    - <SoC EDS installation directory>/host_tools/mentor/gnu/arm/baremetal/share/doc/sourceryg++-arm-altera-eabi

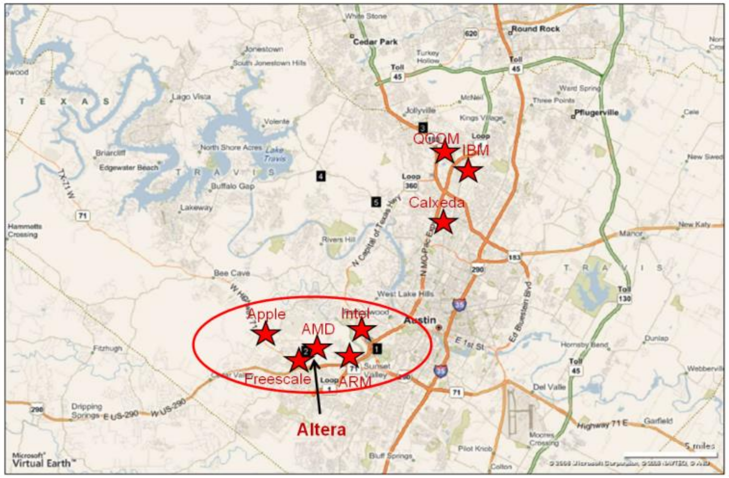
Essential SoC Software Tools Online Videos

- ARM DS-5 Altera Edition Toolchain
  - https://youtu.be/HV6NHr6gLx0
- DS-5 Altera Edition: Bare-metal Debug and Trace
  - https://youtu.be/u_xKybPhcHI
- DS-5 Altera Edition: FPGA-adaptive Linux Kernel Debug and Trace
  - https://youtu.be/lrR-SfVZd18
- Debugging Linux applications on the Altera SoC with ARM DS-5
  - https://youtu.be/ZcGQEjkYWOc
- FPGA-adaptive debug on the Altera SoC using ARM DS-5
  - https://youtu.be/2NBcUv2TxbI
- Streamline Profiling on Altera SoC FPGA. Part 1 - Setup
  - https://youtu.be/X-k9lmXQTio
- Streamline Profiling on Altera SoC FPGA. Part 2 - Running Streamline
  - https://youtu.be/Tzbd7qldKqY

SoC Device Overview

Altera Investment in Embedded Technologies

- Altera established Austin Technology Center (ATC) in 2011
- Altera's primary embedded engineering center
- Austin provides access to one of the richest embedded processing talent bases in the world



Altera SoC Product Portfolio

LOW END SoCs (Lowest Power, Form Factor & Cost)	MID RANGE SoCs (High Performance with Low Power, Form Factor & Cost)	HIGH END SoCs (Highest Performance & System Bandwidth)

HIGH PERFORMANCE

LOW POWER

Cyclone V SoC

- 28nm TSMC
- 925 MHz Dual ARM Cortex™-A9 MPCore™
- 5G Transceivers
- 400 MHz DDR3
- 25 to 110 KLE
- Up to 224 Multipliers (18x19)

Arria V SoC

- 28nm TSMC
- 1.05 GHz Dual ARM Cortex™-A9 MPCore™
- 10G Transceivers
- 533 MHz DDR3
- Up to 462 KLE
- Up to 2136 Multipliers (18x19)

Arria 10 SoC

- 20nm TSMC
- 1.5 GHz Dual ARM Cortex™-A9 MPCore™
- 17G Transceivers
- 1333 MHz DDR4
- Up to 660 KLE
- Up to 3356 Multipliers (18x19)

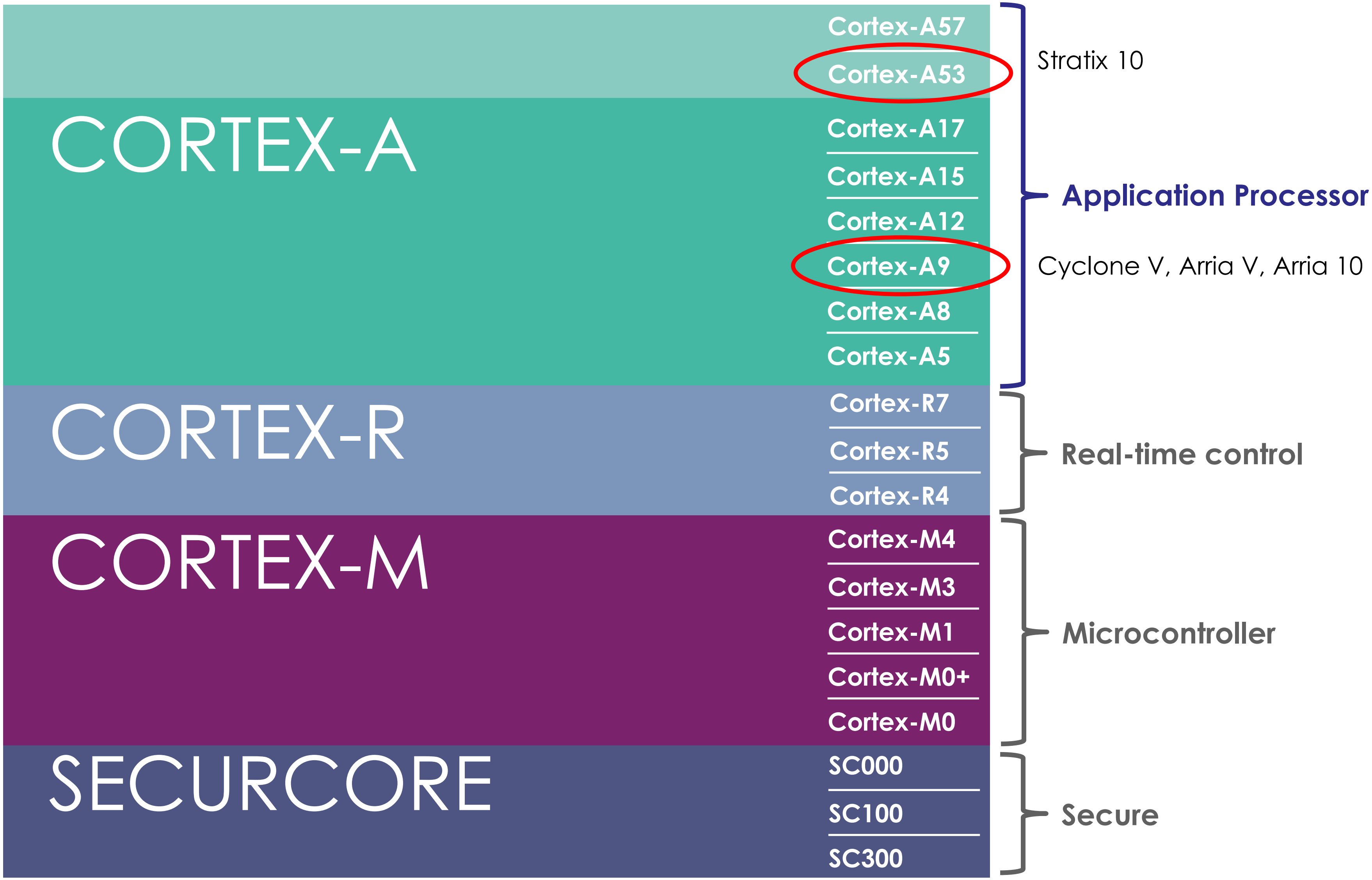
Stratix 10 SoC

- 14nm Intel Tri-Gate
- 64-bit Quad ARM A53 MP Core™
- Optimized for Max Performance per Watt
- Over 4000 KLE

DEVICE AVAILABILITY

Availability of SoC devices across product portfolio ...

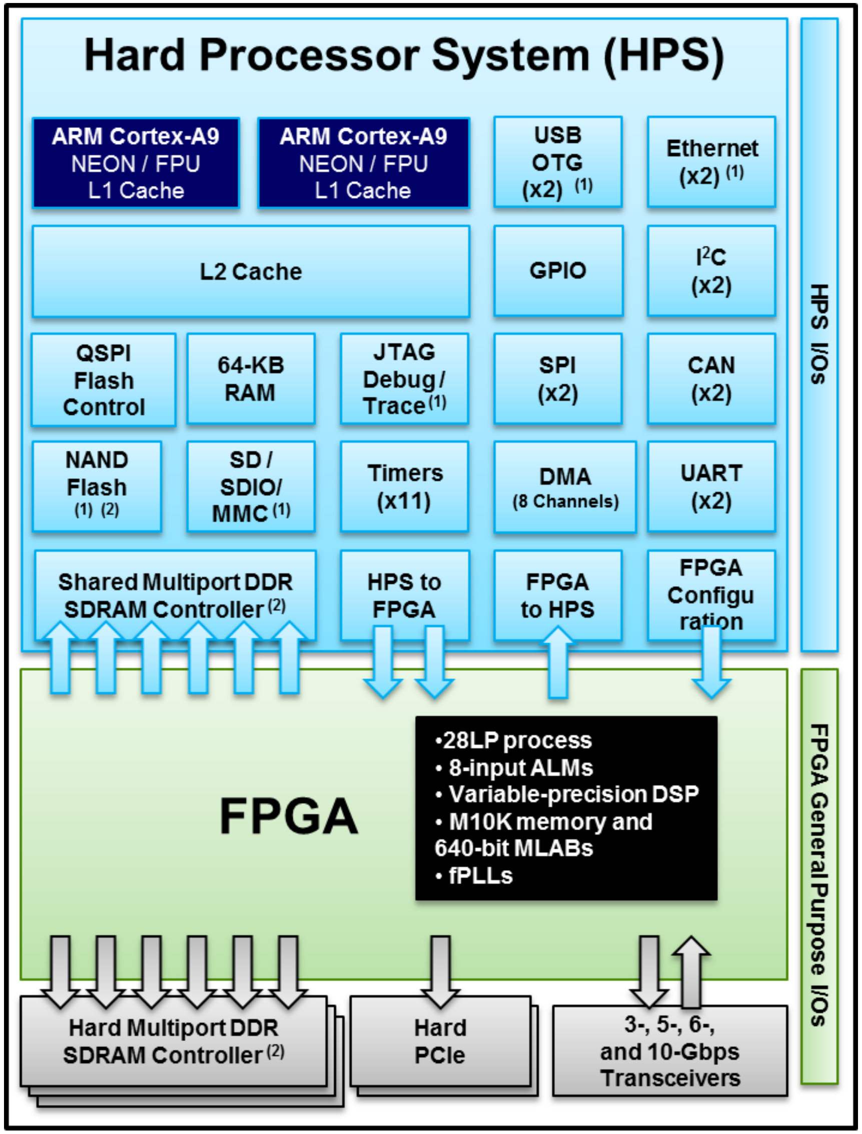
ARM Public Processor Offering



ALTERA

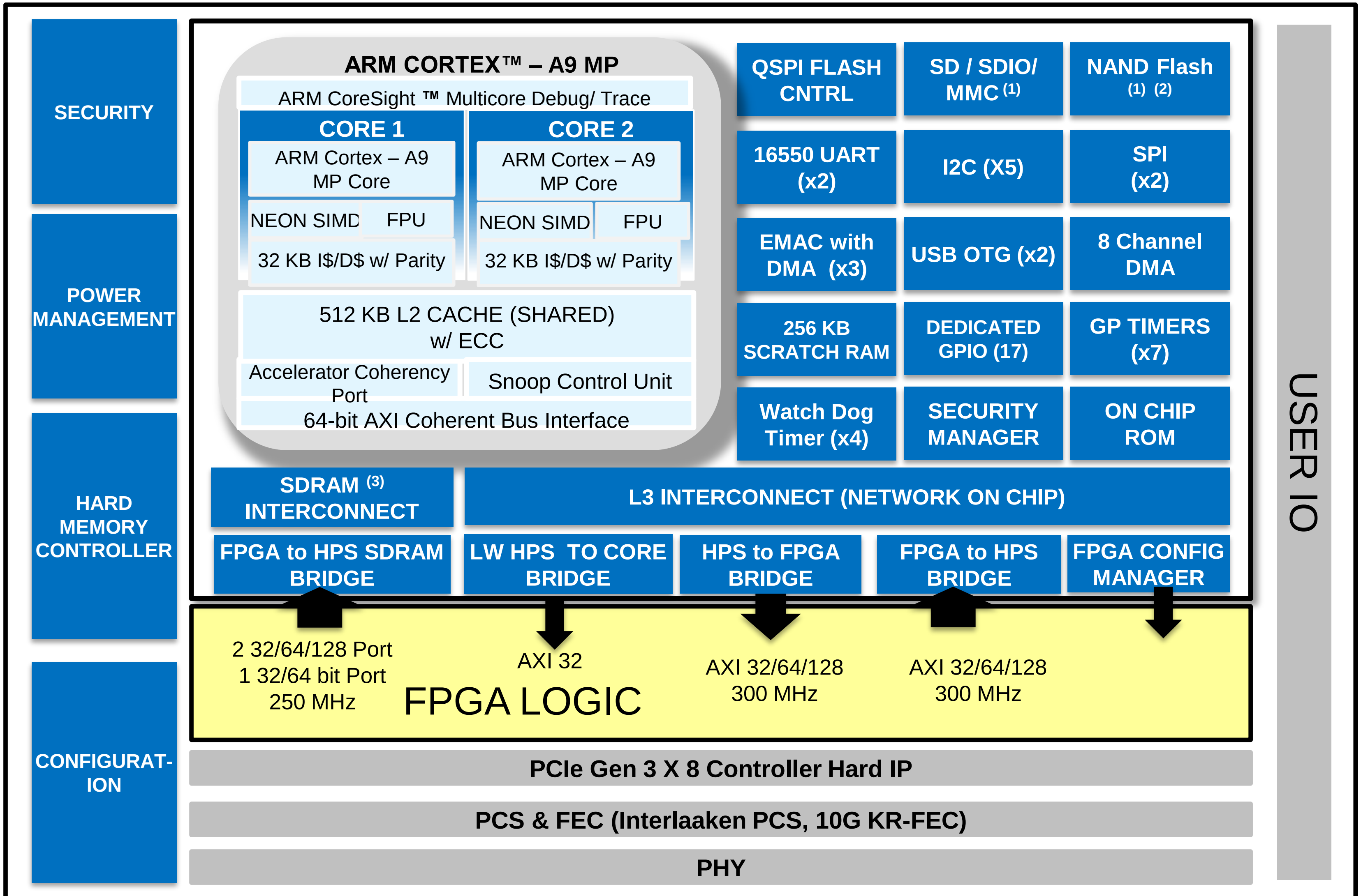
28nm SoC System Architecture

- Processor
  - Dual-core ARM® Cortex™-A9 MPCore™ processor
  - Up to 5,250 MIPS (1050 MHz per core maximum)
  - NEON coprocessor with double-precision FPU
  - 32-KB/32-KB L1 caches per core
  - 512-KB shared L2 cache
- Multiport SDRAM controller
  - DDR3, DDR3L, DDR2, LPDDR2
  - Integrated ECC support
- High-bandwidth on-chip interfaces
  - > 125-Gbps HPS-to-FPGA interface
  - > 125-Gbps FPGA-to-SDRAM interface
- Cost- and power-optimized FPGA fabric
  - Lowest power transceivers
  - Up to 1,600 GMACS, 300 GFLOPS
  - Up to 25Mb on-chip RAM
  - More hard intellectual property (IP): PCIe® and memory controllers



Notes:
(1) Integrated direct memory access (DMA)
(2) Integrated ECC

Arria 10 HPS Block Diagram



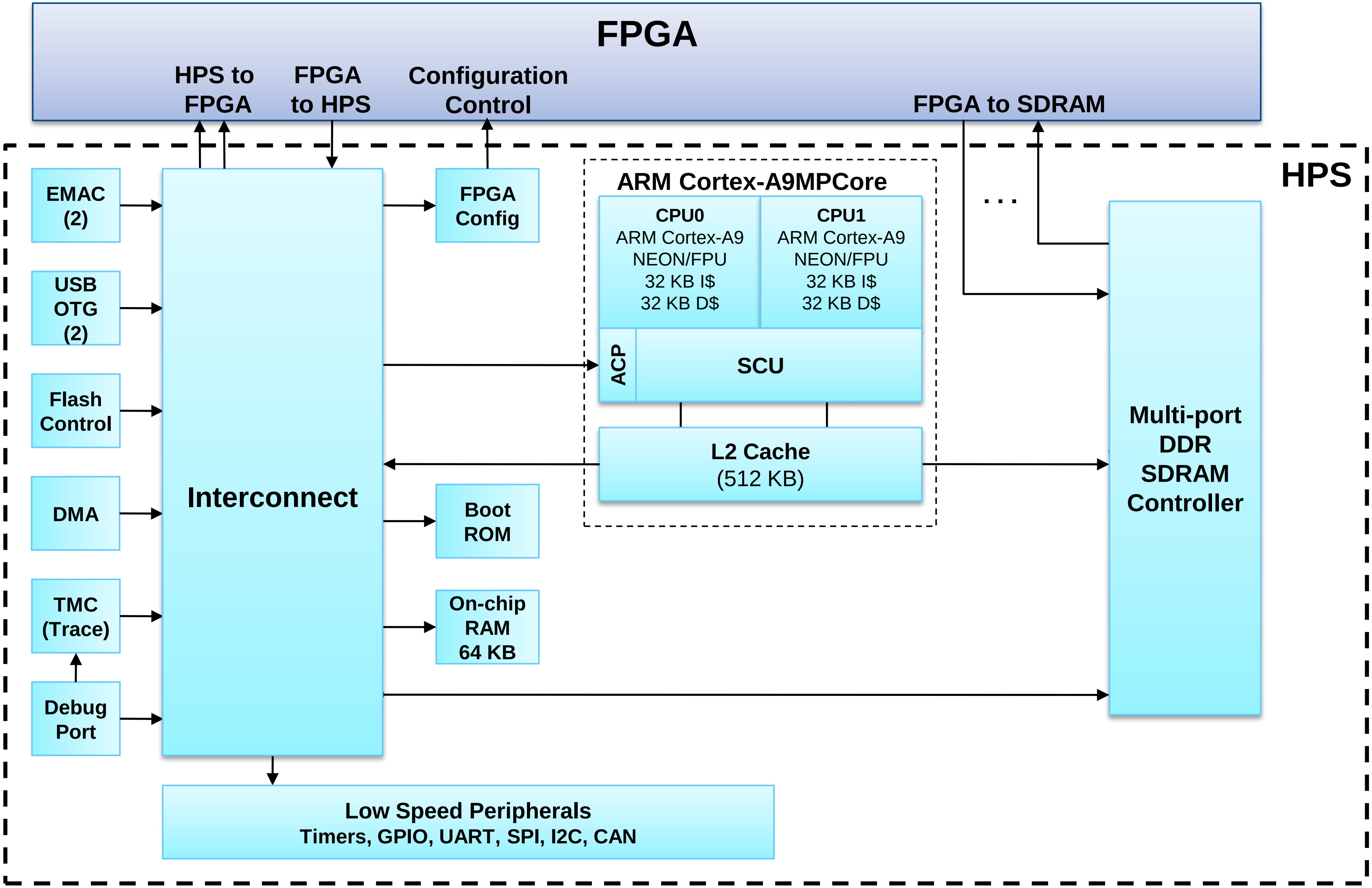
Notes:

(1) Integrated direct memory access (DMA)

(2) Integrated ECC

(3) DDR3/4 & LP DDR3 SDRAM Support fo HPS Memory

High-Level Block Diagram



A Comparison: Cyclone V SoC, Arria V SoC, Arria 10 SoC

Metric	Cyclone V SoC	Arria V SoC	Arria 10 SoC
Technology	28nm	28nm	20nm
Processor Performance	925 MHz	1.05 GHz	1.5 GHz
Total Power Dissipation	100%	100%	60% (40% Lower)
Max PCI Express Hard IP	Gen 2 x4	Gen 2 x8	Gen 3 x8
Memory Devices Supported	DDR2, DDR3, DDR3L, LPDDR2	DDR2, DDR3, DDR3L, LPDDR2	DDR4/3, LPDDR2/3, QDRIV, RLDRAM III, Hybrid Memory Cube
Max. HPS DDR Data-Width	40-bit (32-bit + ECC)	40-bit (32-bit + ECC)	72-bit (64-bit + ECC)
EMAC Cores	EMAC x 2	EMAC x 2	EMAC x 3
NAND Device Supported	8-bit	8-bit	8-bit and 16-bit
SD/MMC devices supported	SD/SDIO/MMC	SD/SDIO/MMC	SD/SDIO/MMC 4.5 with eMMC
FPGA Logic Density Range (LEs)	25 - 110K	370 - 450K	160 - 660K
FPGA Core Performance	260 MHz	307 MHz	500 MHz

Developing Drivers

Developing Drivers Agenda (1 of 4)

- Driver Demonstration Hardware Overview
- Hardware architecture to software development communication options
  - sopc-create-header-files, sopc2dts
- Accessing FPGA hardware from user space
  - /dev/mem
  - mmap(), munmap()
  - runtime device tree inspection
  - application build example
- Driver development environment setup
  - ARCH, CROSS_COMPILE, OUT_DIR
  - Kbuild file, Makefile
- Demonstration Module 1 - the most basic module example
  - Source code licensing
  - module_init(), module_exit(), MODULE_LICENSE(), MODULE_*()
- Demonstration Module 1t - tainted module example
  - kernel tainting, modinfo, insmod, lsmod, rmmod
  - procfs / sysfs module statistics

Developing Drivers Agenda (2 of 4)

- Demonstration Module 2 - introduce module parameters
  - module_param(), module_param_array(), MODULE_PARAM_DESC()
  - sysfs module parameters/* entries
- Demonstration Module 3 - introduce platform driver
  - of_device_id structure
    - .compatible strings
  - MODULE_DEVICE_TABLE()
  - platform_driver structure
    - .probe, .remove, .driver
  - platform_driver_register(), platform_driver_unregister()
  - manually binding/unbinding from user space
- Demonstration Module 4 - extract hardware details from DT
  - platform_get_resource(), platform_get_irq(), clk_get(), clk_get_rate()
  - manually validating device tree information from procfs
- Demonstration Module 5 - reserve memory and enable IRQ
  - down_interruptible(), up(), request_mem_region(), release_mem_region(), ioremap(), iounmap(), request_irq(), free_irq(), resource_size(), ioread32(), iowrite32()
  - basic interrupt handler

Developing Drivers Agenda (3 of 4)

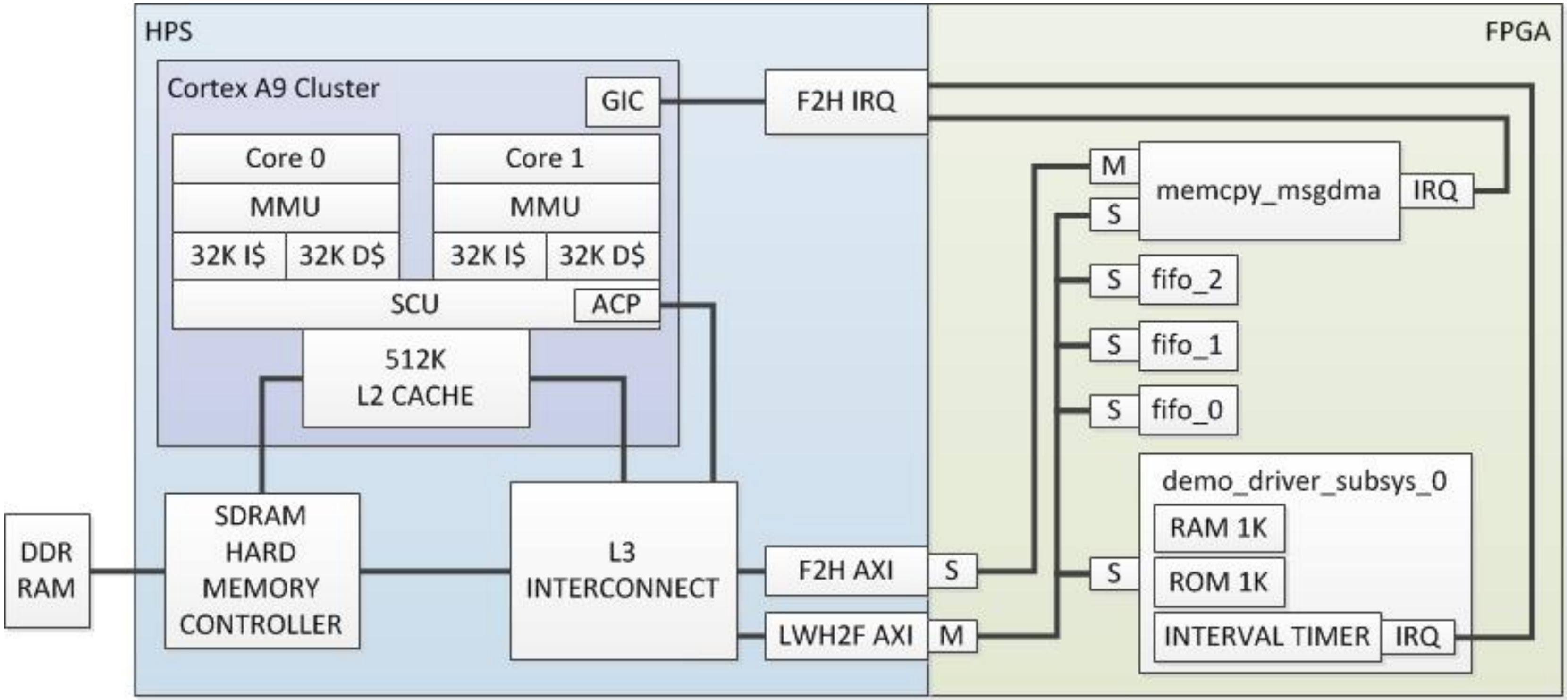
- Demonstration Module 5t - test memory reservation and IRQ
  - resource contention demonstration
- Demonstration Module 6 - introduce sysfs entries
  - driver_create_file(), driver_remove_file(), DRIVER_ATTR(), spin_lock_irqsave(), spin_lock_irqrestore(), spin_lock(), spin_unlock(), scnprintf(), kstrtoul()
  - interrupt handler
  - sysfs show(), store() functions
  - manual interaction with sysfs entries from user space
- Demonstration Module 7 - introduce misc device
  - sema_init(), misc_register(), misc_deregister(), memcpy_fromio(), memcpy_toio(), copy_from_user(), copy_to_user(), get_user(), put_user(), wait_event_interruptible(), wake_up_interruptible(), pgprot_noncached(), remap_pfn_range()
  - file_operations structure
    - .open, .release, .read, .write, .llseek, .unlocked_ioctl, .mmap
  - miscdevice structure
  - struct file.private_data, struct file.f_pos
  - SEEK_SET, SEEK_CUR, SEEK_END
  - ioctl() from user space

Developing Drivers Agenda (4 of 4)

- Demonstration Module 8 - introduce uio device
  - uio_info structure
  - uio_register_device(), uio_unregister_device(), down_trylock()
- Demonstration Module 9 - introduce dma
  - kmalloc(), dma_alloc_coherent(), dma_map_single(), dma_unmap_single(), dma_mapping_error()
- Demonstration Module 10 - multiple device instances
  - INIT_LIST_HEAD(), kzalloc(), list_add(), platform_set_drvdata(), iminor(), list_for_each(), list_entry()
  - struct file.f_flags
- Demonstration Module 11 - introduce custom API
  - EXPORT_SYMBOL()
- Demonstration Module 11t - test custom API

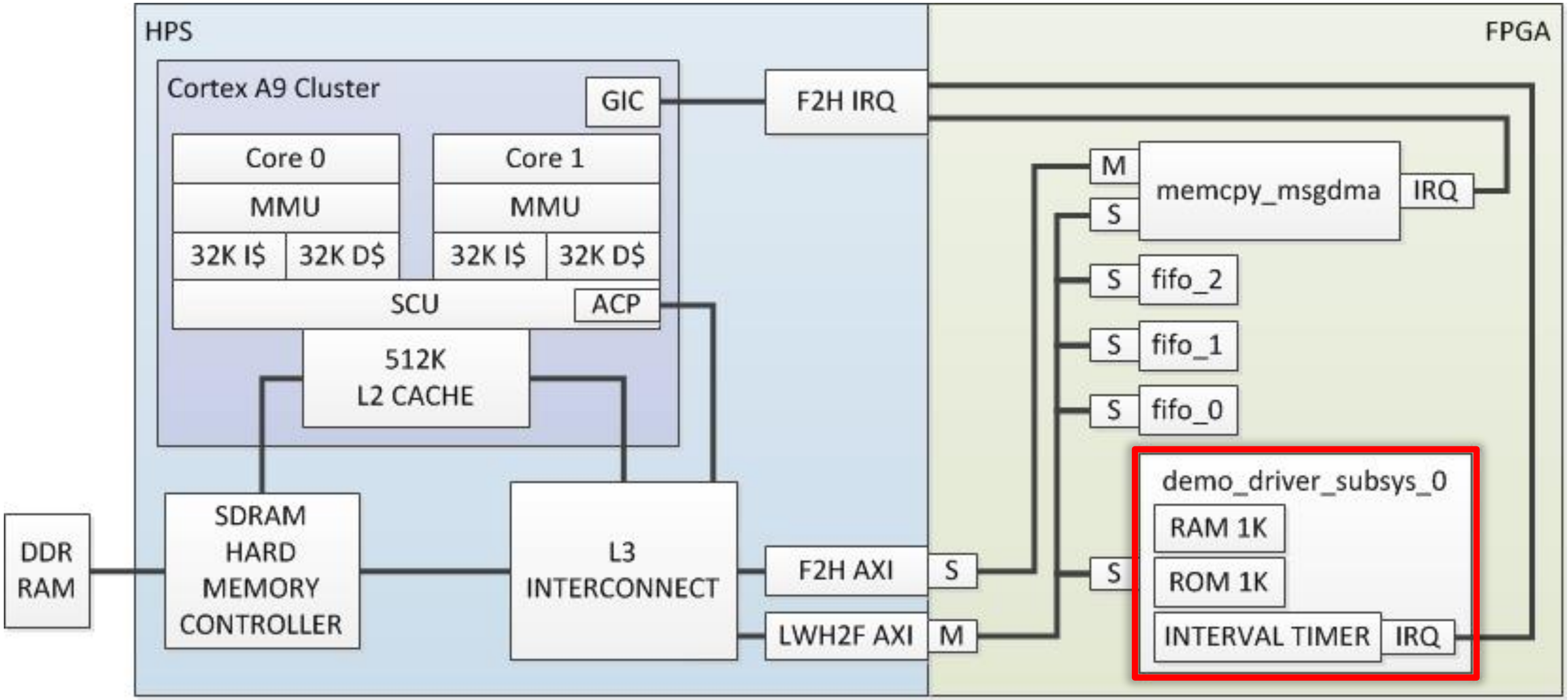
Our demonstration hardware environment for the workshop

Driver Demonstration Hardware



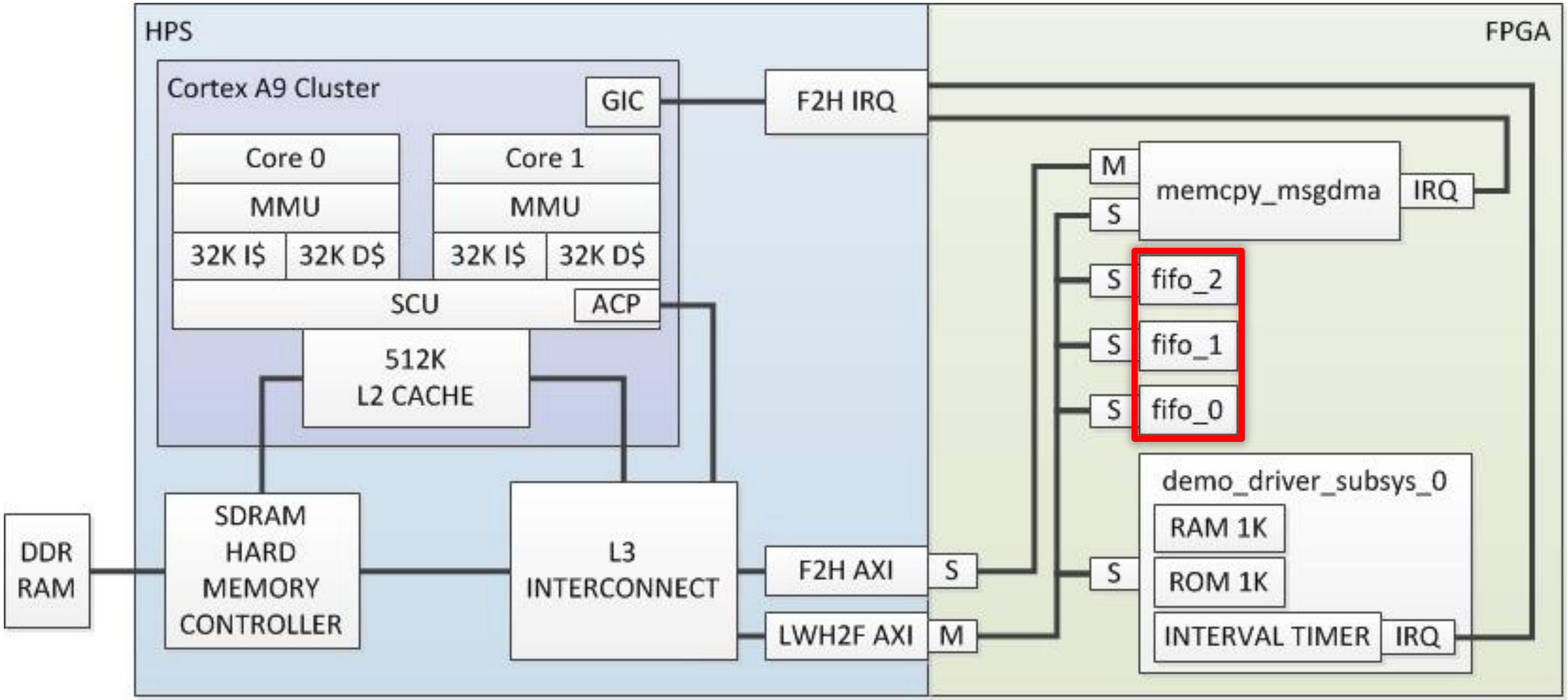
This is a <u>very simplified</u> block diagram of the Altera HPS and FPGA system that is used to demonstrate the concepts described in this workshop. Please see the Qsys system for more detail.

Driver Demonstration Hardware



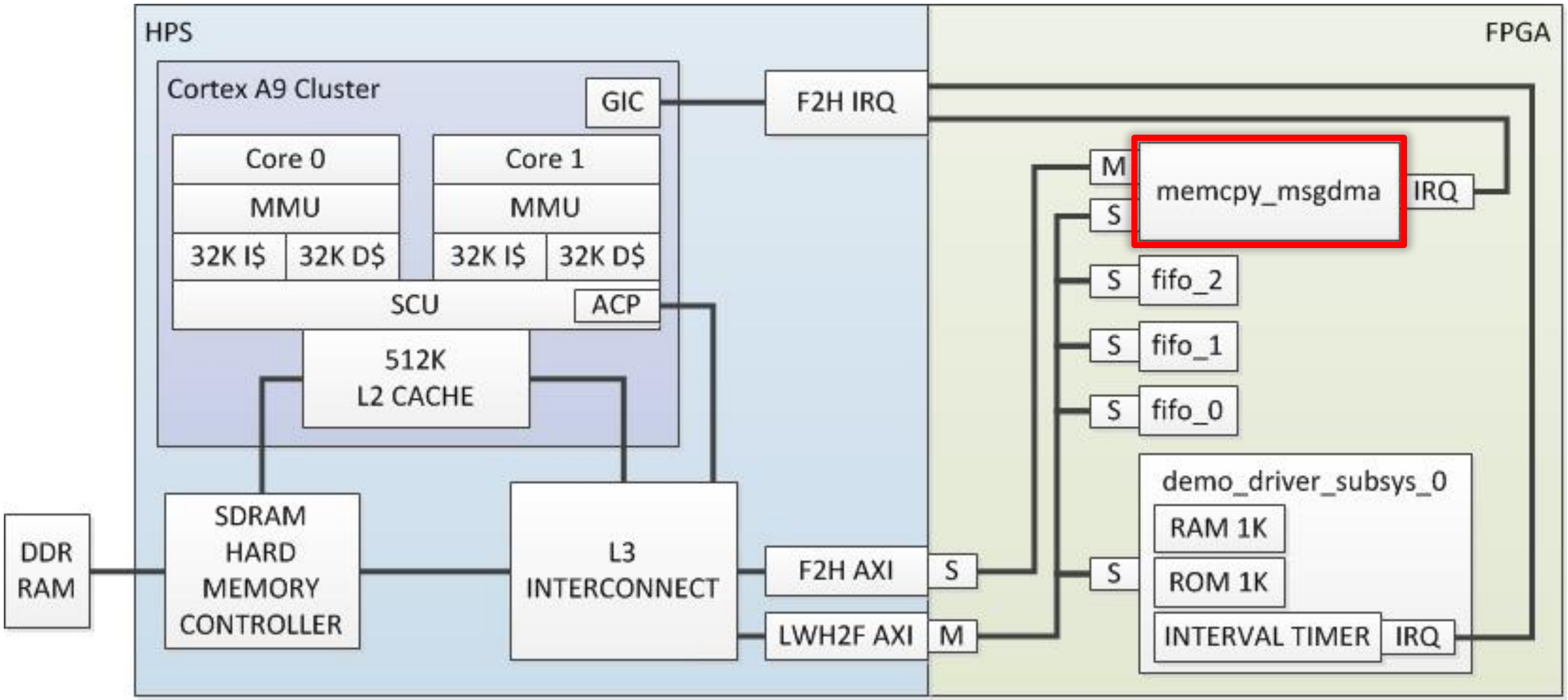
The demo_driver_subsys_0 peripheral is a custom Qsys component that presents one opaque slave interface to the system. Internally it contains a RAM, a ROM, and an Interval Timer that can generate an interrupt.

Driver Demonstration Hardware



There are three FIFO components that have a slave interface for writing data into the FIFO and another slave interface for reading data out of the FIFO. In this block diagram the FIFO is represented with only one slave interface for simplicity.

Driver Demonstration Hardware



The memcpy_msgdma component can be used to DMA data thru the F2H AXI bridge, which allows us to DMA data to/from anywhere in the HPS core or peripherals attached to the LWH2F and H2F AXI bridges in the FPGA.

How do we get our hardware system definition into the software domain?

Hardware details into software development flow

- One of the first challenges that we encounter if we want to write software to interact with our FPGA hardware is communicating the hardware architecture details into our software development environment.
- The Altera Quartus and Qsys tools provide various pieces of output that can help us accomplish this with automated output from the Qsys system generation flow and the Quartus compilation flow.
- The following slides describe what the hardware tools output and some utilities that can be applied to that output to create something that can be easily digested by the software development flow.

Hardware Handoff Files to Software Environment

```
[]$ ls -r *.sopcinfo hps_isw_handoff/*

soc_system.sopcinfo

hps_isw_handoff/soc_system_hps_0:
tclrpt.h              sequencer_defines.h         sdram_io.h
tclrpt.c              sequencer.c                 id
system.h              sequencer_auto_inst_init.c  hps.xml
soc_system_hps_0.hiof sequencer_auto.h            emif.xml
sequencer.h           sequencer_auto_ac_init.c    alt_types.h
```

These files are generated in the Quartus project directory as a result of Qsys system generation and Quartus compilation.

Hardware Handoff Files to Software Environment

```
[]$ ls -r *.sopcinfo hps_isw_handoff/*

soc_system.sopcinfo

hps_isw_handoff/soc_system_hps_0:
tclrpt.h              sequencer_defines.h          sdram_io.h
tclrpt.c              sequencer.c                  id
system.h              sequencer_auto_inst_init.c   hps.xml
soc_system_hps_0.hiof sequencer_auto.h             emif.xml
sequencer.h           sequencer_auto_ac_init.c     alt_types.h
```

The SOPCINFO file is created by Qsys during system generation. This database can be used to create system header files and device trees for the software environment.

The “sopc-create-header-files” utility can generate a variety of header file and macro file formats, and the “sopc2dts” utility can create device trees representing the Qsys system.

Hardware Handoff Files to Software Environment

```
[]$ ls -r *.sopcinfo hps_isw_handoff/*

soc_system.sopcinfo

hps_isw_handoff/soc_system_hps_0:
tclrpt.h               sequencer_defines.h          sdram_io.h
tclrpt.c               sequencer.c                  id
system.h               sequencer_auto_inst_init.c   hps.xml
soc_system_hps_0.hiof  sequencer_auto.h             emif.xml
sequencer.h            sequencer_auto_ac_init.c     alt_types.h
```

The hps_isw_handoff directory is created by Quartus during compilation and contains information about the HPS configuration from Qsys and Quartus. This collection of files is used by the BSP generator to create a customized preloader for the HPS core to run at boot time to configure the HPS core as defined by Qsys and Quartus.

Creating System Header Files

```
[]$ sopc-create-header-files --help

Usage: sopc-create-header-files [<sopc>] [OPTION]...

This utility creates header files from your SOPC Builder system description.
By default, the header files are in cpp format and have a .h suffix.
Other formats may be selected with the appropriate command-line option.

Options:

...cut...

Supported header file formats:
  type suffix             uses                          example
  ==== ====== ========================== ======================
  h      .h     C/C++ header file for cpp  #define FOO 12
  m4     .m4    macro file for m4          m4_define("FOO", 12)
  sh     .sh    shell scripts              FOO=12
  mk     .mk    makefiles                  FOO := 12
  pm     .pm    Perl scripts               $macros{FOO} = 12;

By default, multiple header files are created.  There is one header file for
the entire system and one header file for each master group in each module.
A master group is a set of masters in a module in the same address space.
In general, a module may have multiple master groups.
Addresses and available devices are a function of the master group.

...cut...
```

output formats

Creating System Header Files

```
[]$ mkdir qsys_headers
[]$ sopc-create-header-files soc_system.sopcinfo --output-dir qsys_headers
[]$ ls qsys_headers
axi_bridge_for_acp_128_0.h  fft_sub_sgdma_to_fft.h  lw_mm_bridge.h
fft_ddr_bridge.h            fpga_only_master.h      memcpy_msgdma_mm_read.h
fft_sub_DDR.h               hps_0_arm_a9_0.h        memcpy_msgdma_mm_write.h
fft_sub.h                   hps_0_arm_a9_1.h        soc_system.h
fft_sub_mm_bridge_0.h       hps_0_bridges.h
fft_sub_sgdma_from_fft.h    hps_0.h
```

The sopc-create-header-files utility can create a memory map view from the perspective of any of the masters in the Qsys system. In the default mode demonstrated above, all masters in the Qsys system have a header file created for their view of the memory mapped system. The macros in these headers can be useful when writing applications and drivers that interact with the FPGA based peripherals.

Creating System Header Files

```
[]$ grep "\s[^H].*_BASE" qsys_headers/hps_0_arm_a9_0.h
#define ONCHIP_MEMORY2_0_BASE 0xc0000000
#define SYSID_QSYS_BASE 0xff201000
#define INTR_CAPTURER_0_BASE 0xff206000
#define VALIDATOR_SUBSYS_0_BASE 0xff210000
#define MEMCPY_MSGDMA_CSR_BASE 0xff220000
#define MEMCPY_MSGDMA_DESCRIPTOR_SLAVE_BASE 0xff220020
#define DEMO_DRIVER_SUBSYS_0_BASE 0xff230000
#define FIFO_0_IN_CSR_BASE 0xff240000
#define FIFO_0_IN_BASE 0xff240020
#define FIFO_0_OUT_BASE 0xff240030
#define FIFO_1_IN_CSR_BASE 0xff244000
#define FIFO_1_IN_BASE 0xff244020
#define FIFO_1_OUT_BASE 0xff244030
#define FIFO_2_IN_CSR_BASE 0xff248000
#define FIFO_2_IN_BASE 0xff248020
#define FIFO_2_OUT_BASE 0xff248030
#define FFT_SUB_SGDMA_TO_FFT_CSR_BASE 0xff280000
#define FFT_SUB_SGDMA_TO_FFT_DESCRIPTOR_SLAVE_BASE 0xff290000
#define FFT_SUB_SGDMA_FROM_FFT_CSR_BASE 0xff2a0000
#define FFT_SUB_SGDMA_FROM_FFT_DESCRIPTOR_SLAVE_BASE 0xff2b0000
#define FFT_SUB_DATA_BASE 0xff2c0000
#define FFT_SUB_FFT_STADAPTER_0_BASE 0xff2d0000
```

This is an example of the base addresses of the FPGA based Qsys peripherals from the perspective of the Cortex A9 processor. NOTE: this is a very small portion of the total contents of the header file macros.

Creating System Header Files

```
[]$ grep "\s[^H].*_BASE" qsys_headers/hps_0.h
#define ONCHIP_MEMORY2_0_BASE 0x0
#define SYSID_QSYS_BASE 0x1000
#define INTR_CAPTURER_0_BASE 0x6000
#define VALIDATOR_SUBSYS_0_BASE 0x10000
#define MEMCPY_MSGDMA_CSR_BASE 0x20000
#define MEMCPY_MSGDMA_DESCRIPTOR_SLAVE_BASE 0x20020
#define DEMO_DRIVER_SUBSYS_0_BASE 0x30000
#define FIFO_0_IN_CSR_BASE 0x40000
#define FIFO_0_IN_BASE 0x40020
#define FIFO_0_OUT_BASE 0x40030
#define FIFO_1_IN_CSR_BASE 0x44000
#define FIFO_1_IN_BASE 0x44020
#define FIFO_1_OUT_BASE 0x44030
#define FIFO_2_IN_CSR_BASE 0x48000
#define FIFO_2_IN_BASE 0x48020
#define FIFO_2_OUT_BASE 0x48030
#define FFT_SUB_SGDMA_TO_FFT_CSR_BASE 0x80000
#define FFT_SUB_SGDMA_TO_FFT_DESCRIPTOR_SLAVE_BASE 0x90000
#define FFT_SUB_SGDMA_FROM_FFT_CSR_BASE 0xa0000
#define FFT_SUB_SGDMA_FROM_FFT_DESCRIPTOR_SLAVE_BASE 0xb0000
#define FFT_SUB_DATA_BASE 0xc0000
#define FFT_SUB_FFT_STADAPTER_0_BASE 0xd0000

[]$ grep "_IRQ [^0]" qsys_headers/hps_0.h
#define MEMCPY_MSGDMA_CSR_IRQ 7
#define DEMO_DRIVER_SUBSYS_0_IRQ 8
#define FFT_SUB_SGDMA_TO_FFT_CSR_IRQ 4
#define FFT_SUB_SGDMA_FROM_FFT_CSR_IRQ 3

```

This is an example of the base addresses of the FPGA based Qsys peripherals from the perspective of the HPS AXI bridges.

Below is an example of the IRQ values from the same perspective.

NOTE: this is a very small portion of the total contents of the header file macros.

Creating Device Trees

```
[]$ sopc2dts --help
sopc2dts - 14.1 [f3b984376d464370083edffbe85dd18a2bb42a80]
Usage: sopc2dts <arguments>
Required Arguments:
  --input <sopcinfo file>       The sopcinfo file (optional in gui mode) (Required)

  -i <sopcinfo file>    Short for --input

Optional Arguments:
  --board <boardinfo file>      board description file (can be used multiple times) (Optiona
  --bridge-ranges <{none,bridge,child}> What to describe in bridges address translations (Op
  --bridge-removal <{all,balanced,none}>        Bridge removal strategy (Optional)
  --help                Show this usage info and exit (Optional)
  --verbose             Show Lots of debugging info (Optional)
  --gui         Run in gui mode (Optional)
  --clocks              Show clocks in Device Tree Source / graph (Optional)
  --conduits            Show conduit interfaces in graph (Optional)
  --version             Show version information and exit (Optional)
  --mimic-sopc-create-header-files              Try to (mis)behave like sopc-create-header-f
  --no-timestamp                Don't add a timestamp to generated files (Optional)
  --output <filename>   The output filename (Optional)
  --pov <component name>        The point of view to generate from. Defaults to the first cp
  --pov-type <{cpu,pci}>        The point of view device type (Optional)
  --reset               Show reset interfaces in graph (Optional)
  --sort <{none,address,name,label}>    Sort components by (Optional)
  --streaming           Show streaming interfaces in graph (Optional)
  --type <{dtb,dtb-hex8,dtb-hex32,dtb-char-arr,dts,uboot,kernel}>       The type of output t
  --bootargs <kernel-args>      Default kernel arguments for the "chosen" section of the DTS
  --sopc-parameters <{node,cmacro,all}> What sopc-parameters to include in DTS (Optional)

```

Creating Device Trees

```
[]$ sopc2dts --input soc_system.sopcinfo --output soc_system.dts \
          --board hps_common_board_info.xml --bridge-removal all --clocks \
          --board board_info_ALTERA_CV_SOC.xml

[]$ cat soc_system.dts

...cut...

fifo_0: fifo@0x100040020 {
          compatible = "ALTR,fifo-14.1", "ALTR,fifo-1.0";
          reg = <0x00000001 0x00040020 0x00000004>,
                    <0x00000001 0x00040030 0x00000004>,
                    <0x00000001 0x00040000 0x00000020>;
          reg-names = "in", "out", "in_csr";
          clocks = <&clk_0>;
}; //end fifo@0x100040020 (fifo_0)

...cut...

demo_driver_subsys_0: driver@0x100030000 {
          compatible = "demo,driver-1.0", "demo,driver-1.0";
          reg = <0x00000001 0x00030000 0x00001000>;
          interrupt-parent = <&hps_0_arm_gic_0>;
          interrupts = <0 48 4>;
          clocks = <&clk_0>;
}; //end driver@0x100030000 (demo_driver_subsys_0)

...cut...
```

The sopc2dts utility can create device tree entries for the FPGA based Qsys components represented in the SOPCINFO file.

These device tree entries can be compiled by the “dtc” compiler into a binary device tree blob that is used by the linux kernel to define the peripheral space available on the target.

NOTE: newer u-boot versions are driven by device tree definitions as well as the kernel, however u-boot and the kernel require their own unique device tree definitions, generated separately.

Device Tree Documentation

```
Device Tree for Dummies:
http://events.linuxfoundation.org/sites/events/files/slides/
                                                              petazzoni-device-tree-dummies.pdf





[socfpga-3.10-ltsi]$ ls -R Documentation/devicetree/
Documentation/devicetree/:
00-INDEX  bindings  booting-without-of.txt  usage-model.txt

Documentation/devicetree/bindings:
arc         i2c                   net                 sound
arm         iio                   nvec                spi
ata         input                 open-pic.txt        staging
bus         interrupt-controller  pci                 thermal
c6x         iommu                 phy                 timer
clock       leds                  pinctrl             tty
cpufreq     lpddr2                power               usb
crypto      mailbox               powerpc             vendor-prefixes.txt
dma         marvell.txt           power_supply        video
drm         media                 pwm                 virtio
eeprom.txt  memory-controllers    regulator           w1
fb          metag                 reset               watchdog
fpga        mfd                   resource-names.txt  x86
gpio        mips                  rng                 xilinx.txt
gpu         misc                  rtc
hwmon       mmc                   serial
hwrng       mtd                   serio
```

Some publicly available information on device tree.

Device Tree Documentation

```
[socfpga-3.10-ltsi]$ cat Documentation/devicetree/bindings/clock/altr_socfpga.txt
Device Tree Clock bindings for Altera's SoCFPGA platform

This binding uses the common clock binding[1].

[1] Documentation/devicetree/bindings/clock/clock-bindings.txt

Required properties:
- compatible : shall be one of the following:
        "altr,socfpga-pll-clock" - for a PLL clock
        "altr,socfpga-perip-clock" - The peripheral clock divided from the
                PLL clock.
        "altr,socfpga-gate-clk" - Clocks that directly feed peripherals and
                can get gated.

- reg : shall be the control register offset from CLOCK_MANAGER's base for the clock.
- clocks : shall be the input parent clock phandle for the clock. This is
        either an oscillator or a pll output.

Optional properties:
- fixed-divider : If clocks have a fixed divider value, use this property.
- #clock-cells : from common clock binding; shall be set to 0.

Optional properties:
- fixed-divider : If clocks have a fixed divider value, use this property.
- clk-gate : For "socfpga-gate-clk", clk-gate contains the gating register
        and the bit index.
- div-reg : For "socfpga-gate-clk", div-reg contains the divider register, bit shift,
        and width.
```

An example of some bindings documentation.

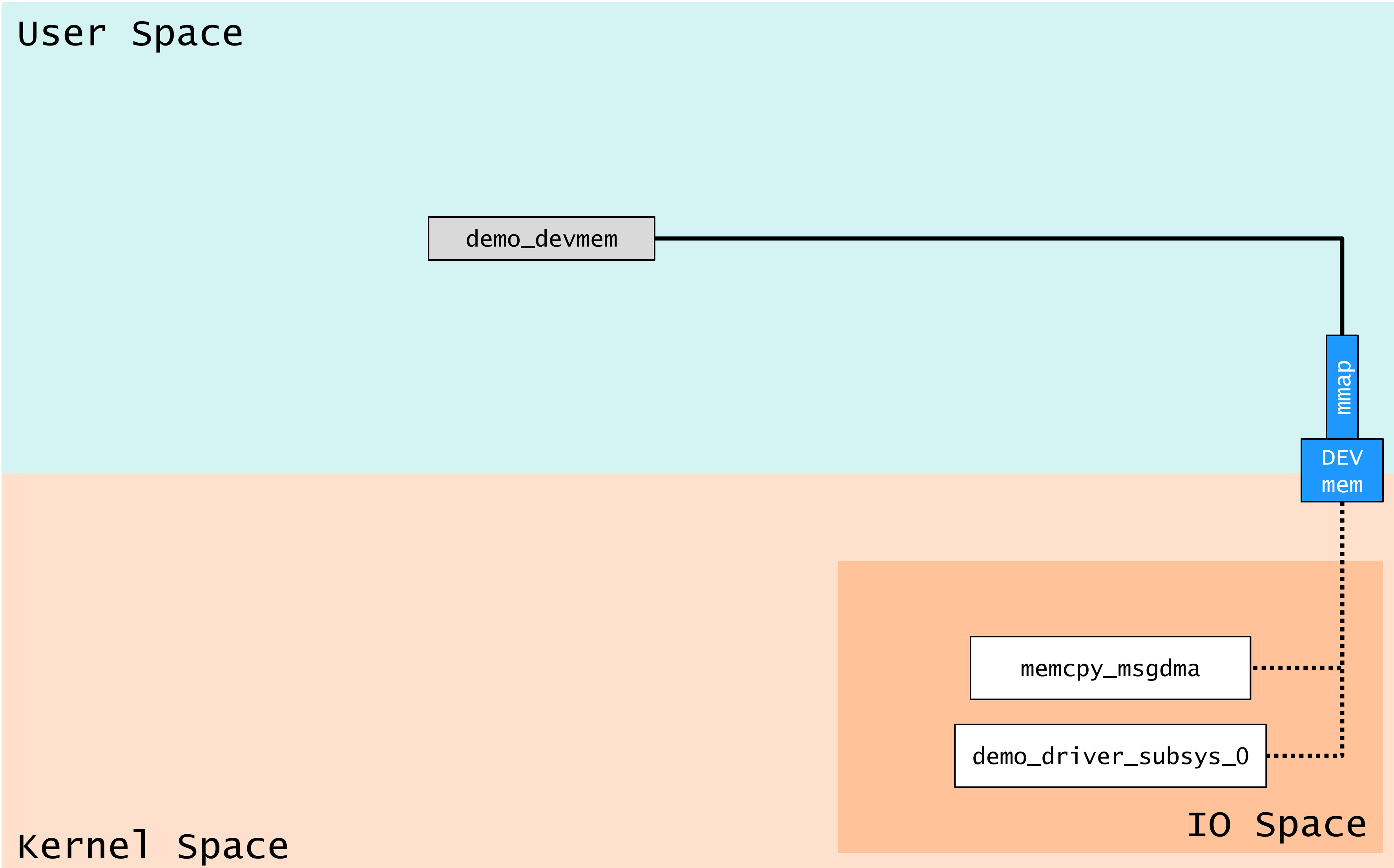
Accessing FPGA hardware, starting off simple with a user space application.

mmap(/dev/mem)

Trivial access to FPGA hardware from user space.

- Often when we begin developing a custom piece of FPGA hardware we know that we will eventually want to have a nice pretty driver that presents a convenient API into user space for applications to interact with, but in the early days of developing the hardware, we don't even know if it's working properly, and we may still have numerous iterations that we apply to the hardware before it stabilizes into the ultimate thing for which we want to create a driver.
- At this early point in time, it can be much less overhead to simply peek and poke at the hardware with some basic stimulus thru a user space application that mmap()'s the hardware using /dev/mem.
  - In some cases this /dev/mem approach may be all you ever need.

demo_devmem



User Space
demo_devmem
mmap
DEV
mem
memcpy_msgdma
demo_driver_subsys_0
IO Space
Kernel Space

Color Key for Code Examples

Color coded indicator of user/kernel space example.

User Space

```
The general context of the code example is not highlighted at all, just black text.

The focus of a code example is highlighted in red text.  If you ignore everything else on
the page in black text, make sure you take note of the red text that is highlighted as it
is relevant to the example.

When the focus of a code example refers to functions and objects provided by the kernel or
standard system calls there is a yellow highlight behind the red text.
```

demo_devmem.c example

User Space

```
int main(int argc, char **argv) {

          int devmem_fd;
          void *demo_driver_map;
          void *memcpy_msgdma_map;
          int result;

          //
          // validate the system features
          //
          validate_system_features();

          //
          // verify that DEMO_DRIVER_PHYS_BASE is page aligned
          //
          if(DEMO_DRIVER_PHYS_BASE & (sysconf(_SC_PAGE_SIZE) - 1)) {
...cut...
          }

          //
          // verify that MEMCPY_MSGDMA_CSR_PHYS_BASE is page aligned
          //
          if(MEMCPY_MSGDMA_CSR_PHYS_BASE & (sysconf(_SC_PAGE_SIZE) - 1)) {
...cut...
          }

          //
          // parse the command line arguments
          //
          parse_cmdline(argc, argv);
```

In the demo_devmem.c example that is provided with this workshop we can see how our user space application starts off. The first few things that it does is attempt to validate its environment.

demo_devmem.c example

User Space

```
        //
        // open() the /dev/mem device
        //
        devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
        if(devmem_fd < 0) {
...cut...
        }

        //
        // mmap() the base of our demo_driver hardware
        //
        demo_driver_map = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ|PROT_WRITE,
                                  MAP_SHARED, devmem_fd, DEMO_DRIVER_PHYS_BASE);
        if(demo_driver_map == MAP_FAILED) {
...cut...
        }

        //
        // mmap() the base of our memcpy_msgdma hardware
        //
        memcpy_msgdma_map = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ|PROT_WRITE,
                                      MAP_SHARED, devmem_fd, MEMCPY_MSGDMA_CSR_PHYS_BASE)
        if(memcpy_msgdma_map == MAP_FAILED) {
...cut...
        }
```

Then we open /dev/mem and we mmap() the base addresses to our demo_driver hardware and our memcpy_msgdma hardware.

demo_devmem.c example

User Space

```
        //
        // perform the operation selected by the command line arguments
        //
        if(g_print_timer             != NULL) do_print_timer(demo_driver_map);
        if(g_dump_rom                != NULL) do_dump_rom(demo_driver_map);
        if(g_dump_ram                != NULL) do_dump_ram(demo_driver_map);
        if(g_fill_ram                != NULL) do_fill_ram(demo_driver_map);
        if(g_dma_rom_ram             != NULL) do_dma_rom_ram(memcpy_msgdma_map);
        if(g_help                    != NULL) do_help();

        //
        // munmap everything and close the /dev/mem file descriptor
        //
        result = munmap(demo_driver_map, sysconf(_SC_PAGE_SIZE));
        if(result < 0) {
...cut...
        }

        result = munmap(memcpy_msgdma_map, sysconf(_SC_PAGE_SIZE));
        if(result < 0) {
...cut...
        }

        close(devmem_fd);
        exit(EXIT_SUCCESS);
```

Then we can use those pointers that we mmap()'ed and then munmap() them when we're finished and close /dev/mem.

demo_devmem.c example

User Space

```
void do_print_timer(void *demo_driver_map) {
		volatile unsigned long *timer_base = demo_driver_map + TIMER_OFST;
		double f_timeout_period = (double)(0.010) / ((double)(1) / (double)(DEMO_DRIVER_FR
		unsigned long timeout_period = f_timeout_period;
		unsigned long timer_snaps_l[4];
		unsigned long timer_snaps_h[4];
		int i;

		//
		// if timer is not running, start it
		//
		if((timer_base[ALTERA_AVALON_TIMER_STATUS_REG] & ALTERA_AVALON_TIMER_STATUS_RUN_MS
			printf("Timer not currently running, initializing and starting timer.\n"
			timeout_period--;
			timer_base[ALTERA_AVALON_TIMER_PERIODL_REG] = timeout_period;
			timer_base[ALTERA_AVALON_TIMER_PERIODH_REG] = timeout_period >> 16;
			timer_base[ALTERA_AVALON_TIMER_CONTROL_REG] = ALTERA_AVALON_TIMER_CONTRO
		}

		//
		// Dump the timer registers
		//
		printf("  status = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_STATUS_REG]);
		printf(" control = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_CONTROL_REG]);
		printf("period_l = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_PERIODL_REG]);
		printf("period_h = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_PERIODH_REG]);
		printf("  snap_l = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_SNAPL_REG]);
		printf("  snap_h = 0x%08lX\n", timer_base[ALTERA_AVALON_TIMER_SNAPH_REG]);
```

We use the mmap()'ed pointer in our C code just like any other pointer.

demo_devmem.c example

User Space

```
void do_dump_rom(void *demo_driver_map) {
        write(STDOUT_FILENO, (void *)(demo_driver_map + ROM_OFST), ROM_SPAN);
}

void do_dump_ram(void *demo_driver_map) {
        write(STDOUT_FILENO, (void *)(demo_driver_map + RAM_OFST), RAM_SPAN);
}

void do_fill_ram(void *demo_driver_map) {
        read(STDIN_FILENO, (void *)(demo_driver_map + RAM_OFST), RAM_SPAN);
}

void do_dma_rom_ram(void *memcpy_msgdma_map) {
        volatile unsigned long *memcpy_msgdma_csr_base = memcpy_msgdma_map;
        volatile unsigned long *memcpy_msgdma_desc_base = memcpy_msgdma_map +
                                                          MEMCPY_MSGDMA_DESC_PHYS_OFST;
        unsigned long temp;

        //
        // make sure the DMA is not busy and the descriptor buffer is empty
        //
        temp = memcpy_msgdma_csr_base[ALTERA_MSGDMA_CSR_STATUS_REG];
        if((temp & ALTERA_MSGDMA_CSR_BUSY_MASK) != 0x00) {
                error(0, 0, "dma is busy before first use");
                exit(EXIT_FAILURE);
        }
        if((temp & ALTERA_MSGDMA_CSR_DESCRIPTOR_BUFFER_EMPTY_MASK) == 0x00) {
                error(0, 0, "dma descriptor buffer is not empty before first use");
                exit(EXIT_FAILURE);
        }
```

A few more examples.

demo_devmem.c example

User Space

```
void validate_system_features(void) {
...cut...
        //
        // test to see that the demo_driver device entry exists in the sysfs
        //
        dirname = DEMO_DRIVER_SYSFS_ENTRY_DIR;
        dp = opendir(dirname);
...cut...
        if(closedir(dp)) {
...cut...
        //
        // test to see that the demo_driver device entry exists in the procfs
        //
        dirname = DEMO_DRIVER_PROCFS_ENTRY_DIR;
        dp = opendir(dirname);
...cut...
        if(closedir(dp)) {
...cut...
        //
        // fetch the clocks value out of our device entry
        //
        filename = DEMO_DRIVER_CLOCKS_ENTRY;
        fd = open(filename, O_RDONLY);
...cut...
        result = read(fd, clocks_array, 4);
...cut...
        if(close(fd)) {
...cut...
```

We use the run time device tree models to validate the system that we're running in before we start touching any hardware.

demo_devmem.h example

```
//
// expected values for the physical base address and clock frequency of the
// demo driver hardware
//
#define DEMO_DRIVER_PHYS_BASE (0xFF230000)
#define DEMO_DRIVER_SYSFS_ENTRY_DIR "/sys/bus/platform/devices/ff230000.driver"
#define DEMO_DRIVER_PROCFS_ENTRY_DIR "/proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x10
#define DEMO_DRIVER_FREQ      (50000000)
#define DEMO_DRIVER_CLOCKS_ENTRY "/proc/device-tree/sopc@0/bridge@0xc0000000/..cut../clocks"
#define H2F_USER1_CLOCK_PHANDLE_ENTRY "/proc/device-tree/clocks/clk_0/linux,phandle"
//
// expected values for the physical base addresses of the memcpy_msgdma core
//
#define MEMCPY_MSGDMA_CSR_PHYS_BASE (0xFF220000)
#define MEMCPY_MSGDMA_DESC_PHYS_BASE (0xFF220020)
#define MEMCPY_MSGDMA_DESC_PHYS_OFST (MEMCPY_MSGDMA_DESC_PHYS_BASE - MEMCPY_MSGDMA_CSR_PHYS_
#define MEMCPY_MSGDMA_REG_NAMES_ENTRY "/proc/device-tree/sopc@0/bridge@0xc0000000/msgdma@0x1
#define MEMCPY_MSGDMA_REG_NAMES_VALUE { 0x63, 0x73, 0x72, 0x00, 0x64, 0x65, 0x73, 0x63, 0x72
0x6f, 0x72, 0x5f, 0x73, 0x6c, 0x61, 0x76, 0x65, 0x00 }
#define MEMCPY_MSGDMA_REG_ENTRY "/proc/device-tree/sopc@0/bridge@0xc0000000/msgdma@0x1000200
#define MEMCPY_MSGDMA_REG_VALUE { 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00
0x00, 0x00, 0x01, 0x00, 0x02, 0x00, 0x20, 0x00, 0x00, 0x00, 0x10 }
//
// demo driver hardware map
//
#define ROM_OFST    (0)
#define ROM_SPAN    (1024)
#define RAM_OFST    (ROM_OFST + ROM_SPAN)
#define RAM_SPAN    (1024)
#define TIMER_OFST  (RAM_OFST + RAM_SPAN)
```

Various system parameters that define our hardware, and device tree values we can validate against at run time from user space.

demo_devmem build script

```
#!/bin/sh

type -t ${CROSS_COMPILE:?}gcc > /dev/null 2>&1 || {
          echo ""
          echo "ERROR: cross compiler tools are not visible in the environment."
          echo ""
          exit 1
}

${CROSS_COMPILE:?}gcc \
          -march=armv7-a \
          -mfloat-abi=hard \
          -mfpu=vfp3 \
          -mthumb-interwork \
          -mthumb \
          -O2 \
          -g \
          -feliminate-unused-debug-types  \
          -std=gnu99 \
          -W \
          -Wall \
          -Werror \
          -Wc++-compat \
          -Wwrite-strings \
          -Wstrict-prototypes \
          -pedantic \
          -o demo_devmem \
          demo_devmem.c
```

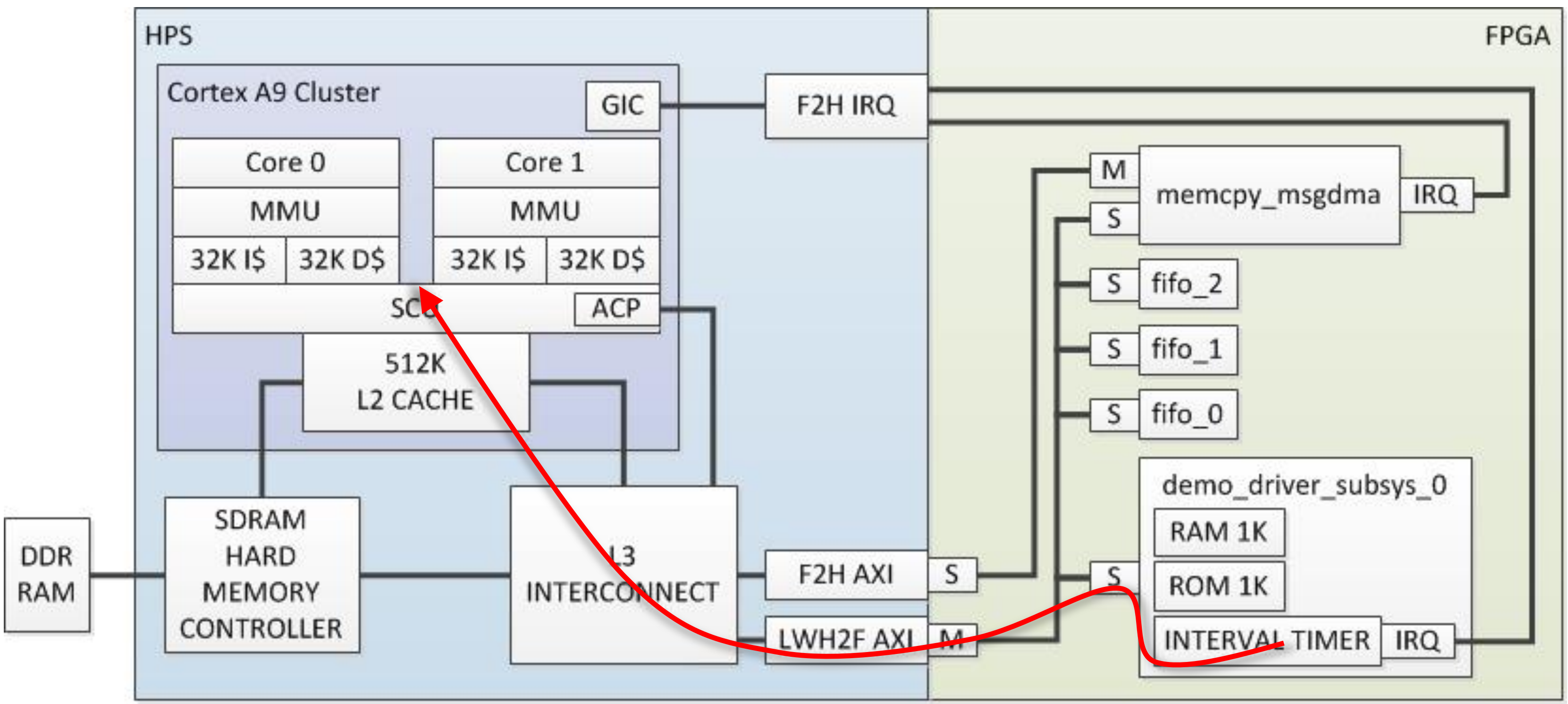
A typical gcc invocation is all it takes to build the application.

demo_devmem target demonstration

```
alias ls='ls --color=never‘                # if you’d like to disabled colored output for ‘ls’
cd /examples/drivers                       # this is where the example programs are installed

./demo_devmem -h
./demo_devmem -t
./demo_devmem -o | hexdump -Cv
./demo_devmem -a | hexdump -Cv
dd if=/dev/zero | ./demo_devmem -f
./demo_devmem -a | hexdump -Cv
./demo_devmem -d
./demo_devmem -a | hexdump -Cv
```

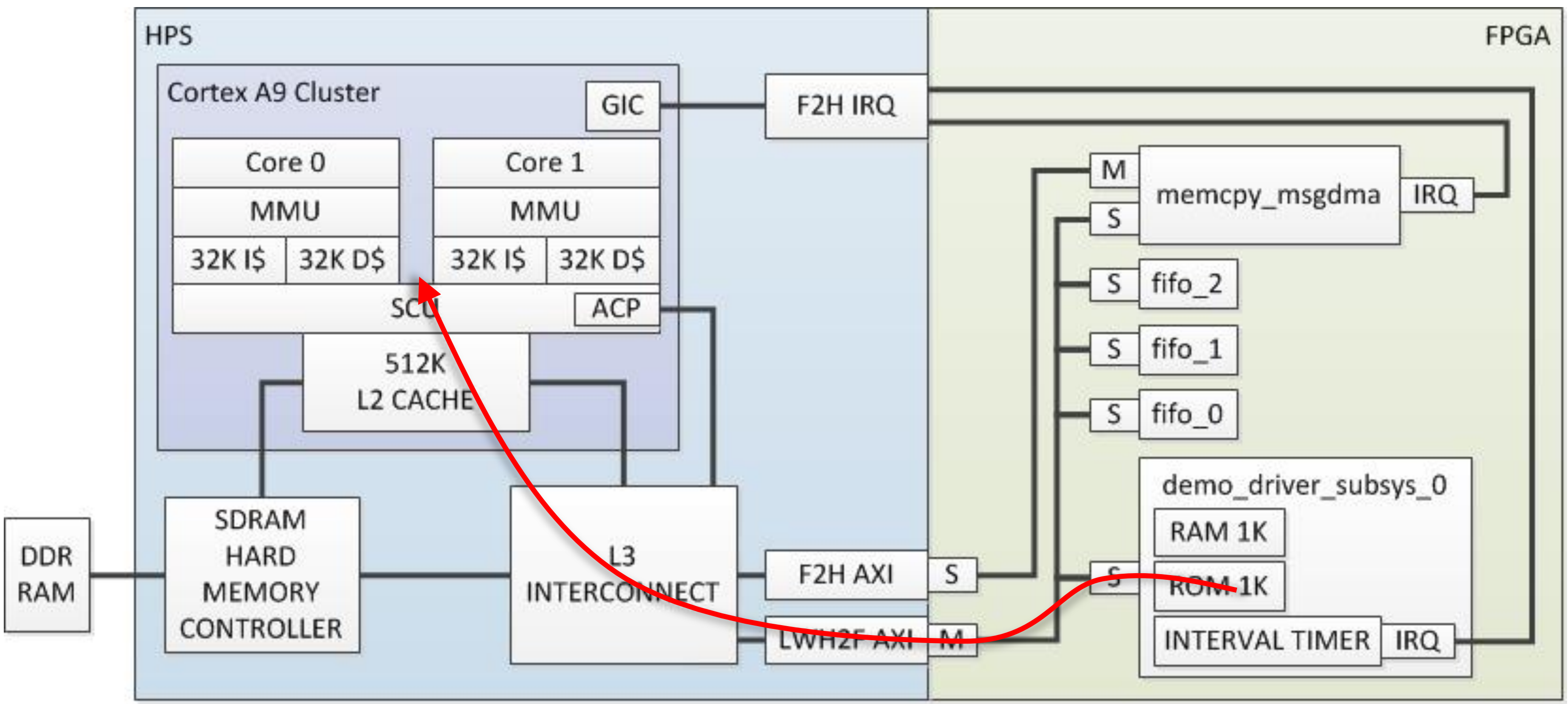
demo_devmem target demonstration

```
./demo_devmem -t

CPU reads from timer registers.
```



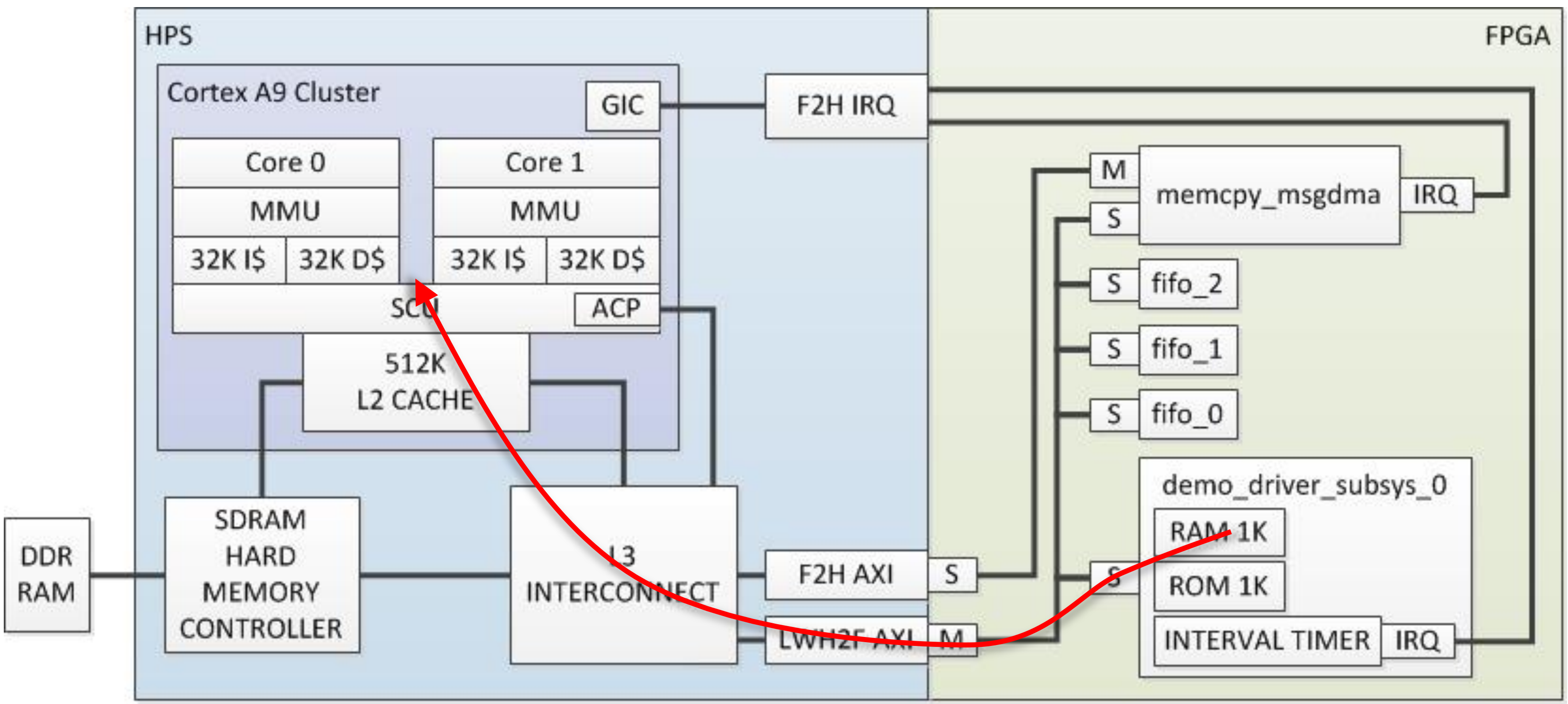
demo_devmem target demonstration

```
./demo_devmem -o

CPU reads from ROM.
```



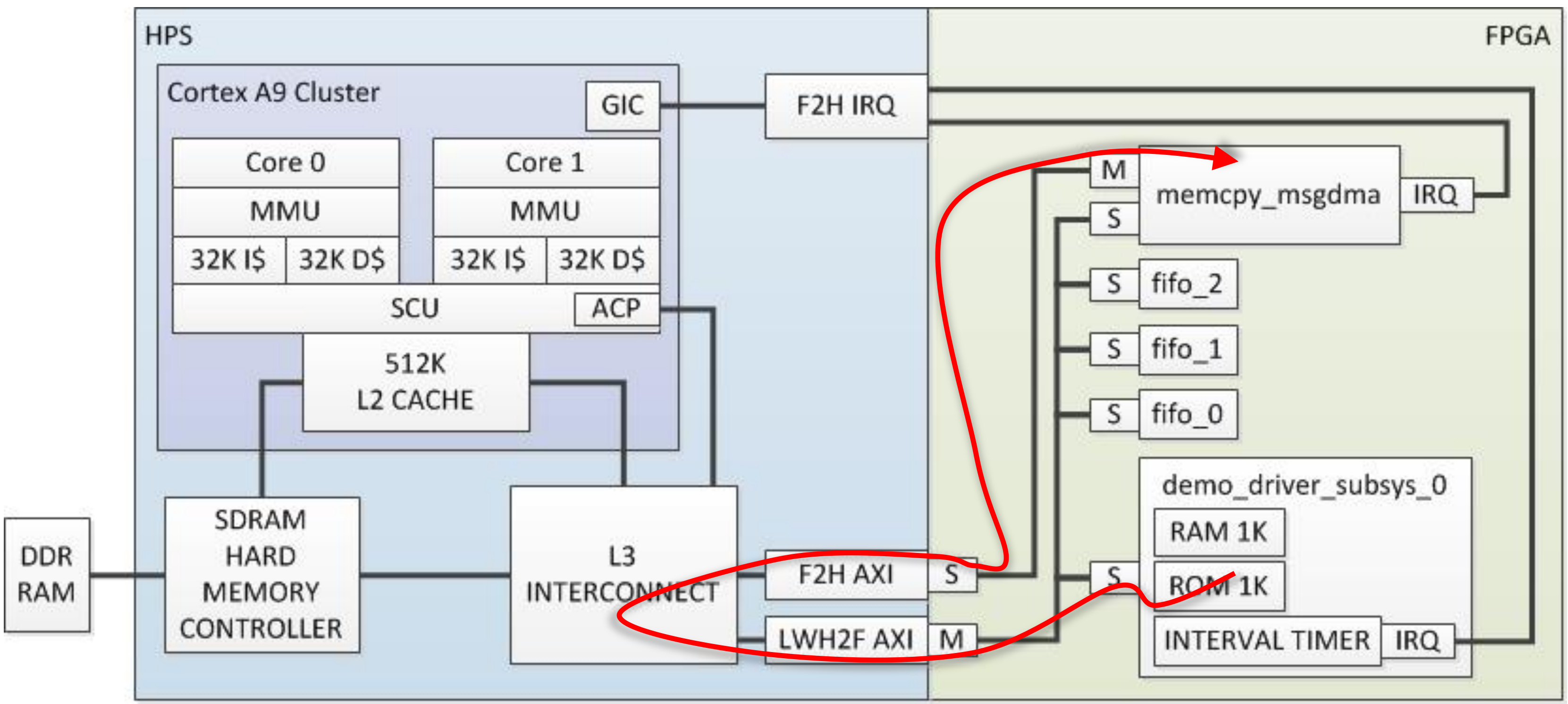
demo_devmem target demonstration

```
./demo_devmem –a

CPU reads from RAM.
```



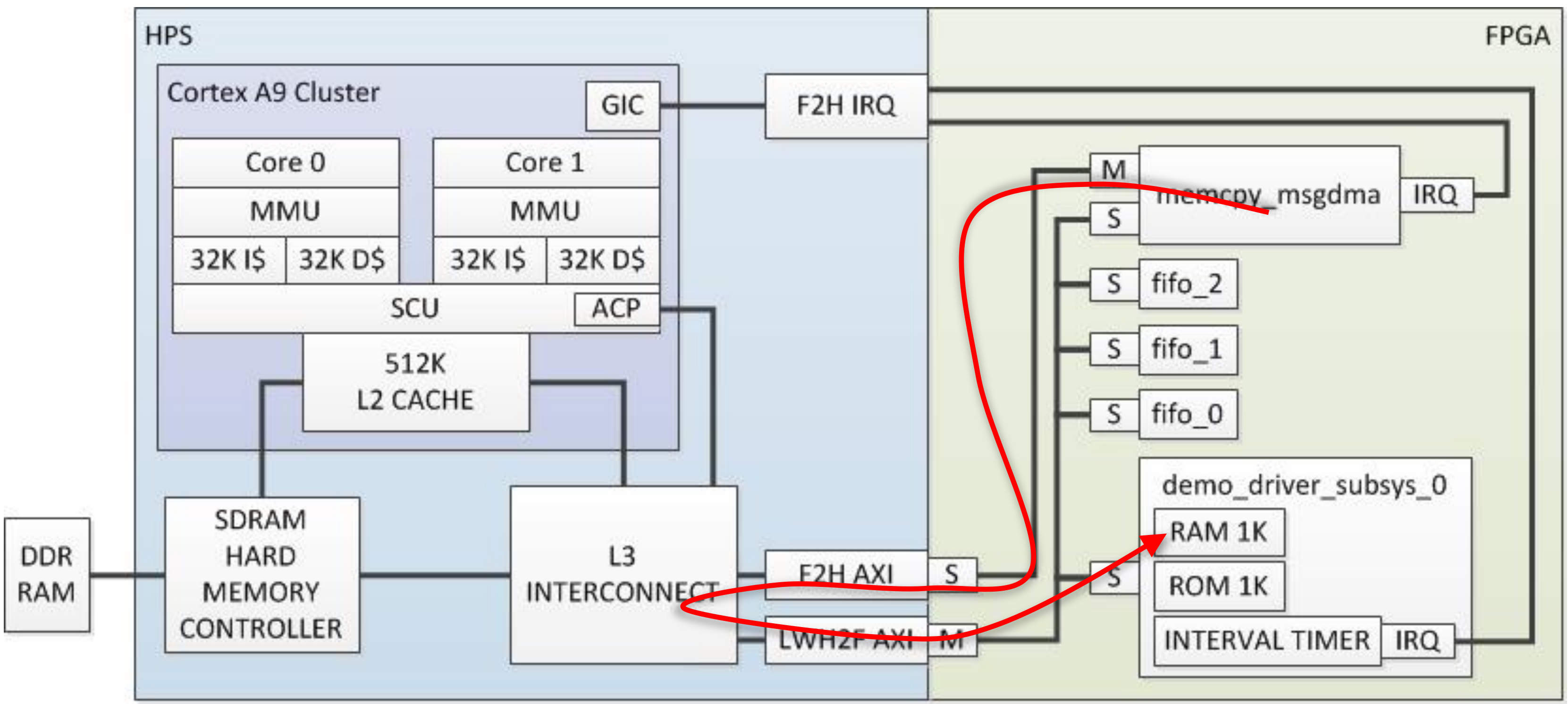
demo_devmem target demonstration

```
./demo_devmem -d

MSGDMA reads from ROM.
```



demo_devmem target demonstration

```
./demo_devmem -d

MSGDMA writes to RAM.
```



/dev/mem summary

- Using /dev/mem to mmap() IO space into your user space application can be quick and convenient with little overhead.
- mmap()'ing /dev/mem allows you to have peek and poke access into the IO space of your FPGA hardware.
- mmap()'ing /dev/mem will NOT allow you have access to kernel space memory, or any other user space memory outside your own application's memory.
- mmap()'ing /dev/mem will NOT allow you to register an interrupt handler or interact with hardware interrupts in any way.

Beginning device driver development

Some helpful pointers and documentation

- The Altera linux socfpga repository is located on rocketboards.org.
  - git clone git://git.rocketboards.org/linux-socfpga.git
- Module writers should be familiar with the Kbuild environment used by the kernel build flow.
  - linux-socfpga/Documentation/kbuild
  - linux-socfpga/Documentation/kbuild/modules.txt
- Module writers should be familiar with the kernel coding style, especially if you plan to upstream your work.
  - linux-socfpga/Documentation/CodingStyle
  - ${OUT_DIR}/scripts/Lindent <source file>
  - ${OUT_DIR}/scripts/checkpatch.pl --file --strict <source file>
- Kernel tainting.
  - linux-socfpga/Documentation/oops-tracing.txt
- GIT resources and documentation.
  - http://git-scm.com/
  - http://gitref.org/

Setup the development environment

```
#
# ARCH should be arm for the Cortex A9
#
export ARCH=arm

#
# CROSS_COMPILE should point into the target cross compiler tools directory and
# contain the prefix used by all the target tools in that directory
#
export CROSS_COMPILE=...cut.../soc_workshop/toolchain/bin/arm-linux-gnueabihf-

#
# OUT_DIR should point into the linux kernel build directory, that's where the
# .config file resides for the kernel that we're building against.
#
export OUT_DIR=...cut.../soc_workshop/socfpga-3.10-ltsi
```

When we’re going to be compiling drivers against the kernel that we built for our target platform, it’s rather convenient to define these three variables in our environment so the kernel makefiles have access to their values.

To compile a kernel module, our architecture is always going to be “arm”, we need to have access to the cross compiler tools that the kernel was built with, and we need to have access to the kernel output directory, so our module can inherit the kernel configuration that we’re running on the target.

Setup the development environment

```
[]$ echo ${CROSS_COMPILE}
...cut.../soc_workshop/toolchain/bin/arm-linux-gnueabihf-
[]$ ls ...cut.../soc_workshop/toolchain/bin/
arm-linux-gnueabihf-addr2line      arm-linux-gnueabihf-gfortran
arm-linux-gnueabihf-ar             arm-linux-gnueabihf-gprof
arm-linux-gnueabihf-as             arm-linux-gnueabihf-ld
arm-linux-gnueabihf-c++            arm-linux-gnueabihf-ld.bfd
arm-linux-gnueabihf-c++filt        arm-linux-gnueabihf-ldd
arm-linux-gnueabihf-cpp            arm-linux-gnueabihf-ld.gold
arm-linux-gnueabihf-ct-ng.config   arm-linux-gnueabihf-nm
arm-linux-gnueabihf-dwp            arm-linux-gnueabihf-objcopy
arm-linux-gnueabihf-elfedit        arm-linux-gnueabihf-objdump
arm-linux-gnueabihf-g++            arm-linux-gnueabihf-pkg-config
arm-linux-gnueabihf-gcc            arm-linux-gnueabihf-pkg-config-real
arm-linux-gnueabihf-gcc-4.9.2      arm-linux-gnueabihf-ranlib
arm-linux-gnueabihf-gcc-ar         arm-linux-gnueabihf-readelf
arm-linux-gnueabihf-gcc-nm         arm-linux-gnueabihf-size
arm-linux-gnueabihf-gcc-ranlib     arm-linux-gnueabihf-strings
arm-linux-gnueabihf-gcov           arm-linux-gnueabihf-strip
arm-linux-gnueabihf-gdb
[]$ ls ${OUT_DIR}
arch           firmware  kernel           Module.symvers   sound
block          fs        lib              net              System.map
COPYING        include   MAINTAINERS      README           tools
CREDITS        init      Makefile         REPORTING-BUGS   usr
crypto         ipc       mm               samples          virt
Documentation  Kbuild    modules.builtin  scripts          vmlinux
drivers        Kconfig   modules.order    security         vmlinux.o
```

The typical Linaro GCC tools for ARM linux development.

The standard kernel source tree/output directory.

Kbuild file

```
[]$ cat Kbuild
obj-m  := demo_module_01.o
obj-m  += demo_module_01t.o
obj-m  += demo_module_02.o
obj-m  += demo_module_03.o
obj-m  += demo_module_04.o
obj-m  += demo_module_05.o
obj-m  += demo_module_05t.o
obj-m  += demo_module_06.o
obj-m  += demo_module_07.o
obj-m  += demo_module_08.o
obj-m  += demo_module_09.o
obj-m  += demo_module_10.o
obj-m  += demo_module_11.o
obj-m  += demo_module_11t.o
```

We start with a standard Kbuild file to define what modules we wish to build in our development directory.

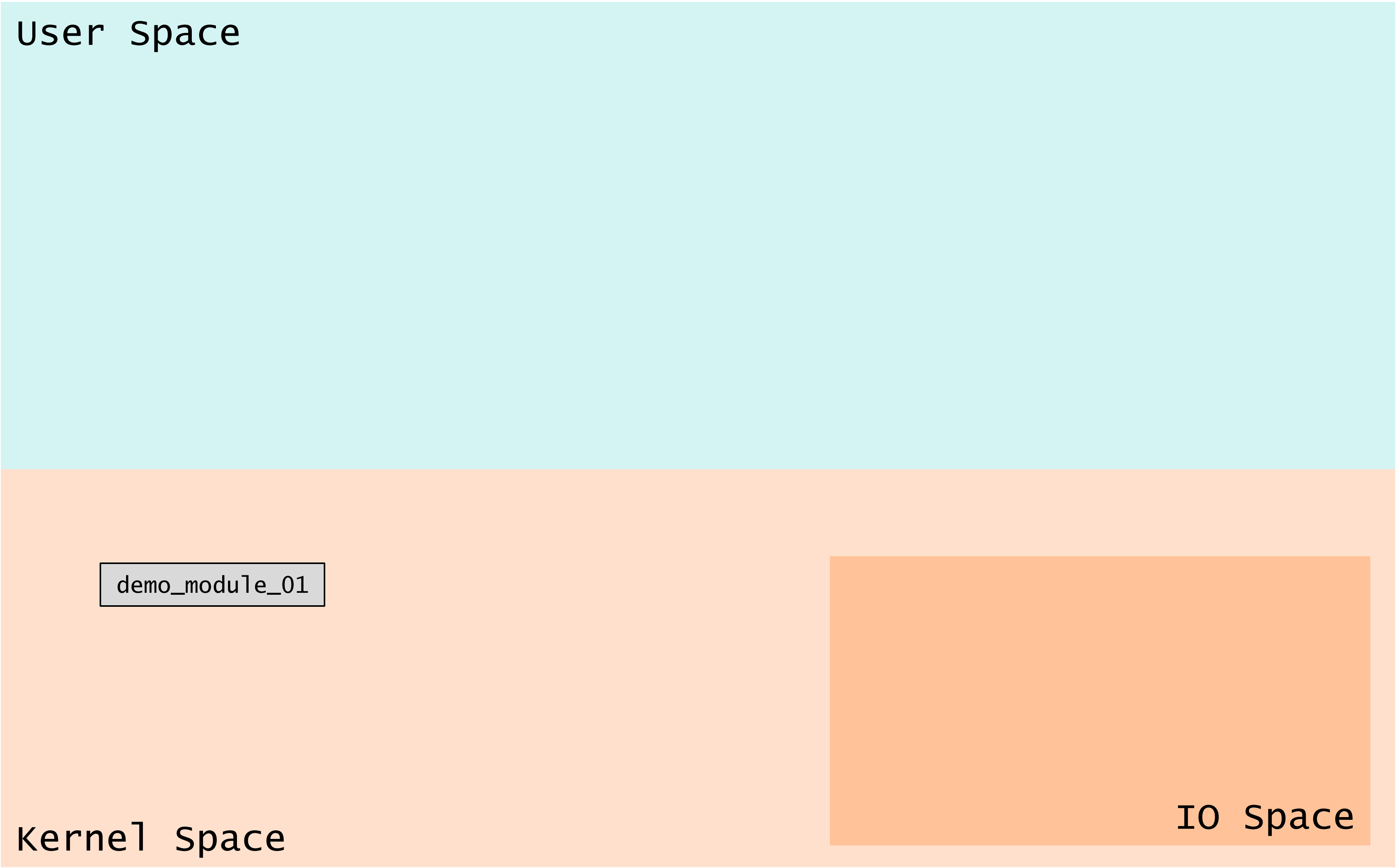
Makefile

```
[]$ cat Makefile
ifndef OUT_DIR
    $(error OUT_DIR is undefined, bad environment,
you point OUT_DIR to the linux kernel build output
directory)
endif

KDIR ?= $(OUT_DIR)

default:
        $(MAKE) -C $(KDIR) M=$$PWD

clean:
        $(MAKE) -C $(KDIR) M=$$PWD clean

help:
        $(MAKE) -C $(KDIR) M=$$PWD help

modules:
        $(MAKE) -C $(KDIR) M=$$PWD modules

modules_install:
        $(MAKE) -C $(KDIR) M=$$PWD modules_install
```

We use a standard module Makefile in our development directory.

Ready to write some code

demo_module_01



Licensing

```
/*
 * Copyright (C) 2015 Altera Corporation
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc., 59 Temple
 * Place, Suite 330, Boston, MA 02111-1307 USA
 */
```

The last thing that most folks care to think about when they start writing a piece of software is the legal licensing requirements of the software they are about to write. But if you ever plan to share your software with anyone else this is one of the most important pieces of the source file, as it should clearly specify what rights anyone else has to use this code and what obligations they assume if they do.

The demonstration modules provided in this workshop all carry the GPLv2 license shown above. For more information on open source software licenses please see: http://opensource.org/licenses

demo_module_01.c – the most basic module

Kernel Space

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

static int demo_init(void)
{
		pr_info("demo_init enter\n");
		pr_info("demo_init exit\n");
		return 0;
}

static void demo_exit(void)
{
		pr_info("demo_exit enter\n");
		pr_info("demo_exit exit\n");
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Driver Student One <dso@company.com>");
MODULE_AUTHOR("Driver Student Two <dst@organization.org>");
MODULE_DESCRIPTION("Demonstration Module 1 - the most basic module example");
MODULE_VERSION("1.0");
```

Every module has an init and an exit function. There are some macros that you could invoke if you just want some rather standard boilerplate init and exit functionality, but every module will have an init and exit entry point.

demo_module_01.c – the most basic module

Kernel Space

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

static int demo_init(void)
{
		pr_info("demo_init enter\n");
		pr_info("demo_init exit\n");
		return 0;
}

static void demo_exit(void)
{
		pr_info("demo_exit enter\n");
		pr_info("demo_exit exit\n");
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Driver Student One <dso@company.com>");
MODULE_AUTHOR("Driver Student Two <dst@organization.org>");
MODULE_DESCRIPTION("Demonstration Module 1 - the most basic module example");
MODULE_VERSION("1.0");
```

These macros export the symbols for the init and exit functions such that the kernel code that loads your module can identify these entry points.

demo_module_01.c – the most basic module

Kernel Space

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

static int demo_init(void)
{
		pr_info("demo_init enter\n");
		pr_info("demo_init exit\n");
		return 0;
}

static void demo_exit(void)
{
		pr_info("demo_exit enter\n");
		pr_info("demo_exit exit\n");
}

module_init(demo_init);
module_exit(demo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Driver Student One <dso@company.com>");
MODULE_AUTHOR("Driver Student Two <dst@organization.org>");
MODULE_DESCRIPTION("Demonstration Module 1 - the most basic module example");
MODULE_VERSION("1.0");
```

There are a collection of macros used to identify various attributes about a module. These strings get packaged into the module and can be accessed by various tools.

The most important module description macro is the MODULE_LICENSE macro. If this macro is not set to some sort of GPL license tag, then the kernel will become “tainted” when you load your module.

Kernel Space

demo_module_01t.c – a tainted module

```
[]$ diff -u demo_module_01.c demo_module_01t.c
--- demo_module_01.c
+++ demo_module_01t.c
@@ -34,8 +34,10 @@
 module_init(demo_init);
 module_exit(demo_exit);

+/*
 MODULE_LICENSE("GPL");
+*/
 MODULE_AUTHOR("Driver Student One <dso@company.com>");
 MODULE_AUTHOR("Driver Student Two <dst@organization.org>");
-MODULE_DESCRIPTION("Demonstration Module 1 - the most basic module example");
+MODULE_DESCRIPTION("Demonstration Module 1t - tainted module example");
 MODULE_VERSION("1.0");
```

demo_module_01t.c demonstrates a tainted module, simply by commenting out the MODULE_LICENCE macro.

Building the module

- Once we have the environment set up with pointers to the cross compiler tools and the kernel output directory, and we've created a Kbuild file and a Makefile for our project, and we've written our module source C file, then all we need to do to build it is run "make".
- The output of a successful module build will be a *.ko file.
- This is the file that we can copy over to our target and load as a module to run it on the target.

modinfo

```
[]$ modinfo demo_module_01.ko
filename:       ...cut.../demo_module_01.ko
version:        1.0
description:    Demonstration Module 1 - the most basic module example
author:         Driver Student Two <dst@organization.org>
author:         Driver Student One <dso@company.com>
license:        GPL
srcversion:     53D0236960C81C1164C0F6D
depends:
vermagic:       3.10.31-ltsi SMP mod_unload ARMv7 p2v8
[]$ modinfo demo_module_01t.ko
filename:       ...cut.../demo_module_01t.ko
version:        1.0
description:    Demonstration Module 1t - tainted module example
author:         Driver Student Two <dst@organization.org>
author:         Driver Student One <dso@company.com>
srcversion:     53C8A51D7CA7ED5FB6C7217
depends:
vermagic:       3.10.31-ltsi SMP mod_unload ARMv7 p2v8
```

Once our module is built, we can use modinfo to see what the MODULE_* macros defined about our module. Notice the lack of a license property for the demo_module_01t.ko module.

demo_module_01.ko target demonstration

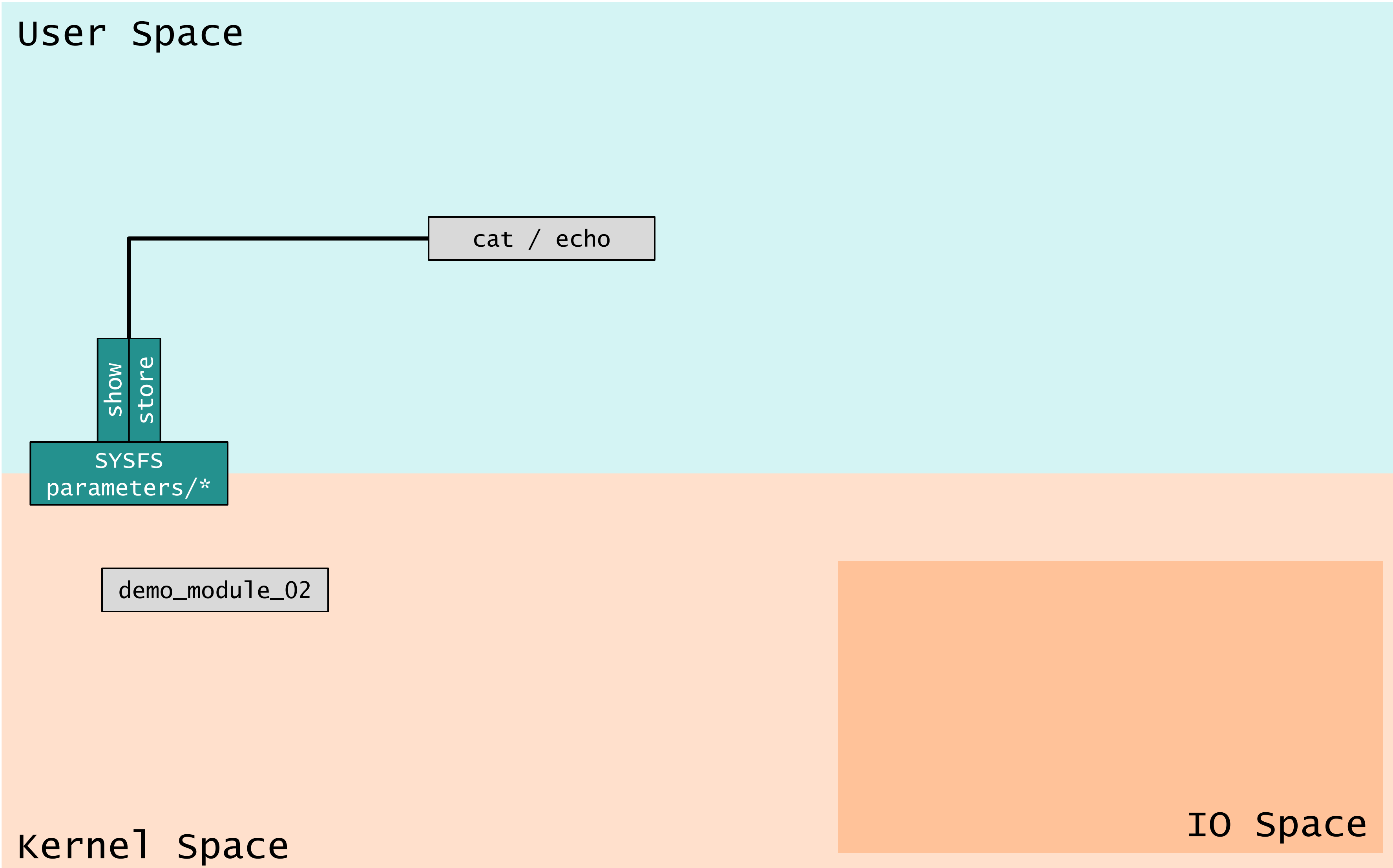
```
cat /proc/sys/kernel/tainted                 # should be 0 = untainted
lsmod                                        # should be "Not tainted"
cat /proc/modules                            # this is what lsmod formats for you
insmod demo_module_01.ko                     # insert our module
cat /proc/sys/kernel/tainted                 # should be 4096 = GPL, out of tree
lsmod                                        # should be "Tainted: G"
insmod demo_module_01t.ko                    # insert our tainted module
cat /proc/sys/kernel/tainted                 # should be 4097 = proprietary, oot
lsmod                                        # should be "Tainted: P"
cat /proc/modules                            # our modules appear (PO) and (O)
find /sys -name "*demo*"                     # find our modules in sysfs
ls /sys/module/demo_module_01                # see what the directory contains
cat /sys/module/demo_module_01/taint         # should be "O" = GPL, out of tree
cat /sys/module/demo_module_01t/taint        # should be "PO" = propprietary, oot
rmmod demo_module_01t                        # remove our modules
rmmod demo_module_01
lsmod                                        # still tainted
cat /proc/sys/kernel/tainted                 # still tainted
ls /sys/module/                              # these are all the modules, in tree
tail /var/log/messages                       # the printk messages from console
```

If you’re target has not been freshly booted, you’re kernel may already be tainted. If so, consider rebooting the target to start with a fresh untainted environment.

Module parameters

For more information:
include/linux/moduleparam.h

demo_module_02



demo_module_02.c – module parameters

Kernel Space

```
/* define module parameters */
static unsigned char param_byte = 0xFF;
static short param_short = 0xFFFF;
static unsigned short param_ushort = 0xFFFF;
static int param_int = 0xFFFFFFFF;
static unsigned int param_uint = 0xFFFFFFFF;
static long param_long = 0xFFFFFFFF;
static unsigned long param_ulong = 0xFFFFFFFF;
static bool param_bool = 1;
static char *param_charp;

module_param(param_byte, byte, S_IRUGO);
module_param(param_short, short, S_IRUGO);
module_param(param_ushort, ushort, S_IRUGO);
module_param(param_int, int, S_IRUGO);
module_param(param_uint, uint, S_IRUGO);
module_param(param_long, long, S_IRUGO);
module_param(param_ulong, ulong, S_IRUGO);
module_param(param_bool, bool, S_IRUGO);
module_param(param_charp, charp, S_IRUGO);


static int param_int_array[] = { 1, 2, 3, 4 };

static unsigned int param_int_array_count;

module_param_array(param_int_array, int, &param_int_array_count, S_IRUGO);
```

We can define parameters for our module which allows us to pass in command line arguments to override these default values at runtime when we insmod or modprobe the module.

We start by defining initialized global variables, these are the default values unless overridden when loaded.

Then this module_param() macro informs the kernel that these variables are parameters to the module.

This is an array parameter example here.

demo_module_02.c – module parameters

Kernel Space

```
MODULE_PARM_DESC(param_byte, "a byte parameter");
MODULE_PARM_DESC(param_short, "a short parameter");
MODULE_PARM_DESC(param_ushort, "a ushort parameter");
MODULE_PARM_DESC(param_int, "a int parameter");
MODULE_PARM_DESC(param_uint, "a uint parameter");
MODULE_PARM_DESC(param_long, "a long parameter");
MODULE_PARM_DESC(param_ulong, "a ulong parameter");
MODULE_PARM_DESC(param_bool, "a bool parameter");
MODULE_PARM_DESC(param_charp, "a charp parameter");
MODULE_PARM_DESC(param_int_array, "an array of int parameters");
```

Remember those MODULE_* macros that we saw before. MODULE_PARAM_DESC will put these strings in the module, which makes them visible to modinfo dumps.

demo_module_02.c – module parameters

Kernel Space

```
static int demo_init(void)
{
        pr_info("demo_init enter\n");

        pr_info("\n");
        pr_info("param_byte   = 0x%02x : %u\n", param_byte, param_byte);
        pr_info("param_short  = 0x%04x : %d\n", param_short, param_short);
        pr_info("param_ushort = 0x%04x : %u\n", param_ushort, param_ushort);
        pr_info("param_int    = 0x%08x : %d\n", param_int, param_int);
        pr_info("param_uint   = 0x%08x : %u\n", param_uint, param_uint);
        pr_info("param_long   = 0x%08lx : %ld\n", param_long, param_long);
        pr_info("param_ulong  = 0x%08lx : %lu\n", param_ulong, param_ulong);
        pr_info("param_bool   = %d\n", param_bool);
        pr_info("param_charp  = \'%s\'\n", param_charp);
        pr_info("\n");
        pr_info("param_int_array_count = %d\n", param_int_array_count);
        pr_info("param_int_array[0] = %d\n", param_int_array[0]);
        pr_info("param_int_array[1] = %d\n", param_int_array[1]);
        pr_info("param_int_array[2] = %d\n", param_int_array[2]);
        pr_info("param_int_array[3] = %d\n", param_int_array[3]);
        pr_info("\n");

        pr_info("demo_init exit\n");
        return 0;
}
```

This example simply dumps all the parameter values from it’s init routine.

demo_module_02.ko target demonstration

```
insmod demo_module_02.ko                        # insert the module without parameters
rmmod demo_module_02                            # remove the module

insmod demo_module_02.ko \
param_byte=0x12          \
param_short=0x3456       \
param_ushort=0x789A      \
param_int=0x3CDEF012     \
param_uint=0x3456789A    \
param_long=0x3CDEF012    \
param_ulong=0x3456789A   \
param_bool=n             \
param_charp=hello-tim    \
param_int_array=5000,-6000,7000         # insert the module with parameters

ls /sys/module/demo_module_02/                            # notice the parameters directory
ls /sys/module/demo_module_02/parameters/                 # notice the parameter files
cat /sys/module/demo_module_02/parameters/param_bool
cat /sys/module/demo_module_02/parameters/param_byte | hexdump -C
cat /sys/module/demo_module_02/parameters/param_charp
cat /sys/module/demo_module_02/parameters/param_int
cat /sys/module/demo_module_02/parameters/param_int_array
cat /sys/module/demo_module_02/parameters/param_long
printf "0x%08X\n" $(cat /sys/module/demo_module_02/parameters/param_long)

rmmod demo_module_02                            # remove the module
```

Observe the output on the printk console.

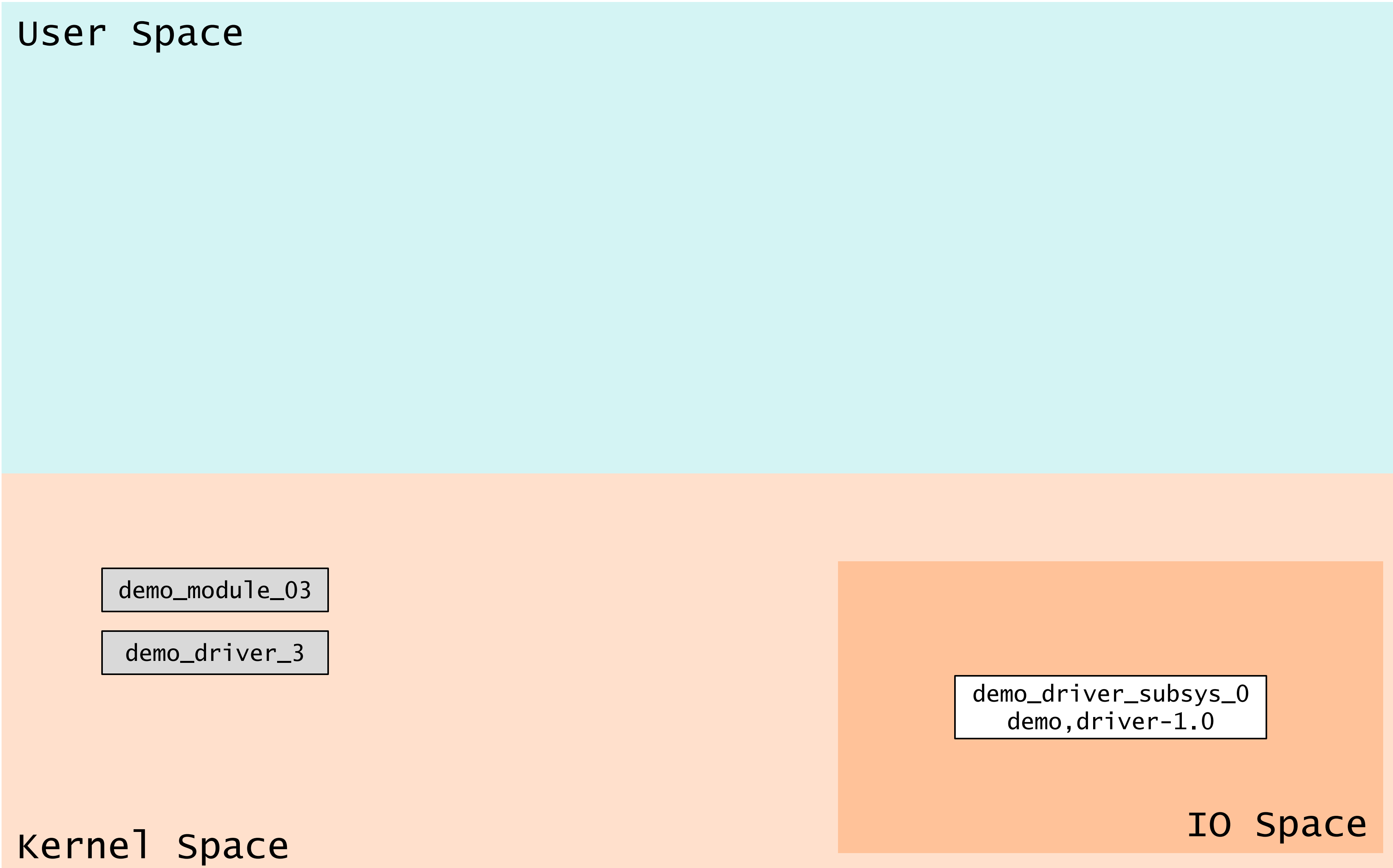
Platform drivers

For more information:

Documentation/driver-model/platform.txt

Documentation/devicetree/

demo_module_03



demo_module_03.c – platform driver

Kernel Space

```
#include <linux/module.h>
#include <linux/platform_device.h>

static struct of_device_id demo_driver_dt_ids[] = {
          {
           .compatible = "demo,driver-1.0“
          },
          { /* end of table */ }
};

MODULE_DEVICE_TABLE(of, demo_driver_dt_ids);

static struct platform_driver the_platform_driver = {
          .probe = platform_probe,
          .remove = platform_remove,
          .driver = {
                       .name = "demo_driver_3",
                       .owner = THIS_MODULE,
                       .of_match_table = demo_driver_dt_ids,
                       },
          /*
             .shutdown = unused,
             .suspend = unused,
             .resume = unused,
             .id_table = unused,
           */
};
```

We create an array of of_device_id’s where we specify “.compatible” strings that the kernel will use to bind our driver to any devices represented in the device tree when our module is inserted. This will automatically trigger our driver probe if the device tree contains a compatible device entry.

We also define an instance for our platform_driver.

demo_module_03.c – platform driver

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
        pr_info("platform_probe enter\n");
        pr_info("platform_probe exit\n");
        return 0;
}

static int platform_remove(struct platform_device *pdev)
{
        pr_info("platform_remove enter\n");
        pr_info("platform_remove exit\n");
        return 0;
}
```

We define our probe and remove functions.

demo_module_03.c – platform driver

Kernel Space

```
static int demo_init(void)
{
        int ret_val;
        pr_info("demo_init enter\n");

        ret_val = platform_driver_register(&the_platform_driver);
        if (ret_val != 0) {
                pr_err("platform_driver_register returned %d\n", ret_val);
                return ret_val;
        }

        pr_info("demo_init exit\n");
        return 0;
}

static void demo_exit(void)
{
        pr_info("demo_exit enter\n");

        platform_driver_unregister(&the_platform_driver);

        pr_info("demo_exit exit\n");
}
```

We register our platform driver in our module init function and we unregister our platform driver in our module exit function.

demo_module_03.ko target demonstration

```
insmod demo_module_03.ko                             # insert the module
find /sys -name "*demo*"                             # find sysfs entries
ls -l /sys/module/demo_module_03/drivers/            # this is a link to the driver entry
ls /sys/module/demo_module_03                        # this looks like the module we saw before
ls /sys/bus/platform/drivers/demo_driver_3           # this is a platform driver entry

# this links to the device entry
ls -l /sys/bus/platform/drivers/demo_driver_3/ff230000.driver

# we can unbind the device
echo "ff230000.driver" > /sys/bus/platform/drivers/demo_driver_3/unbind
ls /sys/bus/platform/drivers/demo_driver_3           # and we see the result

# we can bind the device
echo "ff230000.driver" > /sys/bus/platform/drivers/demo_driver_3/bind
ls /sys/bus/platform/drivers/demo_driver_3           # and we see the result
rmmod demo_module_03                                 # remove the device
```

Observe the output on the printk console.

Extracting information from the device tree

For more information:

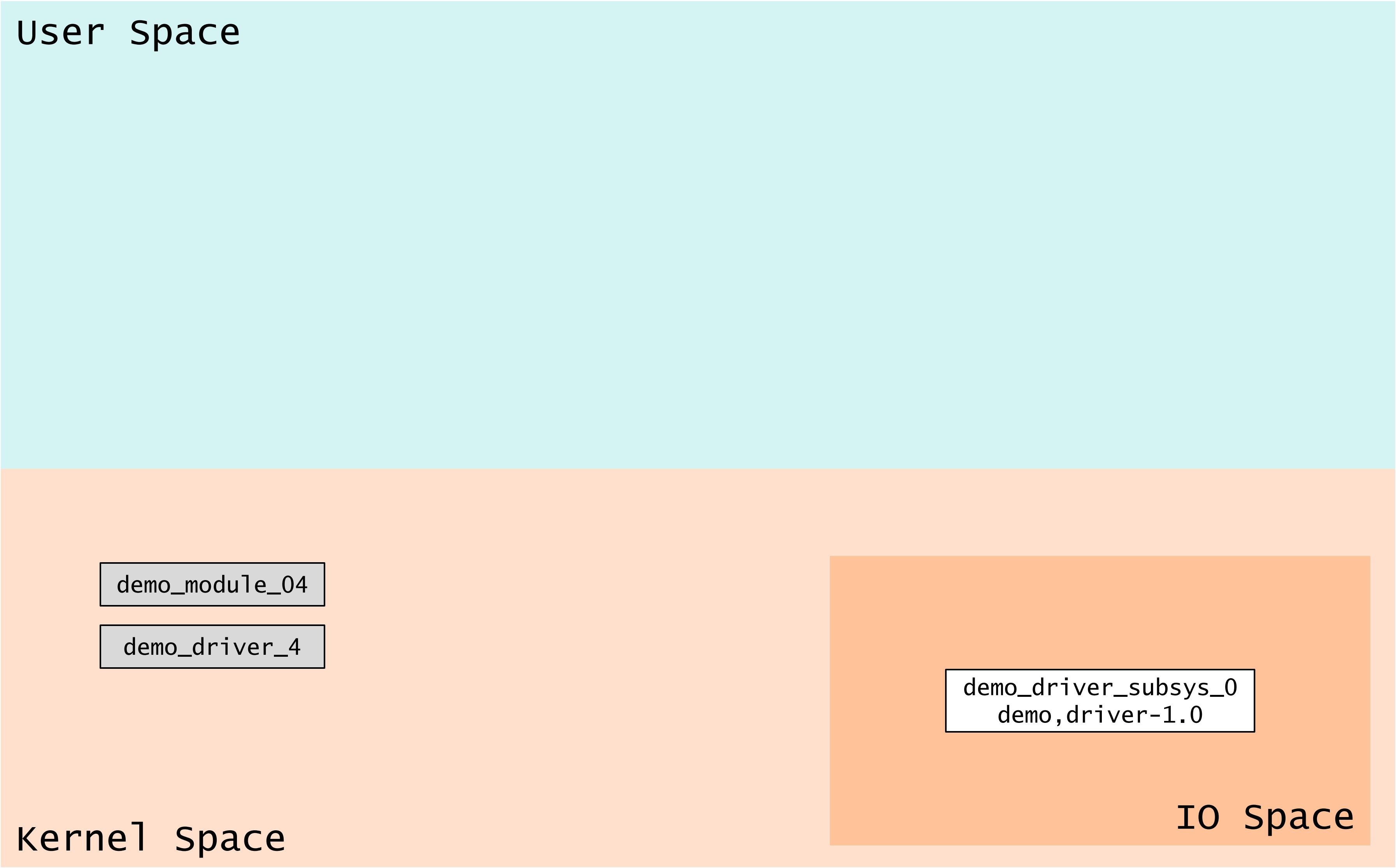
Documentation/devicetree/

Documentation/clk.txt

Retrieving our properties from the device tree

- There are three properties that are of significant interest to most hardware blocks and the software drivers that control them.
  - Memory region
    - every hardware peripheral will likely consume some span of the address map, and the driver needs to know where that is so that it can map it to a pointer that can interact with that region.
  - Interrupt signal
    - peripherals that rely on a hardware interrupt signal need to know what interrupt line that event is signaled on so that an interrupt handler may be registered to handle it.
  - Clock
    - some peripherals that are sensitive to their clock properties will need to know what those details are.

demo_module_04



demo_module_04.c – extracting from device tree

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
        int ret_val;
        struct resource *r;
        int irq;
        struct clk *clk;
        unsigned long clk_rate;

        pr_info("platform_probe enter\n");

        ret_val = -EINVAL;

        /* get our first memory resource */
        r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if (r != NULL) {
                pr_info("r->start = 0x%08lx\n", (long unsigned int)r->start);
                pr_info("r->end = 0x%08lx\n", (long unsigned int)r->end);
                pr_info("r->name = %s\n", r->name);
                if (r->start & ~PAGE_MASK) {
...cut...
                }
                if (((r->end - r->start) + 1) > PAGE_SIZE) {
...cut...
                }
        } else {
                pr_err("IORESOURCE_MEM, 0 does not exist\n");
                goto bad_exit_return;
        }

        g_demo_driver_base_addr = r->start;
```

We retrieve the memory regions allocated by our device by calling platform_get_resource().

demo_module_04.c – extracting from device tree

Kernel Space

```
            /* get our interrupt resource */
            irq = platform_get_irq(pdev, 0);
            if (irq < 0) {
...cut...
            } else {
                        pr_info("irq = %d\n", irq);
            }

            g_demo_driver_irq = irq;

            /* get our clock resource */
            clk = clk_get(&pdev->dev, NULL);
            if (IS_ERR(clk)) {
...cut...
            } else {
                        clk_rate = clk_get_rate(clk);
                        pr_info("clk = %lu HZ\n", clk_rate);
            }

            g_demo_driver_clk_rate = clk_rate;

            pr_info("platform_probe exit\n");
            return 0;

bad_exit_return:
            pr_info("platform_probe bad_exit\n");
            return ret_val;
}
```

We retrieve the irq that our hardware is allocated by calling platform_get_irq().

We retrieve our clock information by calling clk_get() and then clk_get_rate().

demo_module_04.ko target demonstration (1)

```
insmod demo_module_03.ko                        # insert the first module, notice probe
insmod demo_module_04.ko                        # insert the second module, no probe
rmmod demo_module_03                            # remove the first module
rmmod demo_module_04                            # remove the module
insmod demo_module_04.ko                        # insert the module notice probe printk
rmmod demo_module_04                            # remove the module

ls /proc/device-tree/
ls /proc/device-tree/sopc@0/
ls /proc/device-tree/sopc@0/bridge@0xc0000000/
ls /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/compatible
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/name

# reg = [0:7] bridge address, [8:11] span
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/reg

# ranges = [0:7] bridge address, [8:11] processor address, [12:15] span
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/ranges

# interrupts = [0:3] type, [4:7] number, [8:11] flags
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/interrupts
printf "%d\n" <number>                          # convert the hex interrupt number to decimal
expr <number> + 32  # add 32 to the interrupt number for actual GIC IRQ

# this is a phandle
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/interrupt-parent
```

Observe the output on the printk console.

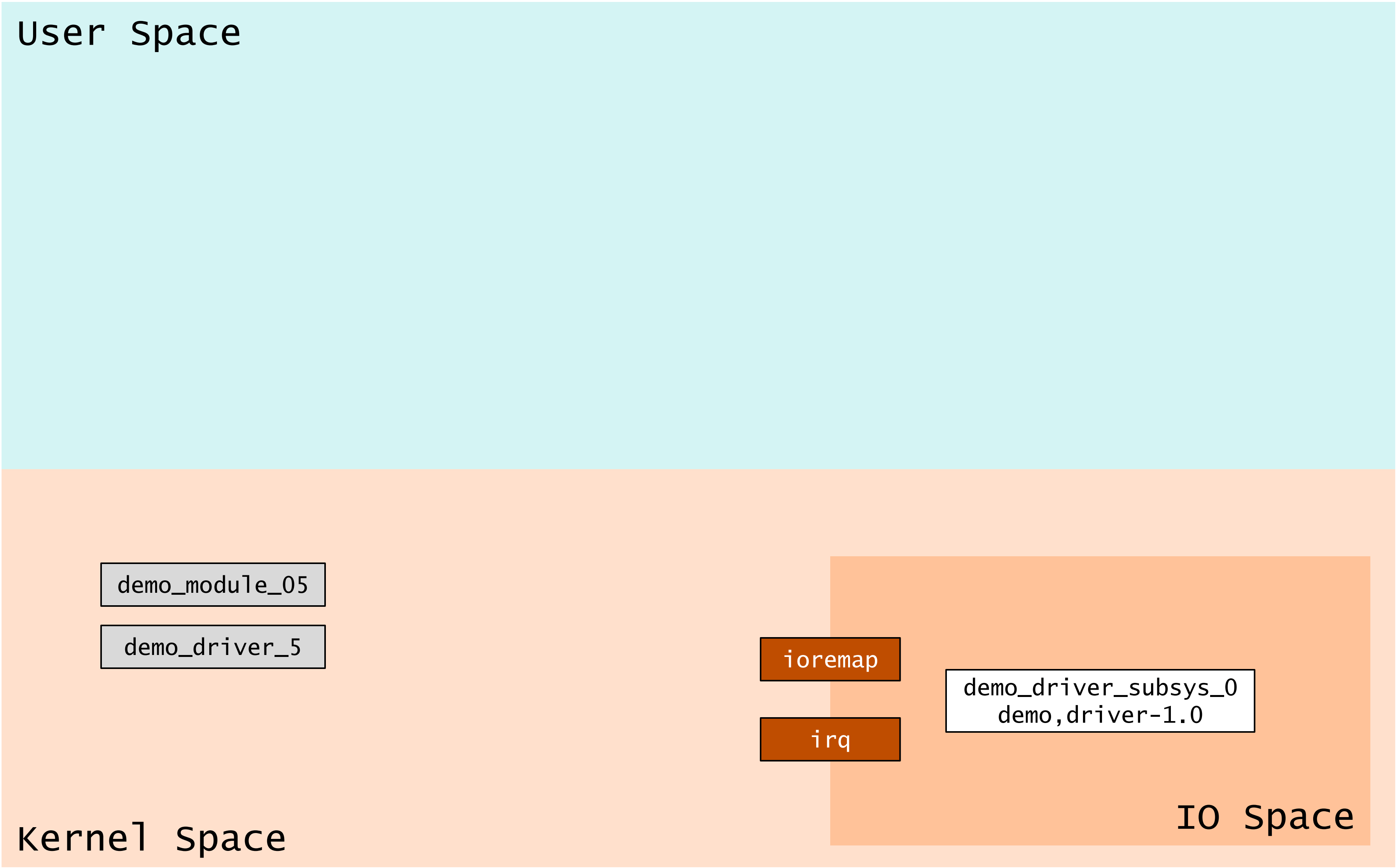
demo_module_04.ko target demonstration (2)

```
# list all the phandles, sorted
find /proc/device-tree -name "linux,phandle" | sort

# list all the phandle values, sorted
# then you can locate the interrupt-parrent phandle
find /proc/device-tree -name "linux,phandle" | sort | xargs hexdump -C

# verify the phandle
ls /proc/device-tree/sopc@0/intc@0xfffed000/
hexdump -C /proc/device-tree/sopc@0/intc@0xfffed000/linux,phandle

# this is a phandle
hexdump -C /proc/device-tree/sopc@0/bridge@0xc0000000/driver@0x100030000/clocks

# verify the phandle
ls /proc/device-tree/clocks/clk_0/
hexdump -C /proc/device-tree/clocks/clk_0/linux,phandle

# verify the clock frequency
hexdump -C /proc/device-tree/clocks/clk_0/clock-frequency
printf "%d\n" <frequency>
```

Observe the output on the printk console.

Reserving memory regions and registering IRQ handlers

demo_module_05



demo_module_05.c – reserve memory, register irq

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
...cut...
          pr_info("platform_probe enter\n");

          ret_val = -EBUSY;

          /* acquire the probe lock */
          if (down_interruptible(&g_dev_probe_sem))
                    return -ERESTARTSYS;

          if (g_platform_probe_flag != 0)
                    goto bad_exit_return;

          ret_val = -EINVAL;

          /* get our first memory resource */
          r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
          if (r == NULL) {
                    pr_err("IORESOURCE_MEM, 0 does not exist\n");
                    goto bad_exit_return;
          }

          g_demo_driver_base_addr = r->start;
          g_demo_driver_size = resource_size(r);
```

We start by retrieving our memory resource.

Note that we acquire a semaphore for the driver resources with the down_interruptible() call.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
/* get our interrupt resource */
irq = platform_get_irq(pdev, 0);
if (irq < 0) {
        pr_err("irq not available\n");
        goto bad_exit_return;
}

g_demo_driver_irq = irq;

/* get our clock resource */
clk = clk_get(&pdev->dev, NULL);
if (IS_ERR(clk)) {
        pr_err("clk not available\n");
        goto bad_exit_return;
} else {
        clk_rate = clk_get_rate(clk);
}
```

Then we retrieve our IRQ number and our clock information.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
        ret_val = -EBUSY;

        /* reserve our memory region */
        demo_driver_mem_region = request_mem_region(g_demo_driver_base_addr,
                                                    g_demo_driver_size,
                                                    "demo_driver_hw_region");
        if (demo_driver_mem_region == NULL) {
                pr_err("request_mem_region failed: g_demo_driver_base_addr\n");
                goto bad_exit_return;
        }

        /* ioremap our memory region */
        g_ioremap_addr = ioremap(g_demo_driver_base_addr, g_demo_driver_size);
        if (g_ioremap_addr == NULL) {
                pr_err("ioremap failed: g_demo_driver_base_addr\n");
                goto bad_exit_release_mem_region;
        }

        g_timer_base = g_ioremap_addr + TIMER_OFST;
```

Now we can reserve our memory region and remap it into an IO pointer that our driver can use.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
/* initialize our peripheral timer hardware */
io_result = ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK |
    ALTERA_AVALON_TIMER_STATUS_RUN_MSK;
if (io_result != 0) {
        pr_err("peripheral timer hardware, incorrect initial state");
        goto bad_exit_iounmap;
}

period_100ms = (clk_rate / 10) - 1;
iowrite32(period_100ms,
        IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
iowrite32(period_100ms >> 16,
        IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));
```

Now that we have an IO pointer we can initialize our timer hardware.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
        /* register our interrupt handler */
        ret_val = request_irq(g_demo_driver_irq,
                                demo_driver_interrupt_handler,
                                0,
                                the_platform_driver.driver.name,
                                &the_platform_driver);

        if (ret_val) {
                pr_err("request_irq failed");
                goto bad_exit_iounmap;
        }

        ret_val = -EBUSY;

        /* start our timer and enable our timer hardware interrupts */
        iowrite32(ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
                  ALTERA_AVALON_TIMER_CONTROL_CONT_MSK |
                  ALTERA_AVALON_TIMER_CONTROL_START_MSK,
                  IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));

        io_result = ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
        io_result &= ALTERA_AVALON_TIMER_STATUS_RUN_MSK;
        if (io_result == 0) {
                /* stop our timer and disable our timer hardware interrupts */
                iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                          IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));

                pr_err("peripheral timer hardware, failed to start");
                goto bad_exit_freeirq;
        }
```

Now we can register our IRQ handler and start our timer hardware to generate interrupts.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
        g_platform_probe_flag = 1;
        up(&g_dev_probe_sem);
        pr_info("platform_probe exit\n");
        return 0;

bad_exit_freeirq:
        free_irq(g_demo_driver_irq, &the_platform_driver);
bad_exit_iounmap:
        iounmap(g_ioremap_addr);
bad_exit_release_mem_region:
        release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);
bad_exit_return:
        up(&g_dev_probe_sem);
        pr_info("platform_probe bad_exit\n");
        return ret_val;
}
```

And we can clean things up and return from our probe function.

Note the bad exit return labels at the bottom of the function.

This construct allows us to unwind any allocations we've made in the opposite order that we created them in.

You can also see that we release the semaphore with up().

demo_module_05.c – reserve memory, register irq

Kernel Space

```
irqreturn_t demo_driver_interrupt_handler(int irq, void *dev_id)
{
        /* clear the interrupt */
        iowrite32(0, IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));

        return IRQ_HANDLED;
}
```

This is what our IRQ handler looks like.

demo_module_05.c – reserve memory, register irq

Kernel Space

```
static int platform_remove(struct platform_device *pdev)
{
        uint32_t io_result;
        pr_info("platform_remove enter\n");

        /* stop our timer and disable our timer hardware interrupts */
        iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                    IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
        /* ensure there is no pending IRQ */
        do {
                io_result =
                    ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
                io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
        } while (io_result != 0);

        free_irq(g_demo_driver_irq, &the_platform_driver);
        iounmap(g_ioremap_addr);
        release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);

        if (down_interruptible(&g_dev_probe_sem))
                return -ERESTARTSYS;

        g_platform_probe_flag = 0;
        up(&g_dev_probe_sem);

        pr_info("platform_remove exit\n");
        return 0;
}
```

Our release function simply releases all the resources that we allocated in probe, and it shuts down our timer hardware.

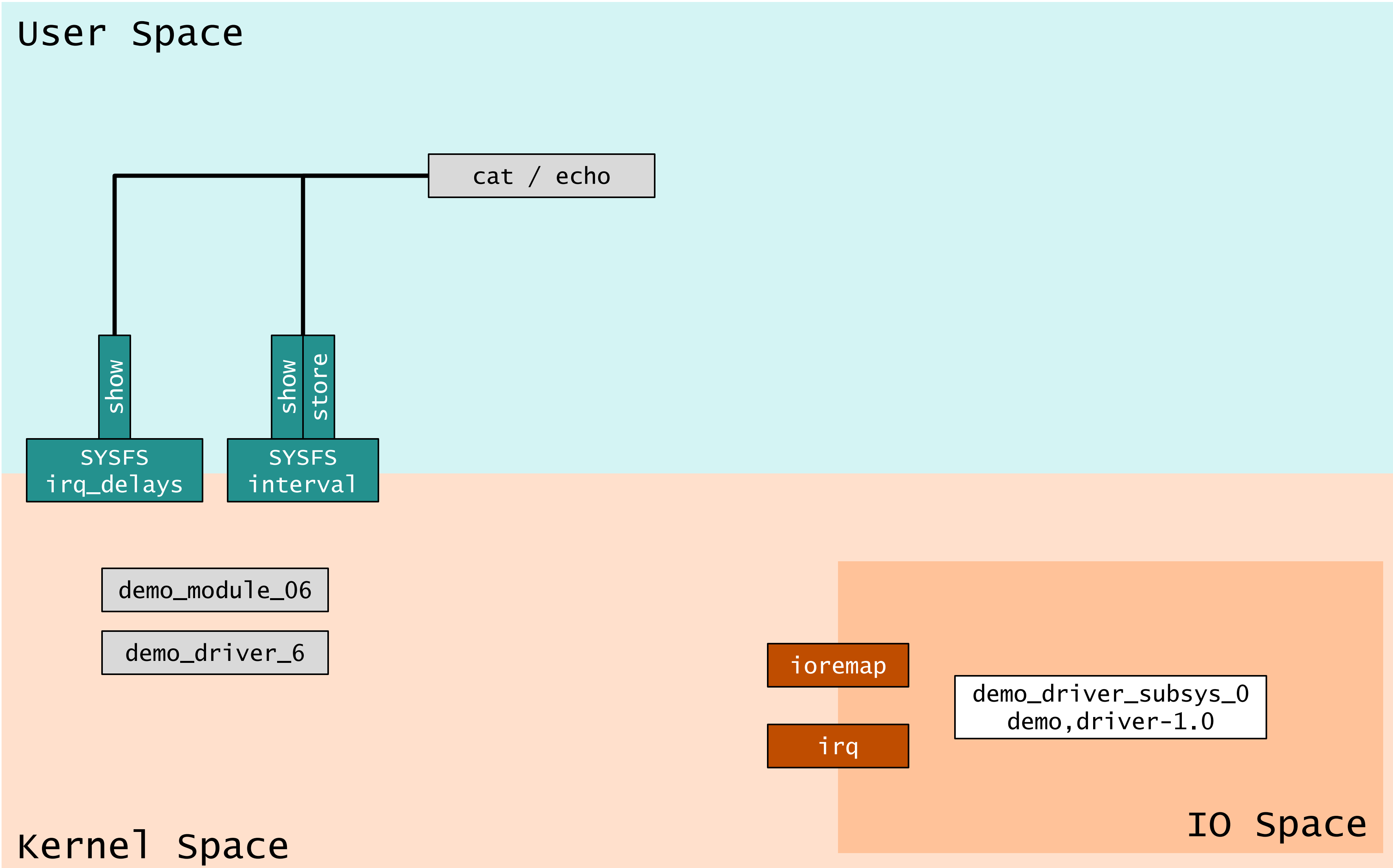
demo_module_05.ko target demonstration

```
./demo_devmem -s                                    # stop the timer hardware
insmod demo_module_05.ko                            # insert module
cat /proc/interrupts | grep "demo"                  # observe IRQ occurring
cat /proc/interrupts | grep "demo"                  # observe IRQ occurring
cat /proc/interrupts | grep "demo"                  # observe IRQ occurring
cat /proc/iomem | grep "demo"                       # observe reserved memory

# demo_module_05t.ko is a test module that we force to reserve the same resources that
# demo_module_05.ko has already reserved.  We can see which of those actions fail and which
# of those actions succeed.
insmod demo_module_05t.ko base=0xff230000 irq=80    # observe printk
                                                    # request_mem_region failed
                                                    # ioremap succeeded
                                                    # request_irq failed
rmmod demo_module_05                                # remove module
```

Observe the output on the printk console.

Creating sysfs files

demo_module_06



demo_module_06.c – sysfs files

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
...cut...
          /* acquire the probe lock */
...cut...
          /* get our first memory resource */
...cut...
          /* get our clock resource */
...cut...
          /* reserve our memory region */
...cut...
          /* ioremap our memory region */
...cut...
          /* initialize our peripheral timer hardware */
...cut...
          /* register our interrupt handler */
...cut...
          /* start our timer and enable our timer hardware interrupts */
...cut...
```

We start our probe function in much the same way as before.

demo_module_06.c – sysfs files

Kernel Space

```
        /* create the sysfs entries */
        ret_val = driver_create_file(&the_platform_driver.driver,
                                     &driver_attr_irq_delays);
        if (ret_val != 0) {
                pr_err("failed to create irq_delays sysfs entry");
                goto bad_exit_stop_timer;
        }

        ret_val = driver_create_file(&the_platform_driver.driver,
                                     &driver_attr_interval);
        if (ret_val != 0) {
                pr_err("failed to create interval sysfs entry");
                goto bad_exit_remove_irq_delays;
        }

        g_platform_probe_flag = 1;
        up(&g_dev_probe_sem);
        pr_info("platform_probe exit\n");
        return 0;
```

Then we create a couple of sysfs files with the driver _create_file() call.

demo_module_06.c – sysfs files

Kernel Space

```
bad_exit_remove_irq_delays:
          driver_remove_file(&the_platform_driver.driver,
                             &driver_attr_irq_delays);
bad_exit_stop_timer:
          iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                    IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
          /* ensure there is no pending IRQ */
          do {
                    io_result =
                        ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
                    io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
          } while (io_result != 0);
bad_exit_freeirq:
          free_irq(g_demo_driver_irq, &the_platform_driver);
bad_exit_iounmap:
          iounmap(g_ioremap_addr);
bad_exit_release_mem_region:
          release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);
bad_exit_return:
          up(&g_dev_probe_sem);
          pr_info("platform_probe bad_exit\n");
          return ret_val;
}
```

If we get an error during probe then we may need to remove the sysfs files we've already created.

demo_module_06.c – sysfs files

Kernel Space

```
static int platform_remove(struct platform_device *pdev)
{
...cut...
          driver_remove_file(&the_platform_driver.driver, &driver_attr_interval);
          driver_remove_file(&the_platform_driver.driver,
                             &driver_attr_irq_delays);

          /* stop our timer and disable our timer hardware interrupts */
          iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                    IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
          /* ensure there is no pending IRQ */
          do {
                    io_result =
                        ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
                    io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
          } while (io_result != 0);

          free_irq(g_demo_driver_irq, &the_platform_driver);
          iounmap(g_ioremap_addr);
          release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);

          if (down_interruptible(&g_dev_probe_sem))
                    return -ERESTARTSYS;

          g_platform_probe_flag = 0;
          up(&g_dev_probe_sem);

          pr_info("platform_remove exit\n");
          return 0;
}
```

Our release function needs to remove the sysfs files in addition to what it already did.

demo_module_06.c – sysfs files

Kernel Space

```
static ssize_t irq_delays_show(struct device_driver *driver, char *buf)
{
            unsigned long flags;
            uint32_t max_irq_delay;
            uint32_t min_irq_delay;

            /* acquire the irq_lock */
            spin_lock_irqsave(&g_irq_lock, flags);

            /* capture the shared data values */
            max_irq_delay = g_max_irq_delay;
            min_irq_delay = g_min_irq_delay;

            /* release the irq_lock */
            spin_unlock_irqrestore(&g_irq_lock, flags);

            if (max_irq_delay == 0)
                        return scnprintf(buf, PAGE_SIZE, "no IRQ delays yet\n");

            return scnprintf(buf, PAGE_SIZE,
                                    "max: 0x%08X = %u\n"
                                    "min: 0x%08X = %u\n",
                                    max_irq_delay, max_irq_delay,
                                    min_irq_delay, min_irq_delay);
}

DRIVER_ATTR(irq_delays, (S_IRUGO), irq_delays_show, NULL);
```

The irq_delays sysfs file that we create registers a show() function, so this file can be read from user space. This file allows us to read the max and min IRQ delay stats, stored by the IRQ handler.

Note the use of spinlocks.

Note the DRIVER_ATTR macro, registering show function only.

demo_module_06.c – sysfs files

Kernel Space

```
static ssize_t interval_show(struct device_driver *driver, char *buf)
{
...cut...
          /* acquire the irq_lock */
          spin_lock_irqsave(&g_irq_lock, flags);

          /* capture the relevant hardware registers */
          raw_status = ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
          raw_periodl =
              ioread32(IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
          raw_periodh =
              ioread32(IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));

          /* release the irq_lock */
          spin_unlock_irqrestore(&g_irq_lock, flags);

          /* calculate the current timer interval */
          raw_status &= ALTERA_AVALON_TIMER_STATUS_RUN_MSK;
          if (raw_status == 0) {
                    interval = 0;
          } else {
                    period = (raw_periodl & 0x0000FFFF) |
                        ((raw_periodh << 16) & 0xFFFF0000);
                    period += 1;
                    interval = g_demo_driver_clk_rate / period;
          }

          return scnprintf(buf, PAGE_SIZE,
                               "irq interval: %u per second\n", interval);
}
```

The interval sysfs file that we create registers a show() function and a store() function, so this file can be read and written from user space. This show function allows us to read the current timer IRQ interval.

Note the use of spinlocks.

demo_module_06.c – sysfs files

Kernel Space

```
static ssize_t interval_store(struct device_driver *driver, const char *buf,
                                    size_t count)
{
...cut...
          /* convert the input string to the requested new interval value */
          result = kstrtoul(buf, 0, &new_interval);
          if (result != 0)
                    return -EINVAL;

          /* range check the requested new interval value */
          if (new_interval > 100)
                    return -EINVAL;

          /* calculate the new period value */
          if (new_interval > 0)
                    new_period = (g_demo_driver_clk_rate / new_interval) - 1;
          else
                    new_period = g_demo_driver_clk_rate;

          /* acquire the irq_lock */
          spin_lock_irqsave(&g_irq_lock, flags);

          /* stop the interval timer */
          iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                      IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
```

This is the interval store() function. It allows us to change the IRQ interval of the timer hardware.

Note the use of spinlocks.

demo_module_06.c – sysfs files

Kernel Space

Note the use of spinlocks.

```
        /* ensure there is no pending IRQ that we are blocking */
        the_status = ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
        the_status &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
        if (the_status != 0) {
                do {
                        /*
                           if we are blocking, release the lock to allow IRQ
                           handler to execute, acquire the lock and check again
                         */
                        spin_unlock_irqrestore(&g_irq_lock, flags);
                        spin_lock_irqsave(&g_irq_lock, flags);
                        the_status =
                            ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS
                                     (g_timer_base));
                        the_status &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
                } while (the_status != 0);
        }

        /* write the new period value */
        iowrite32(new_period, IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
        iowrite32(new_period >> 16,
                  IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));

        /* initialize the MAX/MIN variables */
        g_max_irq_delay = 0;
        g_min_irq_delay = 0xFFFFFFFF;
```

demo_module_06.c – sysfs files

Kernel Space

```
            /* start the timer */
            if (new_interval > 0)
                      iowrite32(ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
                                  ALTERA_AVALON_TIMER_CONTROL_CONT_MSK |
                                  ALTERA_AVALON_TIMER_CONTROL_START_MSK,
                                  IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));

            /* release the irq_lock */
            spin_unlock_irqrestore(&g_irq_lock, flags);
            return count;
}

DRIVER_ATTR(interval, (S_IWUGO | S_IRUGO), interval_show, interval_store);
```

The end of our store() function.

Note the use of spinlocks.

Note the DRIVER_ATTR macro registering a show and store function.

demo_module_06.c – sysfs files

Kernel Space

```
irqreturn_t demo_driver_interrupt_handler(int irq, void *dev_id)
{
...cut...
        spin_lock(&g_irq_lock);

        /* get the current timer value */
        iowrite32(0, IOADDR_ALTERA_AVALON_TIMER_SNAPL(g_timer_base));
        raw_snapl = ioread32(IOADDR_ALTERA_AVALON_TIMER_SNAPL(g_timer_base));
        raw_snaph = ioread32(IOADDR_ALTERA_AVALON_TIMER_SNAPH(g_timer_base));
        snap = (raw_snapl & 0x0000FFFF) | ((raw_snaph << 16) & 0xFFFF0000);

        /* get the current period value */
        raw_periodl = ioread32(IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
        raw_periodh = ioread32(IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));
        period = (raw_periodl & 0x0000FFFF) | ((raw_periodh << 16) & 0xFFFF0000);

        /* calculate response delay and update MAX/MIN variables */
        elapsed_ticks = period - snap;
        if (elapsed_ticks > g_max_irq_delay)
                g_max_irq_delay = elapsed_ticks;
        if (elapsed_ticks < g_min_irq_delay)
                g_min_irq_delay = elapsed_ticks;

        /* clear the interrupt */
        iowrite32(0, IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));

        spin_unlock(&g_irq_lock);
        return IRQ_HANDLED;
}
```

Our IRQ handler gets a little more complicated in this example.

Note the use of spinlocks.

This handler saves off some performance statistics that are read out with the irq_delays file.

demo_module_06.ko target demonstration

```
insmod demo_module_06.ko                             # insert the module
find /sys -name "*demo*"                             # find all sysfs entries
ls /sys/bus/platform/drivers/demo_driver_6           # view the sysfs files

# dump the sysfs files
cat /sys/bus/platform/drivers/demo_driver_6/interval
cat /sys/bus/platform/drivers/demo_driver_6/irq_delays

# run this about 5 times, once per second to verify the 10 IRQ per second speed
cat /proc/interrupts | grep "demo"

# change the IRQ speed to 100 per second
echo "100" > /sys/bus/platform/drivers/demo_driver_6/interval

# run this about 5 times, once per second to verify the 100 IRQ per second speed
cat /proc/interrupts | grep "demo"

# dump the sysfs files
cat /sys/bus/platform/drivers/demo_driver_6/irq_delays
cat /sys/bus/platform/drivers/demo_driver_6/interval

# change the IRQ speed back to 10 per second
echo "10" > /sys/bus/platform/drivers/demo_driver_6/interval

# run this about 5 times, once per second to verify the 10 IRQ per second speed
cat /proc/interrupts | grep "demo"

cat /sys/bus/platform/drivers/demo_driver_6/interval
cat /sys/bus/platform/drivers/demo_driver_6/irq_delays
rmmod demo_module_06
```

Use a calculator to compute the number of CPU clocks the IRQ response takes.

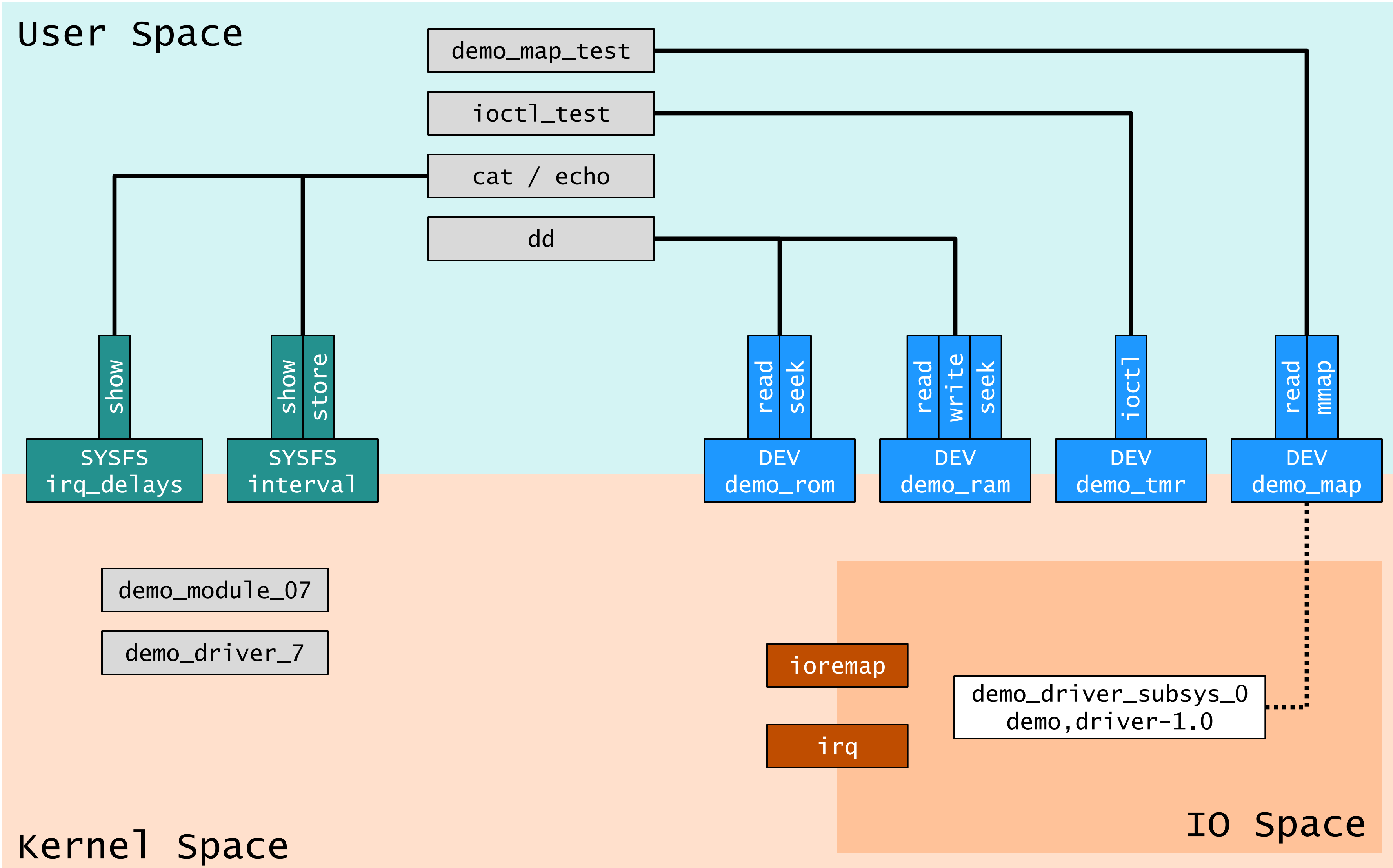
Misc device

For more information:

Documentation/ioctl/ioctl-decoding.txt

Documentation/ioctl/ioctl-number.txt

demo_module_07



demo_module_07.c – misc device

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
...cut...
          /* register misc device dev_rom */
          sema_init(&the_demo_rom_dev.sem, 1);
          ret_val = misc_register(&demo_rom_dev_device);
          if (ret_val != 0) {
...cut...
          }
          /* register misc device dev_ram */
          sema_init(&the_demo_ram_dev.sem, 1);
          ret_val = misc_register(&demo_ram_dev_device);
          if (ret_val != 0) {
...cut...
          }
          /* register misc device dev_tmr */
          sema_init(&the_demo_tmr_dev.sem, 1);
          ret_val = misc_register(&demo_tmr_dev_device);
          if (ret_val != 0) {
...cut...
          }
          /* register misc device dev_map */
          sema_init(&the_demo_map_dev.sem, 1);
          ret_val = misc_register(&demo_map_dev_device);
          if (ret_val != 0) {
...cut...
          }
```

We start our probe function in much the same way as before. We use misc_register() to register 4 different devices with the kernel.

demo_module_07.c – misc device

Kernel Space

```
...cut...
bad_exit_deregister_demo_tmr:
          misc_deregister(&demo_tmr_dev_device);
bad_exit_deregister_demo_ram:
          misc_deregister(&demo_ram_dev_device);
bad_exit_deregister_demo_rom:
          misc_deregister(&demo_rom_dev_device);
...cut...
}

static int platform_remove(struct platform_device *pdev)
{
...cut...
          misc_deregister(&demo_map_dev_device);
          misc_deregister(&demo_tmr_dev_device);
          misc_deregister(&demo_ram_dev_device);
          misc_deregister(&demo_rom_dev_device);
...cut...
}
```

Our probe function ends in much the same way as before. We may have to deregister our misc devices on error conditions.

Our remove function simply removes our misc devices with misc_deregister().

demo_module_07.c – misc device

Kernel Space

```
static const struct file_operations demo_rom_dev_fops = {
            .owner = THIS_MODULE,
            .open = demo_rom_dev_open,
            .release = demo_rom_dev_release,
            .read = demo_rom_dev_read,
            .llseek = demo_rom_dev_llseek,
};

static struct miscdevice demo_rom_dev_device = {
            .minor = MISC_DYNAMIC_MINOR,
            .name = "demo_rom",
            .fops = &demo_rom_dev_fops,
};
```

Our demo_rom device registers an open, release, read and llseek function.

demo_module_07.c – misc device

Kernel Space

```
static int demo_rom_dev_open(struct inode *ip, struct file *fp)
{
          struct demo_rom_dev *dev = &the_demo_rom_dev;
...cut...
          if (down_interruptible(&dev->sem))
                    return -ERESTARTSYS;

          fp->private_data = dev;
          dev->open_count++;

          up(&dev->sem);
          pr_info("demo_rom_dev_open exit\n");
          return 0;
}

static int demo_rom_dev_release(struct inode *ip, struct file *fp)
{
          struct demo_rom_dev *dev = fp->private_data;
...cut...
          if (down_interruptible(&dev->sem))
                    return -ERESTARTSYS;

          dev->release_count++;

          up(&dev->sem);
          pr_info("demo_rom_dev_release exit\n");
          return 0;
}
```

In this example our open and release functions don’t really do much, other than count some statistics for demonstration.

demo_module_07.c – misc device

Kernel Space

```
static ssize_t
demo_rom_dev_read(struct file *fp, char __user *user_buffer,
                          size_t count, loff_t *offset)
{
...cut...
          if (down_interruptible(&dev->sem)) {
                    pr_info("demo_rom_dev_read sem interrupted exit\n");
                    return -ERESTARTSYS;
          }

          dev->read_count++;

          if (*offset > max_offset) {
                    up(&dev->sem);
                    pr_info("demo_rom_dev_read offset > max_offset exit\n");
                    return -EINVAL;
          }

          if (*offset == max_offset) {
                    up(&dev->sem);
                    pr_info("demo_rom_dev_read offset == max_offset exit\n");
                    return 0;
          }

          if (next_offset > max_offset)
                    count -= next_offset - max_offset;

          temp_count = count;
          rom_ptr = g_ioremap_addr + ROM_OFST;
          rom_ptr += *offset;
```

Our demo_rom read routine allows user space to read from our ROM memory in our hardware.

demo_module_07.c – misc device

Kernel Space

```
        while (temp_count > 0) {
                int this_loop_count = IO_BUF_SIZE;
                if (temp_count < IO_BUF_SIZE)
                        this_loop_count = temp_count;

                memcpy_fromio(&dev->io_buf, rom_ptr, this_loop_count);
                if (copy_to_user(user_buffer, &dev->io_buf, this_loop_count)) {
                        up(&dev->sem);
                        pr_info("demo_rom_dev_read copy_to_user exit\n");
                        return -EFAULT;
                }
                temp_count -= this_loop_count;
                user_buffer += this_loop_count;
                rom_ptr += this_loop_count;
        }

        dev->read_byte_count += count;
        *offset += count;

        up(&dev->sem);
        pr_info("demo_rom_dev_read exit\n");
        return count;
}
```

In this example we demonstrate memcpy_fromio() along with copy_to_user(), but in fact on the Cortex A9 it would be safe to simply use copy_to_user() alone. That may not be the case on all CPU architectures however.

demo_module_07.c – misc device

Kernel Space

```
static loff_t demo_rom_dev_llseek(struct file *fp, loff_t offset, int mode)
{
		struct demo_rom_dev *dev = fp->private_data;
		loff_t max_offset = ROM_SPAN;
		loff_t next_offset;
...cut...
		if (down_interruptible(&dev->sem)) {
			pr_info("demo_rom_dev_llseek sem interrupted exit\n");
			return -ERESTARTSYS;
		}

		dev->llseek_count++;

		switch (mode) {
		case SEEK_SET:
			next_offset = offset;
			break;
		case SEEK_CUR:
			next_offset = fp->f_pos + offset;
			break;
		case SEEK_END:
			next_offset = max_offset;
			break;
		default:
			up(&dev->sem);
			pr_info("demo_rom_dev_llseek bad mode exit\n");
			return -EINVAL;
		}
```

Our demo_rom llseek routine allows user space to seek within our ROM space.

demo_module_07.c – misc device

Kernel
Space

```
        if (next_offset < 0) {
                up(&dev->sem);
                pr_info("demo_rom_dev_llseek negative offset exit\n");
                return -EINVAL;
        }

        if (next_offset > max_offset)
                next_offset = max_offset;

        fp->f_pos = next_offset;

        up(&dev->sem);
        pr_info("demo_rom_dev_llseek exit\n");
        return next_offset;
}
```

demo_module_07.c – misc device

Kernel Space

```
static const struct file_operations demo_ram_dev_fops = {
		.owner = THIS_MODULE,
		.open = demo_ram_dev_open,
		.release = demo_ram_dev_release,
		.read = demo_ram_dev_read,
		.write = demo_ram_dev_write,
		.llseek = demo_ram_dev_llseek,
};

static struct miscdevice demo_ram_dev_device = {
		.minor = MISC_DYNAMIC_MINOR,
		.name = "demo_ram",
		.fops = &demo_ram_dev_fops,
};
```

Our demo_ram device registers a write function in addition to the functions that our demo_rom device registered. This allows user space to write and read our RAM hardware.

demo_module_07.c – misc device

Kernel Space

```
static ssize_t
demo_ram_dev_write(struct file *fp,
                        const char __user *user_buffer, size_t count,
                        loff_t *offset)
{
          struct demo_ram_dev *dev = fp->private_data;
          loff_t max_offset = RAM_SPAN;
          loff_t next_offset = *offset + count;
          size_t temp_count;
          void *ram_ptr;
...cut...
          if (down_interruptible(&dev->sem)) {
                    pr_info("demo_ram_dev_write sem interrupted exit\n");
                    return -ERESTARTSYS;
          }

          dev->write_count++;

          if (*offset > max_offset) {
                    up(&dev->sem);
                    pr_info("demo_ram_dev_write offset > max_offset exit\n");
                    return -EINVAL;
          }

          if (*offset == max_offset) {
                    up(&dev->sem);
                    pr_info("demo_ram_dev_write offset == max_offset exit\n");
                    return -ENOSPC;
          }

```

Our demo_ram write function is similar to the read function, just implementing the opposite data flow.

demo_module_07.c – misc device

Kernel Space

```
	if (next_offset > max_offset)
		count -= next_offset - max_offset;

	temp_count = count;
	ram_ptr = g_ioremap_addr + RAM_OFST;
	ram_ptr += *offset;

	while (temp_count > 0) {
		int this_loop_count = IO_BUF_SIZE;
		if (temp_count < IO_BUF_SIZE)
			this_loop_count = temp_count;

		if (copy_from_user
		    (&dev->io_buf, user_buffer, this_loop_count)) {
			up(&dev->sem);
			pr_info("demo_ram_dev_write copy_from_user exit\n");
			return -EFAULT;
		}
		memcpy_toio(ram_ptr, &dev->io_buf, this_loop_count);
		temp_count -= this_loop_count;
		user_buffer += this_loop_count;
		ram_ptr += this_loop_count;
	}

	dev->write_byte_count += count;
	*offset += count;

	up(&dev->sem);
	pr_info("demo_ram_dev_write exit\n");
	return count;
```

In our write we copy_from_user() and memcpy_toio().

demo_module_07.c – misc device

Kernel Space

```
static const struct file_operations demo_tmr_dev_fops = {
            .owner = THIS_MODULE,
            .open = demo_tmr_dev_open,
            .release = demo_tmr_dev_release,
            .unlocked_ioctl = demo_tmr_dev_ioctl,
};

static struct miscdevice demo_tmr_dev_device = {
            .minor = MISC_DYNAMIC_MINOR,
            .name = "demo_tmr",
            .fops = &demo_tmr_dev_fops,
};
```

Our demo_tmr device registers an open, release and ioctl function. This allows user space to interact with our timer hardware.

demo_module_07.c – misc device

Kernel Space

```
static long
demo_tmr_dev_ioctl(struct file *fp, unsigned int cmd, unsigned long arg)
{
...cut...
          if (down_interruptible(&dev->sem)) {
                    pr_info("demo_tmr_dev_ioctl sem interrupted exit\n");
                    return -ERESTARTSYS;
          }
          dev->ioctl_count++;

          switch (cmd) {
          case IOC_SET_INTERVAL:
...cut...
          case IOC_GET_INTERVAL:
...cut...
          case IOC_GET_MAX_DELAY:
...cut...
          case IOC_GET_MIN_DELAY:
...cut...
          default:
                    up(&dev->sem);
                    pr_info("demo_tmr_dev_ioctl bad cmd exit\n");
                    return -EINVAL;
          }

          up(&dev->sem);
          pr_info("demo_tmr_dev_ioctl exit\n");
          return 0;
}
```

Our ioctl boils down to a switch statement.

demo_module_07.c – misc device

Kernel Space

```
case IOC_SET_INTERVAL:
        if (get_user(new_interval, (uint32_t *)arg) < 0) {
                up(&dev->sem);
                pr_info("demo_tmr_dev_ioctl get_user exit\n");
                return -EFAULT;
        }
        /* range check the requested new interval value */
        if (new_interval > 100) {
                up(&dev->sem);
                pr_info("demo_tmr_dev_ioctl new_interval > 100 exit\n");
                return -EINVAL;
        }
        /* calculate the new period value */
        if (new_interval > 0)
                new_period =
                    (g_demo_driver_clk_rate / new_interval) - 1;
        else
                new_period = g_demo_driver_clk_rate;

        /* acquire the irq_lock */
        spin_lock_irqsave(&g_irq_lock, flags);

        /* stop the interval timer */
        iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                  IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
```

Details of switch case SET_INTERVAL. Uses get_user() instead of copy_from_user().

demo_module_07.c – misc device

Kernel Space

manage potential race for spin lock with irq handler

```
/* ensure there is no pending IRQ that we are blocking */
the_status =
    ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
the_status &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
if (the_status != 0) {
          do {
                    /*
                       if we are blocking, release the lock to allow
                       IRQ handler to execute, acquire the lock and
                       check again
                     */
                    spin_unlock_irqrestore(&g_irq_lock, flags);
                    spin_lock_irqsave(&g_irq_lock, flags);
                    the_status =
                        ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS
                                   (g_timer_base));
                    the_status &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
          } while (the_status != 0);
}
/* write the new period value */
iowrite32(new_period,
            IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
iowrite32(new_period >> 16,
            IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));

/* initialize the MAX/MIN variables */
g_max_irq_delay = 0;
g_min_irq_delay = 0xFFFFFFFF;
```

Kernel Space

demo_module_07.c – misc device

```
                /* start the timer */
                if (new_interval > 0)
                        iowrite32(ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
                                    ALTERA_AVALON_TIMER_CONTROL_CONT_MSK |
                                    ALTERA_AVALON_TIMER_CONTROL_START_MSK,
                                    IOADDR_ALTERA_AVALON_TIMER_CONTROL
                                    (g_timer_base));

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);
                break;

        case IOC_GET_MAX_DELAY:
                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                /* capture the shared data values */
                max_irq_delay = g_max_irq_delay;

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                if (put_user(max_irq_delay, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("demo_tmr_dev_ioctl put_user exit\n");
                        return -EFAULT;
                }
                break;
```

Details of switch case GET_MAX_DELAY. Uses put_user() instead of copy_to_user().

demo_module_07.c – misc device

Kernel Space

```
static const struct file_operations demo_map_dev_fops = {
            .owner = THIS_MODULE,
            .open = demo_map_dev_open,
            .release = demo_map_dev_release,
            .read = demo_map_dev_read,
            .mmap = demo_map_dev_mmap,
};

static struct miscdevice demo_map_dev_device = {
            .minor = MISC_DYNAMIC_MINOR,
            .name = "demo_map",
            .fops = &demo_map_dev_fops,
};
```

Our demo_map device registers functions for open, release, read and mmap. This allows user space to mmap() our device, similar to what we did in the /dev/mem example. The read function allows us to emulate IRQ polling in much the same way as a UIO device does, we'll demonstrate a UIO device later.

demo_module_07.c – misc device

Kernel Space

```
static ssize_t
demo_map_dev_read(struct file *fp, char __user *user_buffer,
                      size_t count, loff_t *offset)
{
          struct demo_map_dev *dev = fp->private_data;
          uint32_t cur_irq_cnt;
...cut...
          if (down_interruptible(&dev->sem)) {
                    pr_info("demo_map_dev_read sem interrupted exit\n");
                    return -ERESTARTSYS;
          }

          dev->read_count++;

          if (*offset != 0) {
                    up(&dev->sem);
                    pr_info("demo_map_dev_read offset != 0 exit\n");
                    return -EINVAL;
          }

          if (count != 4) {
                    up(&dev->sem);
                    pr_info("demo_map_dev_read count != 4 exit\n");
                    return -EINVAL;
          }
```

The demo_map read function starts by validating the input arguments for the exact count requirements that we expect to fulfill, anything else is considered an error.

demo_module_07.c – misc device

Kernel Space

```
        cur_irq_cnt = get_current_irq_count();
        while (cur_irq_cnt == get_current_irq_count()) {
                up(&dev->sem);
                if (wait_event_interruptible(g_irq_wait_queue,
                                                        (cur_irq_cnt !=
                                                        get_current_irq_count()))) {
                        pr_info("demo_map_dev_read wait interrupted exit\n");
                        return -ERESTARTSYS;
                }
                if (down_interruptible(&dev->sem)) {
                        pr_info("demo_map_dev_read sem interrupted exit\n");
                        return -ERESTARTSYS;
                }
        }

        cur_irq_cnt = get_current_irq_count();
        if (copy_to_user(user_buffer, &cur_irq_cnt, count)) {
                up(&dev->sem);
                pr_info("demo_map_dev_read copy_to_user exit\n");
                return -EFAULT;
        }

        dev->read_byte_count += count;

        up(&dev->sem);
        pr_info("demo_map_dev_read exit\n");
        return count;
}
```

The heart of the demo_map read function is waiting for the next IRQ event to occur.

demo_module_07.c – misc device

Kernel Space

```
static inline uint32_t get_current_irq_count(void)
{
          uint32_t current_count;
          unsigned long flags;

          spin_lock_irqsave(&g_irq_lock, flags);
          current_count = g_irq_count;
          spin_unlock_irqrestore(&g_irq_lock, flags);
          return current_count;
}

irqreturn_t demo_driver_interrupt_handler(int irq, void *dev_id)
{
...cut...
          spin_lock(&g_irq_lock);
...cut...
          spin_unlock(&g_irq_lock);
          wake_up_interruptible(&g_irq_wait_queue);
          return IRQ_HANDLED;
}
```

The read function leveraged this helper function to query the IRQ count in a safe fashion to avoid race conditions.

The IRQ handler for this example is similar to before with the addition of the wake up event signal, which is what the read function will ultimately pend on.

demo_module_07.c – misc device

Kernel Space

```
static int demo_map_dev_mmap(struct file *fp, struct vm_area_struct *vma)
{
        struct demo_map_dev *dev = fp->private_data;
...cut...
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        dev->mmap_count++;

        if (vma->vm_end - vma->vm_start != PAGE_SIZE) {
                up(&dev->sem);
                return -EINVAL;
        }

        vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

        vma->vm_pgoff = g_demo_driver_base_addr >> PAGE_SHIFT;

        if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, PAGE_SIZE,
                                   vma->vm_page_prot)) {
                up(&dev->sem);
                return -EAGAIN;
        }

        up(&dev->sem);
        pr_info("demo_map_dev_mmap exit\n");
        return 0;
}
```

The demo_map mmap function maps our IO region into a user space safe pointer as noncached memory.

demo_map_test.c

User Space

```
int main(int argc, char **argv) {
...cut...
		//
		// parse the command line arguments
		//
		parse_cmdline(argc, argv);

		//
		// open() the /dev/demo_map device
		//
		dev_demo_map_fd = open("/dev/demo_map", O_RDWR | O_SYNC);
		if(dev_demo_map_fd < 0) {
			perror("dev_demo_map open");
			exit(EXIT_FAILURE);
		}

		//
		// mmap() the base of our demo_driver hardware
		//
		demo_driver_map = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ|PROT_WRITE,
					MAP_SHARED, dev_demo_map_fd, 0);
		if(demo_driver_map == MAP_FAILED) {
			perror("dev_demo_map mmap");
			close(dev_demo_map_fd);
			exit(EXIT_FAILURE);
		}
```

We provide a test application to drive the mmap functionality of our demo_map driver. This looks basically the same as our /dev/mem example, but this example uses /dev/demo_map instead.

ioctl_test.c

User Space

```
int main(int argc, char **argv) {

            int dev_demo_tmr_fd;

            //
            // parse the command line arguments
            //
            parse_cmdline(argc, argv);

            //
            // open() the /dev/demo_tmr device
            //
            dev_demo_tmr_fd = open("/dev/demo_tmr", O_RDWR | O_SYNC);
            if(dev_demo_tmr_fd < 0) {
                        perror("dev_demo_tmr open");
                        exit(EXIT_FAILURE);
            }

            //
            // perform the operation selected by the command line arguments
            //
            if(g_get_interval   != NULL) do_get_interval(dev_demo_tmr_fd);
            if(g_set_interval   != NULL) do_set_interval(dev_demo_tmr_fd);
            if(g_get_max_delay  != NULL) do_get_max_delay(dev_demo_tmr_fd);
            if(g_get_min_delay  != NULL) do_get_min_delay(dev_demo_tmr_fd);
            if(g_help           != NULL) do_help();

            close(dev_demo_tmr_fd);
            exit(EXIT_SUCCESS);
}
```

We provide a test application to drive the ioctl functionality of our demo_tmr driver.

ioctl_test.c

User Space

Here are the get and set interval functions.

```
void do_get_interval(int dev_demo_tmr_fd) {

		int result;
		unsigned long interval;

		result = ioctl(dev_demo_tmr_fd, IOC_GET_INTERVAL, &interval);
		if(result != 0) {
				error(1, errno, "%s:%d ioctl failed", __FILE__, __LINE__);
		}

		printf("Current IRQ interval is %lu interrupts per second.\n", interval);
}

void do_set_interval(int dev_demo_tmr_fd) {

		int result;

		result = ioctl(dev_demo_tmr_fd, IOC_SET_INTERVAL, &g_new_interval);
		if(result != 0) {
				error(1, errno, "%s:%d ioctl failed", __FILE__, __LINE__);
		}

		printf("IRQ interval set to %lu interrupts per second.\n", g_new_interval);
}
```

ioctl_test.c

User Space

Here are the get max and min delay functions.

```
void do_get_max_delay(int dev_demo_tmr_fd) {

		int result;
		unsigned long delay;

		result = ioctl(dev_demo_tmr_fd, IOC_GET_MAX_DELAY, &delay);
		if(result != 0) {
			error(1, errno, "%s:%d ioctl failed", __FILE__, __LINE__);
		}

		printf("Maximum IRQ service delay is %lu ticks.\n", delay);
}

void do_get_min_delay(int dev_demo_tmr_fd) {

		int result;
		unsigned long delay;

		result = ioctl(dev_demo_tmr_fd, IOC_GET_MIN_DELAY, &delay);
		if(result != 0) {
			error(1, errno, "%s:%d ioctl failed", __FILE__, __LINE__);
		}

		printf("Minimum IRQ service delay is %lu ticks.\n", delay);
}
```

demo_module_07.ko target demonstration

```
insmod demo_module_07.ko                                   # insert module
ls /dev | grep "demo"                                      # observe devfs entries
dd if=/dev/demo_rom | hexdump -C                           # read from rom
dd if=/dev/zero of=/dev/demo_rom                           # write to rom
dd if=/dev/demo_ram | hexdump -C                           # read from ram
dd if=/dev/zero of=/dev/demo_ram                           # write to ram
dd if=/dev/demo_ram | hexdump -C                           # verify write
dd if=/dev/demo_ram bs=64 count=1 | hexdump -Cv            # dump first 64 ram bytes
dd if=/dev/demo_rom bs=64 count=1 | hexdump -C             # dump first 64 rom bytes

# demonstrate llseek by copying 4 bytes from offset 56 in rom to offset 16 in ram
dd if=/dev/demo_rom of=/dev/demo_ram bs=1 count=4 skip=56 seek=16
dd if=/dev/demo_ram bs=64 count=1 | hexdump -Cv            # verify the operation
./demo_map_test -h                                         # run demo_map_test
./demo_map_test -o | hexdump -C                            # dump rom
./demo_map_test -a | hexdump -C                            # dump ram
dd if=/dev/urandom | ./demo_map_test -f                    # fill ram
./demo_map_test -a | hexdump -C                            # verify ram
./demo_map_test -t                                         # dump the timer registers
for I in 1 2 3 4 ; do                                      # demonstrate demo_map read
dd if=/dev/demo_map bs=4 count=1 | hexdump -C              # the result is the irq count
done                                                       # returned at the next IRQ event
cat /sys/bus/platform/drivers/demo_driver_7/interval       # demonstrate ioctl
./ioctl_test -g
echo "100" > /sys/bus/platform/drivers/demo_driver_7/interval
./ioctl_test -g
cat /sys/bus/platform/drivers/demo_driver_7/irq_delays
./ioctl_test -x
./ioctl_test -n
rmmod demo_module_07
```

Observe the output on the printk console.

UIO device

User Space I/O

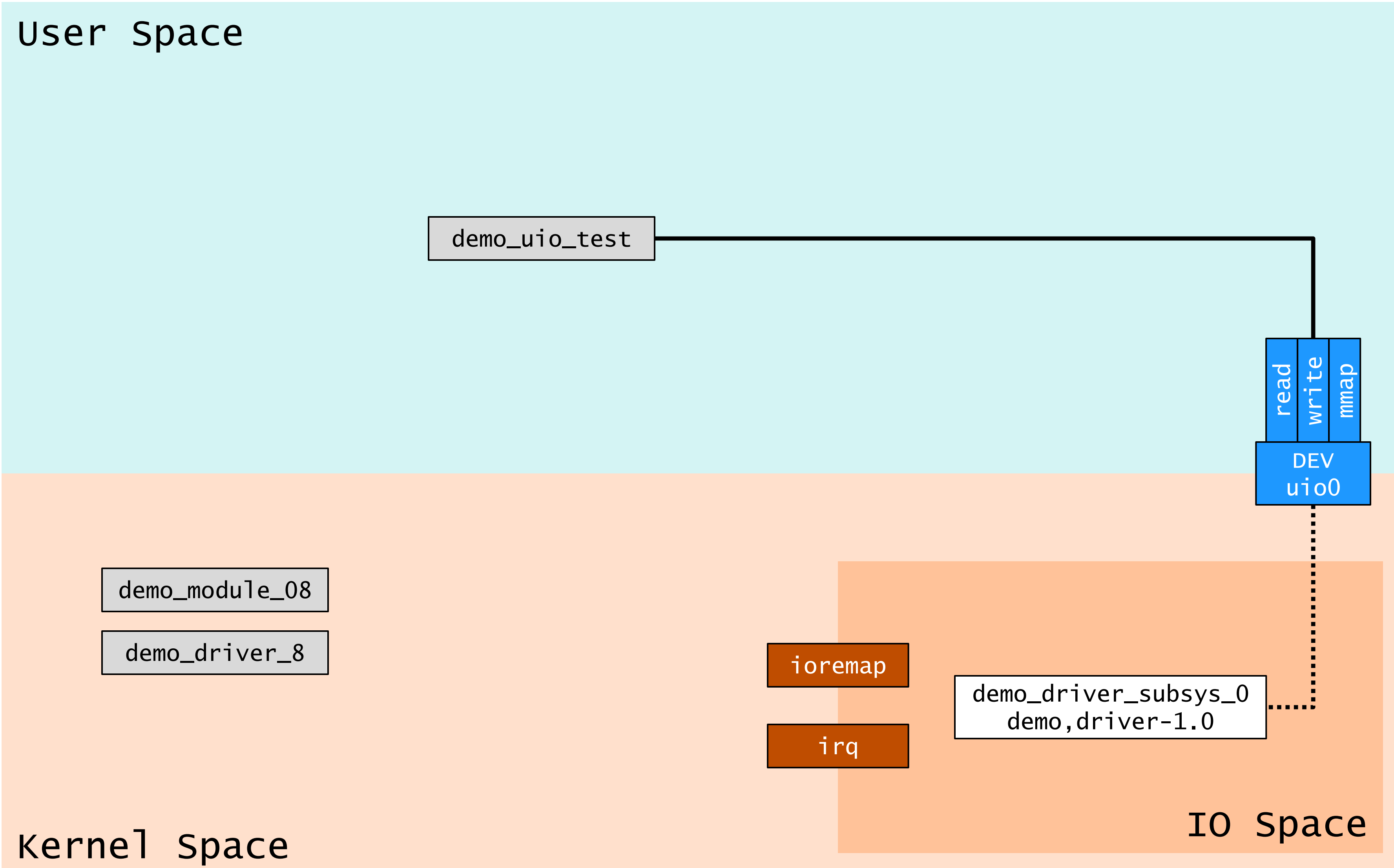
UIO templates in the kernel source tree

```
[socfpga-3.10-ltsi]$ find drivers/uio -name "*.c"
drivers/uio/uio.c
drivers/uio/uio_netx.c
drivers/uio/uio_sercos3.c
drivers/uio/uio_cif.c
drivers/uio/uio_pdrv.c
drivers/uio/uio_dmem_genirq.c
drivers/uio/uio_pruss.c
drivers/uio/uio_pdrv_genirq.c
drivers/uio/uio_aec.c
drivers/uio/uio_pci_generic.c
```

It's worth noting that there are some useful templates in the kernel source tree for simple platform driver implementations driven by a device tree entry, these can be easily modified to provide basic functionality by simply adding a compatibility string to them. The fundamental concept of a UIO driver is that it simply exposes the raw hardware registers to user space and does nothing within kernel space to really control any of the hardware details. Our UIO example does not use the UIO framework in that basic way, instead it builds on the framework that we started with in the previous demonstration modules such that our kernel space code does interact more directly with the hardware in certain situations. We just plug that functionality into the framework that is defined by UIO.

An example of the basic UIO template usage is demonstrated in the lab work that accompanies the WS3 session.

demo_module_08



demo_module_08.c – uio device

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
...cut...
          /* acquire the probe lock */
...cut...
          /* get our first memory resource */
...cut...
          /* get our interrupt resource */
...cut...
          /* get our clock resource */
...cut...
          /* reserve our memory region */
...cut...
          /* ioremap our memory region */
...cut...
          /* initialize uio_info struct uio_mem array */
          the_uio_info.mem[0].memtype = UIO_MEM_PHYS;
          the_uio_info.mem[0].addr = r->start;
          the_uio_info.mem[0].size = resource_size(r);
          the_uio_info.mem[0].name = "demo_uio_driver_hw_region";
          the_uio_info.mem[0].internal_addr = g_ioremap_addr;

          for (i = 1; i < MAX_UIO_MAPS; i++)
                    the_uio_info.mem[i].size = 0;

          /* initialize uio_info irq */
          the_uio_info.irq = g_demo_driver_irq;
```

We start our probe function in much the same way as before. We reserve our memory region and then configure the uio_info struct.

A basic UIO driver would not be concerned about reserving the memory region or ioremap'ing the IO space since the kernel space UIO driver would not be interacting with the hardware directly.

Kernel Space

demo_module_08.c – uio device

```
          /* register the uio device */
          sema_init(&g_demo_uio_dev_sem, 1);
          ret_val = uio_register_device(&pdev->dev, &the_uio_info);
          if (ret_val != 0) {
                    pr_warn("Could not register device \"demo_uio\"...");
                    goto bad_exit_iounmap;
          }

          g_platform_probe_flag = 1;
          up(&g_dev_probe_sem);
          pr_info("platform_probe exit\n");
          return 0;

bad_exit_iounmap:
          iounmap(g_ioremap_addr);
bad_exit_release_mem_region:
          release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);
bad_exit_return:
          up(&g_dev_probe_sem);
          pr_info("platform_probe bad_exit\n");
          return ret_val;
}
```

Then we register our uio device.

demo_module_08.c – uio device

Kernel Space

```
static int platform_remove(struct platform_device *pdev)
{
		pr_info("platform_remove enter\n");

		uio_unregister_device(&the_uio_info);

		iounmap(g_ioremap_addr);
		release_mem_region(g_demo_driver_base_addr, g_demo_driver_size);

		if (down_interruptible(&g_dev_probe_sem))
			return -ERESTARTSYS;

		g_platform_probe_flag = 0;
		up(&g_dev_probe_sem);

		pr_info("platform_remove exit\n");
		return 0;
}
```

Our remove function unregisters the uio_device.

demo_module_08.c – uio device

Kernel Space

```
static struct uio_info the_uio_info = {
          .name = "demo_uio",
          .version = "1.0",
          .irq_flags = 0,
          .handler = demo_uio_interrupt_handler,
          .open = demo_uio_open,
          .release = demo_uio_release,
          .irqcontrol = demo_uio_irqcontrol,
};
```

Our uio_info struct.

demo_module_08.c – uio device

Kernel Space

Our open routine.

```
static int demo_uio_open(struct uio_info *info, struct inode *inode)
{
...cut...
        if (down_trylock(&g_demo_uio_dev_sem) != 0)
                return -EAGAIN;

        /* initialize our peripheral timer hardware */
        io_result = ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
        io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK |
            ALTERA_AVALON_TIMER_STATUS_RUN_MSK;
        if (io_result != 0) {
                pr_err("peripheral timer hardware, incorrect initial state");
                return -EIO;
        }

        period_1s = (g_demo_driver_clk_rate) - 1;
        iowrite32(period_1s, IOADDR_ALTERA_AVALON_TIMER_PERIODL(g_timer_base));
        iowrite32(period_1s >> 16,
                  IOADDR_ALTERA_AVALON_TIMER_PERIODH(g_timer_base));

        /* start our timer and enable our timer hardware interrupts */
        iowrite32(ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
                  ALTERA_AVALON_TIMER_CONTROL_CONT_MSK |
                  ALTERA_AVALON_TIMER_CONTROL_START_MSK,
                  IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));

        return 0;
}
```

A basic UIO driver would rely on user space code to perform this once the device was mmap()'ed.

demo_module_08.c – uio device

Kernel Space

Our release routine.

```
static int demo_uio_release(struct uio_info *info, struct inode *inode)
{
          uint32_t io_result;

          /* stop our timer and disable our timer hardware interrupts */
          iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
                      IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
          /* ensure there is no pending IRQ */
          do {
                    io_result =
                        ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));
                    io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
          } while (io_result != 0);

          up(&g_demo_uio_dev_sem);
          return 0;
}
```

A basic UIO driver would rely on user space code to perform this prior the device being munmap()'ed.

demo_module_08.c – uio device

Kernel Space

```
irqreturn_t demo_uio_interrupt_handler(int irq, struct uio_info *dev_info)
{
        /* snapshot the current timer value */
        iowrite32(0, IOADDR_ALTERA_AVALON_TIMER_SNAPL(g_timer_base));

        /* clear the interrupt */
        iowrite32(0, IOADDR_ALTERA_AVALON_TIMER_STATUS(g_timer_base));

        return IRQ_HANDLED;
}
```

Our irq handler.

A basic UIO driver would rely on user space code to perform this following a read() from the driver.

demo_module_08.c – uio device

Kernel Space

```
static int demo_uio_irqcontrol(struct uio_info *info, s32 irq_on)
{
	uint32_t io_result;

	if (irq_on) {
		/* start our timer and enable our timer hardware interrupts */
		iowrite32(ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
			ALTERA_AVALON_TIMER_CONTROL_CONT_MSK |
			ALTERA_AVALON_TIMER_CONTROL_START_MSK,
			IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
	} else {
		/* stop our timer and disable our timer hardware interrupts */
		iowrite32(ALTERA_AVALON_TIMER_CONTROL_STOP_MSK,
			IOADDR_ALTERA_AVALON_TIMER_CONTROL(g_timer_base));
		/* ensure there is no pending IRQ */
		do {
			io_result =
			    ioread32(IOADDR_ALTERA_AVALON_TIMER_STATUS
					(g_timer_base));
			io_result &= ALTERA_AVALON_TIMER_STATUS_TO_MSK;
		} while (io_result != 0);
	}
	return 0;
}
```

Our irqcontrol routine.

A basic UIO driver would rely on user space code to perform this coordinated with a write to the driver.

demo_module_08.ko target demonstration (1)

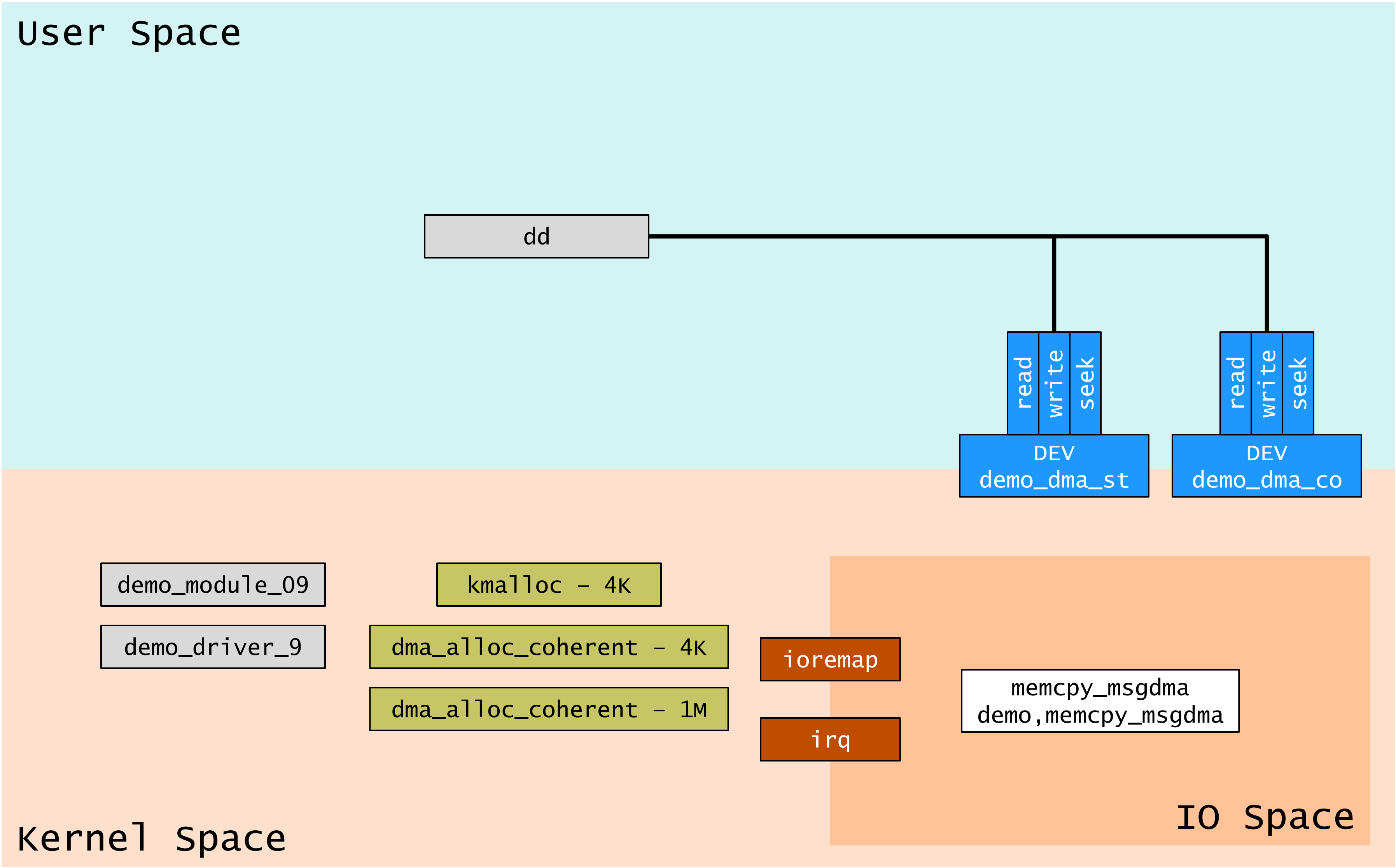
```
insmod demo_module_08.ko                             # insert module
find /dev -name "uio*"                               # find the device node
find /sys/ -name "uio*"                              # find sysfs entries
ls /sys/class/uio/uio0                               # display uio contents
ls /sys/class/uio/uio0/maps/                         # display maps entry
ls /sys/class/uio/uio0/maps/map0/                    # display map0 entry
cat /sys/class/uio/uio0/maps/map0/*                  # dump the map0 entries

# run this block as one copy/paste operation
exec 7<>/dev/uio0                                    # open /dev/uio0 for rd/wr
for I in 1 2 3 4 5
do
dd bs=4 count=1 0<&7 | hexdump -Cv                             # read an IRQ event
done
dd if=/dev/zero bs=4 count=1 >&7                               # disable IRQs
dd bs=4 count=1 0<&7 | hexdump -Cv                             # read an IRQ event

# the last command of the previous block should stall forever, ^-C and run this
# block next
dd if=/dev/zero bs=4 count=1 | tr '\000' '\377' >&7            # enable IRQs
for I in 1 2 3 4 5
do
dd bs=4 count=1 0<&7 | hexdump -Cv                   # read an IRQ event
done

# finally run this command
exec 7<>/dev/null                                    # close /dev/uio0
```

demo_module_08.ko target demonstration (2)

```
./demo_uio_test -h                                  # run demo_uio_test
./demo_uio_test -t                                  # exercise timer regs
./demo_uio_test -o | hexdump -C                     # dump rom
./demo_uio_test -a | hexdump -C                     # dump ram
dd if=/dev/urandom | ./demo_uio_test -f             # fill ram
./demo_uio_test -a | hexdump -C                     # dump ram to verify
ls /sys/class/uio/uio0/                             # observe the event entry
cat /sys/class/uio/uio0/event                       # dump the event entry
dd if=/dev/uio0 bs=4 count=1 | hexdump -Cv          # read another IRQ
printf "%d\n" <hex number>                          # convert the hex number to dec
cat /sys/class/uio/uio0/event                       # verify event count matches
rmmod demo_module_08                                # remove module
```

DMA

demo_module_09



demo_module_09.c – DMA

Kernel Space

```
static struct of_device_id demo_driver_dt_ids[] = {
          {
           .compatible = "demo,memcpy_msgdma"},
          { /* end of table */ }
};

static int platform_probe(struct platform_device *pdev)
{
...cut...
          /* acquire the probe lock */
...cut...
          /* get our csr memory resource */
...cut...
          /* reserve our csr memory region */
...cut...
          /* ioremap our csr memory region */
...cut...
          /* initialize the DMA controller */
...cut...
          /* get our desc memory resource */
...cut...
          /* ioremap our desc memory region */
...cut...
```

For this example we bind with the memcpy_msgdma device.

Our probe function starts off pretty much the same.

demo_module_09.c – DMA

Kernel Space

```
/* allocate some memory buffers */
g_kmalloc_ptr_4k = kmalloc(PAGE_SIZE, GFP_KERNEL);
if (g_kmalloc_ptr_4k == NULL) {
        pr_err("kmalloc failed\n");
        goto bad_exit_iounmap_desc;
}

the_demo_dma_xx.pdev_dev = &pdev->dev;
g_coherent_ptr_4k =
    dma_alloc_coherent(the_demo_dma_xx.pdev_dev, PAGE_SIZE,
                       &g_dma_handle_4k, GFP_KERNEL);
if (g_coherent_ptr_4k == NULL) {
        pr_err("dma_alloc_coherent failed 4KB\n");
        goto bad_exit_kfree;
}

g_coherent_ptr_1m = dma_alloc_coherent(the_demo_dma_xx.pdev_dev,
                                       DMA_DEVICE_BUFFER_SIZE,
                                       &g_dma_handle_1m, GFP_KERNEL);
if (g_coherent_ptr_1m == NULL) {
        pr_err("dma_alloc_coherent failed 1MB\n");
        goto bad_exit_dma_free_coherent_4k;
}
```

We kmalloc a buffer for our streaming driver and we allocate a coherent buffer for our coherent driver.

coherent = non-cacheable.

Then we allocate a coherent buffer for our 1M file emulation buffer.

demo_module_09.c – DMA

Kernel Space

```
        /* register our interrupt handler */
        init_waitqueue_head(&g_irq_wait_queue);
...cut...
        /* enable the DMA global IRQ mask */
...cut...
        /* register misc device demo_dma_co */
        sema_init(&the_demo_dma_xx.sem, 1);
        ret_val = misc_register(&demo_dma_co_device);
        if (ret_val != 0) {
                pr_warn("Could not register device \"demo_dma_co\"...");
                goto bad_exit_freeirq;
        }
        /* register misc device demo_dma_st */
        ret_val = misc_register(&demo_dma_st_device);
        if (ret_val != 0) {
                pr_warn("Could not register device \"demo_dma_st\"...");
                goto bad_exit_deregister_demo_dma_co;
        }
...cut...
}
```

Then we register a couple misc devices, a coherent and streaming driver.

demo_module_09.c – DMA

Kernel Space

```
irqreturn_t demo_driver_interrupt_handler(int irq, void *dev_id)
{
		spin_lock(&g_irq_lock);

		/* clear the IRQ state */
		iowrite32(ALTERA_MSGDMA_CSR_IRQ_SET_MASK,
				g_ioremap_csr_addr + CSR_STATUS_REG);

		spin_unlock(&g_irq_lock);
		wake_up_interruptible(&g_irq_wait_queue);
		return IRQ_HANDLED;
}
```

This is our IRQ handler.

demo_module_09.c – DMA

Kernel Space

```
static const struct file_operations demo_dma_st_fops = {
        .owner = THIS_MODULE,
        .read = demo_dma_st_read,
        .write = demo_dma_st_write,
        .llseek = demo_dma_xx_llseek,
};

static struct miscdevice demo_dma_st_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "demo_dma_st",
        .fops = &demo_dma_st_fops,
};

static const struct file_operations demo_dma_co_fops = {
        .owner = THIS_MODULE,
        .read = demo_dma_co_read,
        .write = demo_dma_co_write,
        .llseek = demo_dma_xx_llseek,
};

static struct miscdevice demo_dma_co_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "demo_dma_co",
        .fops = &demo_dma_co_fops,
};
```

Our streaming and coherent drivers register the same functions, the llseek is actually the same function for both implementations.

Kernel Space

demo_module_09.c – DMA

```
static ssize_t demo_dma_co_read(struct file *fp, char __user *user_buffer,
                                        size_t count, loff_t *offset)
{
...cut...
          ram_ptr = g_dma_handle_1m;
          ram_ptr += *offset;
...cut...
          iowrite32(ram_ptr, g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
          iowrite32(g_dma_handle_4k + next_dma_io_buf_ofst,
                        g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
          iowrite32(this_loop_count, g_ioremap_desc_addr + DESC_LENGTH_REG);
          iowrite32(START_DMA_MASK, g_ioremap_desc_addr + DESC_CONTROL_REG);

          temp_dma_count -= this_loop_count;
          ram_ptr += this_loop_count;
          next_dma_io_buf_ofst += this_loop_count;
          if (next_dma_io_buf_ofst >= PAGE_SIZE)
                    next_dma_io_buf_ofst = 0;
```

descriptor configuration

Our coherent read starts to directly interact with the g_dma_handle_4k.

demo_module_09.c – DMA

Kernel Space

descriptor configuration

```
        while (temp_user_count > 0) {
                if (temp_dma_count > 0) {
                        iowrite32(ram_ptr,
                                g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
                        iowrite32(g_dma_handle_4k + next_dma_io_buf_ofst,
                                g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
                        iowrite32(this_loop_count,
                                g_ioremap_desc_addr + DESC_LENGTH_REG);
                        iowrite32(START_DMA_MASK,
                                g_ioremap_desc_addr + DESC_CONTROL_REG);

                        temp_dma_count -= this_loop_count;
                        ram_ptr += this_loop_count;
                        next_dma_io_buf_ofst += this_loop_count;
                        if (next_dma_io_buf_ofst >= PAGE_SIZE)
                                next_dma_io_buf_ofst = 0;
                } else {
                        while (get_dma_busy() != 0) {
                                if (wait_event_interruptible(g_irq_wait_queue,
                                                        (get_dma_busy() ==
                                                         0))) {
...cut...
                                }
                        }
                }
```

Continues to directly interact with the g_dma_handle_4k.

demo_module_09.c – DMA

Kernel Space

```
                    while (get_dma_fill_level() > 0) {
                            if (wait_event_interruptible(g_irq_wait_queue,
                                                                (get_dma_fill_level() ==
                                                                 0))) {
...cut...
                            }
                    }

                    if (copy_to_user(user_buffer, g_coherent_ptr_4k +
                                     next_user_io_buf_ofst, this_loop_count)) {
                            up(&dev->sem);
...cut...
                    }
                    temp_user_count -= this_loop_count;
                    user_buffer += this_loop_count;
                    next_user_io_buf_ofst += this_loop_count;
                    if (next_user_io_buf_ofst >= PAGE_SIZE)
                            next_user_io_buf_ofst = 0;
            }
...cut...
}
```

Continues to directly interact with the g_dma_handle_4k.

demo_module_09.c – DMA

Kernel Space

```
static ssize_t demo_dma_co_write(struct file *fp,
                                         const char __user *user_buffer, size_t count,
                                         loff_t *offset)
{
...cut...
          while (temp_count > 0) {
                    while (get_dma_fill_level() > 2) {
                              if (wait_event_interruptible(g_irq_wait_queue,
                                                                   (get_dma_fill_level() <=
                                                                    2))) {
...cut...
                    }

                    if (copy_from_user(g_coherent_ptr_4k +
                                          next_io_buf_ofst,
                                          user_buffer, this_loop_count)) {
...cut...
                    }

                    iowrite32(g_dma_handle_4k + next_io_buf_ofst,
                                g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
                    iowrite32(ram_ptr,
                                g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
                    iowrite32(this_loop_count,
                                g_ioremap_desc_addr + DESC_LENGTH_REG);
                    iowrite32(START_DMA_MASK,
                                g_ioremap_desc_addr + DESC_CONTROL_REG);
...cut...
}
```

descriptor configuration

Our coherent write directly interacts with the g_dma_handle_4k as well.

ALTERA

demo_module_09.c – DMA

Kernel Space

```
static ssize_t demo_dma_st_read(struct file *fp, char __user *user_buffer,
                                        size_t count, loff_t *offset)
{
...cut...
          ram_ptr = g_dma_handle_1m;
          ram_ptr += *offset;
...cut...
          dma_handle = dma_map_single(dev->pdev_dev,
                                          g_kmalloc_ptr_4k + next_dma_io_buf_ofst,
                                          this_loop_count, DMA_FROM_DEVICE);

          if (dma_mapping_error(dev->pdev_dev, dma_handle)) {
                    up(&dev->sem);
...cut...
          }

          last_dma_handle_1 = last_dma_handle_0;
          last_dma_handle_0 = dma_handle;

          iowrite32(ram_ptr, g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
          iowrite32(dma_handle, g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
          iowrite32(this_loop_count, g_ioremap_desc_addr + DESC_LENGTH_REG);
          iowrite32(START_DMA_MASK, g_ioremap_desc_addr + DESC_CONTROL_REG);

          temp_dma_count -= this_loop_count;
          ram_ptr += this_loop_count;
          next_dma_io_buf_ofst += this_loop_count;
          if (next_dma_io_buf_ofst >= PAGE_SIZE)
                    next_dma_io_buf_ofst = 0;
```

descriptor configuration

Our streaming read first dma_maps the g_kmalloc_ptr_4k to a dma_handle before the DMA. This performs the cache maintenance for us.

demo_module_09.c – DMA

Kernel Space

```
        while (temp_user_count > 0) {
                if (temp_dma_count > 0) {
                        dma_handle = dma_map_single(dev->pdev_dev,
                                                    g_kmalloc_ptr_4k +
                                                    next_dma_io_buf_ofst,
                                                    this_loop_count,
                                                    DMA_FROM_DEVICE);

                        if (dma_mapping_error(dev->pdev_dev, dma_handle)) {
                                up(&dev->sem);
...cut...
                        }

                        last_dma_handle_1 = last_dma_handle_0;
                        last_dma_handle_0 = dma_handle;

                        iowrite32(ram_ptr,
                                g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
                        iowrite32(dma_handle,
                                g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
                        iowrite32(this_loop_count,
                                g_ioremap_desc_addr + DESC_LENGTH_REG);
                        iowrite32(START_DMA_MASK,
                                g_ioremap_desc_addr + DESC_CONTROL_REG);
...cut...
```

descriptor configuration

Then it continues to dma_map the g_kmalloc_ptr_4k again.

demo_module_09.c – DMA

Kernel Space

```
...
                    } else {
                            while (get_dma_busy() != 0) {
                                    if (wait_event_interruptible(g_irq_wait_queue,
                                                                            (get_dma_busy() ==
                                                                             0))) {
...cut...
                    }

                    while (get_dma_fill_level() > 0) {
                            if (wait_event_interruptible(g_irq_wait_queue,
                                                                    (get_dma_fill_level() ==
                                                                     0))) {
...cut...
                    }

                    if (last_dma_handle_1 != 0) {
                            dma_unmap_single(dev->pdev_dev, last_dma_handle_1,
                                                    this_loop_count, DMA_FROM_DEVICE);
                            last_dma_handle_1 = 0;
                    } else if (last_dma_handle_0 != 0) {
                            dma_unmap_single(dev->pdev_dev, last_dma_handle_0,
                                                    this_loop_count, DMA_FROM_DEVICE);
                            last_dma_handle_0 = 0;
                    }

                    if (copy_to_user(user_buffer, g_kmalloc_ptr_4k +
                                            next_user_io_buf_ofst, this_loop_count)) {
...cut...
}
```

Then it dma_unmaps the dma_handle before using the g_kmalloc_ptr_4k.

demo_module_09.c – DMA

Kernel Space

```
static ssize_t demo_dma_st_write(struct file *fp,
                                              const char __user *user_buffer, size_t count,
                                              loff_t *offset)
{
...cut...
          ram_ptr = g_dma_handle_1m;
          ram_ptr += *offset;
...cut...
          while (temp_count > 0) {
                    while (get_dma_fill_level() > 2) {
                              if (wait_event_interruptible(g_irq_wait_queue,
                                                                  (get_dma_fill_level() <=
                                                                   2))) {
...cut...
                    }

                    if (last_dma_handle_3 != 0) {
                              dma_unmap_single(dev->pdev_dev, last_dma_handle_3,
                                                     this_loop_count, DMA_TO_DEVICE);
                              last_dma_handle_3 = 0;
                    }

                    if (copy_from_user(g_kmalloc_ptr_4k +
                                          next_io_buf_ofst,
                                          user_buffer, this_loop_count)) {
                              up(&dev->sem);
                              pr_info("demo_dma_st_write copy_from_user exit\n");
                              return -EFAULT;
                    }

```

The streaming write uses the g_kmalloc_ptr_4k first.

demo_module_09.c – DMA

Kernel Space

```
                    dma_handle = dma_map_single(dev->pdev_dev,
                                                      g_kmalloc_ptr_4k + next_io_buf_ofst,
                                                      this_loop_count, DMA_TO_DEVICE);

                    if (dma_mapping_error(dev->pdev_dev, dma_handle)) {
...cut...
                    }

                    last_dma_handle_3 = last_dma_handle_2;
                    last_dma_handle_2 = last_dma_handle_1;
                    last_dma_handle_1 = last_dma_handle_0;
                    last_dma_handle_0 = dma_handle;

                    iowrite32(dma_handle,
                            g_ioremap_desc_addr + DESC_READ_ADDRESS_REG);
                    iowrite32(ram_ptr,
                            g_ioremap_desc_addr + DESC_WRITE_ADDRESS_REG);
                    iowrite32(this_loop_count,
                            g_ioremap_desc_addr + DESC_LENGTH_REG);
                    iowrite32(START_DMA_MASK,
                            g_ioremap_desc_addr + DESC_CONTROL_REG);
...cut...
          }

          while (get_dma_busy() != 0) {
                    if (wait_event_interruptible(g_irq_wait_queue,
                                                       (get_dma_busy() == 0))) {
...cut...
          }
```

Then it dma_maps it to a dma_handle before the DMA.

descriptor configuration

demo_module_09.c – DMA

Kernel Space

```
		if (last_dma_handle_3 != 0)
			dma_unmap_single(dev->pdev_dev, last_dma_handle_3,
					this_loop_count, DMA_TO_DEVICE);

		if (last_dma_handle_2 != 0)
			dma_unmap_single(dev->pdev_dev, last_dma_handle_2,
					this_loop_count, DMA_TO_DEVICE);

		if (last_dma_handle_1 != 0)
			dma_unmap_single(dev->pdev_dev, last_dma_handle_1,
					this_loop_count, DMA_TO_DEVICE);

		if (last_dma_handle_0 != 0)
			dma_unmap_single(dev->pdev_dev, last_dma_handle_0,
					this_loop_count, DMA_TO_DEVICE);
...cut...
}
```

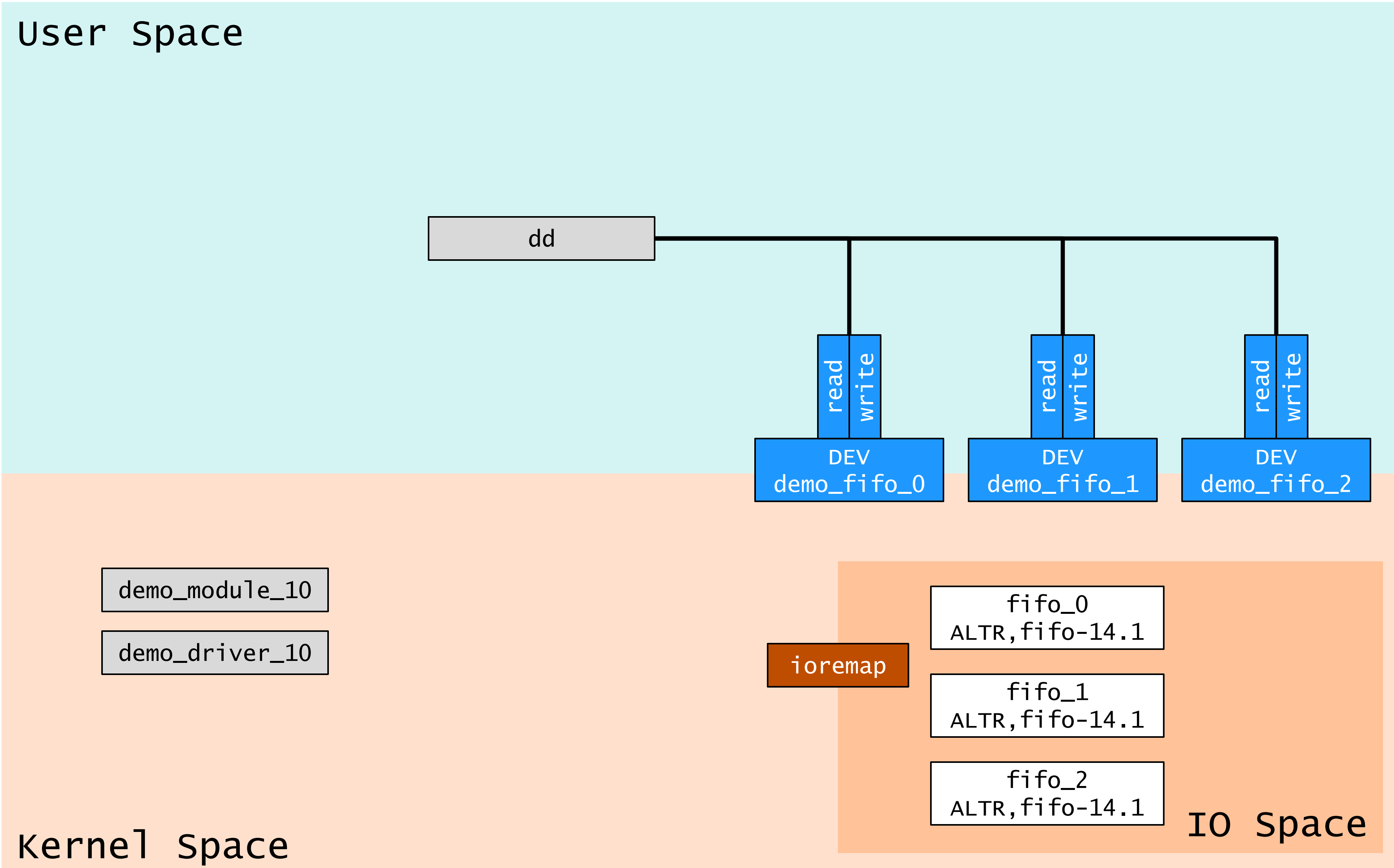
Then it dma_unmaps the dma_handle after the DMA.

demo_module_09.ko target demonstration

```
insmod demo_module_09.ko

# create a couple random files
dd if=/dev/urandom of=random_co.bin bs=1M count=1
dd if=/dev/urandom of=random_st.bin bs=1M count=1

# write to coherent DMA device and read from coherent DMA device
dd if=random_co.bin of=/dev/demo_dma_co bs=1024
dd if=/dev/demo_dma_co of=random_co_data.bin bs=1024

# write to streaming DMA device and read from streaming DMA device
dd if=random_st.bin of=/dev/demo_dma_st bs=1024
dd if=/dev/demo_dma_st of=random_st_data.bin bs=1024

# write to coherent DMA device and read from streaming DMA device
dd if=random_co.bin of=/dev/demo_dma_co bs=1024
dd if=/dev/demo_dma_st of=random_co2st_data.bin bs=1024

# write to streaming DMA device and read from coherent DMA device
dd if=random_st.bin of=/dev/demo_dma_st bs=1024
dd if=/dev/demo_dma_co of=random_st2co_data.bin bs=1024

# compare all the files
md5sum *.bin
rmmod demo_module_09
```

Multiple Device Instances

demo_module_10



demo_module_10.c – Multiple Device Instances

Kernel Space

```
static struct of_device_id demo_driver_dt_ids[] = {
          {
           .compatible = "ALTR,fifo-14.1"},
          { /* end of table */ }
};








static int demo_init(void)
{
          int ret_val;

          INIT_LIST_HEAD(&g_dev_list);
          sema_init(&g_dev_list_sem, 1);
          g_dev_index = 0;

          ret_val = platform_driver_register(&the_platform_driver);
          if (ret_val != 0) {
                    pr_err("platform_driver_register returned %d\n", ret_val);
                    return ret_val;
          }

          return 0;
}

```

We bind this example with devices matching this compatible string.

Our init routine clears a global index that we track each instance of our driver that be bind to each device.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
static int platform_probe(struct platform_device *pdev)
{
		int ret_val;
		struct resource *r0 = NULL;
		struct resource *r1 = NULL;
		struct resource *r2 = NULL;
		struct resource *temp_res = NULL;
		struct demo_fifo_dev *the_demo_fifo_dev;
		uint32_t fifo_level;

		if (down_interruptible(&g_dev_list_sem))
				return -ERESTARTSYS;

		ret_val = -ENOMEM;

		/* allocate a demo_fifo_dev structure */
		the_demo_fifo_dev = kzalloc(sizeof(struct demo_fifo_dev), GFP_KERNEL);
		if (the_demo_fifo_dev == NULL) {
				pr_err("kzalloc failed\n");
				goto bad_exit_return;
		}
```

Our probe routine allocates the storage for our device structure rather than using a statically allocated structure like the previous examples.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
        /* initialize the demo_fifo_dev structure */
        scnprintf(the_demo_fifo_dev->name, NAME_BUF_SIZE, "demo_fifo_%d",
                g_dev_index);

        INIT_LIST_HEAD(&the_demo_fifo_dev->dev_list);
        sema_init(&the_demo_fifo_dev->dev_sem, 1);
        init_waitqueue_head(&the_demo_fifo_dev->wait_queue);

        the_demo_fifo_dev->in_res = NULL;
        the_demo_fifo_dev->out_res = NULL;
        the_demo_fifo_dev->in_csr_res = NULL;

        the_demo_fifo_dev->ioremap_in_addr = NULL;
        the_demo_fifo_dev->ioremap_out_addr = NULL;
        the_demo_fifo_dev->ioremap_in_csr_addr = NULL;

        the_demo_fifo_dev->open_for_read = 0;
        the_demo_fifo_dev->open_for_write = 0;

        the_demo_fifo_dev->miscdev.minor = MISC_DYNAMIC_MINOR;
        the_demo_fifo_dev->miscdev.name = the_demo_fifo_dev->name;
        the_demo_fifo_dev->miscdev.fops = &demo_fifo_fops;

        ret_val = -EINVAL;
```

Then we initialize the newly allocated device structure. The name being the most apparent item that we will see in user space.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
	/* get our three expected memory resources */
	r0 = platform_get_resource(pdev, IORESOURCE_MEM, 0);
	if (r0 == NULL) {
		pr_err("IORESOURCE_MEM, 0 does not exist\n");
		goto bad_exit_kfree_the_demo_fifo_dev;
	}

	r1 = platform_get_resource(pdev, IORESOURCE_MEM, 1);
	if (r1 == NULL) {
		pr_err("IORESOURCE_MEM, 1 does not exist\n");
		goto bad_exit_kfree_the_demo_fifo_dev;
	}

	r2 = platform_get_resource(pdev, IORESOURCE_MEM, 2);
	if (r2 == NULL) {
		pr_err("IORESOURCE_MEM, 2 does not exist\n");
		goto bad_exit_kfree_the_demo_fifo_dev;
	}
```

We expect the FIFO devices to present three memory resources.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
        /* associate the resources for in, out and in_csr */

        if (!strcmp(r0->name, "in"))
                the_demo_fifo_dev->in_res = r0;
        else if (!strcmp(r1->name, "in"))
                the_demo_fifo_dev->in_res = r1;
        else if (!strcmp(r2->name, "in"))
                the_demo_fifo_dev->in_res = r2;

        if (!strcmp(r0->name, "out"))
                the_demo_fifo_dev->out_res = r0;
        else if (!strcmp(r1->name, "out"))
                the_demo_fifo_dev->out_res = r1;
        else if (!strcmp(r2->name, "out"))
                the_demo_fifo_dev->out_res = r2;

        if (!strcmp(r0->name, "in_csr"))
                the_demo_fifo_dev->in_csr_res = r0;
        else if (!strcmp(r1->name, "in_csr"))
                the_demo_fifo_dev->in_csr_res = r1;
        else if (!strcmp(r2->name, "in_csr"))
                the_demo_fifo_dev->in_csr_res = r2;
```

We don’t assume the order in which the resources are presented, our software needs to know which resource belongs to which register set. So we figure that out by the name associated with each resource.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
        /* verify that we found all three resources */
        if (the_demo_fifo_dev->in_res == NULL) {
                pr_err("no resource found for \"in\"\n");
                goto bad_exit_kfree_the_demo_fifo_dev;
        }

        if (the_demo_fifo_dev->out_res == NULL) {
                pr_err("no resource found for \"out\"\n");
                goto bad_exit_kfree_the_demo_fifo_dev;
        }

        if (the_demo_fifo_dev->in_csr_res == NULL) {
                pr_err("no resource found for \"in_csr\"\n");
                goto bad_exit_kfree_the_demo_fifo_dev;
        }
```

And after all that we just verify that we actually found each named resource that we were looking for.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
		/* reserve our memory regions */
...cut...
		/* ioremap our memory regions */
...cut...
		/* initialize the FIFO hardware */
		fifo_level = ioread32(the_demo_fifo_dev->ioremap_in_csr_addr +
							FIFO_LEVEL_REG);
		while (fifo_level > 0) {
			ioread32(the_demo_fifo_dev->ioremap_out_addr + FIFO_DATA_REG);
			fifo_level--;
		}

		fifo_level = ioread32(the_demo_fifo_dev->ioremap_in_csr_addr +
							FIFO_LEVEL_REG);

		if (fifo_level != 0) {
			pr_err("fifo initialization failed");
			goto bad_exit_iounmap_in_csr;
		}

		ret_val = -EINVAL;

		/* register misc device demo_fifo */
...cut...
		/* clean up and exit */
		list_add(&the_demo_fifo_dev->dev_list, &g_dev_list);
		g_dev_index++;
		platform_set_drvdata(pdev, the_demo_fifo_dev);
		up(&g_dev_list_sem);
		return 0;
```

We initialize our FIFO hardware.

And finally we add our device to a global list that we track our device instances with, increment the g_dev_index and store our device structure into the platform driver data.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
static const struct file_operations demo_fifo_fops = {
		.owner = THIS_MODULE,
		.open = demo_fifo_open,
		.release = demo_fifo_release,
		.read = demo_fifo_read,
		.write = demo_fifo_write,
};
```

We can open, release, read and write our FIFO device.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
static int demo_fifo_open(struct inode *ip, struct file *fp)
{
		struct demo_fifo_dev *the_demo_fifo_dev = NULL;
		uint32_t this_minor;
		uint32_t access_mode;
		struct list_head *next_list_entry;
		int found_it = 0;

		if (down_interruptible(&g_dev_list_sem))
			return -ERESTARTSYS;

		this_minor = iminor(ip);

		list_for_each(next_list_entry, &g_dev_list) {
			the_demo_fifo_dev = list_entry(next_list_entry,
						struct demo_fifo_dev, dev_list);
			if (the_demo_fifo_dev->miscdev.minor == this_minor) {
				found_it = 1;
				break;
			}
		}

		up(&g_dev_list_sem);

		if (found_it == 0)
			return -ENXIO;
```

Our open routine needs to search thru our device list to locate our device instance that matches the device that’s being opened.

The device minor number is what identifies this for us.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
	access_mode = fp->f_flags & O_ACCMODE;
	switch (access_mode) {
	case (O_RDONLY):
		if (the_demo_fifo_dev->open_for_read != 0)
			return -EBUSY;
		the_demo_fifo_dev->open_for_read = 1;
		break;
	case (O_WRONLY):
		if (the_demo_fifo_dev->open_for_write != 0)
			return -EBUSY;
		the_demo_fifo_dev->open_for_write = 1;
		break;
	case (O_RDWR):
		if (the_demo_fifo_dev->open_for_read != 0)
			return -EBUSY;
		if (the_demo_fifo_dev->open_for_write != 0)
			return -EBUSY;
		the_demo_fifo_dev->open_for_read = 1;
		the_demo_fifo_dev->open_for_write = 1;
		break;
	default:
		return -EINVAL;
	}

	fp->private_data = the_demo_fifo_dev;

	return 0;
}
```

We ensure that we only allow one file descriptor be be associated with reading or writing the device at a time.

We determine this from the access mode of the file.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
static ssize_t demo_fifo_read(struct file *fp, char __user *user_buffer,
                                    size_t count, loff_t *offset)
{
          struct demo_fifo_dev *the_demo_fifo_dev = fp->private_data;
...cut...
          if (down_interruptible(&the_demo_fifo_dev->dev_sem)) {
...cut...
          }

          if (count & (4 - 1)) {
...cut...
          }

          while (ioread32(the_demo_fifo_dev->ioremap_in_csr_addr + FIFO_LEVEL_REG)
                 == 0) {
                    up(&the_demo_fifo_dev->dev_sem);
                    if (wait_event_interruptible(the_demo_fifo_dev->wait_queue,
                                                      (ioread32
                                                       (the_demo_fifo_dev->
                                                        ioremap_in_csr_addr
                                                        + FIFO_LEVEL_REG) != 0))) {
                              pr_info("demo_fifo_read wait interrupted exit\n");
                              return -ERESTARTSYS;
                    }
                    if (down_interruptible(&the_demo_fifo_dev->dev_sem)) {
                              pr_info("demo_fifo_read sem interrupted exit\n");
                              return -ERESTARTSYS;
                    }
          }

```

We only accept a modulo 4 count.

Our read blocks on an empty FIFO and waits for the write routine to clear the wait queue.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
        fifo_level = ioread32(the_demo_fifo_dev->ioremap_in_csr_addr +
                                FIFO_LEVEL_REG);

        while (fifo_level > 0) {
                temp_data = ioread32(the_demo_fifo_dev->ioremap_out_addr +
                                        FIFO_DATA_REG);
                if (copy_to_user(user_buffer, &temp_data, 4)) {
                        up(&the_demo_fifo_dev->dev_sem);
                        pr_info("demo_fifo_read copy_to_user exit\n");
                        return -EFAULT;
                }
                user_buffer += 4;
                fifo_level--;
                this_count += 4;
                count -= 4;
                if (count == 0)
                        break;
        }

        up(&the_demo_fifo_dev->dev_sem);
        wake_up_interruptible(&the_demo_fifo_dev->wait_queue);

        return this_count;
}
```

Then we read whatever we can out of the FIFO, up to count.

And we wake the wait queue when we're finished.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
static ssize_t demo_fifo_write(struct file *fp,
                                     const char __user *user_buffer, size_t count,
                                     loff_t *offset)
{
          struct demo_fifo_dev *the_demo_fifo_dev = fp->private_data;
...cut...
          if (down_interruptible(&the_demo_fifo_dev->dev_sem)) {
...cut...
          }

          if (count & (4 - 1)) {
...cut...
          }

          while (ioread32(the_demo_fifo_dev->ioremap_in_csr_addr + FIFO_LEVEL_REG)
                 >= FIFO_MAX_FILL_LEVEL) {
                    up(&the_demo_fifo_dev->dev_sem);
                    if (wait_event_interruptible(the_demo_fifo_dev->wait_queue,
                                                          (ioread32
                                                           (the_demo_fifo_dev->
                                                            ioremap_in_csr_addr
                                                            + FIFO_LEVEL_REG) <
                                                           FIFO_MAX_FILL_LEVEL))) {
                              pr_info("demo_fifo_write wait interrupted exit\n");
                              return -ERESTARTSYS;
                    }
                    if (down_interruptible(&the_demo_fifo_dev->dev_sem)) {
...cut...
                    }
          }
```

We only accept a modulo 4 count.

Our write blocks on a full FIFO and waits for the read routine to clear the wait queue.

demo_module_10.c – Multiple Device Instances

Kernel Space

```
		fifo_level = ioread32(the_demo_fifo_dev->ioremap_in_csr_addr +
					FIFO_LEVEL_REG);

		while (fifo_level < FIFO_MAX_FILL_LEVEL) {
			if (copy_from_user(&temp_data, user_buffer, 4)) {
				up(&the_demo_fifo_dev->dev_sem);
				pr_info("demo_fifo_write copy_to_user exit\n");
				return -EFAULT;
			}
			iowrite32(temp_data, the_demo_fifo_dev->ioremap_in_addr +
				FIFO_DATA_REG);

			user_buffer += 4;
			fifo_level++;
			this_count += 4;
			count -= 4;
			if (count == 0)
				break;
		}

		up(&the_demo_fifo_dev->dev_sem);
		wake_up_interruptible(&the_demo_fifo_dev->wait_queue);

		return this_count;
}
```

Then we write whatever we can into the FIFO, up to count.

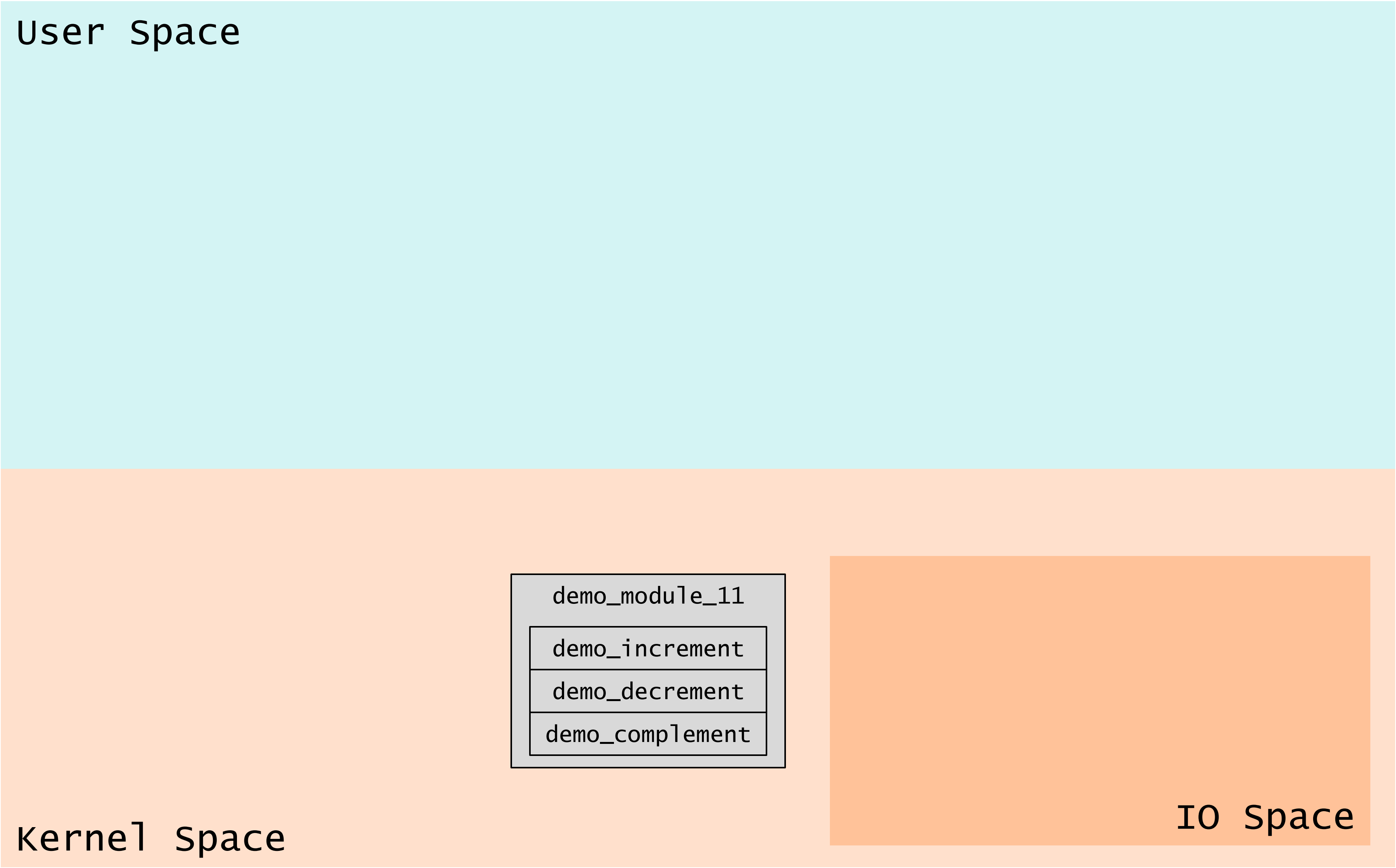
And we wake the wait queue when we’re finished.

demo_module_10.ko target demonstration

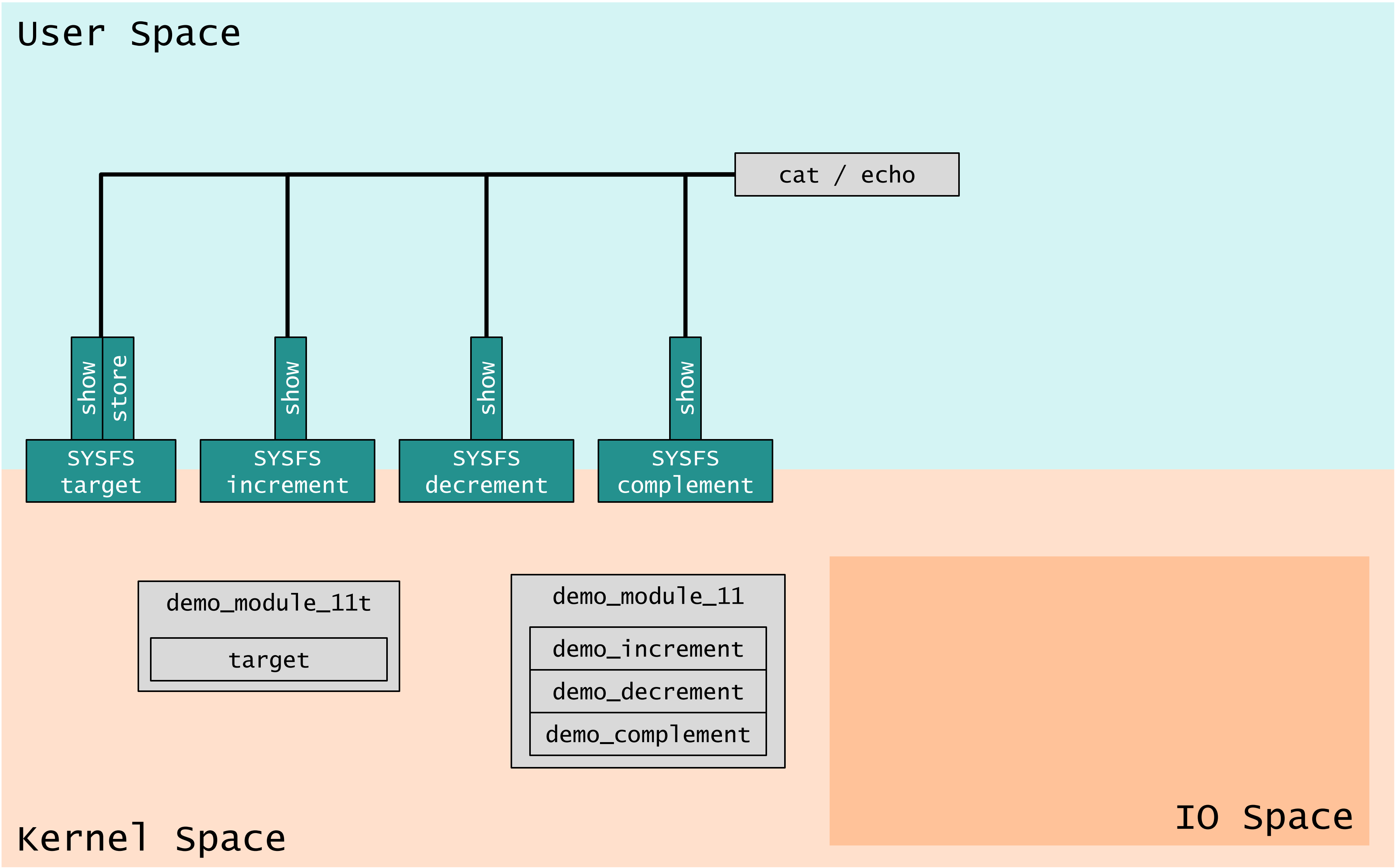
```
insmod demo_module_10.ko                            # insert the module
find /dev -name "*demo*"                            # observe the device nodes
find /sys -name "*demo*"                            # observe the sysfs files
ls /sys/bus/platform/drivers/demo_driver_10         # notice the 3 devices bound

# create 3 random data files
dd if=/dev/urandom of=random0.bin bs=256K count=1
dd if=/dev/urandom of=random1.bin bs=256K count=1
dd if=/dev/urandom of=random2.bin bs=256K count=1

# start 3 separate processes to write the random files to a different fifo
dd if=random0.bin of=/dev/demo_fifo_0 bs=1024 &
dd if=random1.bin of=/dev/demo_fifo_1 bs=1024 &
dd if=random2.bin of=/dev/demo_fifo_2 bs=1024 &

# read back the first 1K of data from each fifo
dd if=/dev/demo_fifo_0 of=random0_out.bin bs=1024 count=1
dd if=/dev/demo_fifo_1 of=random1_out.bin bs=1024 count=1
dd if=/dev/demo_fifo_2 of=random2_out.bin bs=1024 count=1

# read back the remaining data from all the fifos
dd if=/dev/demo_fifo_0 of=random0_out.bin bs=1024 count=255 conv=notrunc seek=1 &
dd if=/dev/demo_fifo_1 of=random1_out.bin bs=1024 count=255 conv=notrunc seek=1 &
dd if=/dev/demo_fifo_2 of=random2_out.bin bs=1024 count=255 conv=notrunc seek=1 &

wait                                                # wait for background processes
md5sum *.bin                                        # validate the in/out data files
rmmod demo_module_10                                # remove the module
```

Custom API

demo_module_11



demo_module_11t



demo_module_11.c – Custom API

Kernel Space

```
void demo_increment(uint32_t *target)
{
	*target += 1;
}
EXPORT_SYMBOL(demo_increment);

void demo_decrement(uint32_t *target)
{
	*target -= 1;
}
EXPORT_SYMBOL(demo_decrement);

void demo_complement(uint32_t *target)
{
	*target = ~(*target);
}
EXPORT_SYMBOL(demo_complement);

static int demo_init(void)
{
	pr_info("demo_init enter\n");
	pr_info("demo_init exit\n");
	return 0;
}

static void demo_exit(void)
{
	pr_info("demo_exit enter\n");
	pr_info("demo_exit exit\n");
}
```

We simply export some entry points that become available in kernel space once our module is loaded.

demo_module_11t.c – Custom API

Kernel Space

```
static int demo_init(void)
{
...cut...
	ret_val = platform_driver_register(&the_platform_driver);
...cut...
	/* create the sysfs entries */
	ret_val = driver_create_file(&the_platform_driver.driver,
	                                   &driver_attr_target);
...cut...
	ret_val = driver_create_file(&the_platform_driver.driver,
	                                   &driver_attr_increment);
...cut...
	ret_val = driver_create_file(&the_platform_driver.driver,
	                                   &driver_attr_decrement);
...cut...
	ret_val = driver_create_file(&the_platform_driver.driver,
	                                   &driver_attr_complement);
...cut...
	return 0;

bad_exit_remove_decrement_file:
	driver_remove_file(&the_platform_driver.driver, &driver_attr_decrement);
bad_exit_remove_increment_file:
	driver_remove_file(&the_platform_driver.driver, &driver_attr_increment);
bad_exit_remove_target_file:
	driver_remove_file(&the_platform_driver.driver, &driver_attr_target);
bad_exit_platform_driver_unregister:
	platform_driver_unregister(&the_platform_driver);
bad_exit_return:
	return ret_val;
```

This test module defines a number of sysfs files that we can interact with our custom API through.

demo_module_11t.c – Custom API

Kernel
Space

```
static uint32_t g_target;

static ssize_t target_show(struct device_driver *driver, char *buf)
{
          return scnprintf(buf, PAGE_SIZE, "0x%08X\n", g_target);
}

static ssize_t target_store(struct device_driver *driver, const char *buf,
                                  size_t count)
{
          int result;
          unsigned long new_target;

          /* convert the input string to the requested new target value */
          result = kstrtoul(buf, 0, &new_target);
          if (result != 0)
                    return -EINVAL;

          g_target = new_target;

          return count;
}

DRIVER_ATTR(target, (S_IWUGO | S_IRUGO), target_show, target_store);
```

These are our “target” variable access routines.

demo_module_11t.c – Custom API

Kernel Space

```
static ssize_t increment_show(struct device_driver *driver, char *buf)
{
		demo_increment(&g_target);

		return scnprintf(buf, PAGE_SIZE, "0x%08X\n", g_target);
}

DRIVER_ATTR(increment, (S_IRUGO), increment_show, NULL);

static ssize_t decrement_show(struct device_driver *driver, char *buf)
{
		demo_decrement(&g_target);

		return scnprintf(buf, PAGE_SIZE, "0x%08X\n", g_target);
}

DRIVER_ATTR(decrement, (S_IRUGO), decrement_show, NULL);

static ssize_t complement_show(struct device_driver *driver, char *buf)
{
		demo_complement(&g_target);

		return scnprintf(buf, PAGE_SIZE, "0x%08X\n", g_target);
}

DRIVER_ATTR(complement, (S_IRUGO), complement_show, NULL);
```

These are our target “increment”, “decrement” and “complement” access routines.

demo_module_11.ko target demonstration

```
insmod demo_module_11t.ko                              # observe module insert fails
insmod demo_module_11.ko                               # insert module
insmod demo_module_11t.ko                              # insert module
find /sys -name "*demo*"                               # observe entries created
ls /sys/bus/platform/drivers/demo_driver_11t           # observe sysfs files

# exercise the sysfs entries
cat /sys/bus/platform/drivers/demo_driver_11t/target
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/decrement
cat /sys/bus/platform/drivers/demo_driver_11t/decrement
cat /sys/bus/platform/drivers/demo_driver_11t/complement
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/increment
cat /sys/bus/platform/drivers/demo_driver_11t/decrement
cat /sys/bus/platform/drivers/demo_driver_11t/complement
rmmod demo_module_11                                   # observe remove module fails
rmmod demo_module_11t                                  # remove module
rmmod demo_module_11                                   # remove module
```

Take Home Lab

What You'll Need

- A supported development kit
  - ALTERA_AV_SOC - Altera
  - ALTERA_CV_SOC - Altera
  - ARROW_SOCKIT - Arrow
  - CRITICALLINK_MITYSOM_DEVKIT - Critical Link
  - DE0_NANO_SOC - Terasic
  - MACNICA_HELIO_12, 13 or 14 - Macnica
- A blank SD Card for your development kit
  - To install the workshop target environment that boots the above boards.
- Communication cables
  - Provides connectivity between host and target.
    - Console UART
    - USB gadget mass storage and networking
- Linux machine, native or VM
  - 4GByte RAM minimum
- Serial terminal application
  - Putty, minicom, or similar, for console UART communication with target

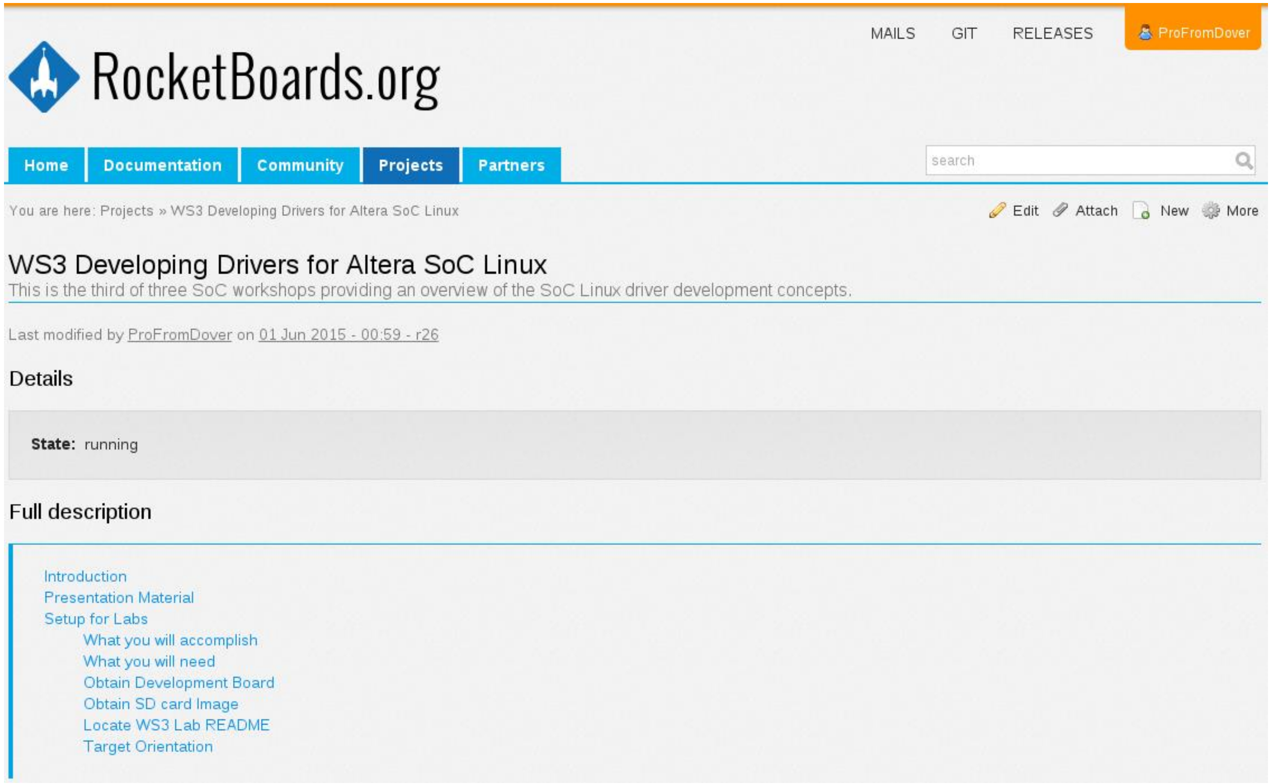
Obtaining Lab Files and Instructions

- SoC SW workshop series getting started page

http://www.rocketboards.org/foswiki/Projects/SoCSWWorkshopSeriesGettingStarted

- WS3 lab instructions are posted to Rocketboards.org

http://www.rocketboards.org/foswiki/Projects/WS3DevelopingDriversForAlteraSoCLinux



What You Will Accomplish

- Configure your host development environment.
- Build an example misc driver.
- Build a test application for the misc driver.
- Modify the uio_pdrv_genirq template and build it.
- Build a test application for the UIO driver.
- Load all of the above onto your dev kit target.
- Run the test applications to verify proper driver functionality.
- Run a validation program.
  - Verifies the lab was completed.
- Submit results & feedback.

Thank You