NOOBS image bootup

xrdp server

remote ssh

startup script

Linux runlevel/ Init:

After the kernel boots up now state is init/runlevel. In this state kernel reads the run level config file from "/etc/inittab" and run the default runlevel folder scripts/executable but in some cases file might be not there then kernel boot is configured with default run level.Unless the user specifies another value as a kernel boot parameter, the system will attempt to enter (start) the default “runlevel”.

**Linux boot up init / runlevel stage**

Few services needs to be launched for user feature accessibility so few init/startup scripts which loads required services. These scripts are loaded at “runlevel”. Where “runlevel” is one of mode of Unix based operating system where all mandatory services are loaded. There are many “runlevel” each runlevel has a certain number of services stopped or started. Conventionally, seven runlevels exist, numbered from zero to six.

Run Level Action  
0 Halt Shuts down system  
1 Single-User Mode Does not configure network interfaces, daemons.  
2 Multi-User Mode Does not configure network interfaces or start daemons.  
3 Multi-User Mode with Networking Starts the system normally.  
4 Undefined Not used/User-definable  
5 X11 As runlevel 3 + display manager(X)  
6 Reboot Reboots the system

As  there are multiple runlevels Meaning that depending the runlevel we can boot up with a different set of programs. There are runlevels for halt and reboot too So when system is moving into halt state it will execute runlevel 0 and when system reboots it will execute runlevel 6. Sometimes you may find few runlevel are same as in ubuntu runlevel 3 and 5 are copied from 2.

If user wants to run a application after booting at init stage then he needs to add user script in root file system, make the entry in “runlevel” and update the “runlevel” file system so next time that service/script will be considered and loaded in “init” mode. Startup script or init script experiment is performed below

**IPC Mechanism :**

**Shared Memory:**

shared memory created at application level no core interaction is there, so access is much faster and quicker but there is no implicit implementation on synchronisation. In shared memory, race conditions can happen very easily. We need to provide external synchronisation mechanisms to avoid race conditions.

When we allocate shared memory a physical separate block is created. when we attach shared memory to process that physical memory block is mapped with process logical address space. For mapping unused segments /pages of process are used and logical mapping is formed.

>>>> Insert Image Here <<<<<<<

**int shmget(key\_t key, size\_t size, int shmflg);**

shmget returns the system identifier to shared memory. it is used to create shared memory or used to obtain an identifier of previously created shared memory if key is not IPC\_PRIVATE.

When a new shared memory segment is created, its contents are initialized to zero values, and its associated data structure

On success, a valid shared memory identifier is returned. On error, -1 is returned, and errno is set to indicate the error.

* size\_t : size must be multiples of page size
* key\_t : IPC Key
* shmflg: shared memory creation type flags
* **void \*shmat(int shmid, const void \*shmaddr, int shmflg);**

**shmat :** attaches the created shared memory to the process, Physical created shared memory mapped with process unused segments. mapping starts from “\*shmaddr” memory location which is passed to function for this you should know your hardware and mapping. If NULL is passed then the process automatically decides the address and maps.

shmat returns an address to the attached memory space. which process can access.

* shmid: shared memory Id returned by “shmget” call
* shmaddr: Shared memory to process logical memory mapping address.
* shmflg: shared memory access permissions decide by this flag

A successful “shmat” call updates the members of the shmid\_ds structure

* **int shmdt(const void \*shmaddr);**

**shmdt:** shmdt is used to detach shared memory from process as an argument it takes address from where shared memory is mapped which return by “shmat”. detaching memory doesn’t delete it just makes memory unavailable to the current process.

* shmaddr: address of attached physical memory
* **int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);**

To control shared system memory like deleting shared memory, changing configuration of shared memory

Note:

* Make sure you are using the key which is available in the IPC section. Check key availability by command “ipcs” and remove key by “ipcrm”.
* before terminating the process/code deleting shared memory by shmctl.

**Semaphore**

what happen if IRQ is doing semaphore lock

semaphore use

explain uni processor scenario

Semaphore is one of the IPC mechanisms of Linux which is mainly used for resource counting and synchronisation purposes where multi-process scenarios appear. semaphore is used by creating semaphore objects. These semaphore objects have their semaphore variable. By only object we can change the semaphore variable. Semaphore variables are at kernel space. variable is of type int and only increment and decrement operation are possible. when this variable hit’s to zero and some other process tries to decrease it further. process will get blocked and it will be blocked only until some other process increment that variable.

Binary Semaphore / Counting Semaphore

* **semget**

**int semget(key\_t key, int nsems, int semflg);**

semget is used to create an semaphore object at kernel level and it returns semaphore identifier to user space. by using this identifier we can perform operations over it.

* key\_t : Argument Needs an IPC key to create a semaphore object at kernel.
* nsems: How many number of semaphore needs to be created
* semflag: Used to define different types of flags while creating semaphores. also permissions are defined.
* **semop**

**int semop(int semid, struct sembuf \*sops, size\_t nsops);**

semop is used to do operations on semaphore variables of particular semaphore id which are returned by semget .

semop() performs operations on selected semaphores in the set indicated by semid. Each of the nsops elements in the array pointed to by sops is a structure

that specifies an operation to be performed on a single semaphore. The elements of this structure are of type struct sembuf, containing the following members:

* unsigned short sem\_num; /\* semaphore number \*/
* short sem\_op; /\* semaphore operation \*/
* short sem\_flg; /\* operation flags \*/

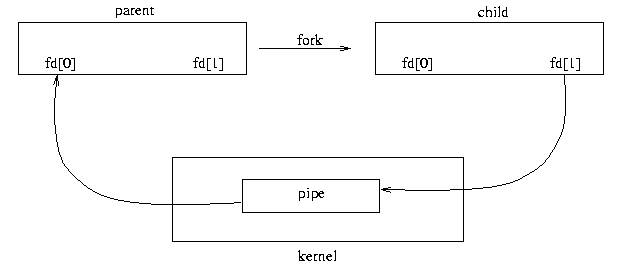
nsops: specifies the number of operations to be done, this is useful when you are managing a set of semaphores under one semaphore id.

* **semctl**

**int semctl(int semid, int semnum, int cmd, ...);**

semctl() performs the control operation specified by cmd on the System depending on the command argument further 4th input argument varies. semctl is used to set initial values of semaphore and get runtime values of semaphore, semaphore object deletion is possible by choosing particular command type.

* **Pipe**

pipe is a one of IPC communication service provided by Linux Software stack. Pipe is a unidirectional data channel that can be used for intra process and inter process communication. Pipe has two ends one as Input (Write) and other is as output (read) end. Pipe is unidirectional so one end can’t be used for reading and writing. pipe can be created by using a “pipe” system call. whenever a pipe is created it creates a temporary file and that temporary file is opened and accessed using file read/write API. Pipe stores data in FIFO format.it is implemented as a queue data structure. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

Pipe has below two version of system call

* int pipe(int pipefd[2]);
* int pipe2(int pipefd[2], int flags);

int pipedf[0],[1] hold file descriptors which refers to the read[0] end and write[1] end of the pipe. flags are used to obtain different behaviour of the pipe.On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**read function** to read from pipe read input, as we discussed pipe creates a temporary file that’s why we can read and write it as a file.

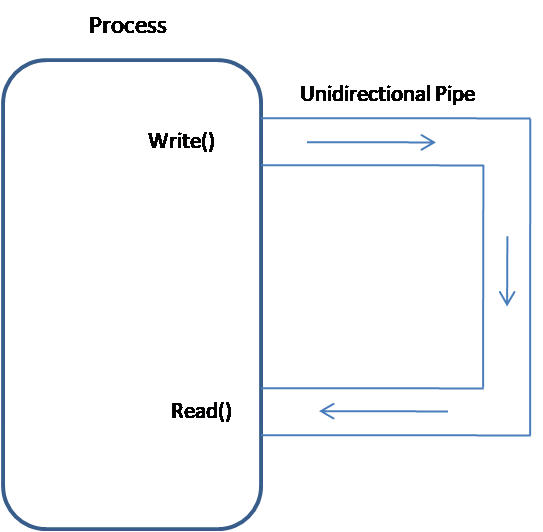
ssize\_t read(int fd, void \*buf, size\_t count);

**write function** call to write in pipe, data will be queued.

**ssize\_t write(int** *fd***, const void \****buf***, size\_t** *count***);**

Normal pipe reading and writing API are of blocking type, which means if the pipe is empty and the “read” api is called then the calling process will get blocked until the write fills into the pipe. Write Api will be blocked if the pipe is full. the pipe buffer is maintained at the kernel end, So every Pipe data goes through the kernel buffer.

so reads get blocked until someone writes to it this mechanism used for process synchronisation in Linux.

****

* **Pipes used for inter process communication (Related process).**

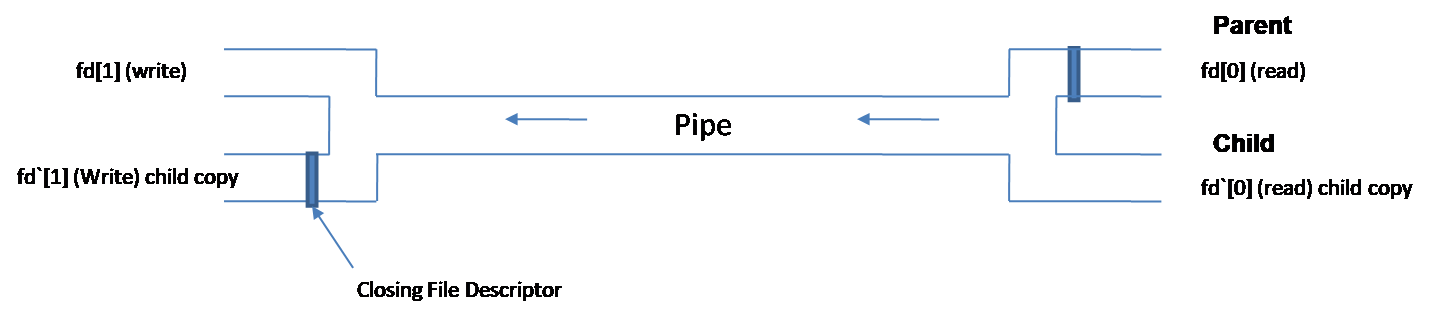
After Creation of the pipe if we do fork it will create a child process which will have the same pipe file descriptor copy.

Once the process has created the pipe successfully it will hold two file descriptors as explained earlier. After Pipe creation if the process does fork and creates the Child process. then the child will also have a copy of this file descriptor. Now we have four file descriptors: two of child and two of parent. Now the pipe will be like the picture below.



Like above Diagram four file Descriptor we have. a child and parent can use this descriptor for their individual own in process communication but data can conflict if both are writing at the same time as both are using a single pipe that's why the “write” operation should be atomic. Pipe follows FIFO operation so all data can be read out in queue format only “write” operation needs to be taken care of. it can be used for parent child communication too. only what we need to do is close the un-necessary descriptor to avoid unwanted read and write operations.

As shown in the picture below, we can close the file descriptor and parent and child can communicate with each other. In case of “exec” command we can create a duplicate file descriptor and pass it as an argument so even if the child is not a copy of parent it can have a descriptor and communicate with parent.



We can block the file unwanted File descriptor and now the parent can write to the pipe and the child will read from it so they can communicate with each other.

>> Extra

* Linux has a VFS (virtual file system) module called pipefs, that gets mounted in kernel space during boot
* pipefs is mounted alongside the root file system (/), not in it (pipe’s root is pipe:)
* pipefs cannot be directly examined by the user unlike most file systems
* The entry point to pipefs is the pipe(2) syscall
* FIFO/ Named Pipes

FIFO are pipes used for unrelated process communication.they are similar as pipe only with few property differences.

* + **Pipes used for inter process communication (Not related process).**

We can create a FIFO/Named pipe by below system call API

int mkfifo(const char \*filename, mode\_t mode);

this will create a named pipe which is a special file and all kind of file operation can be done.As second argument we are passing permissions to that file.as shown in picture "p" shows pipe special file. All kind of file operation from command line can be done on this file. in the fifo can write by using "echo" and "cat" commands.

After creation of pipe special file to do operation in it we need to open the file by "open" system call with flags. one thing to make a note. FIFO is pipe and it has two ends read and Write so File must be opened with O\_RDONLY and O\_WRONLY flags. O\_RDWR is not permitted as read write at one end is not allowed, this may lead to undefined behaviour of program.

**open system call**

int open(const char \*pathname, int flags);

In the case of FIFO one more Flag we have O\_NONBLOCK. it changes how open call processed and read ,write requests are processed with their blocking behaviour. API's may have the following behaviour with open call.

**open(const char \*path, O\_RDONLY);** In this case open call will be blocked until FIFO is opened for writing.

**open(const char \*path , O\_RDONLY | O\_NONBLOCK);** In this case open call will execute Immediately even if no one is opened it for writing.

**open(const char \*path, O\_WRONLY);** In this case open call will be blocked until the some process open same FIFO for reading.

**open(const char \*path, O\_WRONLY | O\_NONBLOCK);**  In this case open call will execute Immediately but if no process open fifo for reading then it return -1 and fifo won't be opened.

Before opening FIFO we need to check weather FIFO special file is created or not we can do that by using access system call

**int access(const char \*pathname, int mode);**

**access()** checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced.

One program(fifo\_p1.c) continuously read from FIFO and print on terminal, Another program(fifo\_p2.c) take command line argument and write to FIFO.In case of FIFO it is synchronised by "open" blockable call back.After read or write operation need to close file and again open for next operation to achieve blocking nature otherwise it continuously runs if try to read again before closing. Need to take care for atomic "write" operations

**Message Queue**

\*\*\*\*\*\*\*\*\*\* Message Queue \*\*\*\*\*\*\*\*\*\*\*\*\*

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by. A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget().The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process.

**message send system call API**

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

The msgsnd() and msgrcv() system calls are used, respectively, to send messages to, and receive messages from, a System V message queue. The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.

struct msgbuf {

long mtype; /\* message type, must be > 0 \*/

char mtext[1]; /\* message data \*/

};

The mtext field is an array (or other structure) whose size is specified by msgsz,a nonnegative integer value. Messages of zero length (i.e., no mtext field) are permitted. The mtype field must have a strictly positive integer value.This value can be used by the receiving process for message selection

The msgsnd() system call appends a copy of the message pointed to by msgp to the message queue

whose identifier is specified by msqid.

If insufficient space is available in the queue, then the default behavior of msgsnd() is to block until space becomes available. If IPC\_NOWAIT is specified in msgflg, then the call instead fails with the error EAGAIN.

Upon successful completion the message queue data structure is updated as follows:

* msg\_lspid is set to the process ID of the calling process.
* msg\_qnum is incremented by 1.
* msg\_stime is set to the current time.

**message Receive Call**

size\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtype, int msgflg);

The msgrcv() system call removes a message from the queue specified by msqid and places. it in the buffer pointed to by msgp.

The argument msgsz specifies the maximum size in bytes for the member mtext of the structure pointed to by the msgp argument.

\* If msgtyp is 0, then the first message in the queue is read.

\* If msgtyp is greater than 0, then the first message in the queue of type msgtyp is read, unless MSG\_EXCEPT was specified in msgflg, in which case the

first message in the queue of type not equal to msgtyp will be read. The msgflg argument is a bit mask constructed by ORing together zero or more of the following flags:

IPC\_NOWAIT

Return immediately if no message of the requested type is in the queue. The system call fails with errno set to ENOMSG. If no message of the requested type is available and IPC\_NOWAIT isn't specified in msgflg, the calling process is blocked until one of the following conditions occurs:

TO DO

>> Queue Max Size can be decided by msg ctl Need to check and change.

>> Max Queue entry also need to check

**message control call**

**int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);**

msgctl() performs the control operation specified by cmd on the System V message queue with identifier msqid.

struct msqid\_ds {

* struct ipc\_perm msg\_perm; /\* Ownership and permissions \*/
* time\_t msg\_stime; /\* Time of last msgsnd(2) \*/
* time\_t msg\_rtime; /\* Time of last msgrcv(2) \*/
* time\_t msg\_ctime; /\* Time of last change \*/
* unsigned long \_\_msg\_cbytes; /\* Current number of bytes in
* queue (nonstandard) \*/
* msgqnum\_t msg\_qnum; /\* Current number of messages
* in queue \*/
* msglen\_t msg\_qbytes; /\* Maximum number of bytes
* allowed in queue \*/
* pid\_t msg\_lspid; /\* PID of last msgsnd(2) \*/
* pid\_t msg\_lrpid; /\* PID of last msgrcv(2) \*/

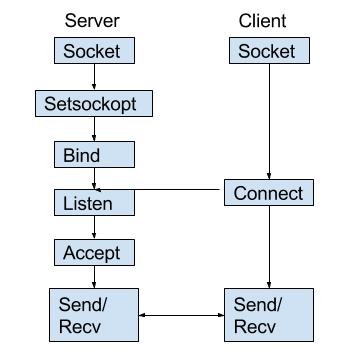
};

By setting/passing this structure we can get/set message queue max entry and message size. These are reconfigurable.

**Socket**

**Socket Program**

Socket Can be used in process communication and between two different machine communication too. The Socket mechanism can implement multiple clients attached to a single server.



local sockets are given as file names in the linux file system, for network socket the file name will be a service identifier (port number/ access point) relevant to the particular network to which clients are connected. Once the socket is established it can be used like a low level file descriptor, providing two way data communication.

Socket is a communication Mechanism that allows client and server machines to connect locally or on a single machine or across a network. The socket mechanism is implemented to connect multiple clients to a single server. Local sockets in a system generally are file in file systems. Sockets are bidirectional communication mechanisms. Usually sockets are the entry door to a communication protocol they are bound to a particular port. Whatever data comes to port it will be streamed to server/ client. Sockets have fast and reliable data transfer rate, easily large chunks of data can be transferred without any data loss.

Main purpose of socket is for server and client type communication where large data needs to be transferred at a high rate. Different machines can be connected on local Networks. so it is very easy to design system

System Call API

**int socket(int domain, int type, int protocol);**

socket creates an endpoint for communication and returns a file descriptor that refers to that end point if the call is successful.

domain: argument specifies in which domain communication is going to happen and selects protocol family which will be used for communication.

* AF\_UNIX, AF\_LOCAL Local communication unix(7)
* AF\_INET IPv4 Internet protocols ip(7)
* AF\_INET6 IPv6 Internet protocols ipv6(7)
* AF\_IPX IPX - Novell protocols
* AF\_NETLINK Kernel user interface device netlink(7)
* AF\_X25 ITU-T X.25 / ISO-8208 protocol x25(7)
* AF\_AX25 Amateur radio AX.25 protocol
* AF\_ATMPVC Access to raw ATM PVCs
* AF\_APPLETALK AppleTalk ddp(7)
* AF\_PACKET Low level packet interface packet(7)
* AF\_ALG Interface to kernel crypto API

above Flags are used to fill “domain” Arguments.

type: specifies which type of socket need to be created, for Example non-Blockable, packet Data read type, etc.

protocols: The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the “communication domain” in which communication is to take place; see protocols(5). See getprotoent(3) on how to map protocol name strings to protocol numbers.

by using socket API we created a end point for communication and obtained file descriptor to it now we need to attach socket to some address and port so client /server can create another endpoint with same address and port

**Bind**

**int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**

bind( ) assigns the address specified

Sockfd is received from socket system call.

by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr.

struct sockaddr {

* sa\_family\_t sa\_family;
* char sa\_data[14];

}

The rules used in name binding vary between address families.

in this document specifying for AF\_UNIX (used to communicate between processes of same machine efficiently) binding.

Traditionally, UNIX domain sockets can be either unnamed, or bound to a filesystem pathname (marked as being of type socket). Linux also supports an abstract namespace which is independent of the filesystem.

**Listen**

**int listen(int sockfd, int backlog);**

listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to **accept incoming connection requests using “accept()” system call.**

The sockfd argument is a file descriptor that refers to a socket of type SOCK\_STREAM or SOCK\_SEQPACKET. The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

accept() system call

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**

The accept() system call is used with connection-based socket types (SOCK\_STREAM, SOCK\_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.

Accept is blocking type api until it forms any connection it will be blocked once it forms connection. it fills client address file descriptor, by using that file descriptor we can do read and write to the socket.

In socket fd read is of blocking type if nothing is there it will be blocked until some data comes up on socket.

Every time while client / server exits a close file descriptor properly otherwise at one end it will show undefined behavior.

Once we get file descriptor we can do read write operation from server amd client both end.

Client will not get connected until server creates socket and get bind with until, until that point if client try to connect , “connect” system call will be failed.

Note: Available socket can be checked by command “netstat”

POSIX Thread

?>>>> Add Information Here

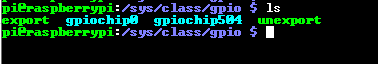
**Accessing GPIO In R-Pi (using Conventional sysfs method).**

R-pi has an Embedded Linux OS stack, so IO interface is more closed to user space not like conventional Linux OS. R-pi has a GPIO controller over it to control GPIO Action.

We can access GPIO by accessing sysfs of R-pi

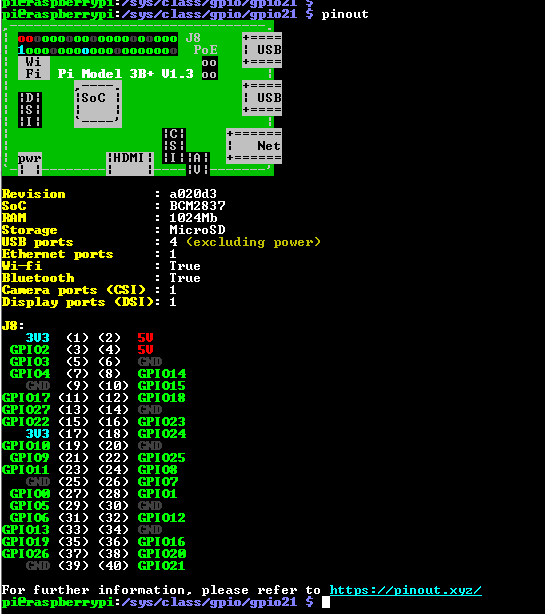
Go to Dir >> /sys/class/gpio/

Most probably you will see below files.



gpiochip0 gpiochip504 are GPIO controller of R-pi.

To check which gpio are available on board you can check by typing “pinout” command on R-pi terminal/shell

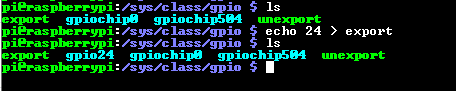


If we want to use or access a gpio we need to make an entry so gpio file can be created, and by the sysfs file system we can use GPIO’s. Sysfs is a pseudo filesystem provided by the Linux kernel that makes information about various kernel subsystems, hardware devices, and device drivers available in user space through virtual files. GPIO devices appear as part of sysfs.

To access a gpio we need to follow below steps

* Make Gpio entry so sysfs file visible
* set GPIO direction
* if pin is as o/p set value as 0/1
* if pin is I/p get value
* Remove entry when gpio are not getting used

we write the pin name/number to the pseudo file /sys/class/gpio/export. This indicates that we want to use a specific GPIO pin and makes it visible in the sysfs file system hierarchy.

to make an Entry of gpio we need to write to “export” file. For example for GPIO 24 we can do like “**echo 24 > export**”. After this we will be able to see a special file as below.

These GPIO files are a special linked File with gpiochip controller. as this special linking we can see in the picture below.

Now to set Direction of GPIO and to set/get value from/to GPIO we need to enter into file gpio24. In the gpio24 file below list of files we can see, which holds GPIO attributes and its value.



each file control gpio differently there details are listed below.

**active\_low:** For pull up and pull down it is configured.

**edge:** used to configure as “rising /falling” edge detection when gpio is configured as input. It can be configured as none

**direction:** Used to set direction of GPIO as “in” or “out”. As an input / output GPIO

**value:** used to write / read value of GPIO.Depending on GPIO direction configuration

we need to set gpio direction first, we can do that by firing command

**echo in > direction / echo out > direction**  we can do this thing in code by accessing files we can write in this file as a normal file but if something else we try to write it will show Error.

After setting Direction we can get or set value by writing into “value” file. for example. “**echo 1 > value**” or we can read value as “**cat value**”. By default GPIO are configured as input.

After GPIO use we need to remove entry of GPIO by unexport for example **echo 24 > unexport.**  gpio24 file will be now invisible from sysfs.

**Linux Software Timer**

To compile this timer related function call we need to link one external library “-lrt” as for thread we link “-pthread”

In Linux timer can be created and we can register to call back which will be invoked when timer Expiry will happen. Basically we need to follow the below things to create a timer.

**\* timer\_create()**: Create a timer.

**\* timer\_settime(2)**: Arm (start) or disarm (stop) a timer.

**\* timer\_gettime(2)**: Fetch the time remaining until the next expiration of a timer, along with the interval setting of the timer.

**\* timer\_getoverrun(2)**: Return the overrun count for the last timer expiration.

**\* timer\_delete(2)**: Disarm and delete a timer.

**timer\_create();**  // created timer and give timer id, the second parameter is about timer attributes. which hold a timer call back and other flags and attributes. There are three different ways of how we can look for expiry.

1. Continues poll to value by getting the current value

2. The thread will be created and registered call back will be executed by a created thread

3. Linux register signal will be generated. for a signal registered to call back will be invoked.

Below is an actual function call declaration.

int timer\_create(clockid\_t clockid, struct sigevent \*sevp,

timer\_t \*timerid);

**clockid**: The **clockid** argument specifies the clock that the new timer uses to measure time. Different clock id can be configured for more reference. We can refer to man Pages. For Experimentation purposes we choose **CLOCK\_REALTIME**.

**sevp**: this is a very important structure that will hold the most of the information like flag attributes, Type of notification and Different Handler. The **sevp** argument points to a **sigevent** structure that specifies how the caller should be notified when the timer expires. For sigevent structure and it's the type of notification and call back registration we might need to refer below man pages.

* **man 7 sigevent**
* **man 2 sigaction** this will be much more helpful to fill sigevent structure.

**timerid**- after timer creation successful timer id is kept in this variable to do further operations on the timer. to control all-timer operation we need timer id.

Once Timer is created with needed notification configuration timer will be disarmed /stopped it needs to start. we can start our timer by setting its time by using system function call **timer\_settime();**

**int timer\_settime(timer\_t timerid, int flags, const struct itimerspec \*new\_value,**

**struct itimerspec \*old\_value);**

we need to pass a **timerid** to which time needs to be set also by flags we can change them. The important argument is **struct itimerspec** which is used to set expiry time and interval time to the timer. Once the timer expires, interval time is reloaded again and again. to stop and have only one expiration we need to define interval time as zero so timer will stop after it's first expiry. more information we can find at **man 2 timer\_settime();**

**For timer arm/start and disarm/stop both operations can be done by using timer\_settime() system function call.**

timer\_delete is used to delete timer and clear its attributes.

**Note:** Three types of timer expiry handler I have Figured Out

* by continuous Polling using the function timer\_gettime().
* By registering a signal to the timer Expiry event signal will generate and the signal handler will be performed as a timer expiry handler.
* A thread will be created with set attributes whenever expiry happens a call back function will be executed by a newly created thread. and parallel callback and main will perform

**GNU GDB Debugger**

No programmer is perfect, some of them make logical mistakes so some are syntactical. Syntax error can be caught by the compiler but the logical error is a bit tough to identify the exact bug. the logical bug may change program response. There are multiple Debugging techniques but one common method should be followed for any type of debugging as follows.

* **Testing:** Finding out what defect or bug exists.
* **Stabilization:**  Making the bugs Reproducible.
* **Localization:** Identify the Line of Code which is creating a bug.
* **Correction:** Fix the bug.
* **Verification:** Making sure the Fix works.

gdb is an exceptionally powerful tool that can help to provide internal states of the program. gdb tool will help with the Localization and Verification step to narrow down the issue gdb is a good tool.

GNU Project Debugger /gdb is a tool used to debug the programs, by using gdb we can analyze programs step by step by putting breakpoint we can observe stack and variable values too, which gives very much help to debug the program.we can do patching work also at run time when debugging is running

Whenever a process crashes in Linux it will create a core dump file in Linux that can be also debugged to identify the crash issues. this "**ulimit -c unlimited**" can be set to generate core dump when a process crashes. gdb is also used with an executable file that is compiled by the GNU compiler. gdb is a text-based application that can be controlled by Few commands.

**Steps For Using GBD (GNU Debugger)**

**~ gdb demo**     (demo is an executable file Generated by GNU compiler).

After this, you may see a few prints of the gdb version. Now the program is at the start point of the main but it is not running yet. This command just started debugging for demo executable but it is paused for now.

After this we will be entered in the gdb shell, now we will be using gdb shell. Linux shell command will not work.

(gdb) **run**

After this run, the command program will start to execute normally and ends normally. We haven't got any chance to see anything (In case of the normal error-free program) because the program hasn't stopped anywhere. If any Bug is there in the program then after the termination line number where a bug occurred it will be shown on gdb shell. (Logical error will not be captured only Error which is crashing the program those only).

we need to stop the program at a certain point to do an analysis of variable value and stack information. We can stop the program at a few particular points by putting breakpoints in the program.

(gdb) **break 5**

by using break command we can stop the program at a particular line number here 5 is the Line number where the breakpoint is planted.

Now once the program is stopped we can do our code observation like we can check present stack variable value by giving command **print "Variable name"**

(gdb) **print j** If j is a variable in program current stack value of j is displayed. this is one way to analyze whether code is behaving properly or not by observing its variable value.

we can also display array elements in one shot by command

(gdb) **display array[0]@5**  by this command array elements are displayed form 0 to 5.

(gdb) **display i**

 By using Display command we can set continues printing on a variable whenever it will cross the point it will print "i"

(gdb) **list**

once we think at a particular breakpoint nearby bug is there we can see cursor surrounding source code by the "list" command.

If we are running a code that has recursion and you want to see the created stack we can use the **backtrace** command.

(gdb) **backtrace** Displays the up to now all created stacks, which provides great help in the recursion program. In this way also we can narrow down the line which has a bug by looking at which function is been called by observing its stack.

(gdb) **cont**

Once for one breakpoint, we are done and we are ready to move to the next breakpoint. We can use the cont command to continue the program. the program will continue until the next breakpoint occurs or the program ends.

(gdb) **disable break 1**

(gdb) **disable display 1**

used to disable display and breakpoint which were earlier mounted on the program.

(gdb) **quit**

used to quit Debugging.

**Compiling a kernel Module**

If not present kernel header files then we need to install kernel header files for R-pi First by command >> **sudo apt-get install raspberrypi-kernel-headers**

**Kernel Headers** contain the C header files for the Linux kernel, which offers the various function and structure definitions required when compiling any code that interfaces with the kernel, such as kernel modules or device drivers and some user programs.

>> mention steps

>> Mention Makefile

>> snap Kernel header file structure