SCIENCE
PASSION
TECHNOLOGY

System Integration (HW - SW - Linux)

Barbara Gigerl, Rishub Nagpal

October 12th, 2022

Part 1
Creating a Custom IP core

Overview

- What we want?
  Extend the existing HW design by our individual IP core

Overview

- What we want?
  Extend the existing HW design by our individual IP core
- What we have?
  A Zybo FPGA board, a hardware design, software

Overview

- What we want?
  Extend the existing HW design by our individual IP core

- What we have?
  A Zybo FPGA board, a hardware design, software

- How do we get there?

IP cores

- IP = Intellectual Property

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior
- Comparable to using a library in C

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior
- Comparable to using a library in C
- Examples:

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior
- Comparable to using a library in C
- Examples:
  - Peripheral controllers like Ethernet, HDMI, VGA, USB, ...

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior
- Comparable to using a library in C
- Examples:
  - Peripheral controllers like Ethernet, HDMI, VGA, USB, ...
  - Crypto cores

IP cores

- IP = Intellectual Property
- Reusable logic component with a defined interface and behavior
- Comparable to using a library in C
- Examples:
  - Peripheral controllers like Ethernet, HDMI, VGA, USB, ...
  - Crypto cores
  - Debug cores

Creating a new IP core in Vivado

1. Tools - Create and Package New IP
2. Create a new AXI4 peripheral
3. Enter name of your choice
4. Next steps: Edit IP
5. Finish
6. IP editor will show 2 files:
   - `<IP_core_name>_v1_0_S00_AXI.v`
   - `<IP_core_name>_v1_0.v`

Editing the IP core

`<IP_core_name>_v1_0_S00_AXI.v`

- Define input ports for user inputs
- Define output ports for output to user
- Specify custom IP core logic
- TODO: Adapt ports and add logic

`<IP_core_name>_v1_0.v`

- AXI wrapper of our IP core
- Instantiates `<IP_core_name>_v1_0_S00_AXI.v`
- TODO: Adapt ports and instantiation

Package and integrate the IP core

1. Select Package IP and choose Merge Changes where necessary
2. Finish packaging with Re-Package IP and close the project
3. Open the block design and select Add IP to add our `<IP_core_name>`
4. Run connection automation
5. For each IO port: Create Port...
6. Validate Design
7. Right click on the block design in Project Manager - Create HDL Wrapper
8. Adapt Constraints file if necessary

Adding SW

1. In Vivado: observe AXI Base Address in the Address Editor
2. Open Vitis SDK as shown before
3. Use observed address to communicate with HW

Adding SW

1. In Vivado: observe AXI Base Address in the Address Editor
2. Open Vitis SDK as shown before
3. Use observed address to communicate with HW

```
//Write
*((int*)0x43c20000) = 0x1;

//Read
int value = *((int*)0x43c20000);
```

Adding SW

1. In Vivado: observe AXI Base Address in the Address Editor
2. Open Vitis SDK as shown before
3. Use observed address to communicate with HW

```
//Write
*((int*)0x43c20000) = 0x1;

//Read
int value = *((int*)0x43c20000);
```

→ not very comfortable!

Part 2
Building, Deploying, and Running Linux

Overview

- What we want?
  Boot Linux and run a C program

Overview

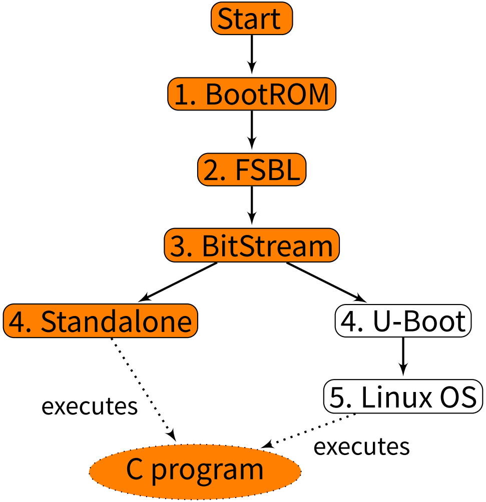
- What we want?
  Boot Linux and run a C program

- What we have?
  A Zybo FPGA board, a hardware design, software, a Linux OS

  - How do we get there?

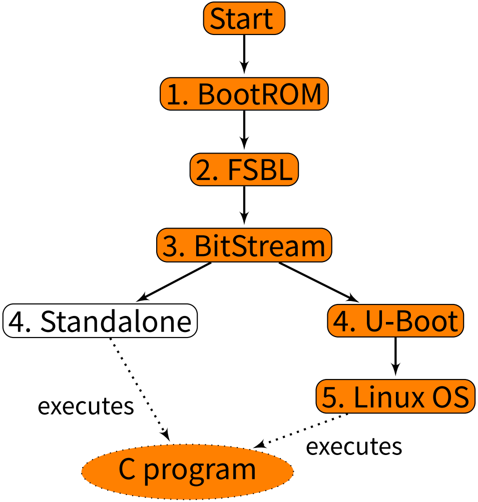
    1. Try Buildroot setup by running simple Linux with Init Ramdisk
    2. Build a device tree for our board
    3. Write a device driver
    4. Use Buildroot to build Linux with correct device tree file and device driver

Last time...



Start
1. BootROM
2. FSBL
3. BitStream
4. Standalone
4. U-Boot
5. Linux OS
executes
executes
C program

Today


Start
1. BootROM
2. FSBL
3. BitStream
4. Standalone
4. U-Boot
5. Linux OS
executes
executes
C program

Part 2a

Building Linux

Buildroot

- Pre-build Linux images might not be suitable.

Buildroot

- Pre-build Linux images might not be suitable.
- Buildroot: automate build process for a specific platform

Buildroot

- Pre-build Linux images might not be suitable.
- Buildroot: automate build process for a specific platform
- Based on makefiles

Buildroot

- Pre-build Linux images might not be suitable.
- Buildroot: automate build process for a specific platform
- Based on makefiles
- Complicated, but much less complicated than building the image without it

Buildroot

- Pre-build Linux images might not be suitable.
- Buildroot: automate build process for a specific platform
- Based on makefiles
- Complicated, but much less complicated than building the image without it
- GUI based on curses

Buildroot

- Pre-build Linux images might not be suitable.
- Buildroot: automate build process for a specific platform
- Based on makefiles
- Complicated, but much less complicated than building the image without it
- GUI based on curses
- Many options to configure (packages, platforms, ...)

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template
- `linux`: the linux kernel

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template
- `linux`: the linux kernel
- `package`: userspace packages, e.g. Python, git, ...

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template
- `linux`: the linux kernel
- `package`: userspace packages, e.g. Python, git, ...
- `fs`: filesystem images

The Buildroot tool directory

- `Makefile`: top-level ”master” Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template
- `linux`: the linux kernel
- `package`: userspace packages, e.g. Python, git, ...
- `fs`: filesystem images
- `boot`: bootloader packages

The Buildroot tool directory

- `Makefile`: top-level "master" Makefile
- `Config.in`: general configurations
- `configs`, `board`: board configuration files
- `arch`: contains config files for supported architectures
- `system/skeleton`: rootfs template
- `linux`: the linux kernel
- `package`: userspace packages, e.g. Python, git, ...
- `fs`: filesystem images
- `boot`: bootloader packages
- `docs`: buildroot documentation

The Buildroot output directory

- After the build process finished, build artefacts are stored in `output`
- Contains a lot of background information
- `output/images`
  - Kernel image,
  - Bootloader image,
  - Root file system image, ...

Yocto

- Buildroot: small, simple, gives quick results

Yocto

- Buildroot: small, simple, gives quick results
- Yocto: needs more build time, requires more disk space, is more complex

Yocto

- Buildroot: small, simple, gives quick results
- Yocto: needs more build time, requires more disk space, is more complex
- Main advantage: more boards supported, more options to configure packages

Yocto

- Buildroot: small, simple, gives quick results
- Yocto: needs more build time, requires more disk space, is more complex
- Main advantage: more boards supported, more options to configure packages
- Both serve the same purpose

Yocto

- Buildroot: small, simple, gives quick results
- Yocto: needs more build time, requires more disk space, is more complex
- Main advantage: more boards supported, more options to configure packages
- Both serve the same purpose
- If you're interested: https://extgit.iaik.tugraz.at/sip/zybo_base_design/-/blob/master/README.yocto.md

Part 2b
Booting Linux

Bootloader

- Task: initialize everything such that OS can be run

Bootloader

- Task: initialize everything such that OS can be run
- Highly processor and board specific

Bootloader

- Task: initialize everything such that OS can be run
- Highly processor and board specific
- Minimum peripheral initialization if needed (wake-on-lan, ...)

Bootloader

- Task: initialize everything such that OS can be run
- Highly processor and board specific
- Minimum peripheral initialization if needed (wake-on-lan, ...)
- Decide on kernel image and load it

Bootloader

- Task: initialize everything such that OS can be run
- Highly processor and board specific
- Minimum peripheral initialization if needed (wake-on-lan, ...)
- Decide on kernel image and load it
- **FSBL:** configure FPGA, prepare processor and basic peripherals, loads the SSBL

Bootloader

- Task: initialize everything such that OS can be run
- Highly processor and board specific
- Minimum peripheral initialization if needed (wake-on-lan, ...)
- Decide on kernel image and load it
- **FSBL:** configure FPGA, prepare processor and basic peripherals, loads the SSBL
- **SSBL:** U-boot or grub, more complex peripherals, load kernel

Bootloader

Buildroot supports many different bootloaders, for example:

- U-Boot

Bootloader

Buildroot supports many different bootloaders, for example:

- U-Boot
- Barebox: derived from U-Boot (has more beautiful code)

Bootloader

Buildroot supports many different bootloaders, for example:

- U-Boot
- Barebox: derived from U-Boot (has more beautiful code)
- Grub: Windows support, bigger bootloader

Bootloader

Buildroot supports many different bootloaders, for example:

- U-Boot
- Barebox: derived from U-Boot (has more beautiful code)
- Grub: Windows support, bigger bootloader
- xloader, AT91bootstrap: for AVR microcontrollers

U-boot

- Boot loader for embedded devices

U-boot

- Boot loader for embedded devices
- Supports 13 architectures and about 300 different boards

U-boot

- Boot loader for embedded devices
- Supports 13 architectures and about 300 different boards
- Used in many projects:
  - ARM-based Chromebooks
  - Amazon Kindle
  - SpaceX

Preparation

- The base demo project has been built and is still available.
  - Including Bitstream
  - Including FSBL
  - Including User application
- Install buildroot into `<BUILDROOT>`

  `git submodule update --init`

Simple Linux with Init Ramdisk

- Test your setup
- Linux without FPGA Bitstream
- Buildroot does not have a default configuration for the Zybo board
  - Adapt the one from Zedboard
  - Can be found in `zybo-buildroot-simple`
- Build commands:
  1. `cd <BUILDROOT>`
  2. `make BR2_EXTERNAL=../zybo-buildroot-simple zynq_zybo_defconfig`
  3. `make`
- `BR2_EXTERNAL`: separate Buildroot from board-specific customizations

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary
  - `zImage`: compressed kernel image (self-extracting)

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary
  - `zImage`: compressed kernel image (self-extracting)
  - Wrapper = 64 byte header before zImage (version, loading position, size, ...)

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary
  - `zImage`: compressed kernel image (self-extracting)
  - Wrapper = 64 byte header before zImage (version, loading position, size, ...)
- `rootfs.cpio.uboot`: initial Linux root file system

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary
  - `zImage`: compressed kernel image (self-extracting)
  - Wrapper = 64 byte header before zImage (version, loading position, size, ...)
- `rootfs.cpio.uboot`: initial Linux root file system
- `zynq-zybo.dtb`: device tree blob

Simple Linux with Init Ramdisk

Output files in `<BUILDROOT>/output/images`

- `uEnv.txt`: U-Boot environment file
- `uImage`: Kernel image with U-Boot wrapper
  - `image`: generic kernel binary
  - `zImage`: compressed kernel image (self-extracting)
  - Wrapper = 64 byte header before zImage (version, loading position, size, ...)
- `rootfs.cpio.uboot`: initial Linux root file system
- `zynq-zybo.dtb`: device tree blob
- `boot.bin`, `u-boot.img`: (U-Boot) images

Hints and (possible) errors

- ```
  You have PERL_MM_OPT defined because Perl local::lib is installed
  on your system. Please unset this variable
  before starting Buildroot, otherwise the
  compilation of Perl related packages will fail
  ```

  Solution: `unset PERL_MM_OPT`
- You might encounter problems when using `gcc >= 10`. If so, either downgrade your compiler (we use `9.4.0` and `9.3.0`) or update buildroot.
- Install `libssl-dev`
  ```

Simple Linux with Init Ramdisk

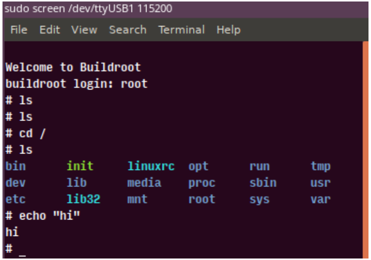
Test your setup:

- Make sure SD card is formatted correctly
  - First partition: FAT32, around 50 MB
  - Second partition: ext4 or other, used as root file system and data storage

Simple Linux with Init Ramdisk

Test your setup:

- Make sure SD card is formatted correctly
  - First partition: FAT32, around 50 MB
  - Second partition: ext4 or other, used as root file system and data storage
- Copy to SD card:
  - `boot.bin`
  - `rootfs.cpio.uboot`
  - `u-boot.img`
  - `uImage`
  - `uEnv.txt`
  - `zynq-zybo.dtb`



Part 2c

Linux Device Trees

The Device Tree

Booting without a device tree

- Kernel image contains the whole hardware configuration.

The Device Tree

Booting without a device tree

- Kernel image contains the whole hardware configuration.
- Bootloader (U-Boot) loads **a single binary**: the kernel image

The Device Tree

Booting without a device tree

- Kernel image contains the whole hardware configuration.
- Bootloader (U-Boot) loads **a single binary**: the kernel image
- Kernel image runs as a bare-metal application on the CPU.

The Device Tree

Booting without a device tree

- Kernel image contains the whole hardware configuration.
- Bootloader (U-Boot) loads **a single binary**: the kernel image
- Kernel image runs as a bare-metal application on the CPU.
- Disadvantage: need to recompile kernel for every specific chip for every specific board

The Device Tree

Booting with a device tree

- Kernel is kernel and hardware config is hardware config

The Device Tree

Booting with a device tree

- Kernel is kernel and hardware config is hardware config
- **Device tree blob**: separate binary containing the hardware description

The Device Tree

Booting with a device tree

- Kernel is kernel and hardware config is hardware config
- **Device tree blob**: separate binary containing the hardware description
- Bootloader (U-Boot) loads **two binaries**: the kernel image and the DTB

The Device Tree

Booting with a device tree

- Kernel is kernel and hardware config is hardware config
- **Device tree blob**: separate binary containing the hardware description
- Bootloader (U-Boot) loads **two binaries**: the kernel image and the DTB
- Decouples the hardware description from the kernel image

The Device Tree

- Device tree: tree data structure with nodes that describe physical devices in system

The Device Tree

- Device tree: tree data structure with nodes that describe physical devices in system
- Formats:
  1. Text file (.dts): source
  2. Binary blob (.dtb): loaded by bootloader
  3. File system in a running Linux: `/proc/device-tree`, node = directory

The Device Tree

- Device tree: tree data structure with nodes that describe physical devices in system
- Formats:
  1. Text file (.dts): source
  2. Binary blob (.dtb): loaded by bootloader
  3. File system in a running Linux: `/proc/device-tree`, node = directory
- Example: https://github.com/Xilinx/linux-xlnx/blob/master/arch/arm64/boot/dts/xilinx/zynqmp.dtsi
- More information: http://xillybus.com/tutorials/device-tree-zynq-1

Device Tree Structure

- Device tree for Linux on Zynq mostly consists of:
  - a part describing the ARM CPUs
  - a part describing the peripherals

Device Tree Structure

- Device tree for Linux on Zynq mostly consists of:
  - a part describing the ARM CPUs
  - a part describing the peripherals
- `cpus`: describes the two ARM cores (which clock is used, frequency CPU supports in a certain voltage domain)

Device Tree Structure

- Device tree for Linux on Zynq mostly consists of:
  - a part describing the ARM CPUs
  - a part describing the peripherals
- `cpus`: describes the two ARM cores (which clock is used, frequency CPU supports in a certain voltage domain)
- Peripherals: LEDs, Switches, ...

Device Tree Structure

- Device tree for Linux on Zynq mostly consists of:
  - a part describing the ARM CPUs
  - a part describing the peripherals
- `cpus`: describes the two ARM cores (which clock is used, frequency CPU supports in a certain voltage domain)
- Peripherals: LEDs, Switches, ...
- `compatible` string: link between hardware and driver
  - Device drivers contain same string in their source code
  - Allows to match hardware and driver

Matching drivers and hardware

In the device tree:

```
myLed_0: myLed@43c30000 {
   compatible = "xlnx,myLed-1.0";
   reg = <0x43c30000 0x10000>;// reg = <address length> =
                              // address range used by device
};
```

Matching drivers and hardware

In the device tree:

```
myLed_0: myLed@43c30000 {
  compatible = "xlnx,myLed-1.0";
  reg = <0x43c30000 0x10000>;// reg = <address length> =
                             // address range used by device
};
```

In the driver's source code:

```
static const struct of_device_id led_of_match[] = {
  {.compatible = "xlnx,myLed-1.0"},
  {},               // Null termination
};
MODULE_DEVICE_TABLE(of, led_of_match);
```

In the userspace program:

```c
#define LED_ADDR ...
//...
char* led_ctrl = (char*)LED_ADDR;
*led_ctrl = 0x12;
```

In the userspace program:

```c
#define LED_ADDR ...
//...
char* led_ctrl = (char*)LED_ADDR;
*led_ctrl = 0x12;
```

In hardware (source of IP core):

```verilog
assign led[0] = slv_reg0[0] ==  1? 1: 0;
```

Device tree generation

- Creating device tree manually is very cumbersome.
- Therefore: Xilinx Device Tree Generator
- Install the DT Generator (in SDK):
  - Clone `https://github.com/Xilinx/device-tree-xlnx`
  - Xilinx - Software Repositories - New Local Repository ...
- Use it:
  - Xilinx - Generate Device Tree
  - Specify `.xsa` file and output directory
- The resulting `dts` and `dtsi` files should be used to replace the ones in `<BUILDROOT>/../zybo-buildroot/board/zynq_zybo/DTS`

Part 2d
Linux Device Drivers

Kernel Modules

Kernel Modules

- Extend the kernel's functionality during runtime
  - Can be loaded during runtime on demand

Kernel Modules

- Extend the kernel’s functionality during runtime
  - Can be loaded during runtime on demand
  - No need to reboot the system

Kernel Modules

- Extend the kernel's functionality during runtime
  - Can be loaded during runtime on demand
  - No need to reboot the system
  - Without kernel modules: include functionality into the kernel image before building

Kernel Modules

- Extend the kernel's functionality during runtime
  - Can be loaded during runtime on demand
  - No need to reboot the system
  - Without kernel modules: include functionality into the kernel image before building
- Most famous example: device drivers

Kernel Modules

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`
- Handling kernel modules
  - Using *kmod* (kernel module daemon)

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`
- Handling kernel modules
  - Using *kmod* (kernel module daemon)
  - *kmod* runs `modprobe` to load module and check dependencies

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`
- Handling kernel modules
  - Using *kmod* (kernel module daemon)
  - *kmod* runs `modprobe` to load module and check dependencies
  - Example: `modprobe test123` to load kernel module `test123`

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`
- Handling kernel modules
  - Using *kmod* (kernel module daemon)
  - *kmod* runs `modprobe` to load module and check dependencies
  - Example: `modprobe test123` to load kernel module `test123`
  - In the background: `insmod` to insert kernel module

Kernel Modules

- See what modules are already loaded: `lsmod` or `cat /proc/modules`
- Handling kernel modules
  - Using *kmod* (kernel module daemon)
  - *kmod* runs `modprobe` to load module and check dependencies
  - Example: `modprobe test123` to load kernel module `test123`
  - In the background: `insmod` to insert kernel module
  - `modprobe -r` or `rmmod` to remove kernel module

Simple Example

See
https://extgit.iaik.tugraz.at/sip/tutorials/-/tree/master/hello_sip
hello_sip.c:

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

static int __init sip_init(void)
{
  printk(KERN_INFO "Hello␣SIP␣students!\n");
  return 0;
}

static void __exit sip_cleanup(void)
{
  printk(KERN_INFO "Goodbye␣SIP␣students!\n");
}
module_init(sip_init);
module_exit(sip_cleanup);
```

Simple Example

Makefile:

```
obj-m += hello_sip.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Simple Example

- Build: `make`
- Infos: `modinfo hello_sip.ko`
- Load: `insmod ./hello_sip.ko`
- Kernel log: `tail /var/log/kern.log` or `dmesg -T`
- Remove: `rmmod hello_sip`

Advanced Example

- `/proc`: one subdirectory for each process
- We use it to access internal kernel structures in general.
- See https://extgit.iaik.tugraz.at/sip/tutorials/-/tree/master/hello_proc

Device drivers

- Allow the kernel to access hardware

Device drivers

- Allow the kernel to access hardware
- Convenient if hardware = file in `/dev/` or `/proc`

Device drivers

- Allow the kernel to access hardware
- Convenient if hardware = file in `/dev/` or `/proc`
- Device driver handels communication with hardware

Device drivers

- Allow the kernel to access hardware
- Convenient if hardware = file in `/dev/` or `/proc`
- Device driver handels communication with hardware
  - Example: `/dev/media0` is connected to SD card driver

Device drivers

- Allow the kernel to access hardware
- Convenient if hardware = file in `/dev/` or `/proc`
- Device driver handels communication with hardware
  - Example: `/dev/media0` is connected to SD card driver
  - Userspace program can use `/dev/media0` without knowing about which SD card or driver is used

Device drivers

- Allow the kernel to access hardware
- Convenient if hardware = file in `/dev/` or `/proc`
- Device driver handels communication with hardware
  - Example: `/dev/media0` is connected to SD card driver
  - Userspace program can use `/dev/media0` without knowing about which SD card or driver is used
  - Writing, e.g. `echo "test"> /dev/media0`, reading, opening, closing, ... has specific functionality

Building blocks of device drivers

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device
  - Functions for open/close/read/write as needed

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device
  - Functions for open/close/read/write as needed
- Standard kernel module:
  - `__init` and `__exit` functions registered with `module_init` and `module_exit`

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device
  - Functions for open/close/read/write as needed
- Standard kernel module:
  - `__init` and `__exit` functions registered with `module_init` and `module_exit`
- Driver specific:
  - `of_device_id`: compatibility

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device
  - Functions for open/close/read/write as needed
- Standard kernel module:
  - `__init` and `__exit` functions registered with `module_init` and `module_exit`
- Driver specific:
  - `of_device_id`: compatibility
  - Inserted into the device table with `MODULE_DEVICE_TABLE`

Building blocks of device drivers

- Module documentation: `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_DESCRIPTION`
- For usage with `/proc`:
  - `file_operations`: struct which defines when reading/writing/opening/closing/... the device
  - Functions for open/close/read/write as needed
- Standard kernel module:
  - `__init` and `__exit` functions registered with `module_init` and `module_exit`
- Driver specific:
  - `of_device_id`: compatibility
  - Inserted into the device table with `MODULE_DEVICE_TABLE`
  - `platform_driver`: specifies `__init` and `__exit` for driver, registered with `module_platform_driver`

Adding a device driver for the Zybo board with Buildroot

1. Create `zybo-buildroot/package/<DRIVER_NAME>` and put the following files there:
2. `Config.in`: Info for the buildroot menu
3. `Kbuild`, `<DRIVER_NAME>.mk`: Makefile
4. `<DRIVER_NAME>.c`: device driver source
5. Enable kernel module build for buildroot by selecting (= `[*]`):
   make menuconfig - External options - `<DRIVER\_NAME>`

Putting it all together

Linux with Root File System and FPGA Bitstream

- Create device tree as shown above
- Copy all the dts and dtsi files to `<BUILDROOT>/../zybo-buildroot/board/zynq_zybo/DTS`
- `cd <BUILDROOT>`
- `make BR2_EXTERNAL=../zybo-buildroot zynq_zybo_defconfig`

Linux with Root File System and FPGA Bitstream

- Configurations can be made:
  - buildroot: `make menuconfig`
  - u-boot: `make uboot-menuconfig`
  - linux: `make linux-menuconfig`
  - busybox: `make busybox-menuconfig`
  - uclibc: `make uclibc-menuconfig`
- Run `make`

Linux with Root File System and FPGA Bitstream

- Copy to first partition of SD card:
  - `<BUILDROOT>/output/images/boot.bin`
  - `<BUILDROOT>/output/images/u-boot.img`
  - `<BUILDROOT>/output/images/uImage`
  - `<BUILDROOT>/output/images/system.dtb`
  - `<BUILDROOT>/output/images/uEnv.txt`
  - The bitstream file: `system_wrapper.bit`
- Create the root file system on the second partition:
- `sudo tar -C <MOUNTPOINT> -xf <BUILDROOT>/output/images/rootfs.tar`

References I

[1] Thomas Petazzoni. *Buildroot: a deep dive into the core*. https://events.static.linuxfound.org/sites/events/files/slides/petazzoni-dive-into-buildroot-core.pdf. Online; accessed 13 October 2020.

[2] Nathan Willis. *Deciding between Buildroot and Yocto*. https://lwn.net/Articles/682540/. Online; accessed 13 October 2020.

[3] Ori Idan Helicon technologies. *Linux Bootloaders for Embedded systems and PCs*. https://www.cs.tau.ac.il/telux/lin-club_files/linux-boot/slide0000.htm. Online; accessed 13 October 2020.

References II

[4] Thomas Petazzoni. *Device Tree for Dummies*. https://events.static.linuxfound.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf. Online; accessed 13 October 2020.

[5] Xillybus Ltd. *A Tutorial on the Device Tree (Zynq)*. http://xillybus.com/tutorials/device-tree-zynq-1. Online; accessed 13 October 2020.

[6] Peter Jay Salzman. *The Linux Kernel Module Programming Guide*. https://tldp.org/LDP/lkmpg/2.6/html/index.html. Online; accessed 13 October 2020.