Embedded Linux on Zynq UltraScale+ MPSoC

Supplement

Tool : Xilinx Vivado & Petalinux 2019.2

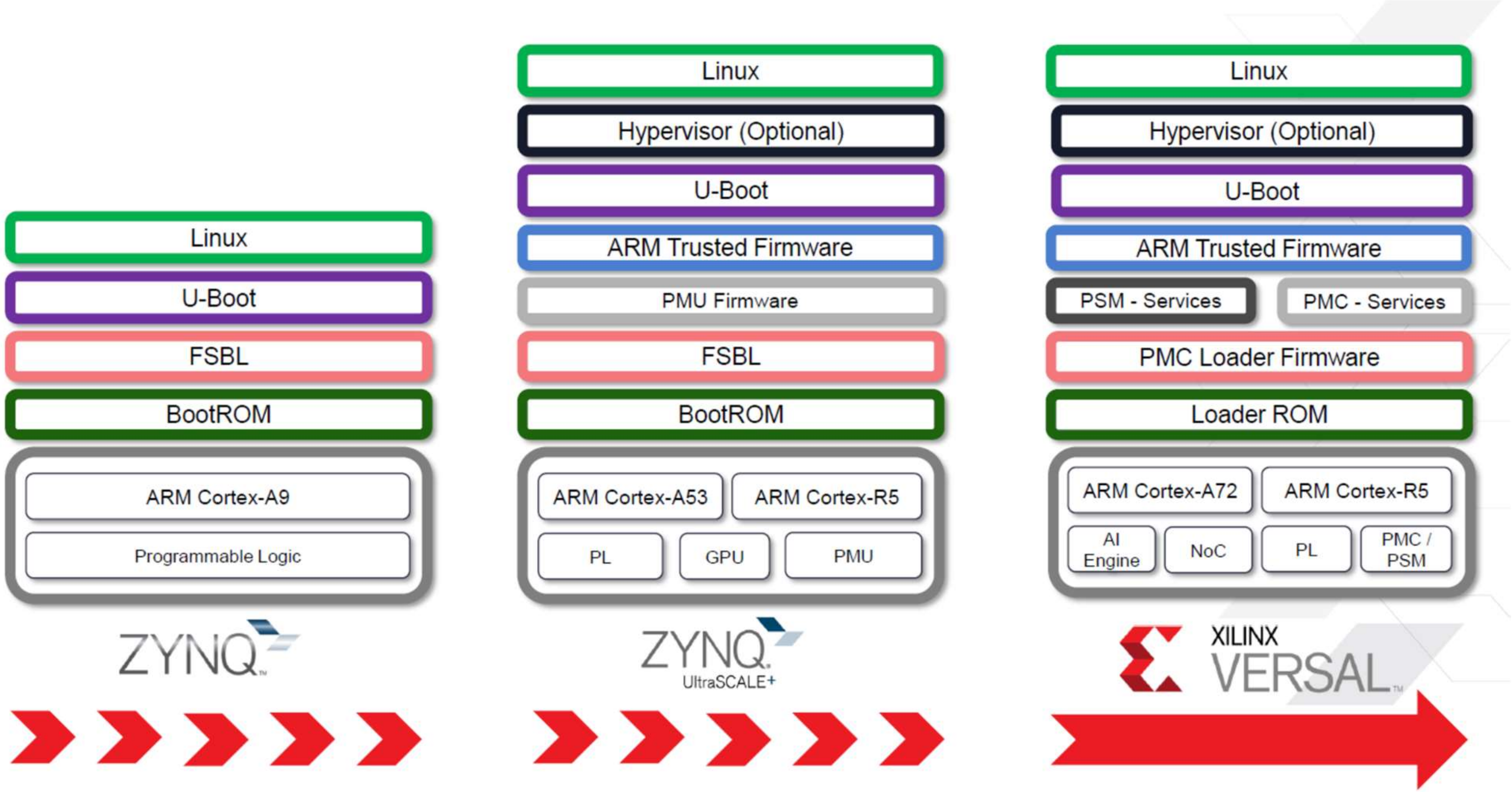
Kit : Ultra96 Training Kit

What is Petalinux?

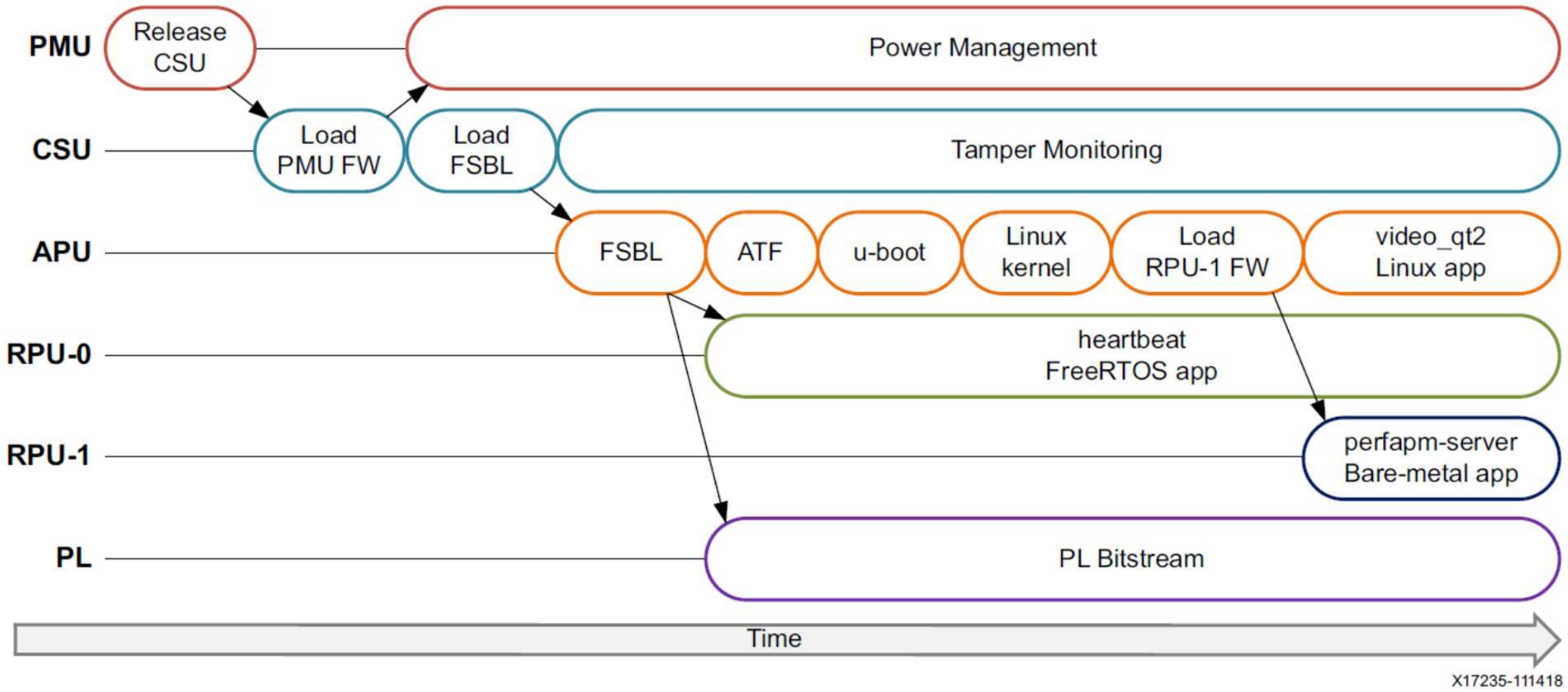
- Embedded Linux System Builder for Xilinx Boards
  - A set of open-source cross-platform development command line and menu tools for use running under an x86 Linux Host OS
  - Just six Petalinux commands (with many options):

    petalinux-create, petalinux-config, petalinux-build, petalinux-package, petalinux-boot, petalinux-util
  - Petalinux tools now use the open-source community Yocto system since v2016.4
- Has Yocto recipes for BSPs, Libraries, Applications
  - Mostly open-source based with a few exceptions for some hardware drivers
  - Petalinux includes and manages the Linux kernel sources and libraries
  - Various hardware drivers and modules
  - Conventional Linux applications and utilities
- Through Yocto, Petalinux can make use of additional embedded Linux applications beyond what Xilinx supplies

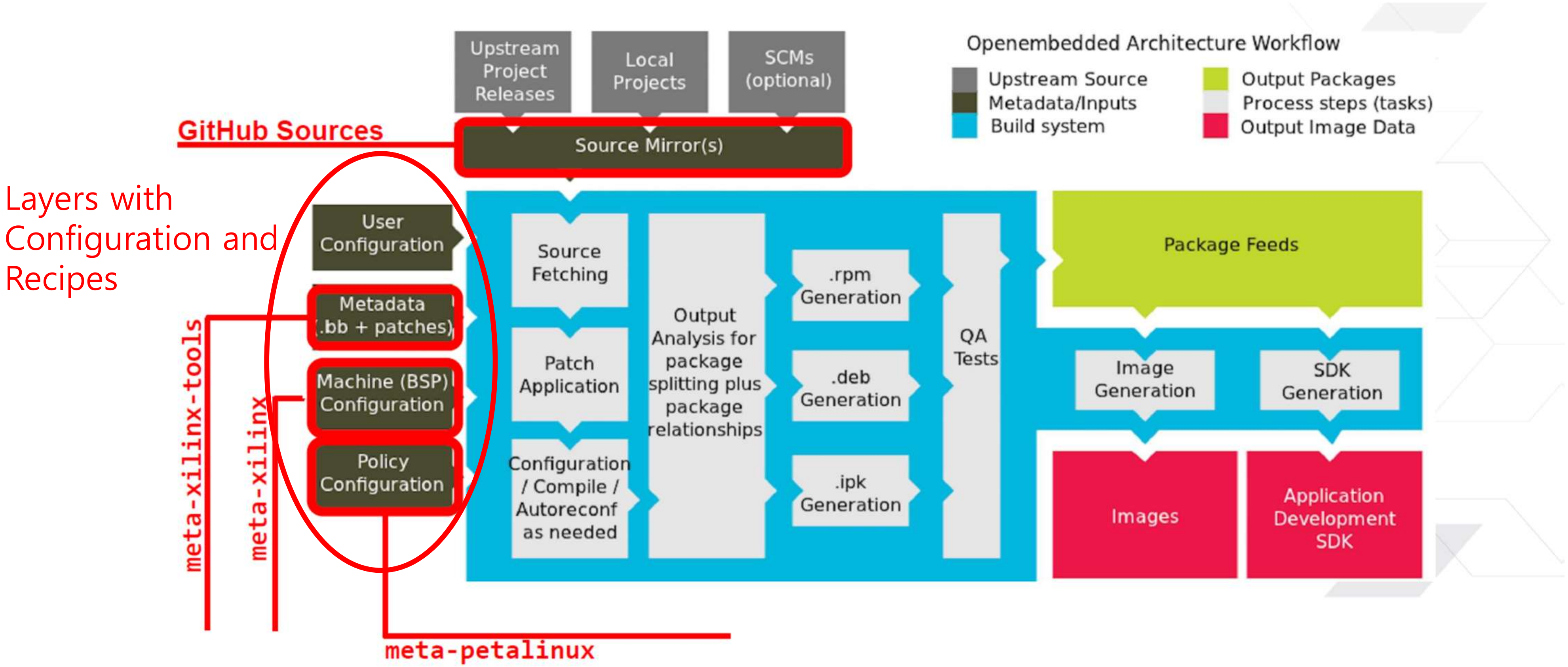
Products built by Petalinux



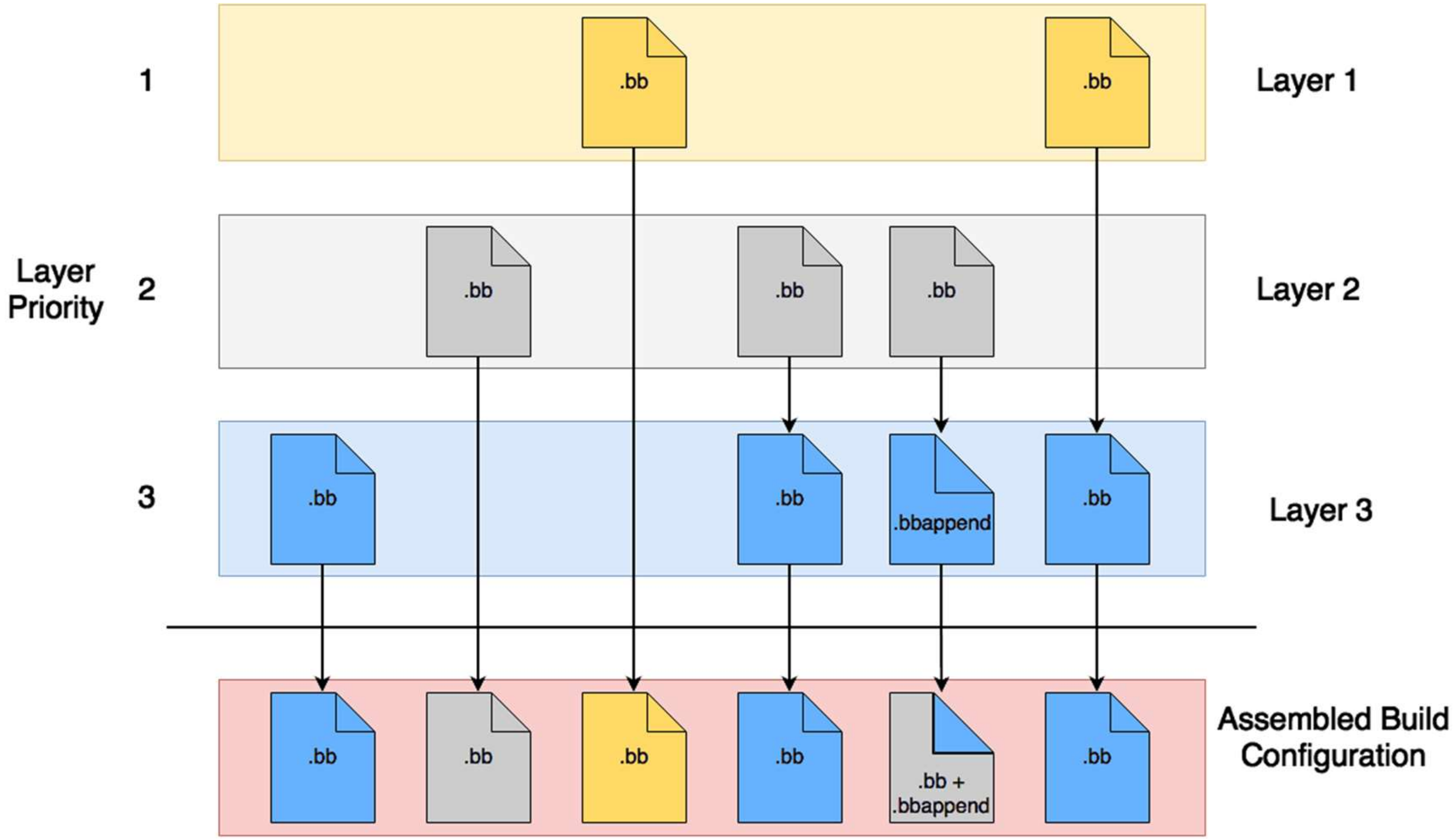
Products built by Petalinux



Petalinux Build Process



Layers with Recipes



Layers

- Petalinux build system is composed of layers
- Layers are containers for the recipes used to construct the system
- Layers are a way to manage extensions and customizations to the system
  - Layers can extend, add, replace or modify recipes
  - Layers are added via BBLAYERS variable in build/conf/bblayers.conf
  - petalinux-create generates meta-plnx-generated and meta-user layers
  - petalinux-config can add user layers(meta-inipro, meta-custom)

Layers

```
meta-inipro
├── conf
│   └── layer.conf
├── recipes-apps
│   ├── vai
│   │   ├── files
│   │   └── vai.bb
│   └── xrtutils
│       ├── files
│       └── xrtutils.bb
├── recipes-bsp
│   ├── u-boot
│   │   ├── u-boot-xlnx
│   │   └── u-boot-xlnx_2019.2.bbappend
│   ├── ultra96-misc
│   │   ├── files
│   │   └── ultra96-misc.bb
│   └── wilc-firmware
│       ├── wilc-firmware_15.00.bb
│       ├── wilc-firmware_15.01.bb
│       └── wilc-firmware_15.2.bb
├── recipes-connectivity
│   └── wpa-supplicant
│       ├── files
│       └── wpa-supplicant_%.bbappend
```

bblayers.conf

```
LCONF_VERSION = "7"

BBPATH = "${TOPDIR}"
SDKBASEMETAPATH = "/media/hokim/data/petalinux/2019.2/components/yocto/source/aarch64"
BBLAYERS := " \
  ${SDKBASEMETAPATH}/layers/core/meta \
  ${SDKBASEMETAPATH}/layers/core/meta-poky \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-perl \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-python \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-filesystems \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-gnome \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-multimedia \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-networking \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-webserver \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-xfce \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-initramfs \
  ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-oe \
  ${SDKBASEMETAPATH}/layers/meta-browser \
  ${SDKBASEMETAPATH}/layers/meta-qt5 \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-bsp \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-pynq \
  ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-contrib \
  ${SDKBASEMETAPATH}/layers/meta-xilinx-tools \
  ${SDKBASEMETAPATH}/layers/meta-petalinux \
  ${SDKBASEMETAPATH}/layers/meta-virtualization \
  ${SDKBASEMETAPATH}/layers/meta-openamp \
  ${SDKBASEMETAPATH}/layers/meta-jupyter \
  ${SDKBASEMETAPATH}/workspace \
  /home/hokim/work/zynqmp_linux/petalinux/ultra96/project-spec/meta-plnx-generated \
  /home/hokim/work/zynqmp_linux/petalinux/ultra96/project-spec/meta-user \
  /home/hokim/work/zynqmp_linux/petalinux/meta-inipro \
  /home/hokim/work/zynqmp_linux/petalinux/meta-custom \
  /home/hokim/work/zynqmp_linux/petalinux/ultra96/components/plnx_workspace \
  "
```

Recipes

- Recipes for building packages
- Recipe Files
  - meta/recipes-devtools/dnf/dnf_2.7.5.bb
- Patches and Supplemental Files
  - Location : meta/recipes-devtools/dnf/dnf/
- Recipes inherit the system configuration and adjust it to describe how to build and adjust it to describe how to build and package the software
- Recipes can be extended and enhanced through append-files from other layers

Recipes

```
SUMMARY = "Package manager forked from Yum, using libsolv as a dependency resolv
er"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://PACKAGE-LICENSING;md5=bfc29916e11321be06924c4fb096fdc
c \
                   "

SRC_URI = "git://github.com/rpm-software-management/dnf.git \
           file://0001-Corretly-install-tmpfiles.d-configuration.patch \
           file://0001-Do-not-hardcode-etc-and-systemd-unit-directories.patch \
           file://0005-Do-not-prepend-installroot-to-logdir.patch \
           file://0029-Do-not-set-PYTHON_INSTALL_DIR-by-running-python.patch \
           file://0030-Run-python-scripts-using-env.patch \
           "

SRCREV = "564c44667c7014843fa6f1732621093114ec59b2"
UPSTREAM_CHECK_GITTAGREGEX = "(?P<pver>\d+(\.\d+)+)"

S = "${WORKDIR}/git"

inherit cmake gettext bash-completion distutils3-base systemd

DEPENDS += "libdnf librepo libcomps python3-iniparse"

# manpages generation requires http://www.sphinx-doc.org/
EXTRA_OECMAKE = " -DWITH_MAN=0 -DPYTHON_INSTALL_DIR=${PYTHON_SITEPACKAGES_DIR} -
DPYTHON_DESIRED=3"

BBCLASSEXTEND = "native nativesdk"

RDEPENDS_${PN}_class-target += " \
  python3-core \
  python3-codecs \
  python3-netclient \
  python3-email \
  python3-threading \
  python3-distutils \
  python3-logging \
  python3-fcntl \
  librepo \
```

```
  python3-shell \
  libcomps \
  libdnf \
  python3-sqlite3 \
  python3-compression \
  python3-rpm \
  python3-iniparse \
  python3-json \
  python3-curses \
  python3-misc \
  python3-gpg \
  "

RRECOMMENDS_${PN}_class-target += "gnupg"

# Create a symlink called 'dnf' as 'make install' does not do it, but
# .spec file in dnf source tree does (and then Fedora and dnf documentation
# says that dnf binary is plain 'dnf').
do_install_append() {
        lnr ${D}/${bindir}/dnf-3 ${D}/${bindir}/dnf
        lnr ${D}/${bindir}/dnf-automatic-3 ${D}/${bindir}/dnf-automatic
}

# Direct dnf-native to read rpm configuration from our sysroot, not the one it w
as compiled in
do_install_append_class-native() {
        create_wrapper ${D}/${bindir}/dnf \
                RPM_CONFIGDIR=${STAGING_LIBDIR_NATIVE}/rpm \
                RPM_NO_CHROOT_FOR_SCRIPTS=1
}

SYSTEMD_SERVICE_${PN} = "dnf-makecache.service dnf-makecache.timer \
                         dnf-automatic.service dnf-automatic.timer \
                         dnf-automatic-download.service dnf-automatic-download.t
imer \
                         dnf-automatic-install.service dnf-automatic-install.tim
er \
                         dnf-automatic-notifyonly.service dnf-automatic-notifyon
ly.timer \
"
SYSTEMD_AUTO_ENABLE ?= "disable"
```

Device Tree

- The principle of Device Tree is to separate a large part of the hardware description from the kernel sources
- Device Tree allows a single kernel image to run on different boards with the the differences being described in the device tree
- Device Tree is a tree of nodes that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board
- Each node can have a number of properties describing various properties of the devices: addresses, interrupts, clocks, etc.
- Written in a specialized language, the Device Tree source code is compiled into Device Tree Blob by the Device Tree Compiler(DTC)

Device Tree

- The DTC checks the device tree syntax but the semantics of the device tree are checked at runtime by the kernel and drivers
- At boot time, the kernel is given a compiled device tree, referred to as a Device Tree Blob, which is parsed to instantiate all the devices described in the device tree
- Some key properties in a device tree node
  - The **compatible** property is used to bind a device with a device driver
  - The **interrupts** property contains the interrupt number used by the device
  - The **reg** property contains the memory range used by the device
- There is limited documentation for the device tree bindings for each device such that driver code inspection may be necessary
  - The docs are in the kernel tree at Documentation/devicetree/bindings

Device Tree – A Simple Example

- A simple example below illustrates a node of a device
  - SomeAn AMBA bus with a GPIO that has registers mapped to 0x4120000 and is using interrupt 91

    91 – 32 = 59, where 32 is the first Shared Peripheral Interrupt
  - The device is compatible with a driver containing a matching compatible string of **"xlnx,simple"**
  - The device driver source code may be the only way to really understand what properties it is expecting from the device tree

```
ps7_axi_interconnect_0: amba@0 {
          #address-cells = <1>;
          #size-cells = <1>;
          compatible = "xlnx,ps7-axi-interconnect-1.00.a", "simple-bus";
          ranges ;
          axi_gpio_0: gpio@41200000 {
                    #gpio-cells = <2>;
                    compatible = "xlnx,simple";
                    gpio-controller ;
                    interrupt-parent = <&ps7_scugic_0>;
                    interrupts = <0 59 4>;
                    reg = <0x41200000 0x10000>;
                    xlnx,is-dual = <0x1>;
          } ;
};
```

Device Tree – Inclusion Example

ps.dtsi (included file)

```
ps7_ttc_1: ps7-ttc@0xf8002000 {
  clocks = <&clkc 6>;
  compatible = "xlnx,ps7-ttc-1.00.a";
  interrupt-parent = <&ps7_scugic_0>;
  interrupts = <0 37 4>,<0 38 4>,<0 39 4>;
  reg = <0xF8002000 0x1000>;
  status = "disabled";
} ;
```

+

system-top.dts (including file)

```
/include/ "ps.dtsi"

&ps7_ttc_1 {
  compatible = "xlnx,psttc", "generic-uio";
  status = "okay";
};
```

Note the “&” used to reference an existing node (rather than creating a new node

=

```
ps7_ttc_1: ps7-ttc@0xf8002000 {
  clocks = <&clkc 6>;
  compatible = "xlnx,psttc", "generic-uio";
  interrupt-parent = <&ps7_scugic_0>;
  interrupts = <0 37 4>, <0 38 4>, <0 39 4>;
  reg = <0xF8002000 0x1000>;
  status = "okay";
} ;
```

The result for the duplicated (red) properties is the same as the including file.

Device Tree – Simplfied Ultra96v2

```
/ {
    compatible = "avnet,ultra96-rev1", "avnet,ultra96", "xlnx,zynqmp-zcu100-revC
", "xlnx,zynqmp-zcu100", "xlnx,zynqmp";
    #address-cells = <0x2>;
    #size-cells = <0x2>;
    model = "Avnet Ultra96 Rev1";
    cpus {
        cpu@0 {
            compatible = "arm,cortex-a53", "arm,armv8";
        };
    };
    amba {
        compatible = "simple-bus";
        serial@ff010000 {
            compatible = "cdns,uart-r1p12", "xlnx,xuartps";
        };
        i2c@ff030000 {
            compatible = "cdns,i2c-r1p14", "cdns,i2c-r1p10";
            i2c-mux@75 {
                compatible = "nxp,pca9548";
                i2c@4 {
                    reg = <0x4>;
                    irps5401@13 {
                        compatible = "infineon,irps5401";
                        reg = <0x13>;
                    };
                };
            };
        };
        mmc@ff170000 {
            compatible = "xlnx,zynqmp-8.9a", "arasan,sdhci-8.9a";
            wilc_sdio@1 {
                compatible = "microchip,wilc3000";
            };
        };
    };
```

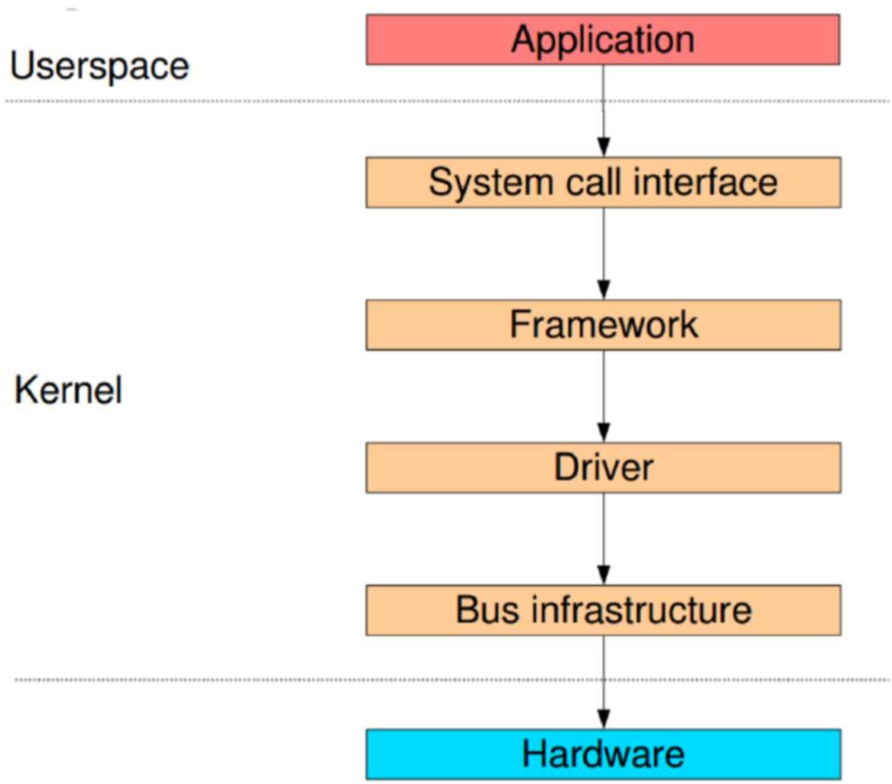
```
    amba_pl@0 {
        compatible = "simple-bus";
        gpio@80000000 {
            compatible = "xlnx,axi-gpio-2.0", "xlnx,xps-gpio-1.00.a";
        };
        i2c@80010000 {
            compatible = "xlnx,axi-iic-2.0", "xlnx,xps-iic-2.00.a";
            pmodtmp2@4b {
                compatible = "inipro,pmodtmp2";
            };
        };
        axi_quad_spi@80020000 {
            compatible = "xlnx,axi-quad-spi-3.2", "xlnx,xps-spi-2.00.a";
            pmodals@0 {
                compatible = "inipro,pmodals";
            };
        };
    };
    chosen {
        bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/m
mcblk0p2 rw rootwait uio_pdrv_genirq.of_id=xlnx,generic-uio cma=512M";
        stdout-path = "serial0:115200n8";
    };
    aliases {
        serial0 = "/amba/serial@ff010000";
    };
    memory {
        device_type = "memory";
        reg = <0x0 0x0 0x0 0x7ff00000>;
    };
};
```

Linux Kernel Frameworks

- Many device drivers are not directly implemented as character devices or block devices. They are implemented under a framework, specific to a device type (framebuffer, V4L, serial, etc.)
- The framework factors out the common parts of drivers for the same type of devices to reduce code duplication
- From userspace, many are still seen as normal character devices
- The frameworks provide a coherent userspace interface (ioctl numbering and semantics, etc.) for every type of device, regardless of the driver
  - The network framework of Linux provides a socket API such that an application can connect to a network using any network driver without knowing the details of the network driver

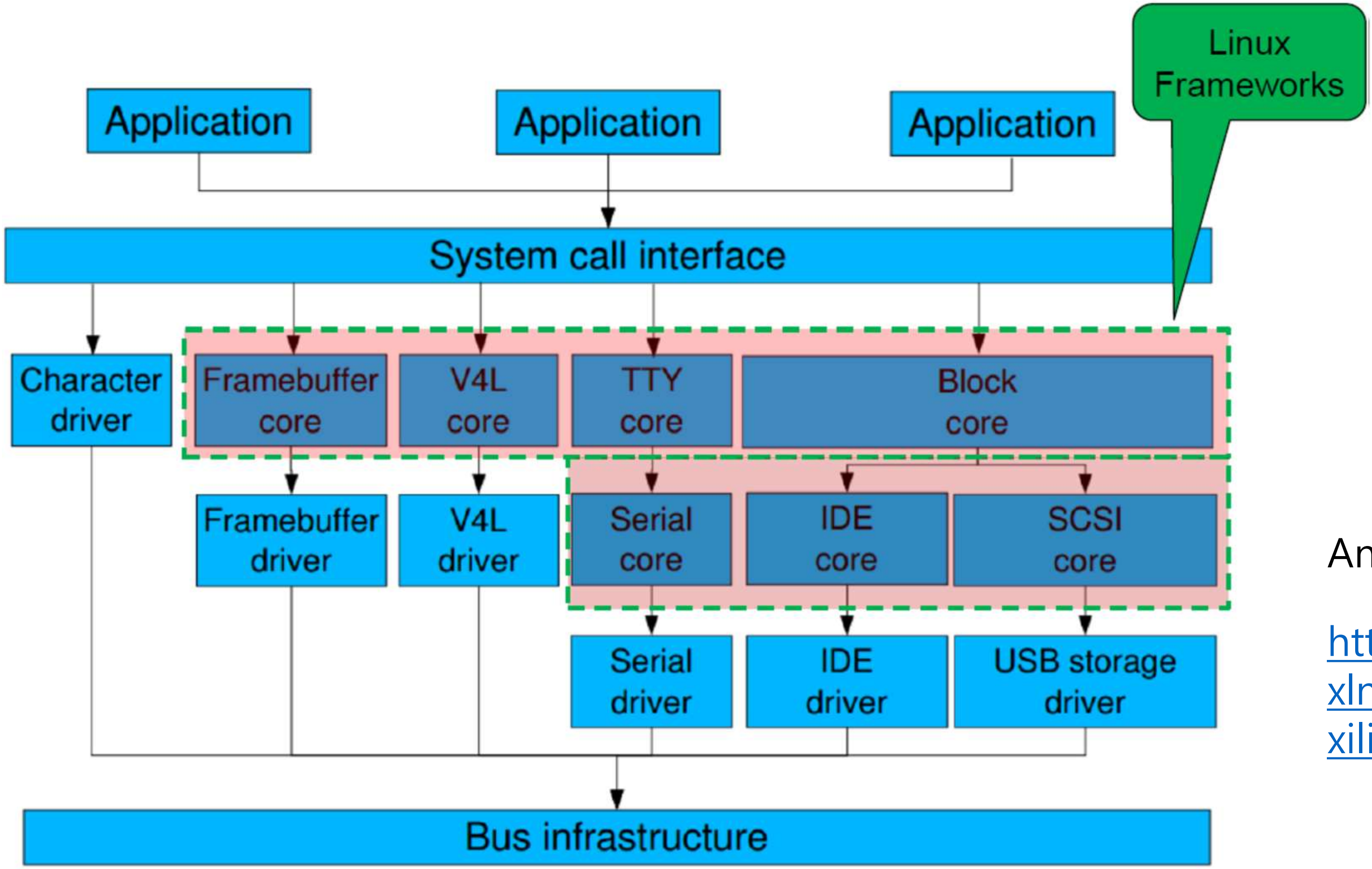
```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Linux Kernel Layers Focused on Frameworks



- A driver is always interfacing with:
  - A framework that allows the driver to expose the hardware features to userspace applications
  - A bus infrastructure(part of the device model), to detect/communicate with the hardware

Example Frameworks



An example driver of spi framework

https://github.com/Xilinx/linux-xlnx/blob/master/drivers/spi/spi-xilinx.c#L846

Virtual File Systems - Overview

- System and kernel information
  - Presented to userspace application as virtual file systems
  - Created dynamically and only exist in memory
- Two virtual filesystems most known to users
  - proc, mounted on /proc, contains operating system related information (processes, memory management parameters...)

  This is an older mechanism that became somewhat chaotic
  - sysfs, mounted on /sys, contains a representation of the system as a set of devices and buses together with information about these devices

  This is the newer mechanism and is the preferred place to add system information

Virtual File Systems - sysfs

- The **sysfs** virtual filesystem is a mechanism for the kernel to export operating details to userspace
- The kernel exports the following items to userspace
  - The bus, device, drivers, etc. structures internal to the kernel
  - **/sys/bus/** contains the list of buses
  - **/sys/devices/** contains the list of devices
  - **/sys/class/** enumerates devices by class (net, input, block...), whatever the bus they are connected to

Kernel Modules

- The Linux kernel by design is a monolithic kernel, but is also modular
- The kernel can dynamically load and unload parts of the kernel code which are referred to kernel modules
- Modules allow the kernel capabilities to be extended without modifying the rest of the code or rebooting the kernel
- A kernel module can be inserted or removed while the kernel is running
  - It can be inserted manually by a root user or from a user space script at startup
- Kernel modules help to keep the kernel size to a minimum and makes the kernel very flexible
- Kernel modules are useful to reduce boot time since time is not spent initializing devices and kernel features that are only needed later
- Once loaded, kernel modules have full control and privileges in the system such that only the root user can load and unload modules

Kernel Modules Details

❖Naming Conversion: <filename>.ko

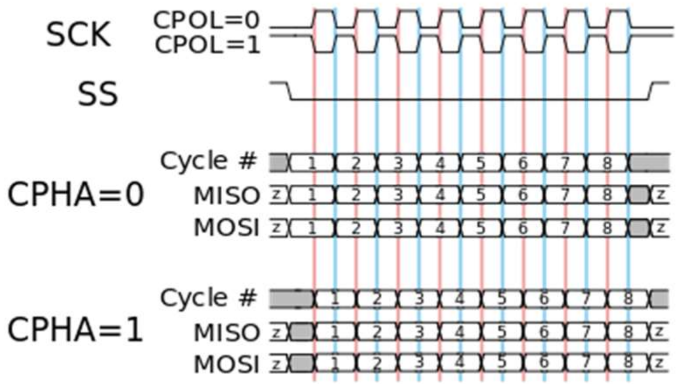
❖Location: /lib/modules/<kernel_version> on the root filesystem

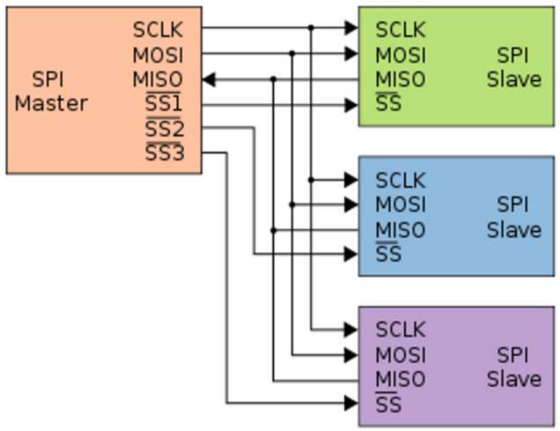
❖Device drivers can be kernel modules or statically built into the kernel image

❖Building a device driver as a module makes the development easier since it can be loaded, tested, and unloaded without rebooting the kernel

SPI

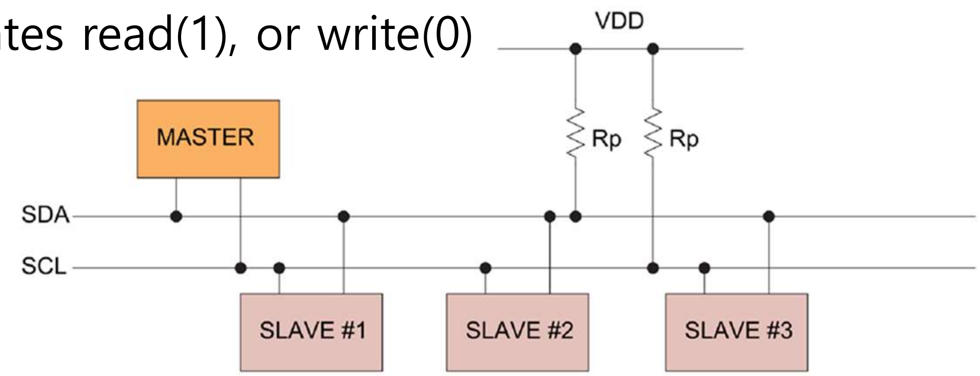
- Synchronous serial digital data link by Motorola
- SPI master controls
  - CLK: synchronization clock
  - SS or CS: slave select or chip-select; when active the slave is allowed to talk
  - MOSI: master out, slave in; carries data from master to slave
  - MISO: master in, slave out; carries data from slave to master





I2C

- Two wires are controlled by the master and slaves according to the protocol
  - SCL: Serial Clock
  - SDA: Serial DAta
- Each have has a 7 bit address
- The 7 MSB of the first transmitted byte are slave address
- The LSB of this byte indicates read(1), or write(0)



Kernel UIO API – Sys Filesystem

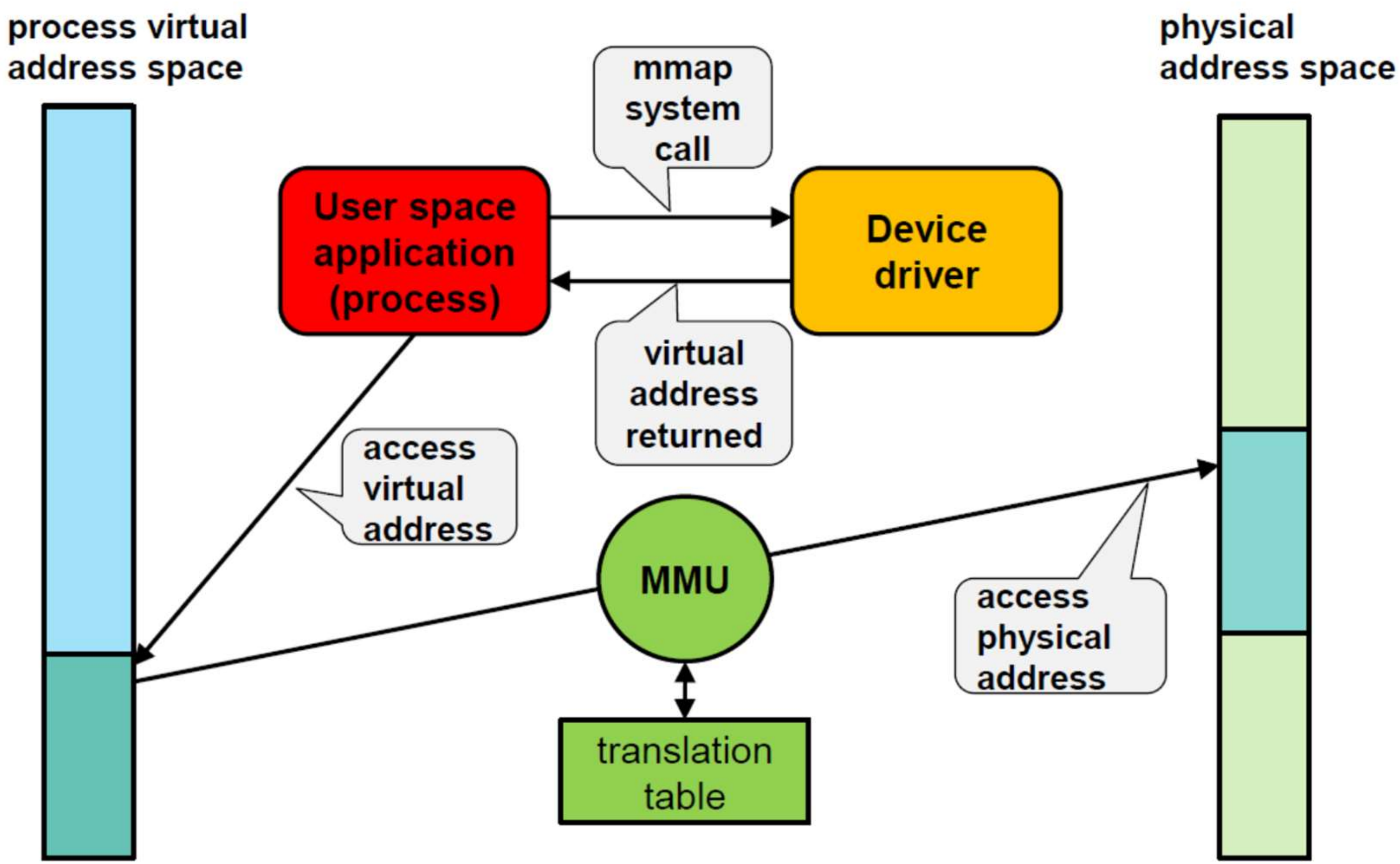
- The UIO driver in the kernel creates file attributes in the sys filesystem describing the UIO device
- /sys/class/uio is the root directory for all the file attributes
- A separate numbered directory structure is created under /sys/class/uio for each UIO device
  - First UIO device: /sys/class/uio/uio0
  - /sys/class/uio/uio0/name contains the name of the device which correlates to the name in the uio_info structure
  - /sys/class/uio/uio0/maps is a directory that has all the memory ranges for the device
  - Each numbered map directory has attributes to describe the device memory including the address, name, offset and size (/sys/class/uio/uio0/maps/map0)

User Space Driver Example

```
#define UIO_SIZE "/sys/class/uio/uio0/maps/map0/size"

int main( int argc, char **argv ) {
    int uio_fd;
    unsigned int uio_size;
    FILE *size_fp;
    void *base_address;

    uio_fd = open( "/dev/uio0", O_RDWR);

    size_fp = fopen(UIO_SIZE, O_RDONLY);
    fscanf(size_fp, "0x%08X",  &uio_size);

    base_address =  mmap(NULL, uio_size,
              PROT_READ | PROT_WRITE,
              MAP_SHARED, uio_fd, 0);

    // Access to the hardware can now occur….

    munmap(base_address, uio_size);
}
```

- ❖Open the UIO device so that it's ready to use
- ❖Get the size of the memory region from the size sysfs file attribute
- ❖Map the device registers into the process address space so they are directly accessible
- ❖Unmap the device registers to finish

Mapping Device Memory Flow

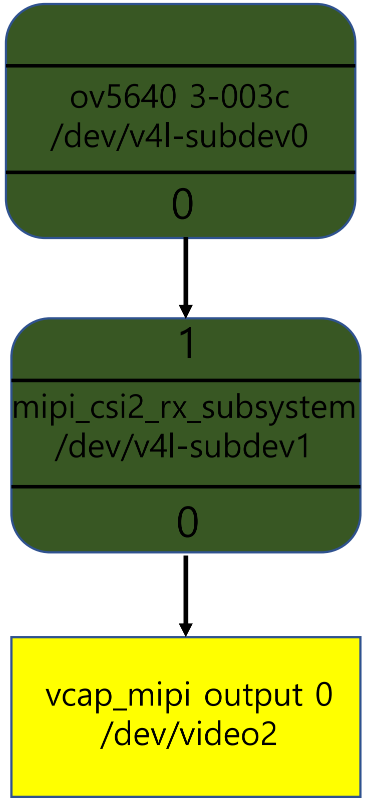


ZOCL Driver Interfaces

- A GEM style driver for Xilinx edge based accelerators
- Accelerator memory allocation is modeled as buffer objects(bo), supports both SMMU based shared virtual memory and **CMA based shared physical memory between PS and PL**. zocl also supports memory management of PL-DDRs and PL-BRAMs. PL-DDR is reserved by zocl driver via device tree. **Both PS Linux and PL logic can access PL-DDRs**
- ioctl command codes and associated structures for zocl driver

https://xilinx.github.io/XRT/master/html/zocl_ioctl.main.html

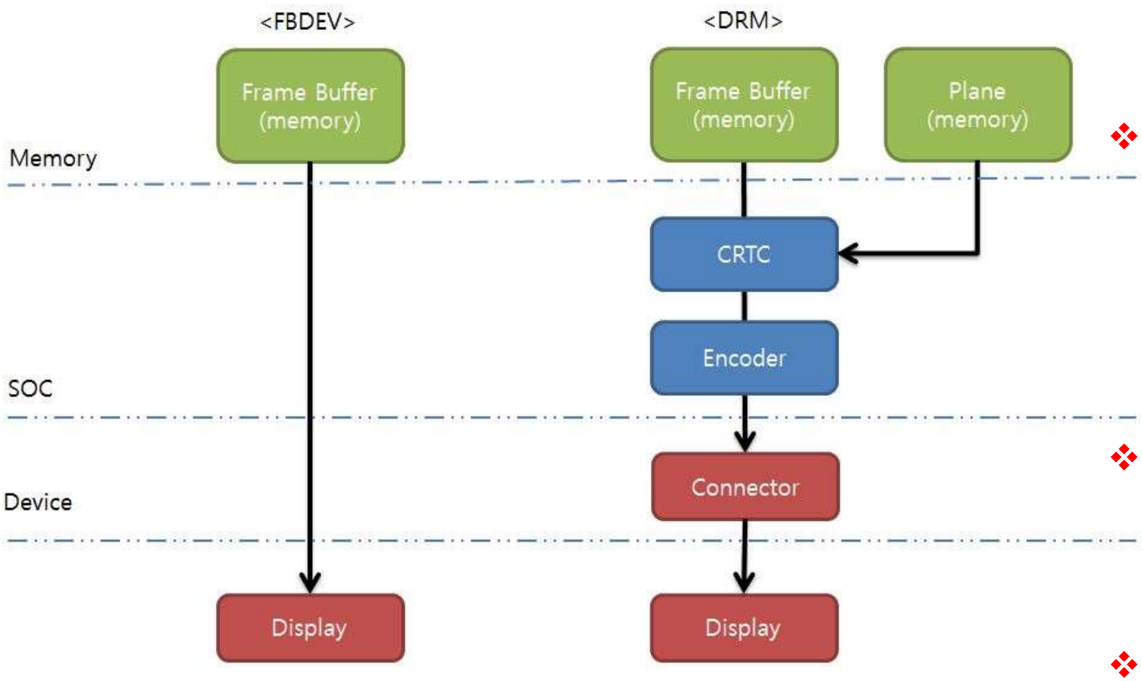
Media pipeline for camera




```
ov5640: camera@3c {
  port {
    ov5640_out: endpoint {
      remote-endpoint = <&csiss_in>;
    };
  };
};

mipi_csi2_rx_subsist_0: mipi_csi2_rx_subsystem@80000000 {
  csiss_ports: ports {
    csiss_port0: port@0 {
      csiss_out: endpoint {
        remote-endpoint = <&vcap_mipi_in>;
      };
    };
    csiss_port1: port@1 {
      csiss_in: endpoint {
        remote-endpoint = <&ov5640_out>;
      };
    };
  };
};
```

```
vcap_mipi{
  vcap_mipi_ports: ports {
    vcap_mipi_port: port@0 {
      vcap_mipi_in: endpoint {
        remote-endpoint = <&csiss_out>;
      };
    };
  };
};
```

DRM Components



- Plane
  - An image layer
  - The final image displayed by the CRTC is the composition of one or several planes
- CRTC
  - Mode information: resolution, depth, polarity, porch, refresh rate, and so on
  - Information of the buffer region displayed
  - Change current framebuffer to new one
- Encoder
  - Take the digital bit-stream from the CRTC
  - Convert to the appropriate analog levels
- Connector
  - Provide the appropriate physical plugs such as HDMI, VGA, DP and so on