What is Device Tree?

- Device Tree is a data structure for describing the hardware.This is where specific information about the hardware is conveyed. It can be used to avoid hard coding of every detail of a device into an Operating System. The device tree is passed to the kernel at boot time

- The Device Tree is where we inform the kernel about a specific piece of hardware that we have added or removed, so that the kernel can select the appropriate driver to handle it

- ".dts" & ".dtsi" files are similar to an XML file which contains data

- .dtsi file acts like a header file and you need to include this in your .dts file if you have added some data in dtsi file which needs to be used by the dts file.

- Recent Linux Kernels doesn't directly Knows about hardware. They have something called device tree source file(.DTS) where in devices are described and pin muxing is done

- DTS file can be compiled by device tree compiler(DTC). DTC generates .DTB which is device tree blob. DTB file is then put into boot partition so that Kernel can come to Know about connected devices during boot time itself.

- New hardware definition can be added in DTS file and developer can compile just this DTS file. So this method doesn't require Kernel recompilation.
  Now, there will be driver files which will contain methods called of_get_property(). These methods uses the parameters defined in DTS files.

How does Uboot pass device tree information to Linux kernel?

The way bootloader works is that after doing some setup, it simply jumps onto Linux entry point. In the old versions, it had a function called TheKernel. I don't know how it is called nowadays but the idea is the same.

```
void TheKernel(char *cmdline, void* dtb);
```

The kernel is passed the command line, and a pointer to the device tree binary, and then the function gets called, simple as that.

From user point of view, these are the steps for booting:

1- set the variable $cmdline to the desired kernel command line
2- use fatload or similar command to read the kernel from the sdcard and put it to some address at the memory, let's say at the address 20000000.
3- use fatload again to read the device tree binary (dtb) to another memory address, like 21000000. (The numbers are all made up)
4- use the bootm (boot from memory) command to start the boot

```
bootm 20000000 21000000.
```

dtc can be **installed** by this command on linux:

sudo apt-get install device-tree-compiler

- ❿ you can **compile** dts or dtsi files by this command:

dtc -I dts -O dtb -o devicetree_file_name.dtb devicetree_file_name.dts

- ❿ you can **convert** dts to dtb by this command:

dtc -I dts -O dtb -f devicetree_file_name.dts -o devicetree_file_name.dtb

- ❿ you can **convert** dtb to dts by this command:

dtc -I dtb -O dts -f devicetree_file_name.dtb -o devicetree_file_name.dts

.dts <- files for board-level definitions

1. .dtsi <- files for included files, generally containing SoC-level definitions (the **i** in dts**i** stands for **Include**)

Build DTC

All commands have to be executed in your DTC source directory. There is a single dtc binary suitable for all architectures. There is not a need to build separate dtc binaries to support MicroBlaze and ARM.

To build dtc:
make

After the build process completes the dtc binary is created within the current directory. It is neccessary to make the path to the dtc binary accessible to tools (eg, the U-Boot build process). To make dtc available in other steps, it is recommended to add the tools directory to your $PATH variable.

U-boot build steps

All commands have to be executed in your u-boot source directory.

- ❿ Clone u-boot git clone https://github.com/Xilinx/u-boot-xlnx.git
- ❿ cd to u-boot directory "cd u-boot-xlnx".
- ❿ Add tool chain to path and then set tool chain as below(tool chain name may vary based on toll chain version).

  Zynq:

  export CROSS_COMPILE=arm-linux-gnueabihf-

  export ARCH=arm

  ZynqUS+:

  export CROSS_COMPILE=aarch64-linux-gnu-

export ARCH=aarch64

Microblaze:

export CROSS_COMPILE=microblazeel-xilinx-linux-gnu-

export ARCH=microblazeel

As an example to build U-Boot for ZC702 execute:
make distclean
make zynq_zc702_defconfig
make

As an example to buildU-Boot for ZCU102 execute:
make distclean
make xilinx_zynqmp_zcu102_rev1_0_defconfig
make

After the build process completes the target u-boot elf-file is created in the top level source directory, named 'u-boot/u-boot.elf'. Additionally in the tools/ directory the 'mkimage' utility is created, which is used in other tasks to wrap images into u-boot format.
To make mkimage available in other steps, it is recommended to add the tools directory to your $PATH.
cd tools
export PATH=`pwd`:$PATH

What is device tree?

Device tree or simply called DT is a data structure that describes the hardware. This describes the hardware which is readable by an operating system like Linux so that it doesn't need to hard code details of the machine.
Linux uses the DT basically for platform identification, runtime configuration like bootargs and the device node population.

Device tree basics

Each driver or a module in the device tree is defined by the node and all its propertities are defined under that node. Based on the driver it can have child nodes or parent node.
For example a device connected by spi bus will have spi bus controller as its parent node and that device will be one of the child node of spi node. Root node is the parent for all the nodes.

Under the root node typically consists of
1) CPUs node information
2) Memory information
3) Chosen can have configuration data like the kernel parameters string and the location of an initrd image
4) Aliases
5) Nodes which define the buses information

Build Linux Kernel

Environment Variables Required

- "CROSS_COMPILE" for gcc cross platform compile settings
- "PATH" for the make procedure being able to find the cross platform compiler tools

Generic instructions for building the kernel

First, configure the Linux kernel:

make ARCH=<architecture> <kernel config>

Build the Linux kernel (requires that the environment is set up for cross compilation):

make ARCH=<architecture> UIMAGE_LOADADDR=<kernel load address> <make targets>

This will create the Linux images used for boot in linux-xlnx/arch/<architecture>/boot/, as well as linux-xlnx/vmlinux.

Linux for Zynq AP SoC

The kernel is configured based on linux-xlnx/arch/arm/configs/xilinx_zynq_defconfig:

```
make ARCH=arm xilinx_zynq_defconfig
make ARCH=arm menuconfig
```

To produce the kernel image:

```
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

In the process, linux-xlnx/arch/arm/boot/Image and linux-xlnx/arch/arm/boot/zImage are created. The Image file is the uncompressed kernel image and the zImage file is a compressed kernel image which will uncompress itself when it starts.
If the mkimage utility is available in the build environment, linux-xlnx/arch/arm/boot/uImage will be created by wrapping zImage with a U-Boot header.

Linux for Zynq UltraScale+ MPSoC

The kernel is configured based on linux-xlnx/arch/arm64/configs/xilinx_zynqmp_defconfig:

```
make ARCH=arm64 xilinx_zynqmp_defconfig
make ARCH=arm64 menuconfig
```

To produce the kernel image:

```
make ARCH=arm64
```

In the process, linux-xlnx/arch/arm64/boot/Image is created. The Image file is the uncompressed kernel image.

Linux for MicroBlaze

The kernel is configured based on linux-xlnx/arch/microblaze/configs/mmu_defconfig:

```
make ARCH=microblaze mmu_defconfig
```

How do I read an ELF file in Linux?

For this readelf and objdump command can be used .
How to use readelf for dumping contents of any section :-
readelf -x section_number elffile

How to use objdump for dumping contents of any section
objdump -s -j section_name elffile

ELF is short for **Executable and Linkable Format**. In other words, it describes how it can be used for binaries and libraries on Linux and other Unix-based systems.

It's a binary format. What this means is that when an OS tries to load a program into memory, it has to examine the program's binary (machine code) for what should be put where, like constants, program code, and such. ELF defines one such way of organizing the program data so the OS can load it into memory, find the 'main' function, and run it.

What is ATF?

ATF acts as a proxy to modify system-critical settings on behalf of the operating system running

Which is the first file in the Linux source code to get executed?

ARM processors. arch/arm/kernel/head.S is the file which get executed first in linux kernel.

After loading on to kernel image to memory, the boot loader will populate r0, r1 and r2 registers with zero, machine-id & atags list pointer before branching on to kernel.

- Usually the kernel image will be a compressed zImage. this needs to uncompressed.

- So during booting, the first piece of code to be executed is arch/arm/boot/compressed/head.S, in which a "start" symbol is present. Bootloader will jump to this address. There might be some architecture specific assembly files also in the same path, which will get executed.

- After which the head.S code will invoke arch/arm/boot/compressed/misc.c - decompress_kernel routine to decompress the kernel image

- Once decompression is complete, the ARM specific kernel initialization code is invoked - arch/arm/kernel/head.S. This will be responsible for identifying processor and machine type. Also initializing of caches, MMU etc

- After processor specific initialisations, the generic kernel startup code is invoked - start_kernel (in init/main.c), kicking off rest of the boot process.

Linux Boot sequence on ARM CPU

Bootloader preparations

Before jumping to kernel entry point boot loader should do at least the following:

1. Setup and initialise the RAM.

2. Initialise one serial port.

3. Detect the machine type.

4. Setup the kernel tagged list.

5. Call the kernel image.

CPU register settings

r0 = 0,

r1 = machine type number discovered in (3) above.

r2 = physical address of tagged list in system RAM, or

physical address of device tree block (dtb) in system RAM

Low level kernel init

Kernel entry point is arch/arm/kernel/head.S:stext

At this point we save values from boot loader, check that CPU is in correct state, enable low level debug if enabled. Prepare MMU and enable it. Call trace is below.

1) ./arch/arm/kernel/head.S:stext()

//For early printk example look the trace below. For stages w/o MMU you should just set omap_uart_phys as one defined in uboot config file.

./arch/arm/kernel/head-common.S:__error_p() -> printascii() -> addruart_current() ->

arch/arm/mach-omap2/include/mach/debug-macro.S:addruart()

2) ./arch/arm/kernel/head.S:__enable_mmu()

3) ./arch/arm/kernel/head.S:__turn_mmu_on()

4) ./arch/arm/kernel/head-common.S:__mmap_switched()

5) init/main.c:start_kernel()

Low level debug

To enable console output as soon as possible, CONFIG_EARLY_PRINTK should be enabled.

Then correct physical and virtual address should be set up (for example see [2]). At this point kernel just writes output symbols directly to specific addresses instead of using kernel log daemon.

If UART address values are set up properly you can use assembler routines like ./arch/arm/kernel/head-common.S:printascii() from assembler code. From kernel code you can

use early_printk() routine, which actually use same assembler code to throw output symbols though console.

Single thread kernel initialization

Code is in init/main.c:kernel_start() does the following things:

1) Obtain CPU id

2) Initialize runtime locking correctness validator

3) Initialize object tracker. (initialize the hash buckets and link the static object pool objects into the poll list)

4) Set up the the initial canary (GCC stack protector support. Stack protector works by putting predefined pattern at the start of the stack frame and verifying that it hasn't been overwritten when returning from the function. The pattern is called stack canary and gcc expects it to be defined by a global variable called "__stack_chk_guard" on ARM. This unfortunately means that on SMP we cannot have a different canary value per task.

5) Initialize cgroups at system boot, and initialize any subsystems that request early init. (cgroups (control groups) is a Linux kernel feature to limit, account and isolate resource usage (CPU, memory, disk I/O, etc.) of process groups.)

==== Disable IRQ ====

6) initialize the tick control (Register the notifier with the clockevents framework)

7) Activate the first processor.

8) Initialize page address pool

9) Setup architecture

a) Setup CPU configuration and CPU initialization

b) Setup machine device tree (tags)

c) Parse early parameters

d) Initialize mem blocks

e) sets up the page tables, initialises the zone memory maps, and sets up the zero page, bad page and bad page tables.

f) Unflatten device tree

g) Store callbacks from machine description

h) Init other CPUs if necessary

i) reserves memory area given in "crashkernel=" kernel command line parameter. The memory reserved is used by a dump capture kernel when primary kernel is crashing.

j) Initialize TCM memory (Tightly-coupled Memory, memory which resides directly on the processor of a computer)

k) Early trap initialization

l) Call machine early_init routine (if exists)

10) Setup init mm owner and cpumask

11) Store command line (We need to store the untouched command line for future reference. We also need to store the touched command line since the parameter parsing is performed in place, and we should allow a component to store reference of name/value for future reference.)

12) Save nubmber of CPU IDs

13) SMP percpu area setup

14) Run arch-specific boot CPU hooks (??? SMP staff)

15) Build zone lists

16) Initialize page allocation

17) Parse early parameters (earlycon, console)

18) Parse other parameters

19) Initialize jump_labels

20) Setup log buffer

21) Initialize pid hash table

22) early initialization of vfs caches

23) Sort the kernel's built-in exception table

24) trap initialization (not implemeted for ARM)

25) Set up kernel memory allocators

26) Set up the scheduler prior starting any interrupts (such as the timer interrupt). Full topology setup happens at smp_init() time - but meanwhile we still have a functioning scheduler.

==== Disable preemption ====

27) initialize idr cache (Small id to pointer translation service.)

28) Initialize performance events core

29) Initialize RCU (Read-Copy Update mechanism for mutual exclusion)

30) Initialize radix tree

31) Early IRQ init (init some links before init_ISA_irqs())

32) IRQ init

33) Initialize priority search tree (A clever mix of heap and radix trees forms a radix priority search tree which is useful for storing intervals.)

34) Init timers

35) Init HR timers

36) Init soft IRQ

37) Initializes the clocksource and common timekeeping values

38) Set machine timer as a system one and initialize it

39) Initialize simple kernel profiler

40) Register CPUs going up/down notifiers

====Enable IRQ====

41) Late initialize of kmem cache

42) Initialize console

43) Fall with panic here if needed

44) Run lock dependency validator

45) Run locking API test suite

46) Check initrd was not overwritten (if needed)

47) Initialize page cgroup

48) Enable debug page allocation

49) Initialize debug memory objects (Called after the kmem_caches are functional to setup a dedicated cache pool, which has the SLAB_DEBUG_OBJECTS flag set. This flag prevents that the debug code is called on kmem_cache_free() for the debug tracker objects to avoid recursive calls.)

50) Initialize kmemleaks (Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector with the difference that the orphan objects are not freed but only reported via /sys/kernel/debug/kmemleak. A similar method is used by the Valgrind tool (memcheck --leak-check) to detect the memory leaks in user-space applications.)

51) Allocate per cpu pagesets and initialize them.

52) Numa policy initialization (Non Uniform Memory Access policy)

53) Run late time init if provided (Machine specific ?)

54) Initialize schedule clock

55) Calibrating delay loop

56) Initialize pid hash table

57) anon_vma_init (?)

58) initialise the credentials stuff

59) Initialize fork (Allocate space for task structures)

60) Prepare proc caches (allocate memory for fork)

61) Allocate kernel buffer

62) Initialize the key management state.

63) Initialize security framework

64) Late gdb initialization

65) Initialize VFS caches

66) Initialize signals

67) Initialize page write-back

68) Initialize proc FS (if enabled)

69) Initialize cgroups (Register cgroup filesystem and /proc file, and initialize any subsystems that didn't request early init.

70) Initialize top_cpuset and the cpuset internal file system

71) early initialization of taskstat (Export per-task statistics to userland)

72) Initialize per-task delay accounting

73) Check write buffer bugs

74) Early initialization of ACPI

75) Late initialization of SFI (Simple Firmware Interface)

76) Ftrace initialization

77) Do the rest non-__init'ed, we're now alive (Create bunch of threads and call schedule to get things moving) Code in init/main.c:rest_init() routine.

a) Create kernel init thread.

b) Create kthreadd thread

c) Prepare scheduler

==== Enable preemption ====

d) call schedule()

Late kernel initialization

kthread() thread

1) Setup a clean context for our children to inherit

2) If kthread_create_list empty just reschedule

3) Create kernel thread for every task in kthread_create_list

4) Go back to step 2

kernel_init() thread

1) Wait for kernel thread daemon initialization completion

2) Setup init permissions:

a) init can allocate pages on any node

b) init can run on any cpu

3) Prepare CPUs for smp

4) Do pre SMP init calls

5) Init lockup detector

6) Enable SMP

7) Initialize SMP support in scheduler

8) Initialize devices in init/main.c:do_basic_setup() (Ok, the machine is now initialized. None of the devices have been touched yet, but the CPU subsystem is up and running, and memory and process management works. Now we can finally start doing some real work..)

a) Finish top cpuset after cpu, node maps are initialized

b) Initialize user mode helper

c) Initialize shmem

d) Initialize drivers

e) Initialize /proc/irq handling code

f) Call all constructor functions linked into the kernel

g) usermodehelper_enable - allow new helpers to be started again

h) Do init calls

9) Open the /dev/console on the rootfs

10) check if there is an early userspace init. If yes, let it do all the work

11) Run late init (Ok, we have completed the initial bootup, and we're essentially up and running. Get rid of the initmem segments and start the user-mode stuff..)

a) finish all async __init code before freeing the memory

b) Free init memory

c) Mark readonly data as RO

d) Set system state to Running

e) Try to execute userspace init command

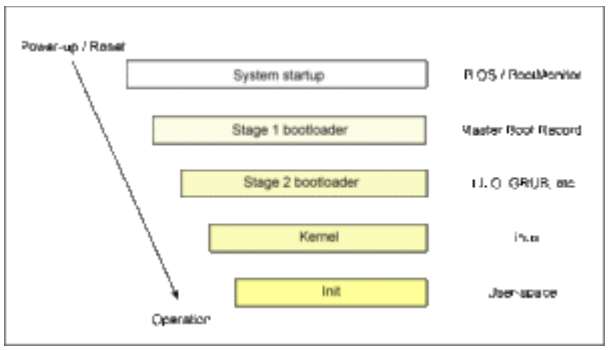
How to Cross Compile ?

arm make distclean make ARCH=arm defconfig ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make all

How A Linux Machine Boots?

The sequence that takes place during booting up a Linux machine is as.

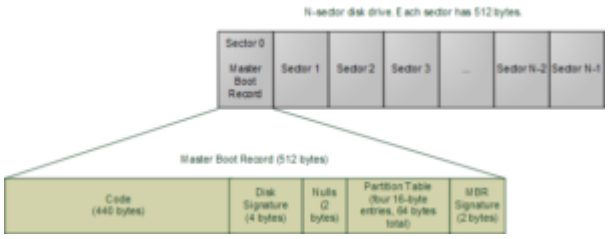
Name	Function
BIOS	Bios – Executes MBR
MBR	MBR- Executes GRUB
GRUB	Grand Unified Bootloader Executes Kernel
KERNEL	Kernel executes /sbin/init
INIT	Init executes Runlevel programme
RUNLEVEL	Runlevel programmes are executed from /etc/rc.d/rc*.d/



BIOS :- Basic Input / Output System:

- ❿ Performs some system integrity checks.
- ❿ Searches, loads, and executes the boot loader programme.
- ❿ It looks for the bootloader in floppy, cd-rom or hard drive. We have options to change the boot sequence by pressing F2
- ❿ Once the bootloader program is detected and loaded into memory, BIOS gives the control to it.
- ❿ So, BIOS loads and executes the MBR bootloader

MBR: Master Boot Record:

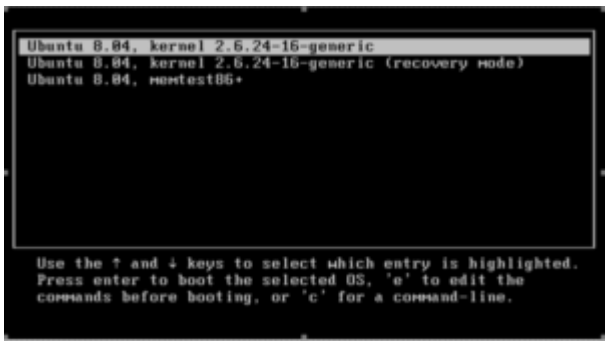


- ❿ It is located in the 1st sector of the bootable disk, typically /dev/hda or /sda.
- ❿ MBR is less than 512 bytes, this has 3 components.

1. Primary bootloader info in 1st 446 bytes
2. Partition table info in next 64 bytes.
3. MBR validates check in last 2 bytes.

It contains information about GRUB (LILO is old systems)

GRUB: Grand Unified Bootloader



- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen. Waits for a few seconds, if you don`t enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem.
- Grub configuration file is /boot/grub/grub/grub.conf (/etc/grub.conf is a link to this).
- Eg for CentOS the grub.conf is.

```
#boot=/dev/sda

default=0

timeout=5

splashimage=(hd0,0)/boot/grub/splash.xpm.gz

hiddenmenu

title CentOS (2.6.18-194.el5PAE)

          root (hd0,0)

        kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/

        initrd /boot/initrd-2.6.18-194.el5PAE.img
```

So, as shown above, both kernel and initrd image are available on grub.conf file.

KERNEL:

- Mounts the root file system as specified in the “root=” in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux kernel, it has the process ID (PID).
- Do, ps –ef | grep init and check the PID
- Initrd = Initial RAM Disk.
- Initrd is used by the kernel as temporary root file system until the kernel is booted and the root file system is mounted. It also contains necessary drivers compiled inside which helps it to access the hard drive partitions and other hardware.

INIT:

Looks at the /etc/inittab file to decide the Linux run Level.

Following are the available run levels.

- 0-Halt
- 1-Single User Mode
- 2- Multiuser without NFS.
- 3-Full multiuser mode
- 4-Unused
- 5- X11
- 6- reboot
- Init identifies the default inilevel from /etc/inittab and uses that to load all appropriate program.
- Execute “grep initdefault /etc/inittab” on your system to identify the default run level.
- Don`t set your default run level to 0 and 6.
- Set to 3 or 5.

Runlevel Programs:

- When the Linux system is booting up, you might see various services getting started, For Example, it might say “Starting sendmail .. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
- Depending on your default init level settings, the system will execute the programs from one of the following directories.

Run level 0 – /etc/rc.d/rc0.d/

Run level 1 – /etc/rc.d/rc1.d/

———-

———-

Run level 6 – /etc/rc.d/rc6.d/

Please note that there are also symbolic links available for these directories under /etc

So, /etc/rc0.d is linked to /etc/rc.d/rc0.d

- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- The program starts with “S” are used during Startup; S= Start-up.
- The program starts with “K” are used during Shutdown; K = KILL.

There are numbers right to “K” & “ S”, they are the sequence number in which the program should start or kill.

Eg :

- S12syslog is to start the syslog daemon, which has the sequence number of 12.
- S80Sendmail is to start the Sendmail daemon, which has the sequence number of 80.

So, syslog program will be started before sending mail.

What are the commonly encountered issues in development of Device Drivers?

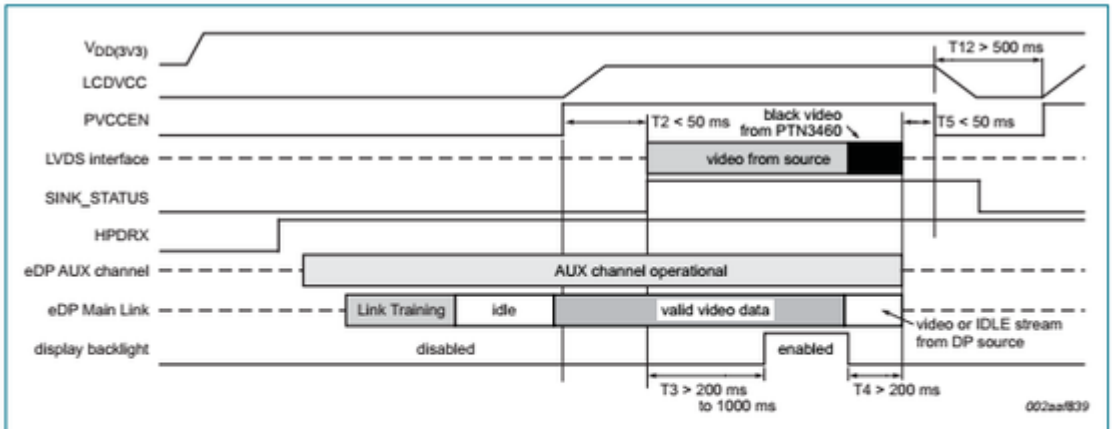
There are quite a lot of issues that can pop up especially in device driver development, few of the prominent ones are:

1. Timing /Device Initialization issues
2. Driver Lock up/Synchronization issues
3. Interrupt/GPIO handling
4. Performance Issues
5. Memory issues

I will elaborate and try to give examples for each issues

Timing /Device Initialization issues

Most of the peripheral devices have specific timing requirement on various operations such as power on/off, IO, initialization etc. For e.g: Consider this Display Port to LVDS bridge IC from NXP(http://www.nxp.com/documents/dat... - section 8.3.3)



T2: Time interval between panel power enable signal (PVCCEN) going HIGH and video data/clock driven on LVDS interface.
T3: Time interval between valid video data/clock on LVDS interface and backlight enable signal (BKLTEN) going HIGH.
T4: Time interval between backlight enable signal (BKLTEN) made LOW and stopping of video data/clock on LVDS interface.
T5: Time interval between stopping of video data/clock on LVDS interface and panel power enable signal (PVCCEN) made LOW.
T12: Time interval for which PVCCEN is held LOW before it can be made HIGH.

Fig 4. Panel power-up/power-down sequence example

This shows timing constraints on power sequence for this IC, w.r.t the display data, interfacing bus line status, power lines and so, if these constraints are not met from the corresponding driver, it may result in quite a few problem; for instance

- the IC may send invalid data to the LVDS connected display hence making it show some garbage screen - momentarily
- Or the power consumption by IC may go for a toss
- or even the IC power on/off may not happen properly causing it to enter an indeterminate state and so on

Similar timing constraint will be there on I/O Operations as well.

Another key factor would be **initialization sequence or commands**, of a particular device - certain commands should be sent to the IC to initialize its operational modes(for e.g: in case of displays, its refresh rates, resolution etc.)So the driver developer need to ensure that these **timing & initialization sequences are also met** .

Driver Lockup

- Most of the driver are makes use of threads or workqueue - which would be initiated to process some requests from userspace or events/interrupts from peripheral.
- For synchronization between each thread/work-queue some primitives such as mutex would be used.
- With poor implementation , of synchronization primitives or corner cases, this would result in lock up between threads and worst cases complete stall of the process or even panic(if watchdog or any debug system is being used)
- So while driver implementation put in effort for identifying such pain points for careful design ensuring proper synchronization (Consider points on performance issues as well) .

Interrupt/GPIO handling

- Most of external peripherals (to an SoC or processor) will have an interrupt line connected - via a GPIO interface
- The behaviour of this line would differ across peripherals. For e.g.:
    - Trigger - Level or edge triggered
    - Edge - rising or falling edge
- If the GPIO interupt controller is configured with wrong interrupt detection logic, it would result in either no interrupt being detected or spurious interrupts to be triggered.
- Another similar problem would be wrong configuration for a pin e.g.
    - configuring as an input while the intended function was to be output
    - configuring alternate function for the pin when it is expected to be a GPIO

Performance issues

As Anand Bhat explained, performance issues in driver can be addressed with proper design decisions i.e. when to use polling an interrupt modes of operation on devices

Some devices will have limited buffers (or FIFO) registers to hold data in I/O operations, so in order to properly handle I/O requests from userspace we need to carefully design data structures and associated processing in kernel drivers. For e.g:

- In case of GPU interface driver in kernel, the driver will play some role in process context management as well as per process command queue management
- This may also have associated buffer queue handling, prioritization and synchronization as well.
- So the driver need to be designed & implemented considering all the necessary points.
- If not done properly, the application through put (in case of GPU - FPS) will be affected.

Memory Issues

- Already Anand Bhat explained this in his answer - excessive usage of kernel memory APIs for granular allocation will result in fragmentation
- In case the hardware has any IOMMU for address translation in a peripheral device - take care of memory mapping and unmapping via provided API set - otherwise will result in page fault for memory operations by the designated peripheral
- When buffers are shared between userspace or kernel (such as overlays or frame buffer), proper
    - Synchronization of buffer access between kernel & userspace
    - Allocation API usage
    - IOMMU or related api usage
    - Caching parameters

There are even more issues, like:

- Peripheral lock up - firmware malfunction
- Bus hang up (if a peripheral is holding a shared bus in case of malfunction) or arbitration issue - which might have resulted because of improper request or sequence from driver
- Improper LDO or power supply configuration in terms of - voltage level or related parameters
- IO memory mapping

Typical issues encountered are:

1> handling hardware errors: If your device is interrupt driven, then there are chances of excessive spurious interrupts when device starts misbehaving. Interrupt handler should be written in such a way that spurious interrupts are either ignored or blocked(if possible). Otherwise system will spend all its time processing interrupts.

2> Memory : If you are expecting lots of operations with each operation requiring some memory to hold data structures, use slab allocator instead f kmalloc all the time. Using kmalloc excessively for different size requests will fragment kernel address space. Using slab will reduce external fragmentation to minimum.

3> Performance: Choose interrupt v/s polling wisely. If your device has DMA capabilities, it is better to switch to polling and look for all completed requests at once rather than getting interrupt for each completed request.

What are the frequently used Linux kernel functions in device driver development?

“Frequently used”? Then I suspect it has got to be “memcpy” or “printk” or something like that. Why? **This is simply because linux kernel functions is completely independent of GNU’s library functions,** and so the simplest of all functions like memcpy(), have to be rewritten by the kernel developer. This is done so that during compilation of linux kernel, there is no dependence on the standard C library at all (look out for the keyword “-nostdlib”, or just Google for it). An actual count is shown below to confirm this guess.

So let's modify it to "**frequently used important kernel functions**". ("important" is subjective as well).

Take any kernel modules and identify all the symbols within, extracting only the API that is NOT declared inside the kernel modules: nm xxx.ko | grep " U ", where "U" indicate that the function is not declared in the kernel module, thus belong to the generic class of external kernel API function.

For example, for fpga.ko:

U __class_create U class_destroy U class_find_device U dev_err U device_add U device_initialize U device_unregister U _dev_info U dev_set_name U __fentry__ U ida_destroy U ida_simple_get U ida_simple_remove U kfree U kmalloc_caches U kmem_cache_alloc_trace U module_put U __mutex_init U mutex_trylock U mutex_unlock U printk U put_device U release_firmware U request_firmware U sprintf U __stack_chk_fail U try_module_get U arch_dma_alloc_attrs U clk_disable U clk_enable U clk_prepare U clk_unprepare U complete U _cond_resched U dev_err U devm_clk_get U devm_ioremap_resource U devm_kmalloc U devm_request_threaded_irq U dma_ops U __fentry__ U fpga_mgr_register U fpga_mgr_unregister U __init_waitqueue_head U ktime_get U memcpy U __platform_driver_register U platform_driver_unregister U platform_get_irq U platform_get_resource U pv_irq_ops U regmap_write U __stack_chk_fail U syscon_regmap_lookup_by_phandle U usleep_range U wait_for_completion U warn_slowpath_null

Many of these functions come in groups, and the important ones I can list them as follows: **mutex, platform_driver, dma, memory allocation/deallocation, clocking, RCU usage, IRQ usage, waitqueue usage etc.** And the order of the usage of these different functions and its context of usage are important.

And if you do a counting of these function, some of the frequently used functions are useful to know how it worked: copy_to_user(), request_region etc.

nm *.ko | grep " U " | sort | uniq -c | sort -n (number on the left indicate the frequency count)

25 U ioport_resource 26 U platform_driver_unregister 26 U __release_region 26 U __request_region 29 U __get_user_1 31 U _copy_to_user 31 U __get_user_4 31 U misc_deregister 31 U misc_register 31 U no_llseek 31 U nonseekable_open 32 U __put_user_4 34 U param_ops_int 39 U printk 45 U param_ops_bool 53 U __fentry__

Linux Kernel: How do the probe function of Device Driver gets called?

The best way to understand the **driver -> probe()** callback has been depicted in this image. Lets consider an example of a platform device driver:

1. The starting trigger function for the driver -> probe() callback is the **module_init()** macro called while loading the driver; the macro is defined in `include/linux/module.h`.
2. **module_init(my_driver_init)** has the callback to **my_driver_init()** function. my_driver_init() function should have a call to **platform_driver_register(my_driver)**.
3. **platform_driver_register(my_driver)** assigns my_driver -> probe() handle to generic drv -> probe() and calls the **driver_register(my_driver)** function.

4. **driver_register(my_driver)** function adds my_driver to the platform bus and calls driver_attach() function.
5. In the same way, even the platform_device needs to attach to the platform bus
6. Finally, only if the **driver_match_device()** returns success based on the **.name** & **.id_table of the driver** matches in the platform devices list that comes either from ACPI/DTS, then the **driver_probe_device()** gets called that has the drv -> probe() callback.

Lets say, you are writing a driver for your platform device.

So, you need to register your driver and device with platform bus.

first, from module_init of your driver requests for driver registration,it search for device in platform device list.

Search based on driver name if your driver name matches with device name list in platfrom devices then probe functions get invoked.

1. your device register with platfrom bus with platform_device_register by passing device details as struct platform_device

struct platform_device your_device_name {

.name = "my_driver_device",

.id = -1,

.....

};

2. your driver register with platform bus with platfrom_driver_register by passing driver details as struct platform_driver

struct platform_driver your_driver_name {

.probe = my_driver_probe,

.remove = my_driver_remove,

.driver = {

.name = "my_driver_device",

.....

},

....

}

How does the ARM Interrupt Vector Table gets initialised in Linux Kernel?

ARM Linux Kernel is slightly different with interrupt initialization. Let's see how it is managed. First thing one should know is the location of the vectors itself.

- In Linux the Vectors are mapped to the higher address. 0xffff0000
- A co-processor instruction cp15 which sets the flag in the control register after reset can remap the interrupts to this address.

Which essentially means the Program Counter (PC) jumps to these addresses on FIQ, IRQ, Undefined, Aborts, SWI etc…

Obviously, these are Virtual address.
So, the physical location of the vectors has to done with the appropriate page tables for this address. Most often we like to have these mapped physically to SCRAM, TCM etc... So that ARM can fetch these vector instructions in 1 cycle.

- In Cortex Cores (ARMV7), the interrupt vectors can be re-mapped to any address.

mcr p15, #0, r0, c12, c0, #0

Important files to know:

1) arch/arm/kernel/entry-armv.S:

This file has the vectors. In 2.6.37 kernel the vectors look like this. Like any other vector table, this translates to a series of branch instructions. It is interesting to note the vector for swi instruction.

__vectors_start:

ARM(swi SYS_ERROR0)

THUMB(svc #0)

THUMB(nop)

W(b) vector_und + stubs_offset

W(ldr) pc, .LCvswi + stubs_offset

W(b) vector_pabt + stubs_offset

W(b) vector_dabt + stubs_offset

W(b) vector_addrexcptn + stubs_offset

W(b) vector_irq + stubs_offset

W(b) vector_fiq + stubs_offset

.globl __vectors_end

__vectors_end:

arch/arm/kernel/traps.c:

This file has the early_trap_init() function places the vectors and stub functions in the right location using memcpy.

memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);

memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);

memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);

It also clears the icache for this region by calling flush_icache_range().

Post this interrupts can be enabled and the vectors are in place.

How does the system know which device has generated the interruption in the case of a shared interrupt (ARM/Linux)?

An ARM CPU typically has two pins (FIQ and IRQ) that are asserted by devices when they want to generate an interrupt. As soon as this happens, the CPU simply switches modes and jumps to the address of the handler.

However, as you have asked in the question, because there are usually more devices than the number of interrupt pins, there's usually an interrupt controller that goes between the CPU and the devices. You can think of this as a hub for connecting more interrupts to the CPU. The interrupt controller can be configured to assert FIQ for certain kinds of interrupts it receives.
The interrupt handler usually asks the interrupt controller which pin caused the interrupt, then calls the appropriate handler. This typically involves reading some sort of memory mapped IO register, which depends on the exact the exact SoC.

Where you have a shared interrupt line, you typically have something outside the processor, in the case of arm, within the chip but outside the arm core itself. Something that does have many interrupt inputs and the output or outputs that go to the processors one or few interrupts. When the processors interrupt occurs then you check with this logic/hardware outside the processor/core to see who caused it, and there you go.

Also, be careful with these shared interrupt systems, it is quite possible to have multiple interrupts happen "at the same time" basically from the time your program starts stopping and the interrupt vector is called, and you do your interrupt startup stuff and eventually read the interrupt status register, more than one can come in. You need to decide how to handle that and depending on the system/logic if you were to return from one of them the others being asserted might bring you back into the interrupt handler, so that impementation would/could be handle only one of the pendings. Other logic might require you to handle all the pendings before you return.

If two interrupts are arrived at the same time, how will the compiler know which ISR is for which interrupt?

This is the exact reason why modern computers are equipped with parallel priority interrupt control.

If two interrupts arrive at the same time, the interrupt with more priority will be handled. Each interrupt has a vector address associated with it which directly addresses the respective interrupt vector (which contains the ISR address to jump to). All interrupt vectors are present in a vector table.

every interrupt has a specific iterrupt service vector associated to it, clubbed by priority. So the runtime will have no ambiguity resolving their service routines

What are the differences between hardware and software interrupts? How would a procedure written for software interrupt be different from one which is written for hardware interrupt?

A software interrupt is made by a program and its priority is usually less than a hardware interrupt. Even though software interrupts have different severity levels, they won't match the level of a hardware interrupt. The procedure for signaling interrupt would be fairly easy for software interrupts, in opcode it would be CC or INT 3. Hardware interrupts are harder to write, and usually come from the micro architecture of the hardware component itself.

To help you understand the interrupt levels, follow this:

SOFTWARE INNTERUPS (Lets keep level 5 as max interrupt level)

1. Process created, allocate memory for it
2. Process wants to display text, use console output driver
3. Process wants to edit another processes memory
4. Process wants part of the kernel(system call)
5. NX violation, process attempted to execute read/write only memory

HARDWARE INTERRUPS (Lets keep level 3 as the highest)

1. User clicked mouse, translate mouse input.
2. CD-ROM has read a bad sector of disk, must flag sector as bad.
3. CPU failed to catch a trap (this is a critical processing failure). Everything stops/halts, OS may trigger a bug check/kernel panic, computer must be reset(restarted).

What is difference between a polled versus vectored interrupts. How they are used to handle multiple interrupt?

Polling means constantly checking if any input data has entered, and if yes, then an input processing function will start to run to process the input. This is call synchronous processing.

Interrupt means whenever an input enters into the system, it will intercept the CPU's current processing, and a interrupt handler procedure executed to process the input. This is call asynchronous processing.

What are the major types of interrupts?

There are mainly three types of interrupts:

1. External interrupts: It arises due to external call from I/O devices. For e.g. I/O devices requesting transfer of data, power failure, etc.

2. Internal interrupts: It arises due to illegal and erroneous use of an instruction or data. For e.g. stack overflow, division by zero, invalid opcode, etc. These are also called traps.
3. Software interrupts: It is initiated by executing an instruction. It can be used by the programmer to initiate an interrupt at the desired point in the program.

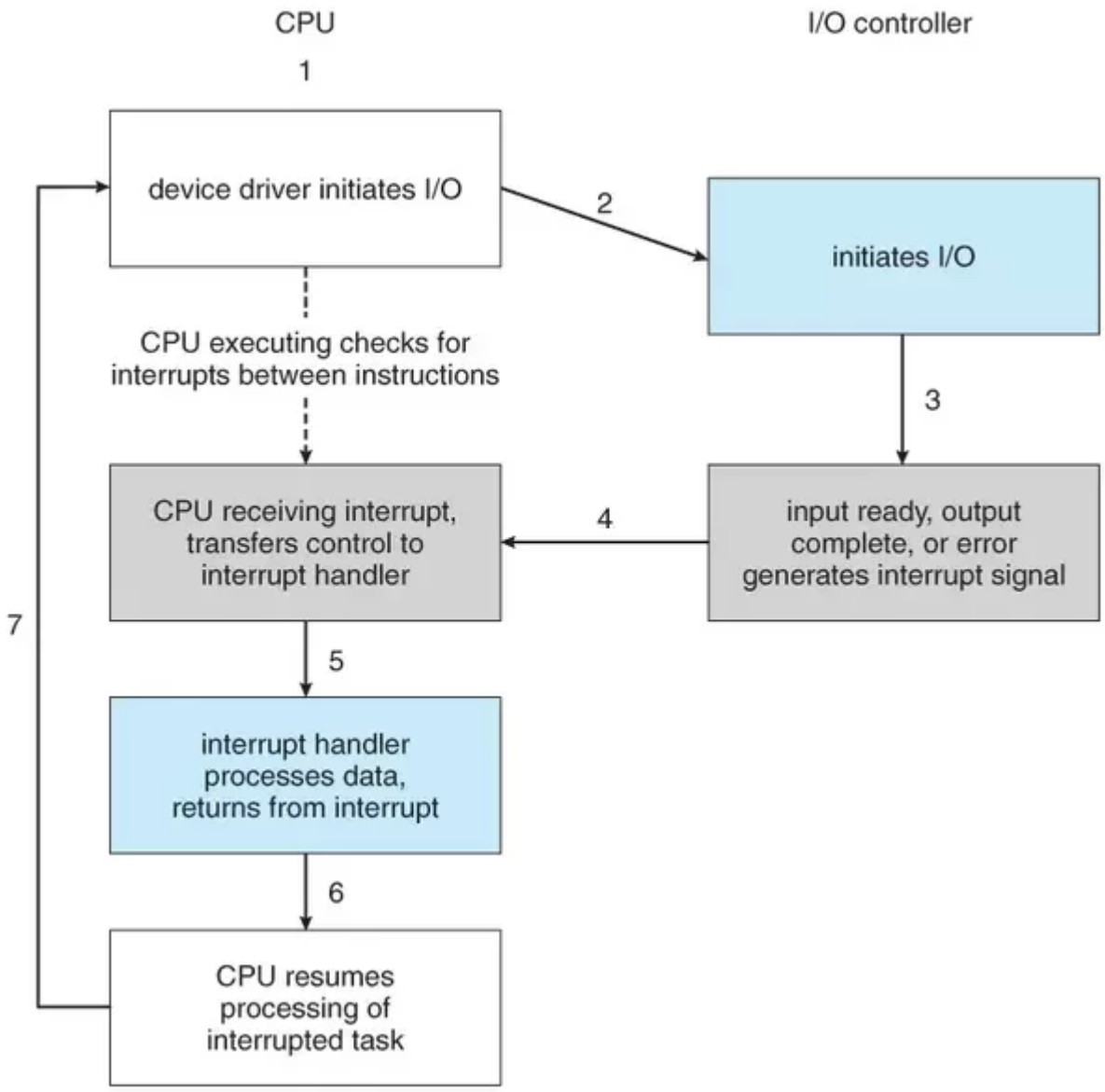
External and internal interrupts are initiated from signals that occur in the hardware of the CPU whereas Software interrupts occur from the instructions

What are "interrupts"?

In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.

There are two types of interrupts:

- **Hardware Interrupts:** These are used by devices to communicate that they require attention from the operating system.Internally, hardware interrupts are implemented using electronic alerting signals that are sent to the processor
- **Software Interrupts:** These are caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed



Each interrupt has its own **interrupt handler**. The number of hardware interrupts is limited by the number of interrupt request (IRQ) lines to the processor, but there may be hundreds of different software interrupts. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Interrupt handler: an interrupt handler, also known as an interrupt service routine or ISR, is a special block of code associated with a specific interrupt condition. Interrupt handlers are initiated by hardware interrupts, software interrupt instructions, or software exceptions, and are used for implementing device drivers or transitions between protected modes of operation, such as system calls.An interrupt handler is a low-level counterpart of event handlers

What are the interrupts and uses of interrupt?

If computers only had 1 dedicated task, there would be no need for interrupts. However, most users want their computers to do more than 1 thing. Interrupts are used to service the needs of various devices. Most devices, if unserviced, would be overrun by incoming events or data. Interrupts are used to stop a running program, service the device and continue with the running program as if nothing had happened. Interrupts are the fundamental building blocks of mult-tasking operating systems. Typically a counter periodically generates an interrupt. The operating system uses these interrupts to perform house keeping tasks like , stopping a program , starting another , etc.

So interrupts make modern operating systems possible.

What is the difference between ISR (Interrupt Service routine) and an Interrupt Vector?

An interrupt vector is stored at a known location in memory, usually in what is called the Interrupt Vector Table.

The IVT can be thought of as an array of memory addresses.

The IVT usually resides at a fixed location in RAM memory. Each entry contains the starting address of the ISR (Interrupt Service Routine) related to it's place in the “array” (IVT).

When an interrupt occurs, the interrupt hardware forces the CPU to save the current state, then jump to the address contained in the IVT related to the specific interrupt that occured. When the ISR has finished executing the CPU restores the previous machine state and normal code execution resumes.

Interrupt vector is the hardware memory space that have the addresses of the methods that must be called when the interrupt happens.

ISR is the piece of software that have the address stored at the interrupt vector and will be executed when the interruption happens.

What is vectored interrupt?

A vectored interrupt is an interrupt that returns an integer value which corresponds to the type of interrupt. This value can then be used to index a table or data structure in memory which would contain the address of the interrupt handler, which is the specific software that the system programmer designs to respond to that specific type of interrupt.

Various types of interrupts could include exceptions (ex. Divide by zero, out of bounds memory access…), a software interrupt (ie an instruction which is designed to trigger an interrupt that may include part of the vector value as part of the instruction), or an externsl hardware interrupt (a reset or bus interrupt that may also contain a part of the vector vslue on the data bus or special interrupt pins).

“Vectored interrupt” refers to a style of interruption handling in processors. This style of interruption handling happens to be quite popular.

Most processors, including those from Intel, AMD … have an interrupt vector table (IVT). This is a data structure that associates an interrupt handler to an interrupt request. In its simplest form, an interrupt vector is the address of the interrupt handler.

The interrupt vector table is used by the processor to determine the correct response to an interrupt or exception.

What is the difference between exception and interrupt in operating systems?

Interrupts and Exceptions both alter program flow. The difference being, interrupts are used to handle external events (serial ports, keyboard) and exceptions are used to handle instruction faults, (division by zero, undefined opcode).

1) Interrupts are handled by the processor after finishing the current instruction.
1) Exceptions on the other hand are divided into three kinds.

- Faults
  - Faults are detected and serviced by the processor before the faulting instructions.
- Traps
  - Traps are serviced after the instruction causing the trap. User defined interrupts go into this category and can be said to be traps.
- Aborts
  - Aborts are used only to signal severe system problems, when operation is no longer possible.

2) Interrupt is an as asynchronous event that is normally(not always) generated by hardware(Ex, I/O) not in sync with processor instruction execution.
2) Exceptions are synchronous events generated when processor detects any predefined condition while executing instructions.

Lastly, An interrupt handler may defer an exception handler, but an exception handler never defers an interrupt handler.

I would say that a **trap** is the way to do system calls/APIs (asking for OS services). This typically involve saving some registers in the process’ kernel stack, switching to kernel mode, and jump to the start of the trap handler.

Exceptions result from executing instructions (e.g., divide by 0, segmentation faults, etc). This will be handled by jumping to the right exception handler.

Interrupts result from external event (i.e., I/O devices) and they happen between instruction execution. I mean in the fetch-decode-execute cycle, a check is made to see if an interrupt has occurred after the execute step in the fetch-decode-execute cycle.

So, traps and exceptions resulted from a running program (i.e., a process). Traps are done when a process wants service from OS (e.g., to read a file), while exception happens when something wrong happened while executing an instruction. Interrupts are external to the process and can happen any time but will be checked by the CPU between instructions.

They are all handled in a similar way. They all are handled by the OS. The hardware needs to save the state of the current process (values of the processor's registers like PC, SP, etc.) somewhere (e.g., stack), switch to kernel mode, and then jump to the start of the handler (i.e., function) that deals with the trap, exception, or interrupt.

What is the difference between signals, interruptions and exceptions?

Signals and interrupts are asynchronous --- meaning, they can arrive at any point in your program.

Exceptions are synchronous --- they happen when and because your program made them happen. Sometimes "making them happen" is inadvertent (some things that start as signals --- e.g. memory-reference errors --- can get turned into exceptions by the low-level signal handler in the runtime).

Signals (which happen to user programs in user space) are modeled on interrupts (which are how the hardware signals the operating system that some hardware event has happened). Both interrupt the program at some unpredictable point, saving the program's context on the stack, and pushing a new context onto the stack. This new context looks pretty much just like a subroutine call to the programmer.

Exceptions, instead of deepening the stack, "crawl back up the stack" looking for an exception-handler of the right type. Somewhere in the program, the exception was raised. When the exception was raised, run-time software began examining the stack, following the call/return chain back up until it finds a stack-frame with an exception handler, which it then executes.

Some run-time systems will provide signal handlers for many signals which turn the signal into an exception.