* Kernel programming * (25/01/23)

⇒ Kernel source file (or) codes contains

- .c
- .h
- Make files
- Kconfig
- Scripts
- Kbuild

} only these files.

⇒ Kernel source code can be downloaded from Kernel.org.

⇒ The Kernel source code contains 80thousand + file In that, majority of files we can see is .c files Now, to compile only .c files at a time excluding other than .c files, we can use make files (or) K config files.

⇒ To compile multiple .c files together, we have to take the help of make files & make utilites.

⇒ A simple make command will generate binaries

❀ Use of [make] command.

⇒ No need to type the command ($ gcc) for recompiling with .c files.

⇒ Compilation time can be reduced.

⇒ It won't recompile the file, which are already compiled, & we can define the targets that you want to generate by compiling only selected files.

⇒ In general, a Make file contains target definitions

⇒ Make file always starts with capital M (Make file)

⇒ we can have only one (1) Make file in 1 directory.

Make file

Target : dependencies recipe (commands)

Target ⇒ a.out

dependencies ⇒ 1.c 2.c (.c files)

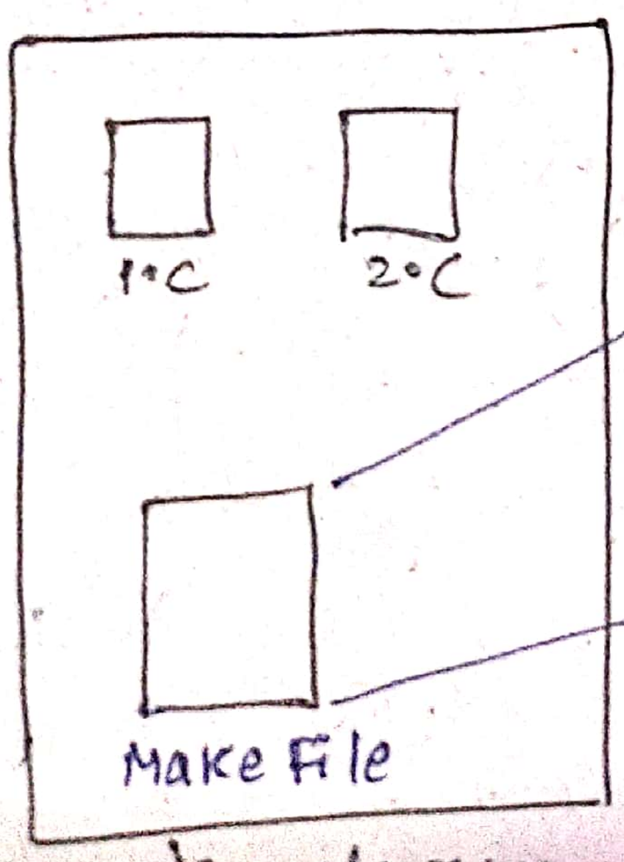
⇒ Even these dependencies can be defined as subtargets

⇒ To write the Make files we have certain rules.

① Target should followed by colon (:)

② Dependencies are separated by spaces.

③ provide <tab> space by using tab key (it will check it).



Make file

a.out : 1.c 2.c <tab> gcc 1.c 2.c (tt) ↓ (recipe)

⇒ Go to the directory where you have Make file & simply run make command $make.

① make command will search for the file (Makefile) in current working directory & process the 1st target only.

② while processing the 1st target, it will check the dependencies are exist in the current working directory or not. If they exist, it will execute the recipe (or) command.

⇒ writing the Makefile in the above Format is meaningless, To make it more effective, modify the make file by defining the sub targets (dependencies).

Make file

Main target

Sub targets

```
a.out : 1.o 2.o
<tab> gcc 1.o 2.o

1.o : 1.c
<tab> gcc -c 1.c

2.o : 2.c
<tab> gcc -c 2.c
```

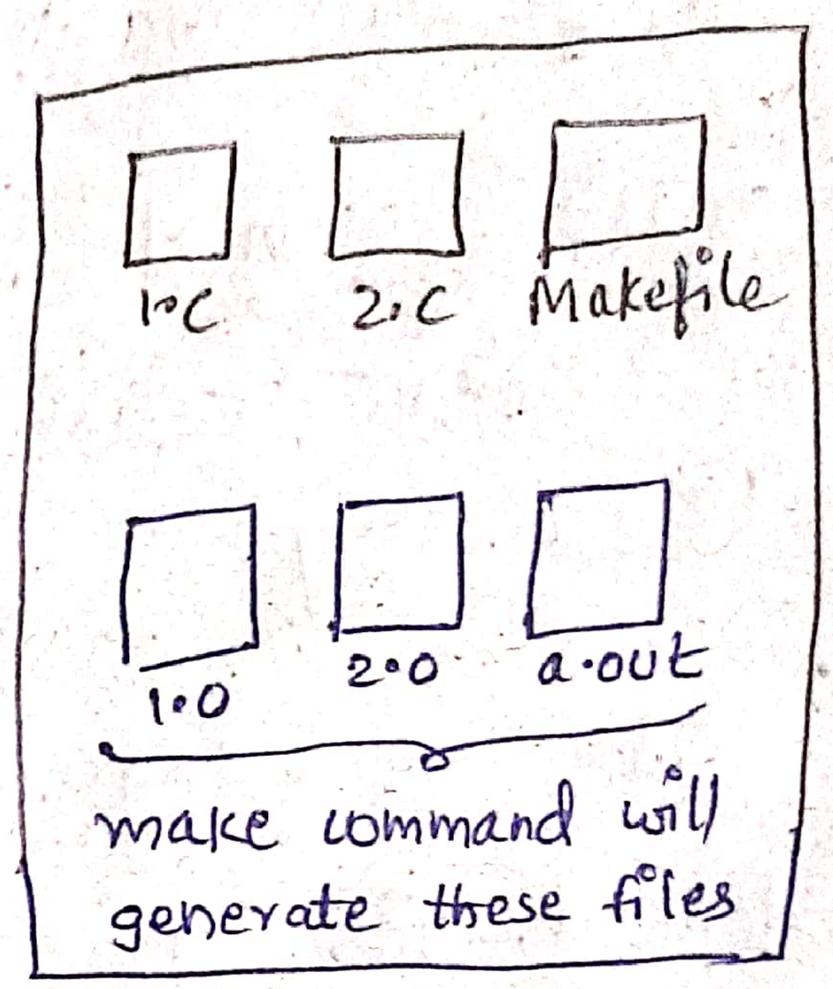
⇒ when we run make command, first it will process the subtargets after that it will process the main target.

Error msg display format

path filename line & fun : error message.

on terminal we can see after using make command

```
$make
gcc -c 1.c
gcc -c 2.c
gcc 1.o 2.o
```



⇒ we can define a target to remove the desired files.

Makefile

```
a.out : 1.o 2.o
<tab> gcc 1.o 2.o

1.o : 1.c
      gcc -c 1.c

2.o : 2.c
      gcc -c 2.c

clean:
      rm *.o a.out
```

(phoney target) → clean:

→ $ make clean

It will remove 1.o, 2.o & a.out files from CD.

⇒ In general, make command is always followed by target name which is defined in particular Make file. (To process the particular target).

⇒ If we don't mention target name, it will process the main target.

⇒ $ make <target name>

← defined in Make file.

⇒ $ make 1.o

⇒ This command will execute & generate 1.o by compiling 1.c.

⇒ It will perform gcc -c 1.c internally.

⇒ If we perform $ make 1.o again, it will not generate a.out again, if there is no modifications in any .c file.

⇒ when we run the make file 2nd time, by using make command if the targets are already exist, it won't process any recipe.

⇒ If the particular dependency file is modified, then it will process that particular targent & its dependencies target.

⇒ $ <u>make clean</u> will deletes .o & a.out files in CD

⇒ we can define variables for dependencies.

```
CC = gcc
TARGETS = 1.o 2.o

a.out : ${TARGETS}
<tab> $(CC) ${TARGETS}
```

⇒ Makefile macros are also known as variables.

1.c 2.c 3.c ⇒ a.out

$ make

$ make all → cuz there are dependencies.

Makefile

```
HEADERS = myheader.h
OBJ = 1.o 2.o 3.o
CC = gcc
CFLAGS = -c

a.out = $(OBJ)
<tab> $(CC) $(OBJ)

1.o : 1.c // $(HEADERS)
      $(CC) $(CFLAGS) 1.c $(HEADERS)
2.o : 2.c // $(HEADERS)
      $(CC) $(CFLAGS) 2.c $(HEADERS)
3.o : 3.c // $(HEADERS)
      $(CC) $(CFLAGS) 3.c $(HEADERS)
```

Makefile

```
HEADERS = myheader.h
OBJ = 1.o 2.o 3.o (or) $(wildcard *.o)
SRC = 1.c 2.c 3.c (or) "
CC = gcc
CFLAGS = -c

all : objs a.out

objs : $(SRC)
       $(CC) $(CFLAGS) $(SRC) $(HEADERS)

a.out : $(OBJ)
        $(CC) $(OBJ)
```

Makefile

```
CC = gcc
SRC = $(wildcard *.c)
OBJS = $(SRC : .c = .o)

a.out : $(OBJ)
        $(CC) -o $@ $^
```

all object files.

Make file

```
CC = gcc
OBJS = 1.o 2.o 3.o

%.o : %.c
<tab> $(CC) -c -o $@ $<

a.out : $(OBJS)
<tab> $(CC) -o $@ $^
```

dependency on .c files

used to generate .o files in (CWD)

Equivalent to

$gcc -c -o 1.o 1.c

⇒ compiling 1.c file to generate .o file with 1.o name.

30/01/23

* Linux Kernel :-

⇒ It is the main core component of the system which requires libraies and applications to provide features to end users.

⇒ Linux was created as a hobby by a ~~finish~~ finnish student Linus toorvald's

Features of Linux kernel :-

⇒ Linux kernel is portable & it has huge hardware support cuz it ~~suns~~ supports most of the architectures.

⇒ The linux will run on super computers as well as tiny devices this shows its scalability.

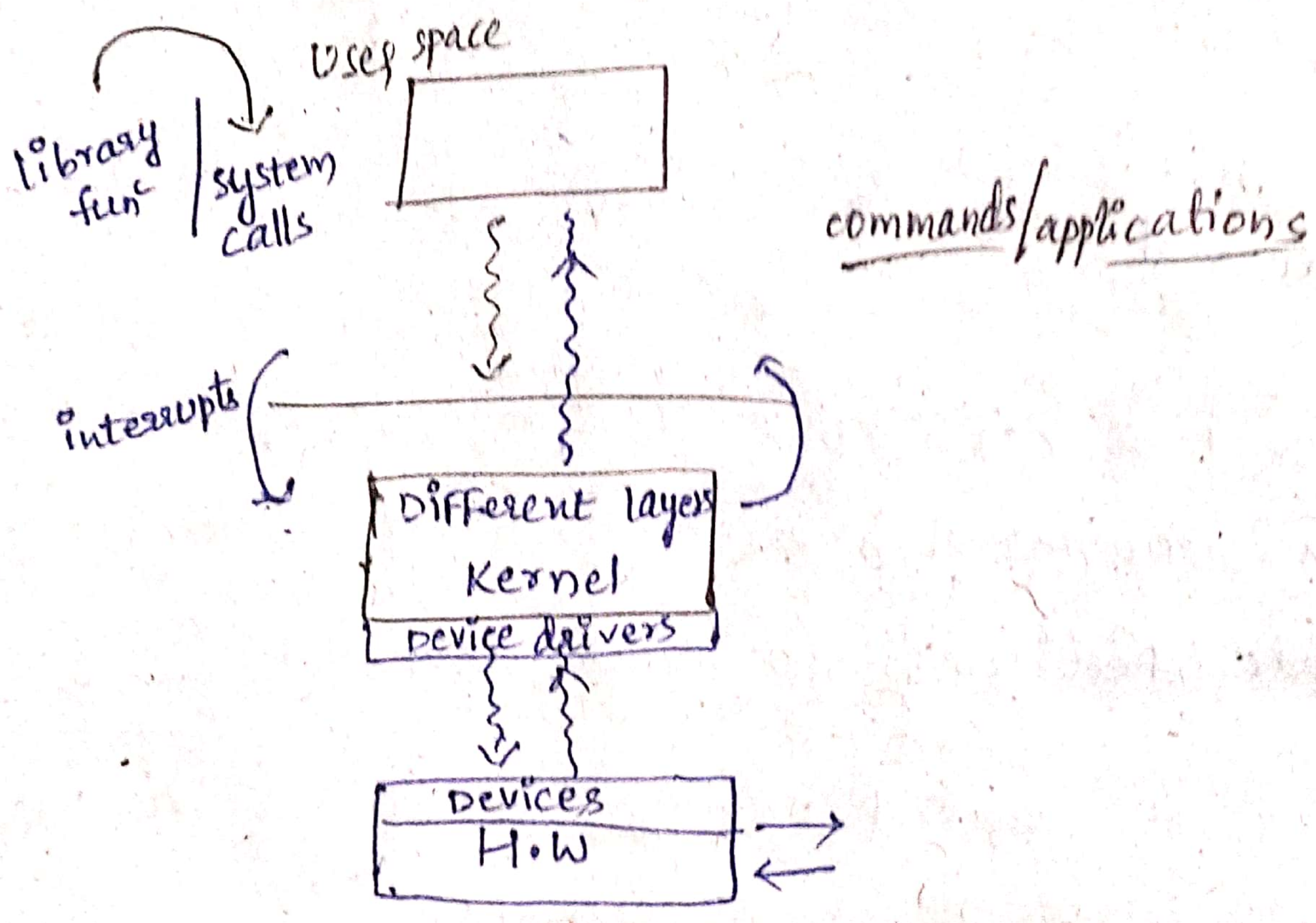
⇒ It is more secure. The code is reviewed by many experts.

⇒ It is more stable & reliable.

⇒ It supports modularity. which means at run time we can load & unload the modules on the ~~flight~~ fly

⇒ It is easy to program, we can learn from the existing code because it is open source.

⇒ It has exhaustive networking support



* Linux kernel Main roles :-

⇒ The kernel manages all the H.W resources like memory, CPU, input output devices (I/O)

⇒ It provides a set of portable, architecture & Hardware independent API's to allow user space applications & libraries to use H.W resources.

⇒ It handles the concurrent access & usage of HW resources from different applications.

* System calls :-

⇒ The main interface b/w kernel & user space is the set of system calls.

⇒ The system calls provides the main kernel services like file & device operations, networking operations, Inter process communication, process management, memory mapping, timers, threads, synchronization premitives Etc.

⇒ The new system calls can also be added by the kernel developers. This system call is wrapped by the C library. The user level applications usually uses C library functions.

* File Systems *

* Pseudo file systems :-

⇒ The linux makes system & kernel information available in user space through this file systems. Sometimes they are also called as Virtual File system.

⇒ These file systems allow applications to see directories & files that doesn't exist on any real storage (H.D) (USB or pendrive).

⇒ There are created & updated (run time) on the fly by the kernel. The two most important pseudo file systems are

① proc virtual file system. It is mounted on /proc location. It maintains operating system related information like process, memory management related parameters etc.

② sys FS File system. It is mounted on /sys directory. It represents the system devices & buses information.

⇒ Inside the linux kernel, we can see the code related to memory management, scheduler, task management, File system layer & drivers device drivers + drivers frame work, low level architecture specific codes, network stack &

device tree (Hardware description).

SVN
Git

To maintain data in cloud

To see this info of kernel :-

① Download the kernel source code Git repository

(or) www.kernel.org. (or) from Vendors website

(or) Open source communities

Generic code size ↑ (GB)

customized kernel source code (size ↓) (KB)

① verify folder names.

② Git commands (understand) → Git cheat sheet

⇒ Linux kernel will supports different architectures

⇒ The architecture supported code we can see in /arch directory.

Git → tool

⇒ It will/can support 32 bit & 64 bit architectures

⇒ we can see these → sub folders/sub directories in this folder with architecture name.

⇒ In kernel source code, we can see more than 50% code is device drivers.

⇒ More than 15% of code is architecture specific code. (±5%)

⇒ File system related code is more than 5%. (FAT32, exfat, ext3 ...)

⇒ sound drivers are more than 4%. (ac 33 ..)

⇒ Network specific code is more than 4% (net folder)

⇒ Header files more than 3% (include folder)

/usr/src ⇒ kernel headers

/usr/include ⇒ user level headers

⇒ Documentation upto 3%.

⇒ Tools more than 1%.

⇒ Kernel specific code more than 1% & so-on.

⇒ Linux header files are different from user level header files.

⇒ Kernel header files are located in /usr/src (source)

⇒ user level application header files are located in /usr/include

⇒ The entire linux source code is written in 'C' language. A little bit assembly language is used for CPU & machine initialization & exceptions & for critical library routines.

⇒ NO C++ is used.

⇒ (so, we can't use online compilers to compile kernel source code.)

⇒ To compile the kernel source code we can't use ANSI C compilers. so, we have to download the separate tool chain for the target boards

⇒ In kernel source code, no 'C' library is used. i.e, we can't use any user level 'C' library functions in kernel source code.

⇒ There is no library concept in kernel programming

⇒ Every kernel func definition is included in the kernel image itself.

⇒ Every kernel func definitions are present in text segment.

⇒ Kernel will have its own same set of kernel functions similar to user level 'C' library functions.

⇒ In Kernel source code, every directory will contains 1 Makefile & 1 Kconfig File.

⇒ portable → same code can be used for diff architecture.

⇒ Linux Kernel source code is designed to be portable that means the same source code can be used for different architectures.

⇒ In Kernel programming, we should not use any floating point numbers. (float, double)

⇒ No stable linux internal API's (functions) are used

(*) API ⇒ Application programming interface (functions)

(*) ABI ⇒ Application Binary interface (executable files)

⇒ Kernel memory constraints (memory is less)

① No memory protection.

② Kernel doesn't try to recover from attempts to access illegal locations. It just dumps oops msg on the system console.

③ Kernel will have fixed size of stack i.e., 4 KB (or) 8 KB. There is no mechanism was implemented to make it grow like in user space

④ Swapping is also not implemented for Kernel memory either.

⇒ In /arch directory we can find architecture specific code.

<u>Linux</u> folder

arch/<ARCH>/mach-<machie> → machine (or) Board specific code
↳ Architecture name.

arch/<ARCH>/include/asm → architecture specific headers

arch/<ARCH>/boot/dts → Device tree source files

arch/<ARCH>/config/ → Kernel configuration files for various targets.

arch/<ARCH>/block → block layer core code.

arch/<ARCH>/copying → License info (GPL terms & cond)

arch/<ARCH>/credits → Linux main contributors

arch/<ARCH>/documentation → Kernel documentation sources

arch/<ARCH>/drivers → All device drivers code except sound drivers.

arch/<ARCH>/fs → contains file system specific code

arch/<ARCH>/include → Kernel headers

arch/<ARCH>/include/linux → Linux kernel core headers

arch/<ARCH>/include/uapi → User space API headers (System call related header files)

arch/<ARCH>/init → Linux Kernel initialization code.

arch/<ARCH>/ipc ⟷ IPC mechanisms code.
Eg (pipe)

" /Kbuild file → Kernel build system code

" /Kconfig file → contains Kernel configuration parameters

" /kernel folder → Linux kernel code code

" /Make file → contains target definitions to build kernel image, modules & installation etc.

⇒ The Makefile which we ~~seen~~ see in the Kernel Source code top directory is known as top of the tree make file.

" /mm folder → contains memory mngmt specific code.

" /net folder → contains network specific code.

" /sound folder → sound supported code

* /arch → file system folder

* arch → source code folder

31/01/23

⇒ The linux kernel is licensed under GNU General public license version 2 (GPL V.2)

⇒ This license gives you the right to use, modify study & share the software freely.

⇒ You must mention the license under driver code during the development of driver.

⇒ It is illegal to distribute a binary kernel that includes statically compiled properitery drivers.

* Advantages of GPL drivers :-

⇒ You don't have to write driver from scratch. You can reuse code from similar free software drivers.

⇒ You could get free community contributions support, code review & testing. It happens only with code submitted for the main line kernel.

* Advantages of intree kernel drivers (already exist from since)

⇒ Once your sources are accepted in the main line tree there are many more people reviewing your code allowing to get cost free security fixes & improvements.

⇒ You can also get changes from people modifying internal kernel API's.

⇒ Accessing your code is easier for users.

⇒ You can get contributions from your own customers.

⊛ Drawbacks when we write drivers in User Space

① Increase in interrupt latency (response time).

② Security issues.

⊛ Kernel source code Management tools (or) Navigation tools

CLI → cscope — $ sudo apt install cscope

Ctags — $ sudo apt install universal-ctags (or) exuberant-ctags

GUI → Eclipse + Source code → Linux

Source insight → windows based. (.exe)

WINE ⇒ windows Emulator ⇒ with this we can install/add source insight

↓

(install this tool)

To install Eclipse by terminal

$ snap install --classic eclipse.

01/02/23

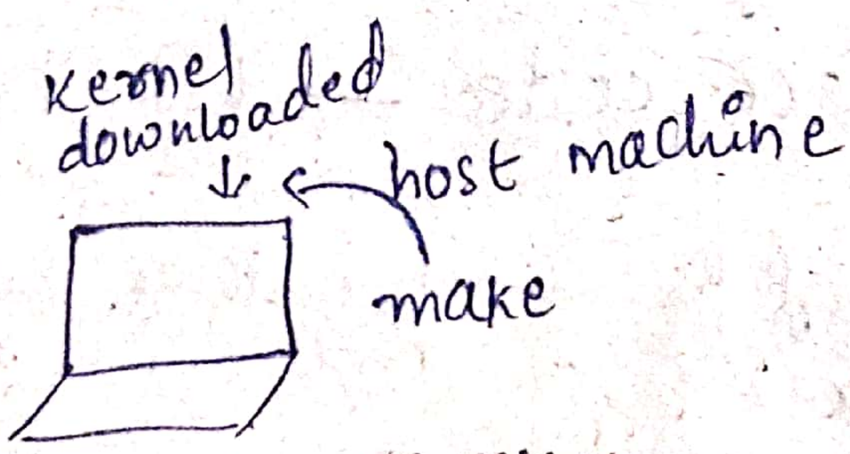
⊛ Kernel configuration :-

⇒ Kernel configuration is the process of defining the set of ~~operations~~ options with which you want your Kernel to be compiled

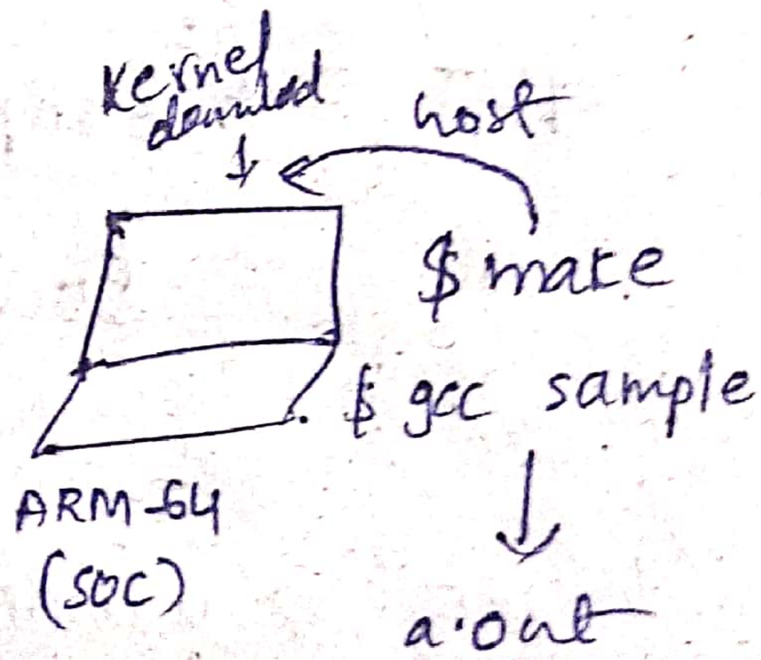
⇒ The set of options depends on the target architecture & hardware and on the capabilities you would like to give to your Kernel such as network capabilities, file system, real time etc.

⇒ By default, the Kernel build system assumes that the Kernel is configured for host architecture

⇒ Tool chain is architecture dependent. For different architectures, we have different tool chains.



$ gcc sample.c
↓
a.out → By default, it will execute on X-86 (host machine)



↓
It will execute on ARM-64 bit (SOC) by default.

* Kernel source code :-

3 options

thousands of .c files (which are dependent on basic configuration)

.c → .o
.c → .o
.c → .o
⋮ ⋮
(*)

} ⇒ Vmlinux ⇒ Vmlinuz

(elf format)

(uncompressed) debugging

(compressed kernel image).

+ System.map

(kernel symbol table).

(M)

.c → .o → .ko
.c → .o → .ko
.c → .o → .ko
⋮ ⋮

{modules}

space bar (key) from keyboard

object files

kernel object files.

(space)

.c
.c
.c
⋮

Kconfig file

Kbuild file

Make file

Top of the tree Make file

$ make menuconfig

Target name defined in top of the tree Makefile.

⇓

After selection is done, save

⇓

→ generate file .config Kernel configuration for host (or) target.

/arch/x-86/config/ ____

configuration

/arch/arm/config/ ____

bcm 8375 defconfig (for raspberry pi)

⇒ $ make ARCH = arm menuconfig

CROSS_COMPILE =

(*) → To select .c files & make kernel image = (y)

(M) → To select .c files to make kernel object files = (m)

(space) → To remove / not involve in building files. = [Not set]

1. make menuconfig

2. make

3. make modules

sudo 4. make modules_install → /lib/modules/kernel version/drivers.

sudo 5. make install → /boot

⇒ Need super user permission.

6. sudo update-grub

→ Target boards 7. Reboot

(cross platform)

02/02/23

⇒ Kernel configuration & Build system :-

⇒ KC&BS is based on multiple make files.

The only one main Makefile present at top ~~the~~ directory of kernel source tree will interacts. This interaction takes place using make tool which passes the make file through various targets defining which action to be done

(configuration, compilation, installation etc --).

⇒ $ make menuconfig

⇒ menu config is the target name.

⇒ The configuration is stored in the .config file

It is a simple textual file defines a key=value

⇒ For selecting the options we can run below commands

$ make xconfig (or) make gconfig (for graphical interface)

$ make menuconfig (or) make nconfig (for textual interface).

⊛ Initial configuration

⇒ By default the configuration is set for the host machine from /boot/config-Name
↳ kernel version name.

⇒ For Embedded platforms, the default configuration files are available in arch/ARCH name/configs

⇒ To load a default configuration file just run below commands to generate .config file with that configuration.

$ make cpu_defconfig
↓
.config

⇒ This command will overwrite the existing .config file. Now we can make configuration changes

by using make menuconfig or make nconfig.

→ To create your own default configuration file

$ make save defconfigname

→ This kernel image is a single executable binary file. so, it is generated by linking multiple object files corresponds to the features enabled in the configuration. so, this kernel image gets loaded in the memory (RAM) by the boot loader.

→ There are some features that can be compiled as a modules. These modules are pluggins that can be loaded or unloaded dynamically to add (or) remove features to the kernel. so modules can be device drivers (or) file system drivers etc.

→ Each Module is stored as a separate file in the file system in /lib/module directory.

→ The kernel options can be selected during the make menuconfig as part of kernel image (or) separate module (or) can be ignored. For this, we have different type of options.

① bool options → They are either True (or) false

(*) True means included the feature in kernel

(space) False means to exclude the feature from kernel.

② Tristate options :-

(*) True means to include the feature in kernel image

(M) module. module to include the features as a separate module.

(space) False means To Exclude the feature from kernel.

⇒ while selecting the options, some options are automatically selected based on dependencies, some options we have to enable manually for dependencies.

⇒ For make xconfig it requires debian (or) Ubuntu packages ~~multiple~~ packages
Qt-5

⇒ Based on selection, we can see the keys defined with the values either m (or) y (or) Not set

⇒ For make menuconfig it requires libncurses-dev Packages. tly for make nconfig too

1. NO .config

1st time $ make menu config

	key
* →	y
M →	m
space →	Not set

⇒ It will generate .config file

Saved with .config.old

2nd time modifications done

$ make menuconfig
↓
save
⇓
.config (new file)

⇒ Choose (or) select a compiler to compile kernel source code. This compiler is invoked by the kernel Make file with a variable defined as

$(CROSS_COMPILE) gcc (defined in top of the tree make file)

⇒ For native (or) host machines it is not defined.

⇒ For building to a target board download the tool chain first either export the path (or) set the path in .bashrc file. Tool chain can be downloaded from vendor's website (silicon or board vendors)

⊛ How to export path

⇒ By using export followed by path.

* Cross tool-chain set up :-

⇒ This methods are applicable only for target boards for different architectures.

method - I Temporary (If we close the terminal exported path will be lost)

By exporting path one time

```
export ARCH = arm
export CROSS_COMPILE = _____
```

Tool chain → Raspberripi
↳ Downloaded from silicon or board vendors

Downloads/arm-linux-gnu-eabi/bin/arm-linux-gnu
" " -objdump
" " -nm

$ make ARCH=arm CROSS_COMPILE = arm-linux-

method - II

By mentioning the path every time while using make command.

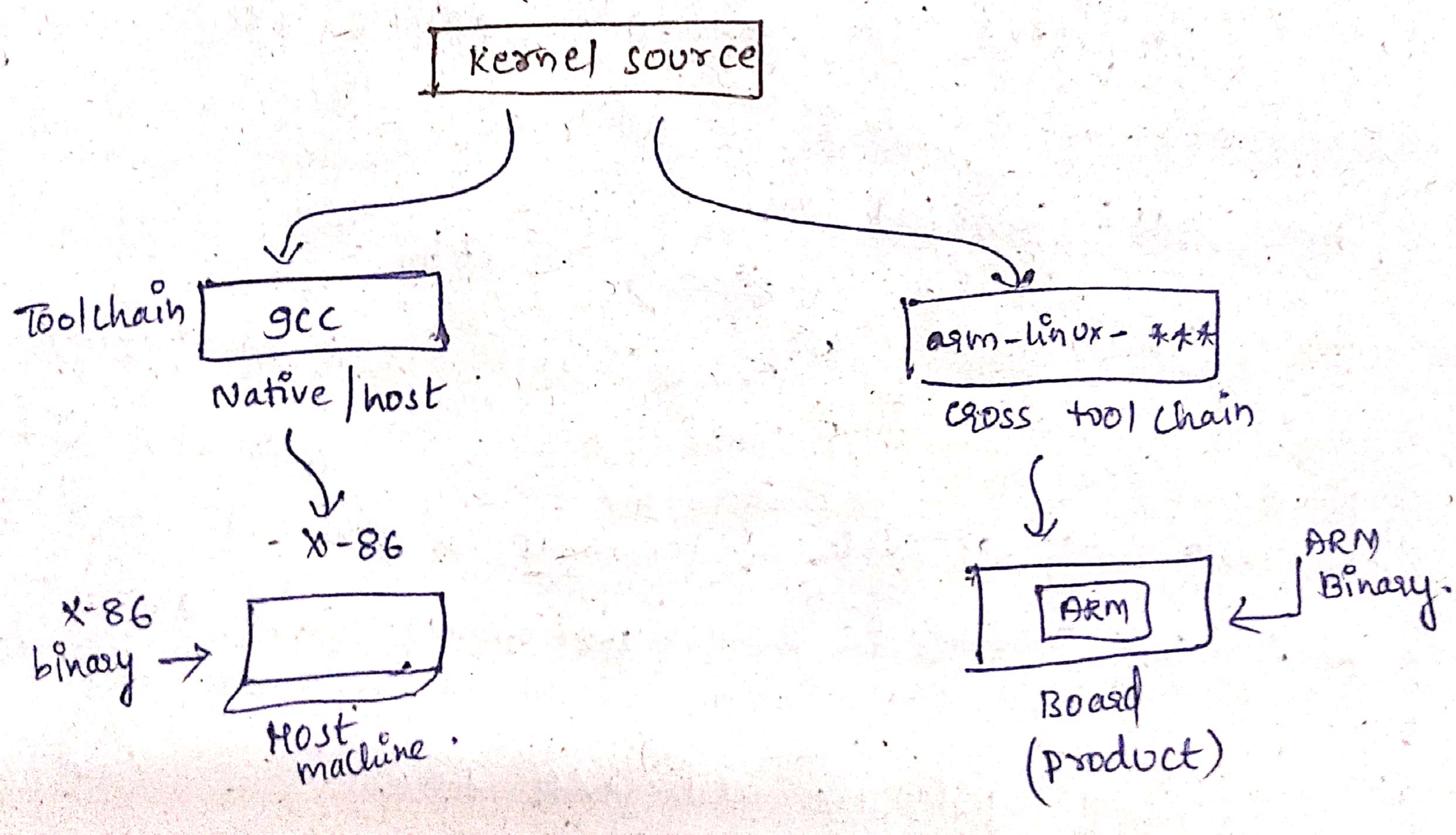
$ make ARCH=arm CROSS_COMPILE = /home/engineer/Downloads/arm-linux-gnu-eabi/bin/arm-linux-

method - III

Permanent (set the path in .bashrc folder which is present in Home directory in hidden files)

use

$ make ARCH = arm CROSS_COMPILE = arm-linux-



make command :-

⇒ This will generates Vm linux, the raw uncompressed kernel image ~~useful for debuggin~~ which is in elf Format used for debugging but cannot be booted.

⇒ The final compressed kernel image that can be booted is generated in arch/ARCH/boot/imagename

for x-86 we can see bzimage & Vm linux

For ARM we can see zImage

⇒ The device tree blob is generated in arch/ARCH/boot/dts/***.dtb

⇒ Device tree source files are compiled with device tree compilers to generate device tree binary.

.dts (source)
.dtsi
} ⟹ .dtb (device tree blob)
Device tree compiler

⇒ Even it will generate system.map & also configuration files which starts with config-

⇒ make modules :- (The options which are selected with M. They are converted into kernel objects (modules))

⇒ This will generate all .ko files

.c files ⟶ .o ⟶ .ko

⇒ objdump & nm tool can be used on .o as well as .ko

* make install :-

⇒ It does the install system for the host system by default. so it must run as a root user (sudo make install).

⇒ It install the generated kernel image vmlinuz copies ~~from~~ to /boot/vmlinuz-kernel-version from arch/ARCH/boot directory along with system.map and config files (kernel configuration file)

⇒ make install is rearly used in embedded development only for host machines.

* make modules-install :-

⇒ It must be run as a root user. It does the installation for host system by default.

⇒ It installs the modules that are generated (.KO files) are copied to /lib/module/kernel version/ directory.

⇒ It will generates modules.alias, modules.alias.bin, modules.dep, modules.dep.bin, modules.symbols, modules.symbols.bin.

modules.alias, modules.alias.bin → used for modules loading & unloading

modules.dep, modules.dep.bin → used for module dependencies

modules.symbols.bin → The symbols belongs to which module.

* Kernel cleanup targets :-

⇒ To cleanup the generated files, we have to use

$ make clean.

⇒ $ make mrproper → It removes all generated files along with .ko files.

⇒ $ make dist clean → It also removes backup files, patchfiles, config files & all generated files.

~~Cross compi~~ 03/02/2023

* Tool chain :-

⇒ On a linux ~~work~~ station (or) host machine, we can see native tool chain. This tool chain runs on your work station & generates the code for x-86 (or) host machine

⇒ For Embedded system (target boards) (ARM architecture) development, it is usually impossible (or) not intersting to use a native tool chain.

⇒ The target is too restricted in terms of storage & memory.

⇒ The target (1.3 GHz) is very slow when compared to your work station. (3.4 GHz)

→ we may not want to install all development tools on your target board. Therefore, cross compiling tool chains are generally used on your work station but generates code for your target boards.

⇒ To generate / create the tool chain, we should understand 3 types of machines.

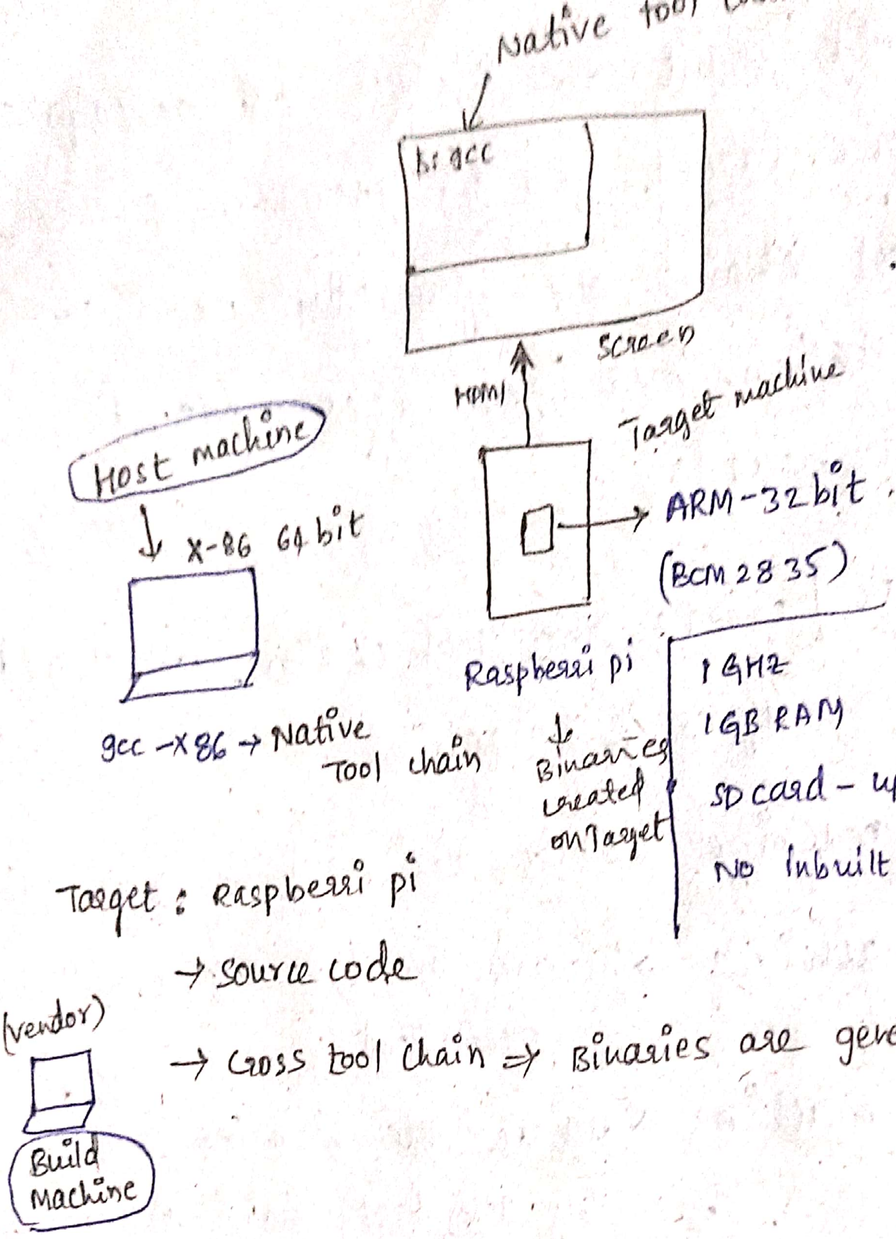
① Build machine → where the tool chain is built

② Host machine → where the tool chain will be executed.

③ Target machine → where the binaries created by the tool chain are executed.

⇒ To build the cross compilied tool chain we need these following tools.

① Binary ~~utilities~~ utils, (Bin)
② Kernel headers,
③ C (or) C++ libraries,
④ C(or)C++ compilers,
⑤ Gdb debuggers & other development & debugging tools



Source code can be downloaded from vendors website (or) Github

⇒ <u>Bin utils</u> is a set of tools to generate & mainipulate binaries for a given cpu architecture like assembler, linker, debugging tools like objdump, readelf, archive tools, nm & so-on---.

⇒ <u>Kernel headers</u>

The c library & compiled programs needs to interact with kernel with available system calls & their numbers, constant definition & data structures etc.

⇒ <u>c library</u> :-

⇒ The 'c' library is an essential component of a linux system to interface b/w applications & kernel provides the well known standard c

API to ease application development.

⇒ ~~Building a tool chain manually~~

⇒ Building a cross compiling tool chain by self is a painful & difficult task. we need to learn details of about a lot of components like c library versions, API's, floating point mechanisms, components etc.

⇒ Get a ~~pre~~ compiled tool chain from the vendors website or from distributions

⇒ There is another solution to build the tool chain with an automated process like <u>build root</u> (tool used to generate own compiler, FS, etc).

⇒ By using build root we can automate the process to generate cross compiler, kernel images, device tree blobs, boot loader image, root file system image.

<u>Download</u> (ppt)

Build root : a nice, simple & efficient Embedded linux build system.

x-86 → grub boot loader

ARM → U-boot loader

04/02/23

(Important tool)

How do u generate patch file ?

⇒ It is a file used to find & solve the bug

⇒ It can be downloaded by diff tool.

⇒ It consists of old & new details info (difference of changes) we can add or delete the data from two files.

Scanned with OKEN Scanner

06/08/23

* Kernel Modules :-

⇒ Modules make it easy to develop the drivers without rebooting. At run time we can load & unload the module on fly by using simple commands

⇒ The Modules are useful to keep kernel image size to minimum & reduces the boot time.

⇒ Modules can be loaded as a root user.

⇒ when a module is loaded, the related info is available in the kernel log. The Kernel keeps its messages in a circular buffer. we can see the kernel logs by using a commond dmesg (diagnostic message)

⇒ The default log level is 5.

⇒ This dmesg is extracting the kernel logs from a specific path or file var/log/syslog

⇒ The kernel module is different from user level application. we can see this module with .ko extension (kernel object).

⇒ Each Module will have one (1) initialization Fun^c & one(1) clean up function. These fun^c names are

⇒ while loading the module, the initialization fun^c gets executed & now the module is ready to receive the request coming from application or Hardware.

⇒ when you unload the module, cleanup fun^c gets executed (or) exit fun^c gets executed.

⇒ while loading & unloading the prints comes from these functions are redirected to Kernel logs.

⇒ when a module is loaded the initialization fun^c symbol is added to the kernel symbol table. we can see the kernel symbol table in 2 locations i.e, /boot/ directory

① /boot/

system.map-*** → This is generated (After kernel compilation using $ make command)

when we use $make install system.map-*** will add in to /boot directory

② After booting the kernel, the dynamic kernel symbol table is loaded in to /proc/kallsyms

↑ dynamic kernel symbol table

→ when a module is loaded the initialization & exit fun^c entries are updated in to dynamic kernel symbol table (i.e, kallsyms) & the definitions are added in to the segment of running/boot

→ Each Kernel module fun^c is applied with static keyword.

→ For every module we need to write a separate Make file.

basic module.c

static int hello_~~module~~ init (void)

→ If a kernel fun^c is not receiving any inputs you must specify it as void. (to avoid errors)

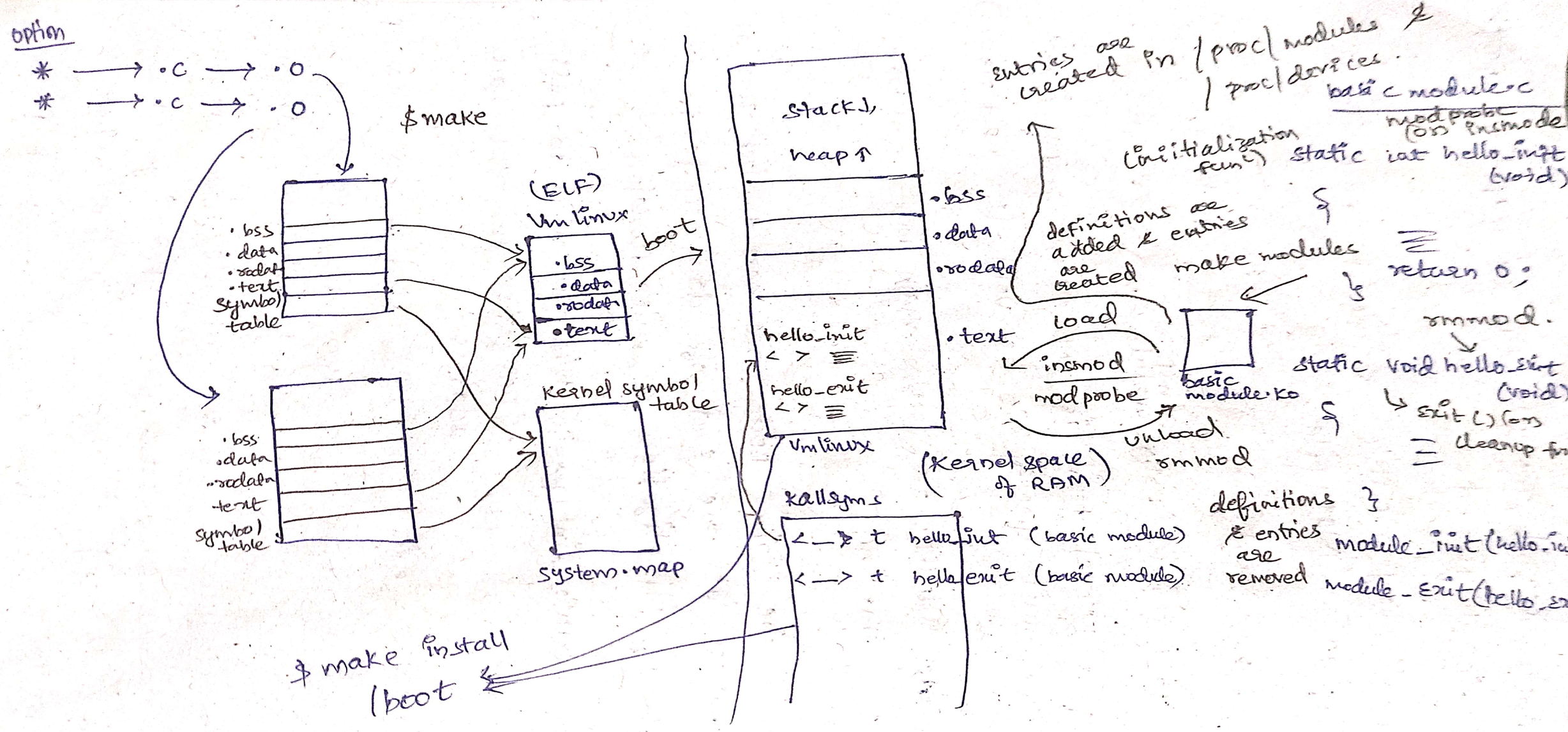
→ ~~We~~ we should not use main fun^c in kernel programming.

→ In the driver code, we have to use two (2) kernel macros at the end of driver code to specify initialization & exit function.

→ module_init() → To this, we need to pass initialization functions base address

→ module_exit( ); → we need to pass clean up fun^c base address or exit fun^c base address.

→ when you unload the module, the clean up fun^c gets executed & the corresponding entries are deleted in ① /kall syms & ② /proc/modules & ③ /proc/devices. definitions are removed from kernel image text segment.


option
* → .c → .o
* → .c → .o
$make
.bss
.data
.rodata
.text
symbol table
(ELF)
Vmlinux
boot
Kernel symbol table
System.map
$make install
/boot
Stack
heap
hello_init
hello_exit
Vmlinux
(Kernel space of RAM)
kallsyms
t hello_init (basic module)
t hello_exit (basic module)
entries are created in /proc/modules & /proc/devices.
basic module.c
mod probe (or) insmod
(initialization fun) static int hello_init (void)
definitions are added & entries are created
make modules
return 0;
rmmod.
static void hello_exit (void)
exit() fun
cleanup fn
load
insmod
modprobe
basic module.ko
unload
rmmod
definitions & entries are removed
module_init(hello_in
module_exit(hello_ex

make menuconfig

options

{*} CONFIG_***

{M}

↓

.config

CONFIG_*** = y

" = m

Make File

obj-(CONFIG_***) = modules.o

make ⇓

obj-y = module.o

obj-m = module.o

make → ↓

make modules → .KO

$ make

Makefile

obj-m = basic module.o

make -C /usr/src/Kernel version M = $(shell pwd) modules

To change directory

(or)

/lib/modules/$(shell uname -r)/build

~~Makefile~~

Kernel headers are present in

/usr/src/kernel version/

Top of the ~~tree~~ make file

modules

(targets)

Kernel logs :-

⇒ Kernel debugging :-

⇒ We can't use printf func in Kernel programs we have an equivalent func in Kernel i.e, printk which is used in old Kernel versions still the latest Kernels will have printk support.

Instead of this we have to use a set of family of functions with pr_ prefix with log levels. These log levels are defined in include/linux/printk.h header file.

⇒ Kernel log levels are present in Linux-5.10.165/include/linux/kern_levels.h

⇒ Kernel printk log levels defined in include/linux/printk.h.

eg: printk (KERN_INFO "Module is loaded\n");

(or)

pr_info ("Module is loaded\n");

~~⇒ For debugging you can use a set of~~

⇒ For debugging we can use pr_debug or set of family functions that starts with dev_prefix.

→ They are defined in include/linux/device.h

⇒ Each message is associated ~~with~~ to a priority ranging from 0 for Emergency to 7 for debug as specified in include/linux/kern_level.h header file.

⇒ All the msgs, regardless of their priority, are stored in the Kernel log ring buffer typically accessed by dmesg command.

⇒ Some of these messages may appear on console, depending on their priority & the configuration of log level of the Kernel Parameter which defines the priority above which msgs are displayed on the console are ~~The~~ detailed ~~are~~ in admin-guide/kernel-parameters and the value of /proc/sys/kernel/printk which allows to change at runtime the priority above which messages are displayed on the console. It's details are detailed in documentation/sysctl/kernel.txt.

(*) Find out the header files in which module_init, module_exit are defined.

linux/module.h.

~~/linux/modules.h~~

~~linux/init.h~~

⇒ module_init MACRO internally invokes

init_module function

⇒ module_exit MACRO internally invokes cleanup_

module function.

(static applied data type has it in symbol table)

module.c

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

// static int hello_module (void)

int init_module (void)
{
    pr_info ("My basic module is \n");
    pr_info ("loaded successfully \n");
    // printk (KERN_INFO "loaded successfully \n");
    // printk ("<6>" "loaded successfully \n");
    return 0;
}

// static void hello_exit_module (void)
void cleanup_module (void)
{
    printk (KERN_INFO "My basic module is \n");
    printk (KERN_INFO "OFF loaded successfully \n");
}
```

```
// module_init(hello_module);
// module_exit(hello_exit_module);

MODULE_AUTHOR("SUNIL KUMAR");
MODULE_LICENSE("GPL_V2");
MODULE_DESCRIPTION("Basic module");
```

Make file

```
#obj-m += test_mod.o
 obj-m = module1.o

all :
        make -C /lib/modules/$(shell uname -r)/build
                M=$(shell pwd) modules
# make -C /usr/src/linux-headers-3.16.0-77-
        generic M=$(shell pwd) modules.

clean :
        rm -rf *.o *.ko *.mod.c.
```

* __INIT , __EXIT MACROS :-

* __init MACRO is used before initialization function.

```
static int __init hello_module(void)
```

* initialization func is required only while loading the module. After loading, this func is not required. that's memory is wasted & it

is not used by the kernel. To use that memory by deleting its definition from running kernel image text segment we have to use __init macro before fun^c name. In this case this fun^c definition is not added to the text segment, it is added to the .init segment

After executing this fun^c once, this definition is removed automatically. My __exit macro is applicable to cleanup fun^c (or) exit fun^c.

⇒ To load the module we need to use $insmod. & $modprobe. (for lib/modules).

⇒ To unload the module, we need to use $rmmod

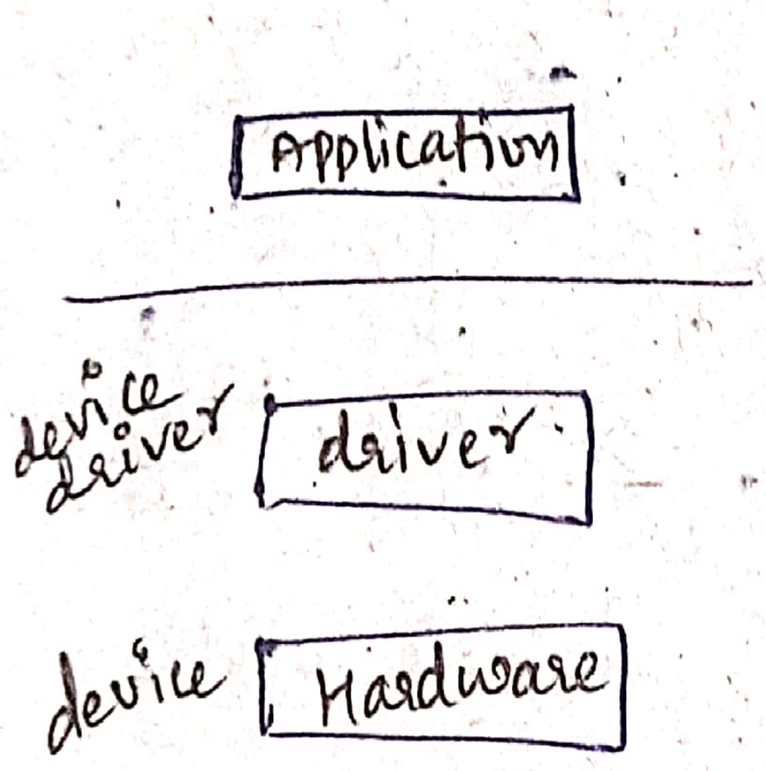
⇒ To see the modules which are loaded into kernel we need to use lsmod, this will fetch the info from /proc/modules

⇒ To see the information of module, we need to use $modinfo.

⊛ Device drivers :- (07/02/23)

⇒ Device drivers is a piece of code which will be written to control a particular device (or) to communicate with a device (or) to utilize the features of a device.

⇒ Device is a hardware, device driver is a software we should write corresponding application in user space to utilize the features provided by the driver.



commands (or) Applications	driver	device
✓	✓	✓
✓	✓	✓
✗	✓	✓
✓	✗	✓
✓	✓	✗

NO communication

⇒ For proper communication, we should have application in user space, driver in kernel space & Hardware in the system.

⇒ They are catagorized into 3 types of device drivers. They are :-

1. character device drivers
2. Block device drivers
3. Network device drivers

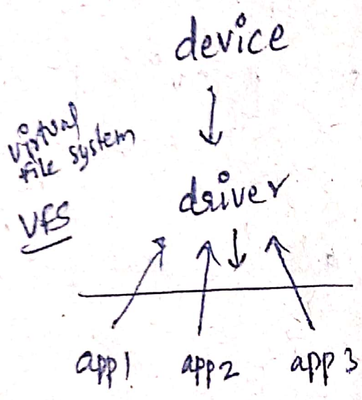
⇒ character device drivers will send the data byte by byte. Each byte will raise 1 interrupt

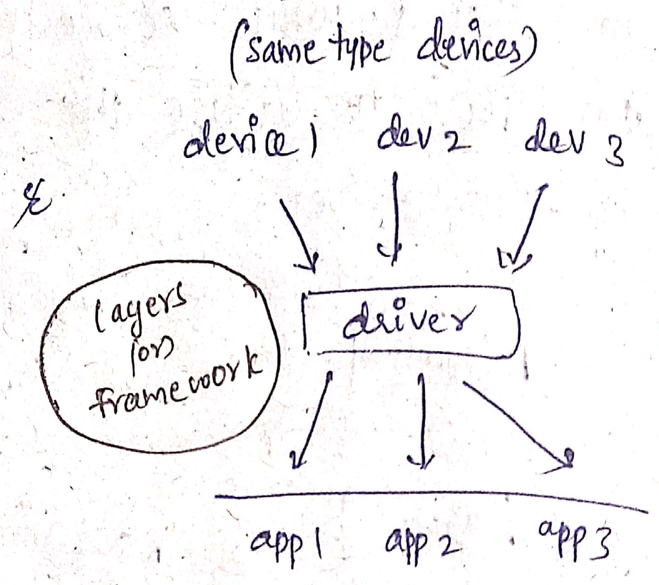
1000 bytes → 1000 interrupts.

⇒ Block drivers will send the data by block by block.

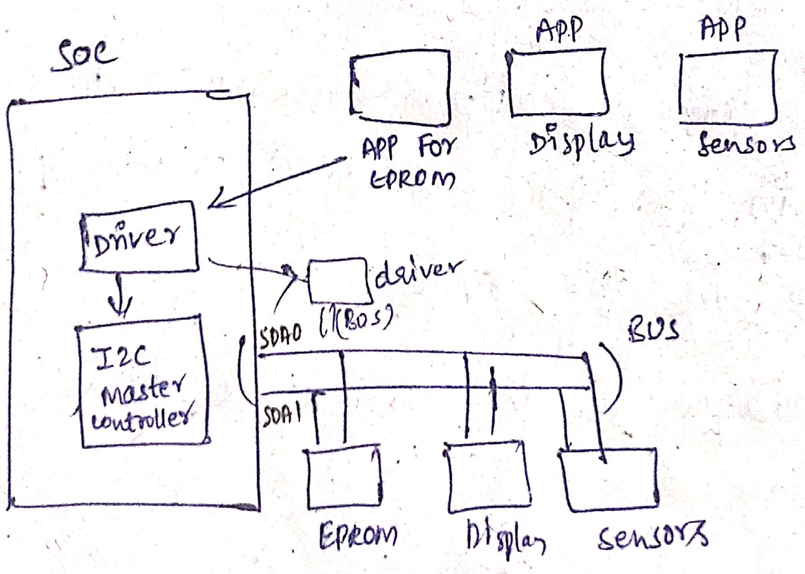
⇒ Network device drivers will send the data in terms of packets (with some limitations).

⇒ 85% of device drivers comes under character drivers.

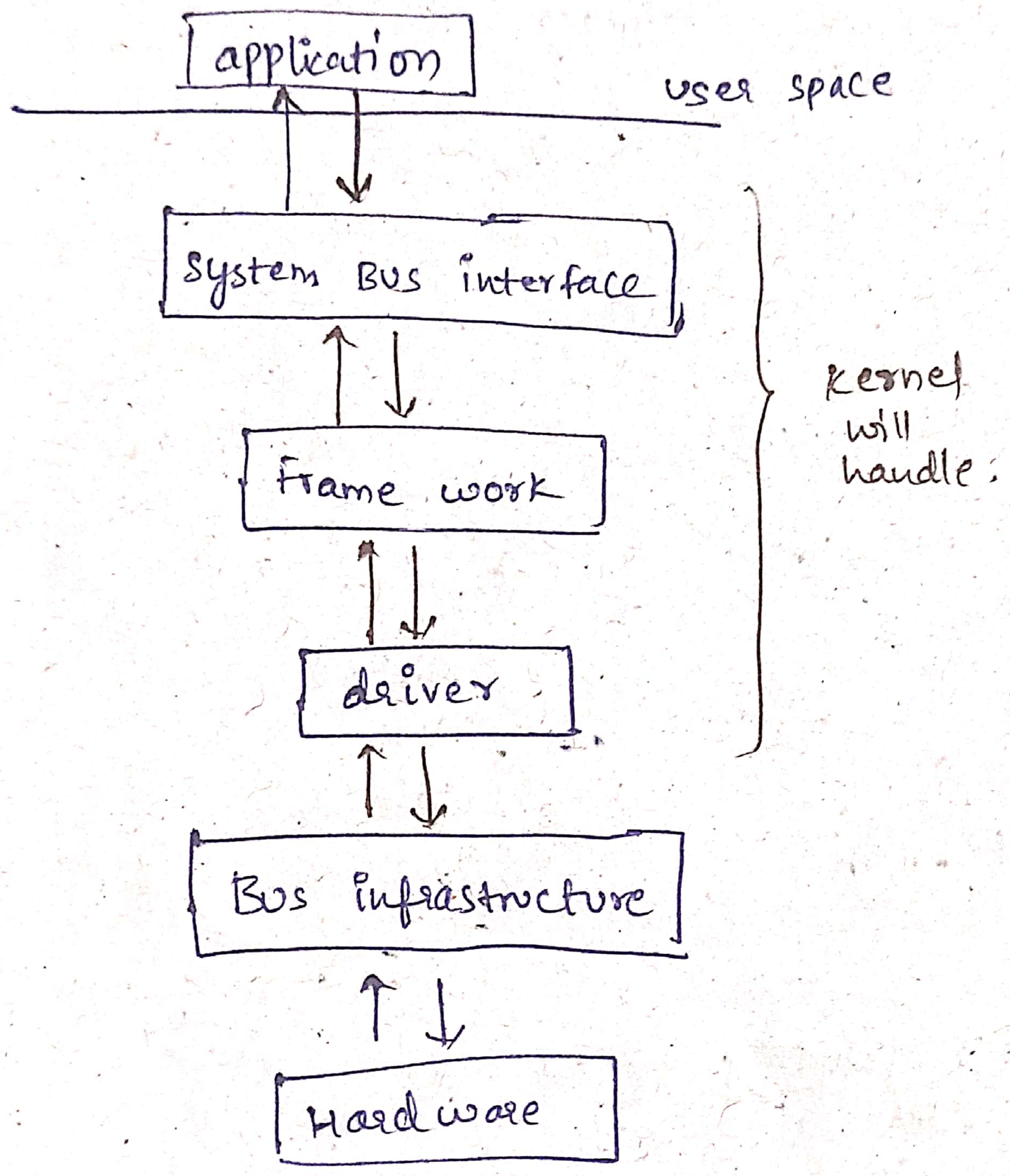




multiple applications



→ In linux, a driver is always interfacing with a frame work that allows driver to expose the hardware features to user space applications & a bus infrastructure, the part of the device model to detect or communicate with the hardware.



8/02/23

a.out

/dev/stdin → (Keyboard driver)

/dev/stdout
/dev/stderr } → Terminal driver
(or)
display driver
(or)
console driver

opens this 3 File objects defaultly.

file_operations structure

Function pointers

open func pointer
read "
write "
release "
ioctl "

← { open ( ) system calls
read ( )
write ( )
close ( )
ioctl ( ) }

cdev File operations
device no
&
no. of devices

device no

3 dev files
c (42,0)
c (42,1)
c (42,2)

i_rdev
i_cdev
member in inode structure

device files → type of file
c
B
N
→ statically
→ dynamically

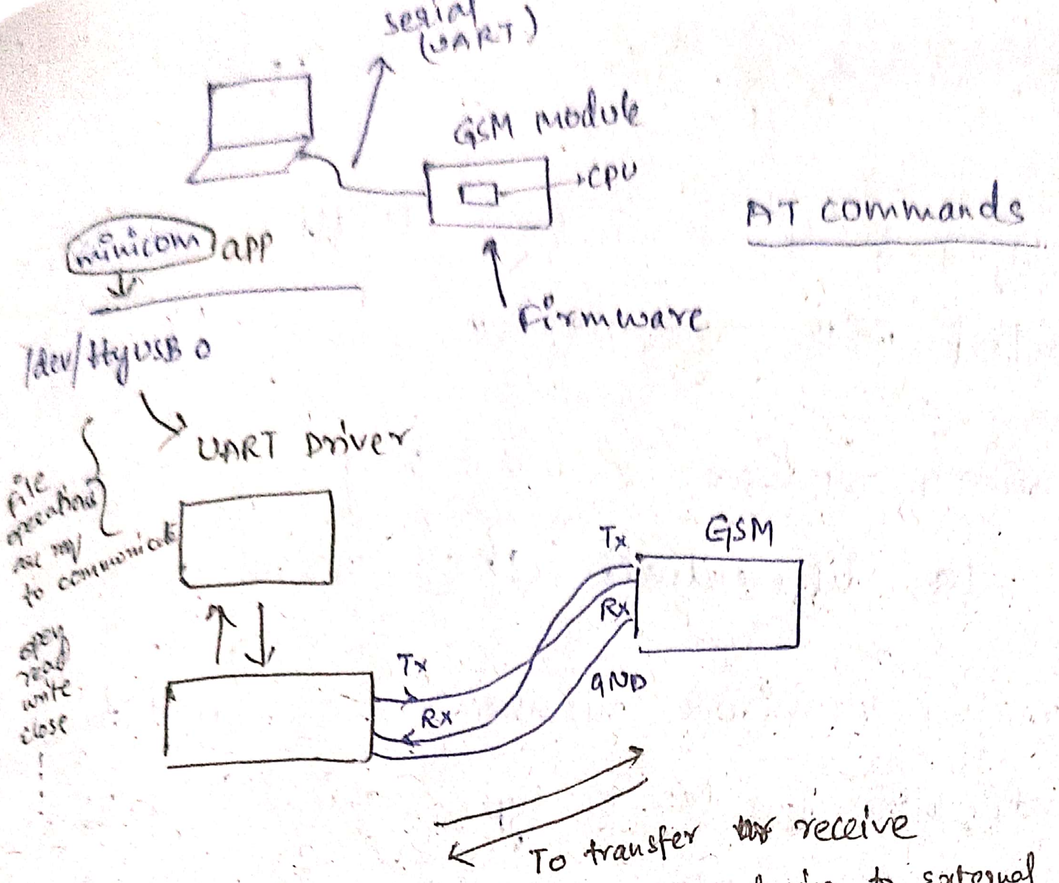
APP1 APP2 APP3

System call interface

driver (42, 0 to 2)

device no → dev_t (32 bit value)
↓
major no + minor no

device 1 (42/0)
device 2 (42/1)
device 3 (42/2)



To transfer ~~or~~ receive data from device to external hardware, the code is written in interrupt handler func.

⇒ interrupt handler ( )

{

device → external device
internal

(errors).

}

⇒ Linux requires a mechanism to transfer data from Kernel to user space & from user space to Kernel. This transfer of data is handled via device nodes (or) device files (or) virtual nodes (or) virtual files (all are same)

⇒ These device files exists within the root system in /dev directory.

⇒ To perform the operations, from user space these applications should open these device files.

Major number & Minor numbers :-

⇒ The major & minor number identifies the driver associated with the device.

⇒ The minor number is used only by the device driver to differentiate the control.

⇒ The major & minor numbers are stored together in a variable of dev_t type which is a 32 bit value. In this 12 bits are reserved for Major number & 20 bits are reserved for Minor no.

⇒ ~~This can be combined~~

⇒ The combination of Major number & starting minor no. can be generated by using a predefined MACRO dev_t dev no;

```
dev no = MKDEV (42, 0);
```

Major number — starting minor no.

~~This~~

⇒ The major numbers are already reserved by the Kernel for specific drivers.

⇒ A major no. which is already assigned to the driver by the Kernel (or) which is already loaded in the Kernel can't be used for other devices.

⇒ 42 major number is used for practical purpose

⇒ we can extract the major number & minor no.

from device number by using predefined MACRO.

extracts major no

= MAJOR (dev_t devno);

minor no.

= MINOR (dev_t devno);

(Q) How do u extract the major no. & minor no. by using inline function?

(A) unsigned imajor (struct inode *inode);

inode

device file inode object base address.

unsigned iminor (struct inode *inode);

Assignment

(*) Find out the header files in which this dev_t (type def) & the above macro's are defined

↳ Kdev_t.h

⇒ Registering a character driver with the kernel can be done along with major no. can be done by using below kernel function.

device no

⇒ int register_chrdev_region (dev_t devno, unsigned int count, char *name);

no. of devices

driver name.

⇒ on success, it will return '0' (zero)

on failure, it will return Error.

It will create entry in /proc/devices with some info.

⇒ The opposite of this func is

unregister_chrdev_region.

⇒ void unregister_chrdev_region(dev_t devno,
unsigned int count);

on success
⇒ It will delete the entry in /proc/devices directory

⇒ This func fails when the driver is in communication with devices (or) applications.

⇒ If the major number has to assigned by the Kernel (or) we don't know which major number to be assigned we have to use below Kernel function. i.e., alloc_chrdev_region.

address of device no

int alloc_chrdev_region(dev_t *devno, unsigned int
~~starting no~~
minor no, unsigned int count, char *name);

⇒ On success, it will return 0.
on failure, it will return error (or) -1.

⇒ Each character device (or) driver is associated with a cdev object.

⇒ cdev object is a predefined structure which is of type struct c_dev;

⇒ It is defined in linux/cdev.h header file.

⇒ we can initialize the cdev object with file operation objects by using cdev_init func.

⇒ Before using this func, we have to define file operation structure it is a predefined structure which is of type struct file_operations. It contains file operations func pointers. This structure must be initialized in your driver. we need to assign driver operations (driver routines) base addresses to this func pointers. It is defined in linux/fs.h header file.

⇒ From a driver (or) device point of view, we need to maintain our own structure which is called device context info (or) driver context info (or) per device context info.

⇒ This cdev object, buffers, reference counts, indexes total no. of characters, IRQ line no, resources starting address & Kernel data structures should be the member of this device context info.

⇒ Now, Once the file operation structure is initialized, we have to link cdev object with file operation structure.

cdev.h

⇒ void cdev_init (struct cdev *, const struct file_operations *);

⇒ struct cdev *cdev_alloc (void);

⇒ int cdev_add (struct cdev *, dev_t, unsigned);

cdev object base address — device no — no. of devices.

⇒ void cdev_del (struct cdev *);

⇒ cdev_init( ) :-

⇒ This func will take 2 i/p's, cdev object base address & file object base address & creates the link.

⇒ By using cdev_add func, we are informing to the kernel that this driver will perform these many operations with corresponding devices with associated major & minor numbers.

⇒ we can create this cdev object dynamically by using a func cdev_alloc func.

⇒ cdev_init, cdev_alloc, cdev_add() these funcs should be used in initialization func.

⇒ cdev_del() is used in cleanup (or) exit ( ).

→ Under initialization Fun^c

① Register the driver with the kernel with the corresponding major no, minor no & no. of devices

② Link the ~~file~~ cdevobject with file operation object & pass it to the kernel. If we have kernel data structures even initialize them

③ Driver related buffers, counters, no. of characters must be initialized.

④ Initialize the synchronization kernel data structures

⑤ Hardware device initialization should be done (resources access & register the interrupt handler fun^c with IRQ)

→ cleanup fun^c (or) exit ()

① The exact opposite of initialization fun^c is done under cleanup fun^c...

→ (ls mod (or) cat /proc/devices)

→ To see the registered drivers with major numbers, we can use

```
$ ls mod
   (or)
$ cat /proc/devices.
```

⇒ Entries are created when driver gets registered.

⇒ drivers gets registered when we use register_chrdev_region.

⇒ cdev_init links the driver & file operations (cdev object)

⇒ The link gets created b/w app & driver is when we open the file in user space

⇒ After initialization & cleanup (), use insmod to load the driver.

⇒ insmod command, CPU will executes initialization fun^n then the driver gets registered & is loaded.

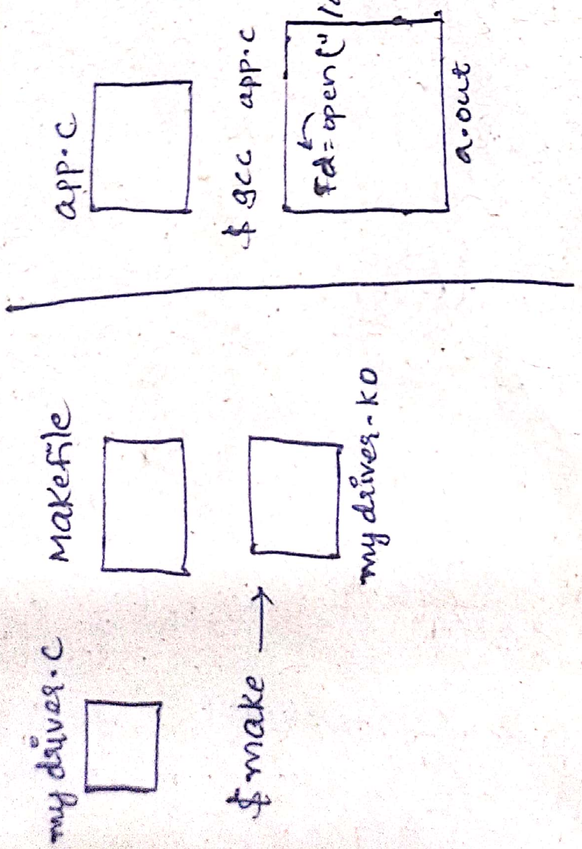
⊛ Device File Creation :-

⇒ It can be done statically & dynamically.

⇒ For statically we have to use a command

$ sudo mknod /dev/ <file name defined by user> c <Major no.> <Minor no.>

b → for block devices



mydriver_info info

```
char rbuf[100];
char wbuf[100];
unsigned int rc;
"   int wc;
int rr;
int rf;
int wr;
int wf;
int irq;
void *reg
struct cdev mycdev;
owner;
struct file_operations *ops;
```

dev_t devno;

(42,0)

/proc/device

entry gets created which means register_chrdev_region() ✓

& also

/proc/kallsyms

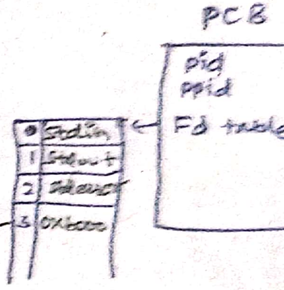
```
<> t mydriver_open
<> t "  _read
<> t    _write
<> t "  _close
<> t "  _init
<> t "  _exit
```

vmlinux

```
mydriver_open:
mydriver_read:
mydriver_write:
mydriver_close:
mydriver_init
```

.text

cdev_init()



int fd = open("/dev/mydev", O_RDONLY)

sys_open( )

0xc000 0xb000

```
mydriver_open (struct inode *inode, struct file *file);
{
    =
}

my driver_read ()
{
    =
}

my driver_write ()
{
    =
}

my driver_close ()
{
    =
    =
}

my driver_initialize ()
{
    =
}

mydriver_exit ()
{
    =
}
```

09/02/2023

⇒ we have to maintain the module usage count in the driver by using below func.

⇒ `try_module_get (THIS_MODULE);`

↳ It passes the address where this module is loaded.

// It will increase the count of no. of devices communicating to driver.

⇒ The opposite of this func is module_put, to this we need to pass THIS_MODULE.

~~put~~ `module_put (THIS_MODULE);`

// This will decrement the count by 1.

(*) NOTE :-

⇒ If the module usage count is not zero, we cannot unload the driver. i.e., rmmod command will fail. So, rmmod command will verifies the module usage count.

⇒ private data member (file object) it is used only for particular device communication of same type. for specific application. In this member we will update device context info (or) driver context info base address.

⇒ In open driver routine, increase the count by 1.

⇒ If it is for a specific device communication update the ~~private data~~ device context info in private data.

⇒ Verify the file it is opened under which mode?

⇒ In close driver routine, decrement the module usage count by 1.

⇒

⇒ copy_from_user & copy_to_user are the kernel functions used to copy the data from user space buffer to kernel space buffer & vice versa

⇒ To transfer (read/write) 1 byte (or) 4 bytes of data at a time we will use get_user() & put_user() functions are used

⇒ copy_from_user inputs are destination address, source address & length.

```
copy_from_user (tdev->buf, buff, len);
```

⇒ In write driver routine, we will copy the data from user space to kernel space by using copy_from_user func & returns the no. of characters copied.

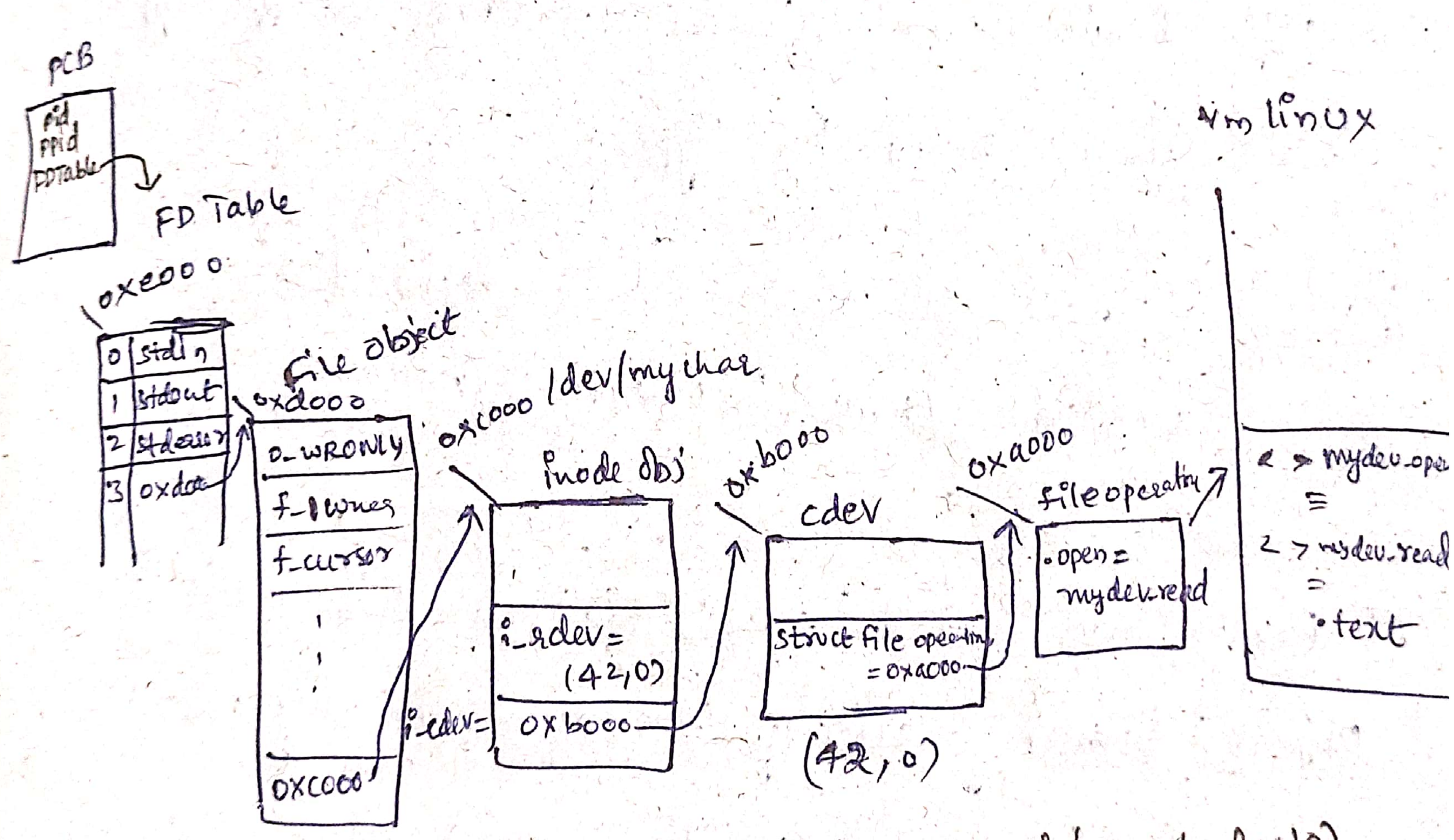
⇒ In read driver routine, we are copying the data from kernel space buffer to user space buffer & reducing the total no. of characters count and also returning the no. of bytes copied to user space.

mknod /dev/mychar c 42 0

int fd = open("/dev/mychar", O_WRONLY);

---

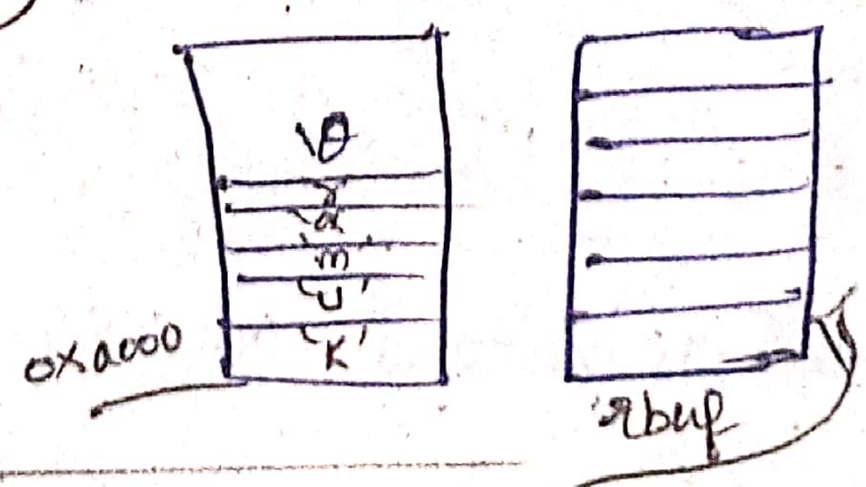
sys_open( )



read (fd, rbuf, 10)

write (fd, buf, strlen(buf))

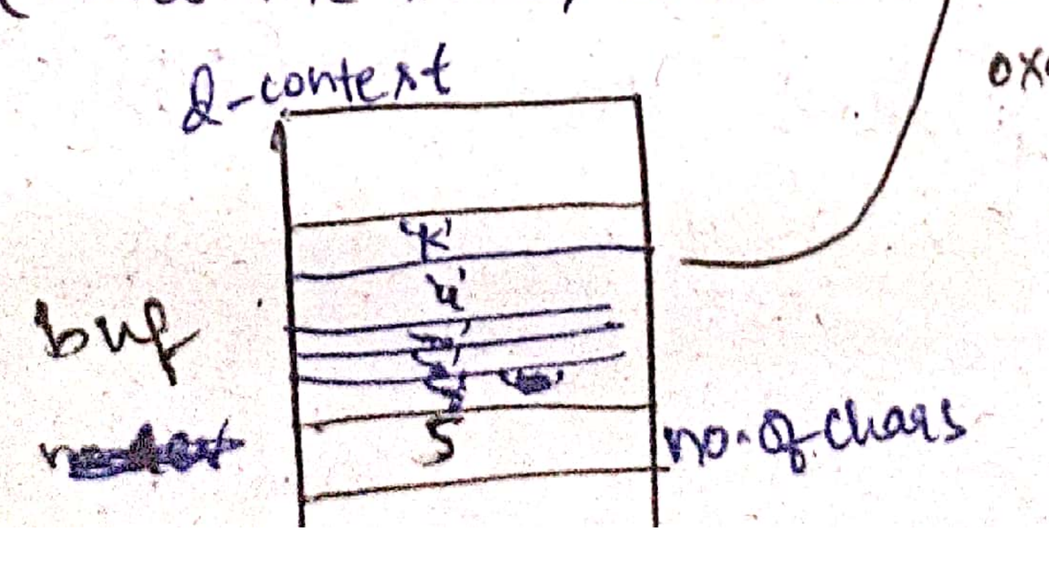
3 0xa000 5



---

mychar_write(struct file *file, const char *buf, size_t len, loff_t * offset)

0xd000 0xa000 5 0



mychar_read( )

{ ≡ }

10/02/23

* Blocking Mechanisms in Drivers

⇒ The Blocking mechanism can be implemented in a driver by using wait queue objects.

→ This Blocking mechanism is implemented in driver level by using wait event calls with proper conditions.

⇒ For wait queue objects (or) predefined structures defined in kernel. This structure is of type wait_queue_head_t type.

Q) Find where the structure is defined?

A) home/engineer/linux/include/linux/wait.h

→ First, we need to initialize this wait queue object either statically (or) dynamically.

→ linux/wait.h header file should be included.

statically :-

→ For static ~~declaration~~ initialization, we have to use a predefined macro.

~~declare_que~~

DECLARE_WAIT_QUEUE_HEAD (~~object~~ queue object name);

↳ Global declaration

⇒ For dynamic initialization, first declare a variable of type wait_queue_head_t mywq;
~~mywait queue~~;

⇒ This can be initialized under fun^n

⇒ init_waitqueue_head (&mywq);

↳ address of that declared variable

local declaration in (initialization driver routine).

⇒ The wait_event() & wait_event_interruptable() calls behaves as a blocking call (or) unblocking that depends upon the condition. So, before executing this call, it verifies the condition. If the cond^n is true, this will unblock & executes the remaining code. If the cond^n is false, it behaves as a blocking call until it executes wakeup() & wake_up_interruptable() on same wait queue object. & before wait_event() (or) wait_event_interruptable() execution it verifies the condition again.

wait queue object variable name

⇒ wait_event (wq, condition); → It goes into uninterruptable deep sleep.

⇒ wait_event_interruptable (mywq, cond^n);

⇒ wake_up (&wq);

⇒ wake_up_interruptable (&wq);

↳ address of witqueue object variable.

case (i) wait_event (wq, cond) (T)

executes normally (unblocking cond^n)

case (ii) wait_event (wq, cond) (F)

blocks

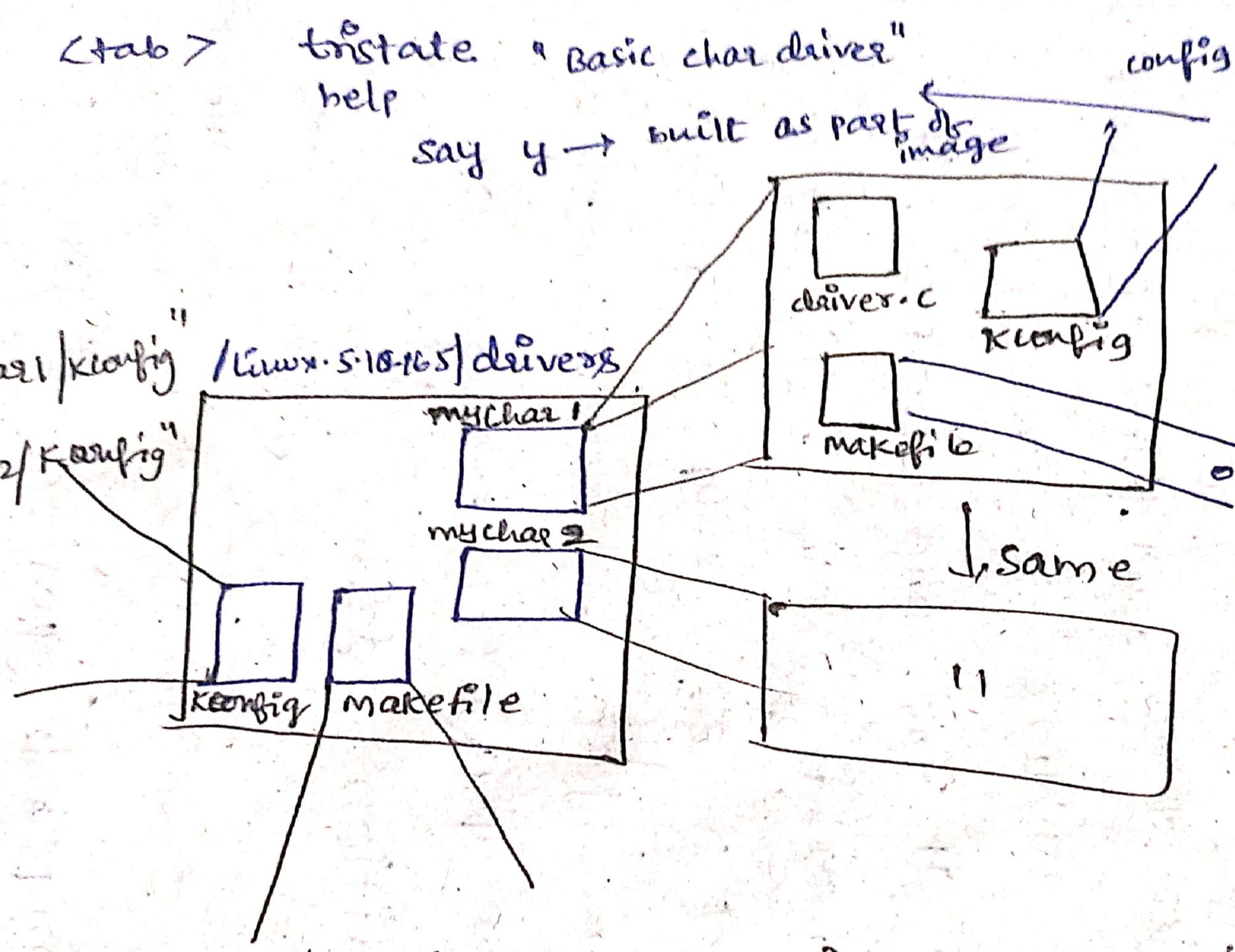
① check the cond again

(T) Unblock

(F) Block

wake_up (&wq);

How to add your own driver into kernel source code



obj-$(CONFIG_MYCHAR1) += mychar1/

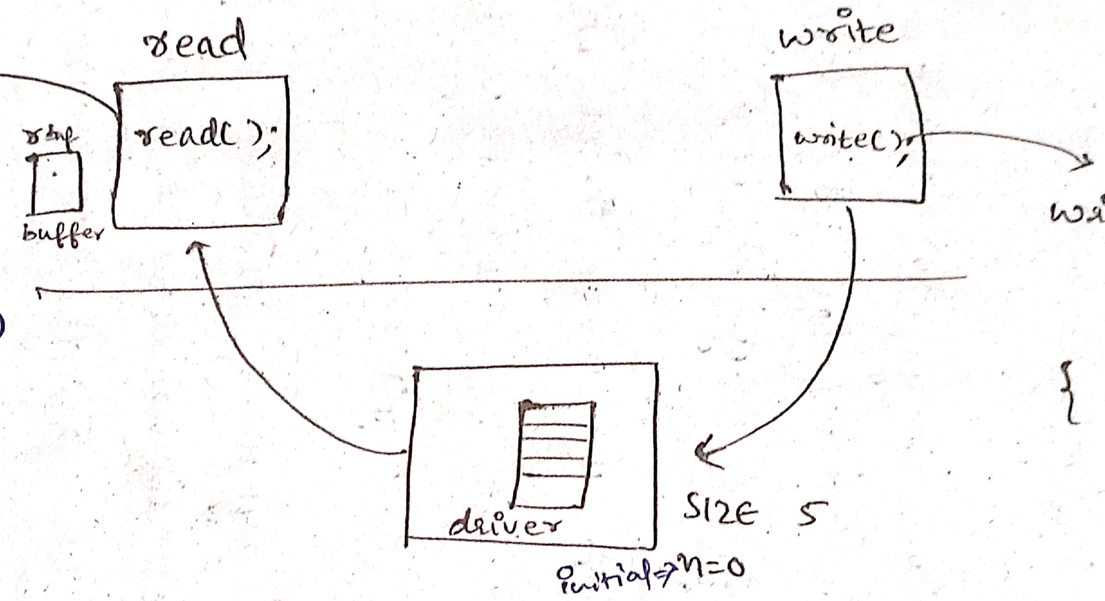
obj-$(CONFIG_MYCHAR2) += mychar2/

<tab> tristate "Basic char driver"
help
say y → built as part of image

config MY_CHARDEV

obj-$(CONFIG_MY_CHARDEV) += driver.o

```
read_driver_routine (  ,  , len,  )
{
  wait_event_interruptible (rwq, n>0)
    if (n < len)
      uc = n;
    Else
      uc = len;
  copy_to_user (kbuf -> user buf, uc);
      n = n-uc;
  wake_up_interruptible (&wwq);
    return uc;
}
```




```
write_driver_routine( struct file *fil
  const char *buf, size_t len, loff_t
  *offset
{
  wait_event_interruptible (wwq, n<SIZE
    a = SIZE - n;
    if (len > a)
      len = a;
  copy_from_user (user buf -> kernel buf,
                  len);
    n = n + len;
  wake_up_interruptible (&rwq);
  return len;
}
```

⊛ Dynamic creation of device files :-

⇒ The methods of managing device nodes or device files became clumsy & difficult. To overcome this, we have to create the device files on the fly when we load the driver. ~~For~~

⇒ For dynamic creation they have created udev method which creates the device nodes on the fly as they are needed.

⇒ In udev ⇒ u stands for user & indicates that the device nodes must be created, removed & modified from user space.

⇒ udev handles the dynamic creation of device nodes but it cannot ~~dis~~ handles the discovery of hardware devices & its management device nodes (major number & minor no & file to be created).

⇒ HAL (Hardware abstraction layer) (Demon process) uses dbus (device bus) infrastructure. It maintains a dynamic database of all connected hardware devices & is closely coupled to the hardplug facility. ~~The~~ devices are added or removed from the system, working with hardpluggable subsystem, the udev acts upon the notification of

events to create & remove device nodes.

* The information necessary to create them with the right name, major no., minor no. & permissions etc are gathered by examination of information already registered in sysfs file system & a set of configuration files.

⇒ udev program gets loaded with initramfs image.

external device (pendrive)
↓
Hardware
↓ info
hot plugable facility
↓
HAL
↓ (kernel)
device driver load
→ sysfs → entries will gets created
(demon process) ← udev
↓ info
creates device node/device file.

⇒ we need to create an entry in sysfs file system when the driver is loaded. For this,

① To create an entry in sysfs file system by using below kernel func^c. [linux/device.h]

class_create( );
↓ on success
it will create an entry in sys/class
& creates a Kernel object ~~structure~~ of type struct class *.

// used in

⇒ struct class * class_create (struct module * owner, const char * name);

on success Entry will created sys/class/<name>

Address of driver by using THIS_MODULE

" Entry name wish to create "

⇒ opp of the above fun^c is class_destroy

⇒ void class_destroy (struct class *class);
<name>

// used in cleanup fun^c

// when this class gets success, the entry in sysfs & kernel object gets destroyed.

⇒ After creating the class, now we have to create device file by using

⇒ struct device * device_create (struct class * cls, struct device * parent, dev_t devno, const char name, ...);

on sucess it will create an Entry in /dev/_ with major & minor no. combination.

⇒ (if it has no parent device file object or we are creating it for the 1st time, 2nd argument should be NULL)

⇒ If we don't have parent file object, we need to pass it as NULL, otherwise pass that device file base address.

→ opposite of the above fun^c is

void device_destroy(struct device *dev);

// If it is successful. the entry in /dev directory gets deleted.

11/02/23

* Proc virtual File System Entries :-

⇒ This maintains each process running information in a separate folder

⇒ This file system reads the entries from the kernel & maintains it.

⇒ From a driver point of view, we need to update some info in user space i.e., in proc virtual file system. It maintains some kernel information entries like ① /proc/devices contains registered character & block driver's major numbers

② /proc/iomem → it maintains on system physical ram & bus device addresses.

③ /proc/io ports → it gives the each peripheral port address registered with the kernel in X-86 system.

④ /proc/interrupts → it maintains registered interrupt request no.

⑤ /proc/kallsyms → it maintains dynamically loaded kernel symbol table

⑥ /proc/modules → it maintains currently loaded modules info.

⑦ /proc/cpuinfo → it maintains info about CPU's on the system.

⇒ proc file operation structure

① Define the ~~file~~ proc virtual file system operation structure which is of type struct proc_ops which contains func pointers for read, write, release & so-on ---.

⇒ proc_open(), proc_write(), proc_release() ---
func pointer

⇒ This structure must be initialized with corresponding func's to create entry in proc virtual File system.

⇒ To create a directory under proc virtual FS we have to use a func proc_mkdir. It will creates a directory of type struct proc_dir_entry

* proc_mkdir (const char * name, struct proc_dir_entry * parent)

name of directory, address of parent directory

⇒ In this we need to create a file by using proc_create ( ).

return type.

name of file

mode

⇒ struct proc_dir_entry * proc_create (const char *name, umode_t mode,

Struct proc_dir_entry *parent, const struct proc_ops *ops);

directory name in which u need to create file

add of performing fun pointer

⇒ This we have to do under initialization func

⇒ To delete the entry i.e complete directory, we have to use proc_remove ().

~~void~~

~~struct proc_dir_entry * pr~~

void proc_remove (struct proc_dir_entry *);

⇒ we must include linux / proc_fs.h header file

Verify example in embetronicx

* ioctl :- (i/p o/p control)

⇒ To change the device configuration from User space (or) to read ~~some~~ device specific info from driver

⇒ we have to define our own ioctl commands by using predefined macros.

_IO (or) _IOW (or) _IOWR (or) _IOR.

⇒ we must include include/uapi/asm-generic/ioctl.h header file

#define _IO (type, nr) _IOC (_IOC_NONE, (type), (nr), 0)

#define _IOR (type, nr, size) _IOC(_IOC_READ, (type), (nr), (_IOC_TYPECHECK(size)))

#define _IOW (type, nr, size) _IOC(_IOC_WRITE, (type), (nr), (_IOC_TYPECHECK(size)))

#define _IOWR (type, nr, size) _IOC(_IOC_READ|_IOC_WRITE (type), (nr), (_IOC_TYPECHECK(size

⇒ we must define a magic number for these set of commands. Each command will have a command number also.

type (magic no.)

An 8-bit number, often a character literal, specific to a subsystem or driver, and listed in Ioctl numbers.

nr (command no.)

An 8-bit no. identifying the specific command, unique for a given value of 'type'.

datatype

The name of the data type pointed to by the argument, the command number encodes the sizeof (data_type) value in a 13-bit (or) 14 bit integer, leading to a limit of 8191 bytes for the max size of the argument.

NOTE:-

Do not pass sizeof (data_type) type into _IOR/_IOW/_IOWR, as that will lead to encoding sizeof (sizeof(data_type)) i.e., sizeof (size_t). _IO does not have a data_type parameter.

⇒ ioctl () system call.. (from user space)

⇒ #include <sys/ioctl.h>

⇒ int ioctl (int fd, unsigned long request, ...);

⇒ This ioctl command is a total of 32 bit value so

⇒ This 32 bit is a combination of type no. ~~(type)~~, command no, direction & type.

⇒ The ioctl commands defined for that particular driver should be accessable even for the application

⇒ <u>Steps</u>

① we need to define ioctl commands in separate header file. This header file should be accessed to both application & driver.

② Invoke ioctl() from application to perform ~~the~~ particular operation. This will invoke the corresponding ioctl driver routine to perform the operation. Now this operation is executed (or selected) based on command. For each & every command we need to define the operations by using switch case (in user & kernel level).

⊛ UART ⊛ (Universal asynchronous receiver transmitter)

* **communication:-** It is nothing but exchanging the data between two devices.

⇒ A communication based device will have memory in terms of hardware registers. we can see serial communication & parallel communication

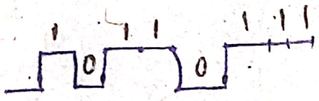
⇒ In serial communication, data will be transmitted bit by bit over a single physical line.

⇒ In parallel communication, data will be transmitted on 'n' no. of physical lines at a time.

⇒ UART - comes under ~~recei~~ Asynchronous receiver & transmitter.

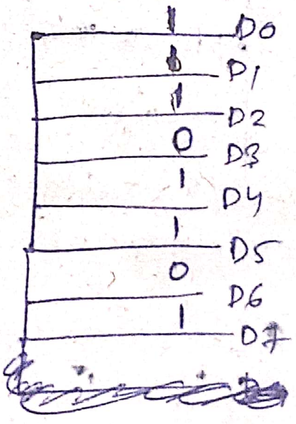
⇒ HWR serial

10110111 →



HWR parallel GPIO's.

8 physical lines



⇒ X-86 Archictecture supports IO memory mapping

⇒ ARM " supports memory mapping I.O

UART

start bit → active low

stop bits → active high

parity bits → NO parity
Even "
odd "
sticky " → 0
→ 1

Data bits
5/6/7/8

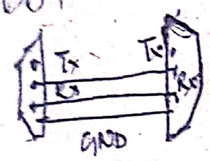
baud rate

2400
4800
9600
⋮
115200
1Mbit/sec
2Mbit/sec

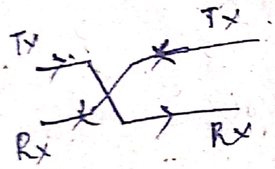
+15V
+5V

-5V
-15V

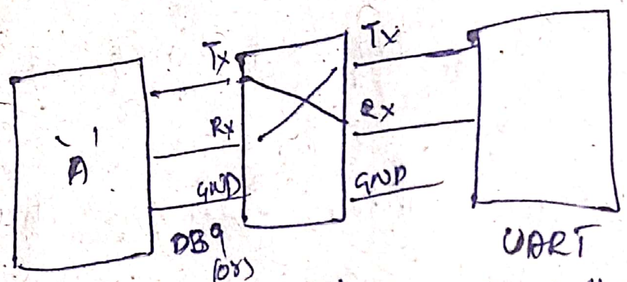
DB9



Straight cable



Cross connection



UART (controller) DB9 (or) 3 physical cables UART (controller)

To understand about UART

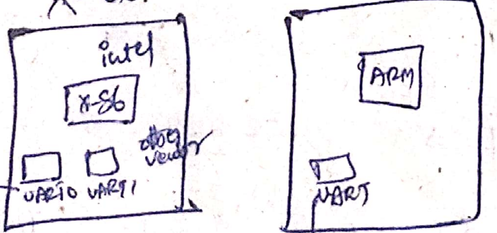
→ protocols

→ interfacing through boards (configuration)
manuals

→ SOC User data manual
H/w Register description
(addresses)
(mapping info) etc

HWR

→ Data registers
→ control "
→ status "



0x375 IOmem
0x27F
(Specific address) to map

MEMIO
Use mapped registers

13/02/23

⇒ use $ cat /proc/ioports & search for UART device.

⇒ Your UART controller must be mapped with the registers (specific address) $ 0X3F8 for (IRQ4), 0x2F8 (IRQ3), 0X3E8 (IRQ4), 0X2E8 (IRQ3)

$ → Hexa decimal 0X

⇒ when we write the driver code for a device,

① Set the communication parameters in the corresponding hardware registers. Before this we need to request the kernel to allocate the resources (like Hardware registers (HWR))

② Register IRQ line no. with interrupt handler func.

③ Disable the interrupts. we have not started the communication.

→ when the communication starts, then only we have to enable the interrupts. that means we need to open/run the application from the user space.

Q) What is an interrupt?

A) Raising an interrupt is ntg but activating the IRQ line no. to send (or) to receive the data (or) to handle the errors. we need to

write interrupt handler fun^c

interrupts.

⇒ This interrupt handler fun^c should be registered with the corresponding IRQ line no. with the kernel.

⇒ This registration can be done by using a Kernel fun^c request_irq ( ).

⇒ To access the driver resources, we have to use fun^c request_region ( ). ⇒ It will give virtual addr

⇒ When this call is success, an entry gets created in /proc/iomem

⇒ In x-86 architecture, to update the data to read the data from Hardware registers (HWR) we have to use below functions

IN { inb() outb() / inl() outl() / inw() outw() } → STR (store) 8/16/32.

8/16/32 bit

↓

This functions internally invokes an assembly instruction IN

⇒ Ly for ARM architecture, we have

readb() writeb()
readl() writel() → kernel functions
readw() writew()

⇒

(*) NOTE :-

⇒ To access HARDware resources (Registers) we have to use another kernel functions request - ?

⇒ For UART communication, the IRQ line no is 4 (Four).

⇒ when you open the application, then only we will enable the receive interrupt.

⇒ when you perform write operation from the application, then only we have to enable transmit ~~application~~ interrupt.

⇒ when an interrupt is raised PIC will identify ~~from~~ which device has raised the interrupt based on IRQ line number & informs the CPU to execute the corresponding interrupt handler fun^c. and the CPU will suspend the normal process execution & executes the interrupt handler fun^c. this means interrupt is serviced.

⇒ In interrupt handler fun^c, for transmit interrupt, data will be copied from driver buffer to THR register & for receive interrupt, the data received through RX line in RBR register is copied to driver buffer.

Now, the data copied in THR register will be transmitted outside through Tx line with (or) without FIFOS. In this case, for every byte transmit, it raises transmit interrupt & for every byte received, it raises receive interrupt. ~~For handling the~~ ~~errors~~, we need to write the code for handling the errors in interrupt handler fun

⇒ The opposite of request_region() is release_region()

This will delete the entries from /proc/ioports

⇒ The opposite of request_irq() is free_irq()

~~which~~ ~~was~~

it releases the ~~irq~~ interrupt handler fun() ~~for~~ corresponding irq line no. for particular driver from interrupt vector table.

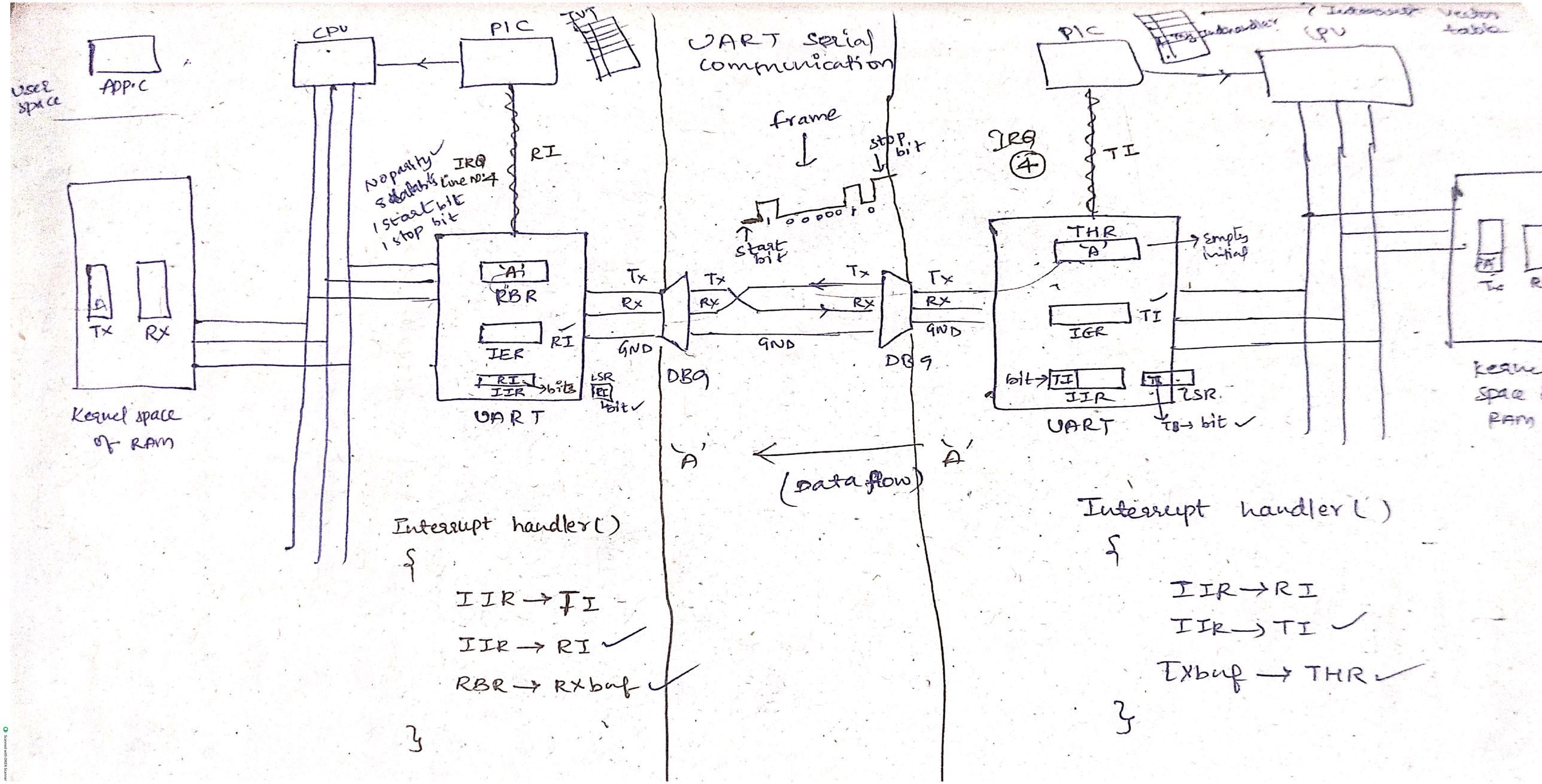
⇒ sometime GPIO pins are also used as IRQ line no.

⇒ when an interrupt is raised, interrupt should be served as early as possible.

⇒ Based on this, the data processing activity should be done over a period of time, for this the interrupt handler is divided into 2 parts

① Top Halves ② Bottom Half / Halves.

Scanned with OKEN Scanner

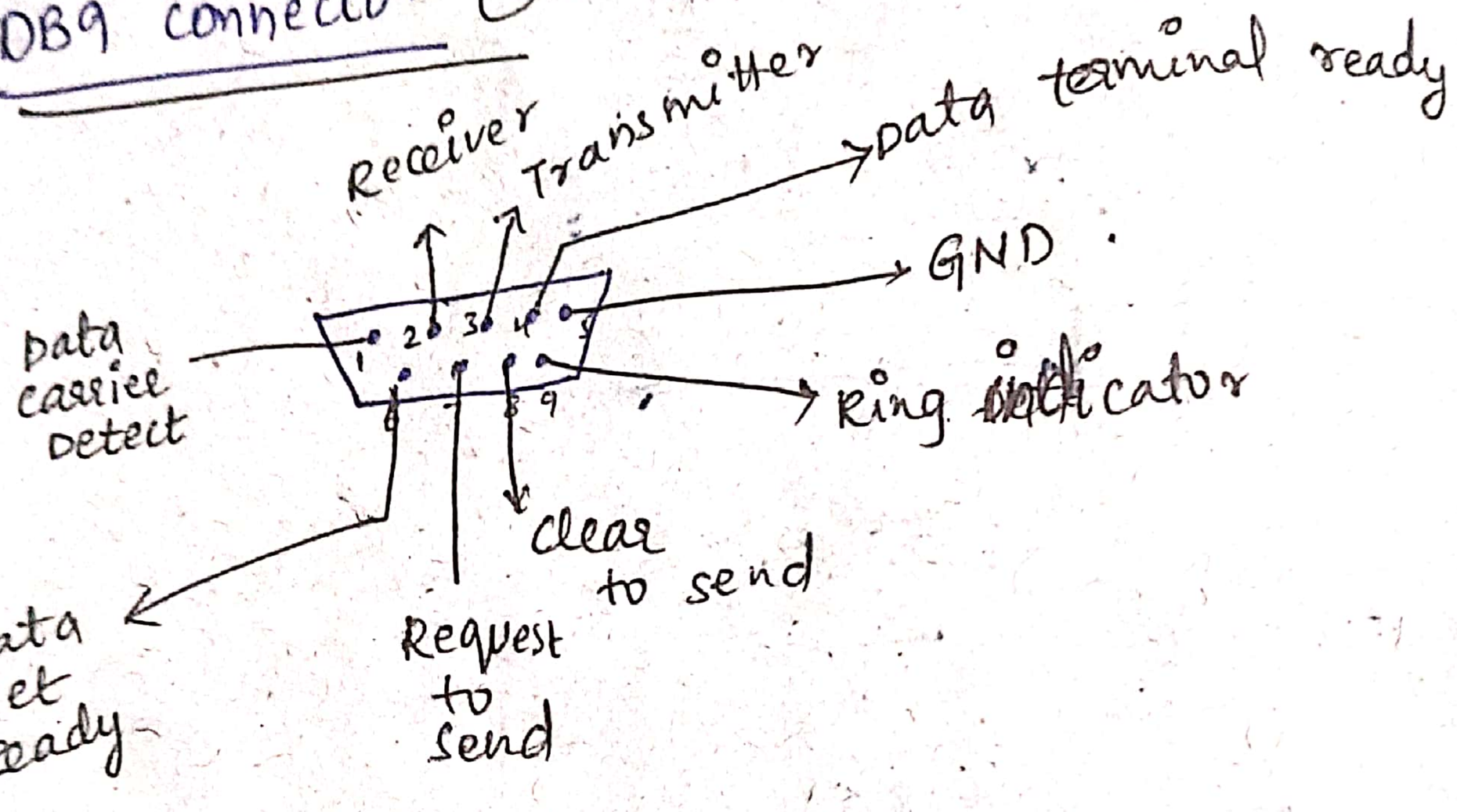

User space
APP.C
CPU
PIC
IVT
UART Serial communication
frame
stop bit
start bit
No parity
1 start bit
1 stop bit
IRQ line No:4
RI
TI
IRQ 4
Kernel space of RAM
TX
RX
'A'
RBR
IER
RI
IIR
bits
LSR
RI bit
UART
Tx
Rx
GND
DBG
THR
'A'
empty initial
IER
TI
bit
IIR
LSR
TB bit
UART
Kernel space RAM
'A'
(Data flow)
'A'
Interrupt handler()
{
IIR → TI
IIR → RI
RBR → RXbuf
}
Interrupt handler()
{
IIR → RI
IIR → TI
Txbuf → THR
}

→ In top half mechanism 1st we will verify the interrupts

→ In Bottom half (we will schedule the task) mechanism, we will perform data processing activity.

tasklets, workques etc.

DB9 connector



UART Hardware Registers (12):-

1. RBR (Receive Buffer Register)
2. THR (Transmitter holding register)
3. IER (Interrupt Enable register)
4. IIR (Interrupt identification register) (R)
5. FCR (FIFO control register) (WR only)
6. LCR (Line control register) (R & W)
7. MCR (Modem control register) (R/W)
8. LSR (Line status register) (R)
9. MSR (Modem status register) (R)
10. SCR (scratch Register) (R & W)
11. DLL (LSB)
12. DLM (MSB) } Divisor latch registers (R & W)

11/08/23

Brief flow of UART

read

copy to user

↑

Rx line → RBR → RxBuf.

inb()

APP

Driver

RI ↑ ↓ TI (or) error

Device

write

copy from user

(TI)

THR Empty ✓ → interrupt ↑

interrupt handler()

← (4)

TXbuf → THR

outb( )

request_irq

←

Modem

AUX → (1)

communication ~~protocols~~ parameters

LCR → Data

parity

stop

DLAB bit

(1) → To set baud rate

↓

DLL
DLH } → Baud rate set

(or)

(0)

DLAB

└ THR | RBR / IER.

IER → TI/RI/LS Interrupt.

FCR (FIFOS)

1

4

8

14

request_region()

→ To access Hardware registers

(HWR)

Data ⇄

control registers

Status " → IIR

LSR

putty
minicom
Picocom } apps

download

Host root directory

/boot → vmlinuz-5.15.40, System.map, initramfs, .config.

[& update-grub]

To delete unwanted kernel sources

Delete

⊛ Kernel Timers :-

→ Time is one of the most used resources in the memory. It is used to do almost everything such as differ the work, sleep, timer scheduling etc.

→ There are 2 categories of time :-

① Absolute time.

② Relative time.

① Absolute time :- It is used to know the date & time of the ~~data~~ day.

② Relative time :- It is used to scheduling the task or to create delays ~~in high precison~~.

→ For Absolute time, we depend on a hardware chip called real time clock (RTC).

→ For ~~scheduli~~ Relative time, it uses processor timers. These are from kernel point of view we will call it as kernel timers.

→ Kernel timers are classified into 2 different parts. They are :-

① standard timer (or) system timers - (msec)

② High Resolution timer. (μsec)

(jiffies → 32 bit machine
jiffies_64 → 64 bit machine)

① Standard timers

unsigned long int
↓ (64 bit value)

→ These are measured in terms of jiffies.

→ A Jiffy is a kernel unit of time declared in a linux/jiffies.h header file.

→ Jiffies is defined in terms of constants called Hertz's which is the no. of times jiffies is incremented in one second. Each increment is called a tick. Indirectly we can say hertz represents the size of a jiffy. Hertz'es depends on the Hardware & on the Kernel version and also determines how frequently clock interrupt fires. so, this is configurable on some architectures & fixed on some others it means that jiffies is incremented hertz'es time for every second.

⇒ If hertz = 1000, then it is increment 1000 times. that means 1 tick it is for 1/1000 sec

⇒ Depending on the platform, jiffies can lead to overflow. In 32 bit machine it is different & in 64 bit machine it is different

⇒ Jiffies can be incremented from system boot up time.

⇒ To generate the delay we will take the help of jiffies.

⇒ To generate the delay in terms of micro & nano seconds, we will use high resolution timers.

⇒ The timer is represented by a kernel data structure struct timer_list which is defined in linux/timer.h header file.

⇒ we can schedule a func (or) task to be executed over a period of time by using timers.

⇒ The global kernel timers are maintained as a running queue to which we need to add our kernel timer list object after initialization

⇒ we can set up the timer (or) initialize the timer by using timer_setup( ).

1st i/p ⇒ base address of the timer_list variable

2nd i/p ⇒ func's base address which executed and expiry time

3rd argument ⇒ flags.

⇒ This internally invokes init_timer func

⇒ on success it will return ntg, the func will be added to the global timer list.

→ Setting the expire time when the timer is initialized, we need to set its expiry before the call back gets fired that can done by using mod_timer( )

1st arg → timer_list variable's base address

2nd arg → unsigned long expires ~~(jiffies + expiry)~~

In 2nd arg we need to pass jiffies + the equivalent jiffies value for expiry.

→ To get the jiffies value for the given time can be done by using ~~millisec~~ msecs_to_jiffies( ).

i/p → time in terms of milliseconds.

→ For releasing the timer we can call del_timer( )

```
void del_timer(struct timer_list *timer);
```

→ The del_timer returns void, it returns zero(0) on an inactive timer & 1 on active timer.

→ To check the timer is running or not we have to use timer_pending( ).

→ Jiffies is a variable which is defined globally & can be accessed directly in any driver. (we will have read only permissions).

② High Resolution Timer

⇒ The standard timers are less accurate, don't suit for real time applications.

⇒ This HRT are introduced in 2.6 kernel version

⇒ To enable (or) to check the support of High resolution timer, we need to enable an option in Kernel configuration i.e., CONFIG_high_res_timers

~~should~~ [it should be selected as y)

⇒ By using this HRT, we have resolutions/support in microseconds upto nanoseconds depending upon the platform compared to milliseconds on the standard timers.

⇒ standard timers depends upon jiffies, whereas HRT is implemented on Ktime.

⇒ HRT is architecture dependent.

⇒ we must include linux/hrtimer.h header file

⇒ we must use a predefined structure

struct hrtimer

⇒ we can initialize the hrtimer by using

void hrtimer_init ( ); //initialization func.

1st arg ⇒ struct hrtimer variable's base address

2nd arg ⇒ clock_id_t which_clock // for monolithic use CLOCK_MONOTONIC

3rd arg ⇒ enum hrtimer_mode mode

REL

→ Before hrtimer initialization, we need to set up ktime which represents the time duration.

→ starting the timer can be done by using

int hrtimer_start ( ).

1st arg → struct hrtimer variable's base address.

2nd arg → Ktime_t time

3rd arg → const enum hrtimer_mode mode.

Here, mode represents the expiry mode → for absolute time value ~~HRtimer_mode_abs~~ HRTIMER_MODE_ABS

→ For relative time value use HRTIMER_MODE_REL

⇒ For hrtimer cancellation, we have to use below func.

int hrtimer_cancel (struct hrtimer *timer);

→ we can dependently check whether the timer callback is still running by using

int hrtimer_callback_running (struct hrtimer *timer);

return value 0 → Inactive
1 → Active

① verify in /boot/ config file whether there is support of HRT (or) not

② & also /proc/ configs.gz

③ By looking into the timers list.

$ use $cat → $ cat /proc/timer_list

④ By using clock-gettime

⇒ we can generate the desired delays for nanosec, microsec, milliseconds by using below fun cs.

ndelay (unsigned long nsec); // For nanoseconds

udelay (unsigned long usec); // For microsec

mdelay (unsigned long msec); // For millisec

⇒ These functions works under atomic context such as interrupt service routine i.e., they can't sleep, they can't be scheduled thats why they go into busy loops.

NOTE:

⇒ we shouldn't use any ~~data~~ blocking calls under interrupt service routine & it shouldn't be delayed & it should be executed as soon as possible.

⇒ To set the hrtimer value we have to use ~~ktet~~. ktime_set( ).

1st i/p → starting value.

2nd " → Last value.

⇒ This will return the ~~base address of~~ ktime.

15/02/23

* Exporting kernel objects to sys FS file system through drivers :-

⇒ sys FS is a ram based file system initially based on ramfs. It provides a means to export kernel data structures their attributes & linkages b/w them to user space.

⇒ sys FS is the commonly used method to export system information from kernel space to user space for specific devices.

⇒ The sys FS is tied to the device driver model of the kernel. The heart of the sys FS model is kobject. Kobject is the glue that binds the sys FS & the kernel.

⇒ sys FS is always compiled if config_sysfs is defined.

⇒ For every kobject that is registered with the system a directory is created for it in sys FS. The directory is created as a subdirectory of the kobject's parent expressing the internal object hirearchies to user space.

* Attributes :

⇒ Attributes can be exported for kernel objects in the form of regular file in the file system.

⇒ sysfs forwards the file io operations to methods defined for the attributes providing a means to read & write kernel attributes. So, attributes should be ASCII textual files preferrably with only 1 value per file.

⇒ An attribute definition is simply struct attribute

```
struct attribute
{
    char * name;
    struct module * owner;
    mode_t mode;
};
```

⇒ `int sysfs_create_file (struct kobject *kobj, struct attribute *attr);`

To remove the file

```
void sysfs_remove_file (struct kobject *kobj,
                        struct attribute *attr);
```

Scanned with OKEN Scanner

⇒ A bare attribute contains no means to read or write the values of the attributes.

⇒ subsystems are encouraged to define their own attribute structures & wrapper functions for adding & removing attributes for specific object type.

For example:- The driver model defines struct device_attribute.

```
struct device_attribute
{
    struct attribute attr;
    ssize_t (*show) (struct device *dev,
                     struct device_attribute *attr, char *buf);

ssize_t (*store) (struct device *dev, struct device_attrib
                  -ute *attr, const char * buf, size_t coun
};
```

⇒ The functions associated with it is

```
int device_create_file( struct device *dev, const struct
                        device_attribute *attr);
```

To remove the file

```
void device_remove_file (  "  ,  "  );
```

⇒ we have a helper attributes i.e.,

define DEVICE_ATTR(_name, _mode, _show, _store)

⇒ struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)

⇒ To create a directory under sys fs file system under specific group we have to use below function

⇒ Kobject_create_and_add(directory name, objects name in which u want to create it);

in kernel folder → kern_kobj

in firmware → firmware_kobj

⇒ Kobject_put() func will reduce the reference count. It will also deletes the object directory.

⇒ Th

derekmollog.ie

work differing mechanism :-

→ Differing is a method by which you schedule a piece of work to be executed in future. There are 3 differing work mechanism kernel will provide. * Bottom Half mechanisms

① soft irq's → executed in an atomic context. (synchronization without any sleep)

② Tasklets ⇒ Executed in an atomic context

③ work queues ⇒ " " a process "

①
⇒ A soft irq (or) a software interrupt is a differing mechanism used only for very fast processing in an interrupt context.

⇒ There are only networks & block device subsystems using soft irq's.

② Tasklets are an instantiation of soft irq's & will be sufficient in almost every case.

⇒ soft irq's are a form of bottom half processing that run in interrupt context & are suitable for long running non-blocking handlers.

⇒ They are executed once all interrupt handlers have completed, before the kernel resumes the scheduling processes so sleeping is not allowed.

⇒ A soft irq is executed but remains interruptable so that the execution can be pre emitted by any top of.

⇒ soft irq's cannot be used by the device drivers because they are reserved for various Kernel subsystems because of this there is a fixed no. of soft irq's defined at Kernel compilation time.

⇒ ~~high soft irq~~

⇒ Each soft irq has a specific purpose

⇒ The following types are defined for the current Kernel version as

enum → Enumeration

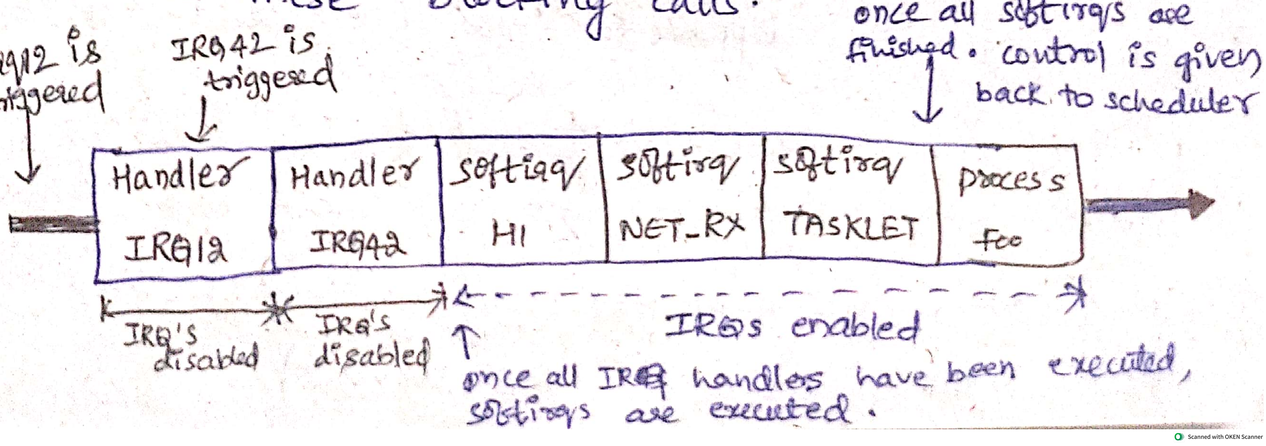
```
enum {
    HI_SOFTIRQ = 0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,
    NR_SOFTIRQS
};
```

→ 1st variable initialized & every variable gets initialized with same value if we use enum.

- HI_SOFTIRQ & TASKLET_SOFTIRQ → running tasklets
- TIMER_SOFTIRQ → running timers
- NET_TX_SOFTIRQ and NET_RX_SOFTIRQ → Used by the networking subsystem.
- BLOCK_SOFTIRQ → Used by the IO subsystem
- BLOCK_IOPOLL_SOFTIRQ → Used by the IO subsystem to increase performance when the iopoll handler is invoked.
- SCHED_SOFTIRQ → load balancing.
- HRTIMER_SOFTIRQ → Implementation of high precision timers.
- RCU_SOFTIRQ → implementation of RCU type mechanisms

⇒ The highest priority is the HI_SOFTIRQ type softirqs, followed in order by the other softirqs defined. RCU_SOFTIRQ has the lowest priority.

⇒ softirqs are running in interrupt context, which means that they can not call blocking functions. If the softirq handler requires calls to such functions, work queues can be scheduled to execute these blocking calls.



(*) Tasklets :-

⇒ A tasklet is a special form of defered work that run in interrupt context just like softirq.

⇒ Tasklets can be allocated dynamically & these can be used by device drivers.

(*) NOTE :-

(*) softirqs cant be allocated dynamically.

⇒ A Tasklet is represented by the kernel struct tasklet_struct. Tasklets are executed within the HI and tasklet softirqs. They are executed with all interrupts enabled but a given tasklet is guaranteed to execute on a single CPU at a time.

~~but~~

⇒ A pre initialized tasklet can be initialized as follows :-

MACRO

⇒ Statically declare_tasklet (tasklet structure variable name, tasklet func, tasklet data);

⇒ declare_tasklet_disable (name, func, data);
(same as above)

⇒ There is one difference b/w these 2 functions
1st one creates a tasklet already enabled & ready to be scheduled without any other func call; done by setting the count field to zero (which is the member of struct tasklet)

whereas, the 2nd one creates a tasklet by disable & the count field is set to 1. To enable this tasklet we have to call tasklet_enable ( ) before the tasklet is scheduled.

→ For declaring a tasklet dynamically we have to use a func tasklet_init ( ).

⇒ void tasklet_init (struct tasklet_struct *t, void (*func) (unsigned long), unsigned long data);

NOTE

⇒ Enabling & disabling.

⇒ If the count field is zero, the tasklet is disabled that means we have to enable for execution.

→ Enabling & disabling the tasklet :-

⇒ To enable the tasklet we have a kernel func

```
void tasklet_enable ( tasklet, base address);
```

structure → struct tasklet_struct *

⇒ This func simply enables the tasklet

For old kernel version, we have to use a func

```
void tasklet_hi_enable ( struct tasklet_struct *);
```

⇒ To disable the tasklet, we have to use a kernel func

```
void tasklet_disable (struct tasklet_struct *);
```

⇒ This func returns only when the tasklet has ~~complete~~ terminated its execution.

* Scheduling the tasklet :-

⇒ There are 2 functions used to schedule the tasklet based on priority normal & high.

For normal priority :-

```
void tasklet_schedule (struct tasklet_struct *t)
```

For high priority :-

```
void tasklet_hi_schedule (struct tasklet_struct *t);
```

⇒ The Kernel maintains normal & high priorities tasklets in 2 different lists

⇒ tasklet_schedule adds the tasklets into normal priority list scheduling the associated softirq with a TASKLET_SOFTIRQ flag

⇒ with tasklet_hi_schedule, the tasklet is added into high priority list scheduling the associated softirq with a ~~HIGH_SO~~ HI_SOFTIRQ flag.

⇒ Latency :- The response time b/w raise time & execution time.

→ The high priority tasklets are meant to be used for soft interrupt handlers with low latency requirements.

→ The calling tasklet_schedule on a tasklet already scheduled, but whose execution has not started will do nothing resulting in the tasklet ~~should~~ being executed only once.

→ tasklet_schedule can be called in a tasklet that means it can be rescheduled by itself.

→ high priority tasklets are executed before normal ones.

→ Don't schedule all tasklets on high priority that will degrade the performance of the system by increasing the system latency.

→ You can stop a tasklet by using tasklet_kill ().

So, if a tasklet is under execution, it will wait until its completion before killing. If it is scheduled & it is not in execution will be killed.

```
void tasklet_kill (struct tasklet_struct *t);
```

→ NOTE

→ Tasklet should not sleep.

→ work queues can sleep.

(*) Work Queues :-

⇒ work queues are build on top of kernel threads.

⇒ work queues are executed under process context

⇒ A kernel thread called worker picks up the item from the work queue & executes the funcs associated with those items.

⇒ Kernel maintains a global work queue list

⇒ The task scheduled by the work queue will be added to the global work queue list.

⇒ workqueues are suitable for long running & lengthy task, they can be used inside & outside of the context of interrupt handling

⇒ If there is no work queue is scheduled, the worker thread will be in idle state (sleep)

⇒ worker threads are controlled by worker POOL which take care of the level of concurrency (simultaneously running worker threads) & the process management

⇒ We can see there are 2 types of workqueues

① struct work_struct → it schedules a task to run at a later time.

② struct delayed_work → it schedules a task to run after atleast a given time interval.

→ The delayed work uses a timer to run after a specified timer interval.

→ The calls with this type of work are similar to those for struct work_struct but has _delayed in the function's name.

→ There are 2 types of macros that can be used

ONE That declares & initializes the work item & the other one that only initializes the work item.

linux/workqueue.h → header file.

→ we have a macro DECLARE_WORK (name, void (*function) (struct work_struct *));

For delayed one,

DECLARE_DELAYED_WORK (" , ");

2nd macro

INIT_WORK (struct work_struct *work, void (*func) (struct work_struct *));

For delayed

INIT_DELAYED_WORK (struct delayed_work *work, void (*func) (struct work_struct *));

DECLARE_WORK() & DECLARE_DELAYED_WORK()

declare & initialize a work item, &

INIT_WORK() and INIT_DELAYED_WORK() initialize an already declared work item.

⇒ The following sequence declares & initiates a work item:

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

void my_work_handler (struct work_struct *work);
DECLARE_WORK (my work, my_work_handler);
```

or, if you want to initialize the work item separately,

```
void my_work_handler (struct work_struct *work);
struct work_struct my_work;
INIT_WORK (&my_work, my_work_handler);
```

⇒ Once declared & initialized, a work instance can be scheduled into workqueue through schedule_work() & schedule_delayed_work(). This func enqueues the given work item on the local CPU workqueue, but does not guarantee its execution on it. It returns true if the given work is successfully enqueued, or false if the given work is already found in the workqueue. Once queued, the func associated with the work item is executed on any of the available CPUs by the relevant kworker thread.

schedule_work);

schedule_delayed_work(struct delayed_work *work,
unsigned long delay);

⇒ you can wait for a workqueue to complete running all of its work items by calling flush_scheduled_work():

```
void flush_scheduled_work (void);
```

16/02/23

⇒ we can schedule the work on a particular CPU scheduled_delayed_work_on().

```
int schedule_delayed_work_on(int cpu, struct delayed_
work *work, unsigned long delay);
```

⇒ If we want to schedule the work on all processors CPU we can use schedule_on_each_CPU().

```
                                  func pointer
int schedule_on_each_cpu( void (*func) (struct work_
                                        struct *));
```

⇒ work queue is represented by struct workqueue_struct type, it can created by using below func

```
struct workqueue_struct * create_workqueue (const char*name);
```

⇒ This funⁿ uses one thread for processing in the system

```
struct workqueue_struct * create_singlethread_workqueue
        (const char * name);
```

⇒ This funⁿ uses single thread.

⇒ To add a task in the new queue, we can use queue_work() (or) queue_delayed_work()

```
int queue_work(struct workqueue_struct *queue,
        struct work_struct *work);
```

For delayed work, queue_delayed_work()

```
int queue_delayed_work(struct workqueue_struct *queue,
        struct delayed_work * work, unsigned long delay);
```

⇒ To wait for all item works to finish call flush_workqueue()

```
void flush_workqueue(struct workqueue_struct *queue);
```

⇒ To destroy the work queue call destroy_workqueue

```
void destroy_workqueue(struct workqueue_struct *q);
```

Scanned with OKEN Scanner

(*) I2C Protocol (*)

Inter Integrated Circuit

I2C, I^2C, IIC

⇒ It is designed by phillips semiconductors now it is called NXP semiconductors

⇒ Low Band width / low frequency devices are connected to this I2C.

⇒ It functions as Master-slave communication (or) I^2C known

⇒ Always the request comes from the master, but not from the slave.

⇒ It comes under 8 bit synchronous serial communication. It has 2 pins.

1. serial data line (SDA)
2. serial clock line (SCL).

⇒ Data will be transmitted along with the clock. Along with each data bit we can (it will transfer) see one clock pulse on clock line

⇒ The combination of SDA & SCL lines is known as I2C bus. To this I^2C bus, slave devices are connected to pull up resistors (connected to SDA line)

⇒ Slaves on these I2C bus will be in idle state / inactive state defaultly.

⇒ Each connected slave is identified by 7 bit (or) 10 bit address.

⇒ communication parameters are:-

- Start cond^n → To start the communication
- Stop " → To stop " "
- Acknowledgement → for the response
- NO Acknowledgement → No response
- speed (clock frequency setting.
- Data bits → How many bit? 8 bit (or) 16 bit
- 7 bit address (or) 10 bit address for identifying slave
- Direction bit etc - - -
  ↳ To read / write operation.

⇒ The slave devices that we can connect to I2C are:-

I2C EPROMS
" sensors.
" ADC
" RAM
" display
" PMIC
" RTC

⇒ Data is read by the master (or) slave when the clock is high state.

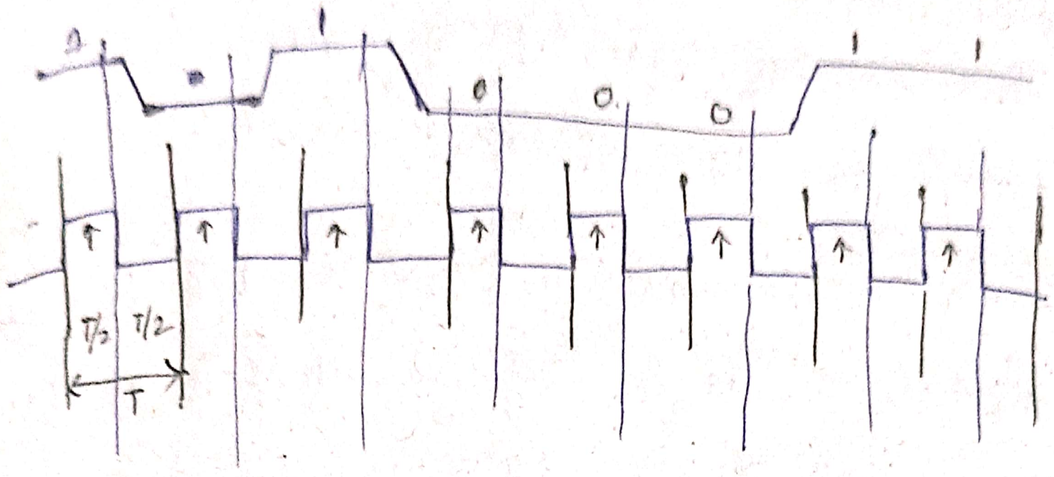
⇒ when the clock is low, data may change i.e, it won't consider the data.

⇒ In I2C (& SPI) communication, MSB bit will transfer first. In advance controllers we can change this options using registers (mentioned in data sheet).

Scanned with OKEN Scanner

Eg:- 0XA3

1010 0011

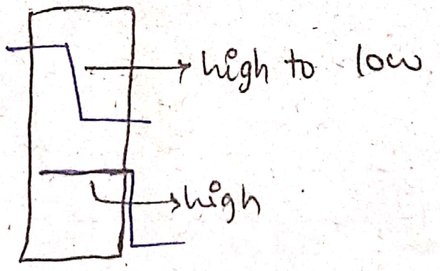


100 Kbits/sec
400 Kbits/sec
1.2 Mbits/sec
3.4 Mbits/sec
} → two way of communication

5 Mbits/sec → → one way of communication.

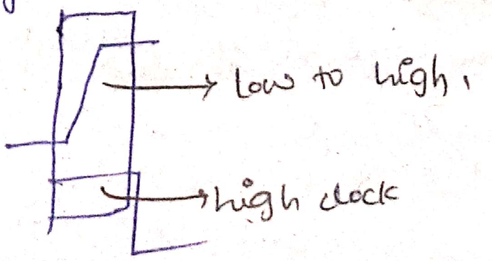
* <u>start condition</u> :-

⇒ A high to low transaction on SDA line when clock is high state. i.e., known as start condition. It is always generated by the master.



* <u>stop condition</u> :-

⇒ A low to high transaction on SDA line when clock is high state is known as stop condⁿ.



⇒ It is generated by master

* Acknowledgement :-

⇒ pulling the SDA line to low for 1 complete clock cycle is ntg but Acknowledgement. This can be generated by master or slave based on communication.

⇒ Data will be transmitted on the I2C bus either master or slave but the clock is always transmitted by the master.

⇒ Acknowledgment is always generated as 9th Bit.

⇒ when master issues a start cond^n on I2C bus, all slaves comes under Active state & master will send slave address along with direction bit on I2C bus. only the particular slave will respond on the I2C bus (whose address is matched) will pull the SDA line to low that means acknowledgement is generated. Now, only that particular slave & master are in active mode & the remaining slaves are in inactive mode.

⇒ Based on direction bit the data will be transmitted from master to slave (or) slave to master.

⇒ For every one byte transmission we can find one acknowledgement.

⇒ For every acknowledgement, interrupt will be generated.

NACK (No acknowledgement)

(Q) When NACK is generated?

(A) (1) When no slave device is available on the I2C bus with the address then, it will generate NACK.

(2) When master is sending the data to slave, there is no more memory is available to store the data then slave will generate NACK.

(3) When master is reading the data from the slave, when master wants to stop reading the data from slave. Master will generate NACK.

(4) Based on Real time scenarios, when slave is busy with some task which is already requested by master, still the work is not yet completed slave will generate the NACK without responding.

Direction bit :-

⇒ If the direction bit is zero (0) that is ntg but write operation.

⇒ If the direction bit is One (1) that is ntg but read operation.

⇒ In master receiver mode, First master will perform write operation then it performs read operation. It depends upon the device.

Scanned with OKEN Scanner

⇒ In master transmitter mode, Always it will be in write mode.

⇒ In master receiver mode, master will generate repeated start

Write mode (Master transmitter mode)

Start cond^n	slave address along with direction bit	ACK	Data	ACK	Data	Ack	Data	Ack	Stop
M	M (d→0)	S	M	S	M	S	M	S	M

Read mode (Master receiver mode)

Start cond^n	7bit address + d→0	ACK	Memory address	ACK	repeated start cond^n	7bit slave address & D→1	ACK	Data
M	M	S	M	S	M	M	S	S

ACK	Data	ACK	Data	NACK	Stop cond^n
M	S	M	S	Stop M	M.

* Kernel threads *

→ Kernel threads are the basis of work queue mechanism. They run the kernel code ~~under~~ in process context.

→ To create a kernel thread we need to use kthread_create() kernel func.

→ Kernel threads are represented by a structure struct task_struct type. we must include linux/kthread.h header file.

```
struct task_struct *
    kthread_create ( int (*thread) (void *data), void *data
                     const char *name format[], ...);
```

fun^c pointer

return type

1st i/p → thread func base address that will run by kernel thread

2nd i/p → Data passed as i/p to thread func

3rd i/p → name format represents kernel thread name.

→ To start the kernel thread we have to call wake_up_Process ( ).

```
int wake_up_process(struct task_struct *P);
```

Structure base address.

⇒ we can create & run a kernel thread by using kthread_run( ). [It will internally invokes kthread_create & wakeup_process functions].

```
struct task_struct * kthread_run (int (* thread) (void *data)
                                  void *data, const char name format[], ...);
```

⇒ To stop the kernel thread we have to use kthread_stop( ). This fun^c works by sending a signal to that thread.

```
int kthread_stop (struct task_struct *);
```

⇒ we can bind the kernel thread to a particular CPU by using kthread_bind( ).

```
void kthread_bind (struct task_struct *k,
                   unsigned int cpu core);
```

⇒ we can free the kernel thread by using a kernel fun^c free_kthread_struct( ).

```
void free_kthread_struct (struct task_struct *k);
```

* Kernel locking Mechanism *

→ locking is a mechanism that helps to share resources b/w different threads or processes.

→ A shared resource is a data (or) device that can be accessed by atleast 2 users simultaneously or not.

→ Two processes accessing the same device, we have to use locking mechanisms. (while accessing global resources).

① MuterBS (deep sleep)
② semaphorses synchronization mechanism.
③ spin locks

① Mutexes-

→ Mutual exclusion is most used locking mechanism

→ we must include linux/mutex.h header file.

→ Kernel represents these mutexes by a data structure struct mutex.

→ It has a counter, 1 means unlocked state & 0 means locked state.

→ Mutexes internally uses spin locks.

→ First we need to initialize mutex statically by using define_mutex macro. It will take mutex variable name.

DEFINE_MUTEX (mutex variable name)

⇒ Dynamically, we can create by using mutex_init() to which we have to pass mutex variable's base address. By default these mutexes are in unlocked state when we initialize it. i.e, counter value will be '1'.

⇒ To acquire the lock we have to use below functions.

① `void mutex_lock (struct mutex * lock);`

⇒ This func will locks the mutex i.e., the counter is set to zero.

② `int mutex_lock_interruptible (struct mutex * lock);`

⇒ To unlock or release the mutex, we have to use mutex_unlock().

③ `void mutex_lock (struct mutex * lock);`

⇒ Some time we need to check whether a mutex is locked or not to avoid dead lock scenario. so, this can be done by using a fun mutex_ease_locked().

④ `int mutex_ease_lock (struct mutex * lock);`

⇒ This func will return 0 on locked state & 1 on unlocked state.

* mutex is already locked for this we need to use mutex_trylock ().

(5) int mutex_trylock (struct mutex *lock);

* It returns 0 if it is already locked and 1 if it is unlocked.

* There is a func which can send a signals by killing the corresponding process by using mutex_lock_killable ().

(6) (*) Rules to be followed w.r.t mutex :-

* only 1 task can hold the mutex at a time.

* Multiple unlocks are not permitted they must be initialized through API's.

* A task holding the mutex may not exist since, the mutex will remain locked & the possible contenders will wait forever

* Memory areas where held locks reside must not be freed.

* held mutexes must not be reinitialized since, ~~mutexes~~ tasklets & timers involves rescheduling so mutexes may not be used in atomic context

⊛ Spin locks :-

⇒ Like mutex spin locks is a mutual exclusion mechanism it has two states, locked & unlocked. locked means acquired, unlocked means release.

⇒ Any thread that needs to acquire the spin locks will go in to active loop until the lock is acquire. Since, spin locks heavily consumes the CPU while looping. It should be used for very quick acquires especially when the time to hold the spin lock is less than the time to reschedule

⇒ Spin lock should be released as soon as the critical task is done.

⇒ spinning on a processor (or) CPU means no other tasks can run on that CPU or ~~other~~ processor then it makes no sense to use spin locks on a single core machine. so, in this case it will slow down the system or goes into dead lock scenarios also.

⇒ On a single processor core system, you should use spin_lock_irq save ( ). Before it goes to locking state it will read the state

⇒ ~~spin_lock_irqsave( ) and~~ spin_unlock_irqrestore( ) which will respectively disable the interrupts on the

concurrency

* In driver code, it is recommended to use spin_lock_irqsave() & spin_unlock_irqrestore()

* First we need to initialize the spin lock by using spin_lock_init(spinlock_t variable name);

* we need to use linux/spinlock.h header file.

* spin_lock_irqsave(base address of the variable, flags);

* spin_unlock_irqrestore(base address of variable, flags);

* To check whether the spin lock is locked or not we have to use spin_ease_lock().

(*) inline func ?

→ substitutes the func definition same as macro.
→ simple task
→ length shouldn't be more than 5-6 lines
→ 2nd stage substitutions will be done.
→ MACROS substitutions will be done in 1st stage.
→ we need to use inline before the func name.

20/2/21

I2C client driver code.

21/2/23

SPI (Serial peripheral interface)

* Interview important Questions *

Q) Importance of volatile keyword ?

Q) Implement ur own size of operator & container of operator.

Q) without using How do u get the size of structure.

Q) A data which is defined in 1 file can be accessed in another file which is applied with static keyword.

Q) Importance of void pointer

Q) Differences b/w inline func & main func

" " & MACROS

MACROS & main func.

Q) Storage classes

Q) memory sections (To verify)

Q) Dynamic memory allocation

Q) Max req memory by using malloc() → limitation

Q) Differences b/w malloc(), calloc(), alloc().

Q) Importance of free().

Q) Mutex locks, semaphores differences

Q) Differences b/w threads & processes

Q) " " Mutex locks & conditional variables

Scanned with OKEN Scanner

(A) Fork () , exec family of calls.

(Q) signals.

(Q) explain scenarios when signals are delivered to a process.

(Q) segmentation Fault ?

(Q) Difference b/w abort () & exit ().

(Q) verify which signal is generated (By using gdb tool) (Ans)

(Q) Return values → ? (bash/shell terminal).

(Q) ltrace, strace.

(Q) valgrind tool → To verify memory leaks

(Q) Holes.

(Q) structure padding (Holes reduction)
↓
(Q) pragma pack, __attributes.

projects

(1) kernel compilation
driver code → How to add to source code ?

(2) understanding GPIO subsystems,

How to control device by using some peripherals by raising an interrupts

(3) Buildroot tool used to generate kernel image for target boards.

(4) UART communication.

(5) I2C

(6) SPI

(7) Socket programming.

(8)