

Internship Training Cum Project Report

Final Internship Evaluation

At **PHYTEC INDIA**

For the period 06/02/2023 to 06/08/2023

Submitted to

Department of Electrical and Electronics Engineering

In partial fulfillment of the award of the degree of

Bachelor of Technology

In

Electrical and Electronics Engineering

By

Deepak Kumar Beniya

1901227415

8th Semester



Electrical and Electronics Engineering

C.V. Raman Global University, Bhubaneswar, Odisha -752054

(Formerly C.V. Raman College of Engineering)



C.V. Raman Global University

Bhubaneswar, Odisha -752054

BONAFIDE CERTIFICATE

I hereby declare that work related to a Report titled “Internship at **PHYTEC INDIA**” submitted herein, has been carried out at **Skill Development Institute, BBSR**. The work is original and has not been submitted earlier as a whole or in part for the award of any degree at this or any other institution.

SIGNATURE

DR. Ashwani kumar Sahoo

HEAD OF THE DEPARTMENT

Electrical and Electronics Engineering

SIGNATURE

Gopalakrishna polimera

Tehcnical Manager

Instructors

Embedded Full Stack IoT Analyst



C.V. Raman Global University

Bhubaneswar, Odisha -752054

CERTIFICATE OF APPROVAL

This is to certify that we have examined and approved the final Internship report for the 8th semester in **Electrical and Electronics Engineering** entitled “**Linux Internals and Rugged Board**” submitted by **Deepak Kumar Beniya (1901227415)**.

We here by accord our approval of it as an Internship work carried out and presented in a manner required for its acceptance for the partial fulfillment of the award of the degree of **Bachelor of Technology(B.Tech)** in **Electrical and Electronics Engineering(EEE)** Department for which it has been submitted. This approval does not necessarily endorse or accept every statement made, opinion expressed, or conclusions drawn as recorded in this internship report, it only signifies the acceptance of the internship report for the purpose it has been submitted.

(Project Guide)

(External Examiner)

(Internal Examiner)

ACKNOWLEDGMENT

We would like to express our special thanks of gratitude to our Trainer **Gopalakrishna polimera and Chaman Kumar** as he gave us the excellent opportunity to do this wonderful Internship on the Module “**Linux Internals and Rugged Board**”, which also helped us in doing a lot of Research and we came to know about so many new things. I am thankful to him.

We express enough thanks to the college for their continued support and encouragement. We offer our sincere appreciation for the learning opportunities provided by our college.

Our completion of this Module could not have been accomplished without the support of our parents. Thanks to them as well. The countless times they kept us during our hectic schedules will not be forgotten. It was a great comfort and relief to know that they were willing to provide management of Ours SDI campus activities while I completed my work.

Their encouragement when the times got rough is much appreciated and noted. Our heartfelt thanks.

Thanks to the fellow interns as well for their continued support and comfort.

Thanks to everyone associated with the Module at every micro level.

Thanks to our college and all members associated with the college.

CONTENTS

1. Introduction (Objective etc)	
2. Course Module Contents	
a. Linux Internals	
1. Linux Intro & Installation.....	8
2. Linux Shell Commands.....	11
3. Shell Scripting.....	14
4. Process Management.....	17
5. File Operation.....	19
6. Signals.....	22
7. Linux Scheduler & Memory Management.....	23
8. Linux Multi-Threading Programming.....	25
9. Inter Process Communication.....	29
10. Network Programming in Linux.....	31
b. Rugged Board	
1. Introduction.....	34
1.1. PhyCORE_A5D2x SOM Architecture.....	36
1.2. PhyCORE_A5D2x Components Placement Diagram.....	37
1.3. PhyCORE_A5D2x SOM: Power Source Types.....	38
1.4.1 Types of Signals.....	39
2. Functional Description.....	40
2.1. PhyCORE_A5D2x Primary “System Power (VDD_3V3)”.....	41
2.2. System Control.....	42
2.3. Ethernet PHY (10BASE-T/100BASE-TX).....	42
2.4. QSPI Memory.....	43
2.5. EEPROM Memory.....	43
3. Power Supply Connections and Timing Sequences:.....	44
3.1. Power Configuration #1.....	44
3.2. Power Supply Configuration #2.....	44
3. Training & Evaluation Schedule	
a. Training Schedule	
b. Completion Status	

- c. Evaluation Schedule
- 4. Evaluation Result with signature from competent authority of the company
- 5. Project Work
 - a. Project Schedule
 - b. Project Description
 - c. Experiments and Results
- 6. Certificate (signed from competent authority of the company)

ABSTRACT

This report presents the findings and experiences gained during a 6-month internship training focused on Linux internals and rugged board technology. The internship provided an opportunity to delve into the intricacies of Linux operating system internals and gain hands-on experience with rugged board systems.

The report begins by discussing the background and objectives of the internship, highlighting the significance of understanding Linux internals and the relevance of rugged board technology in various industries. It outlines the methodology employed, which involved a combination of theoretical study, practical implementation, and collaborative projects.

The internship primarily focused on exploring Linux internals, including the scheduler, memory management, and file operations. The report highlights the key concepts and mechanisms involved in these areas and their impact on system performance and efficiency. Additionally, it discusses the Linux scheduler's role in managing task scheduling and the memory management subsystem's importance in efficient resource utilization.

Furthermore, the report touches upon network programming in Linux, emphasizing the development of applications that communicate over the network using sockets and various protocols. It outlines the socket programming model, internet protocols, network addressing, and available libraries and frameworks for network programming in Linux.

The internship also involved working with rugged board systems, which are designed to withstand harsh environments and offer enhanced reliability. The report provides insights into the architecture and features of rugged boards, highlighting their applications in industries such as industrial automation etc.

Throughout the internship, collaboration with experienced professionals and the guidance of a dedicated supervisor played a crucial role in deepening understanding and fostering growth. The report acknowledges the support received from the host organization, supervisor, technical staff, and fellow interns, as well as the contributions of family, and friends.

In conclusion, the internship on Linux internals and rugged board provided a comprehensive understanding of the internal workings of the Linux operating system and practical insights into the development and deployment of applications on rugged board systems. The gained knowledge and experiences will serve as a solid foundation for future endeavours in these domains.

Introduction

Linux is an open-source operating system widely known for its stability, security, and flexibility. It has become the preferred choice for various applications, ranging from personal computers to embedded systems and rugged environments. In this report, we will explore the internals of Linux and its compatibility with rugged boards.

Linux, developed by Linus Torvalds in 1991, has evolved into a powerful operating system with a vast community of developers and contributors worldwide. Its source code is freely available, allowing users to modify and distribute it under open-source licenses. This approach has fostered innovation and collaboration, making Linux a robust and reliable platform.

Internally, Linux consists of a kernel, which acts as the core component, and a set of utilities and libraries that provide additional functionalities. The kernel, responsible for managing system resources and facilitating communication between hardware and software, forms the foundation of the operating system. Its modular design allows for easy customization and scalability, enabling Linux to run on a wide range of hardware architectures.

One area where Linux has found significant application is in rugged boards. Rugged boards, also known as single-board computers (SBCs), are specially designed to withstand harsh environmental conditions such as extreme temperatures, vibrations, and shock. They are commonly used in industrial automation, aerospace, defense, and other demanding industries.

The robustness of Linux, coupled with its adaptability, makes it an ideal choice for rugged board deployments. Linux's small footprint and efficient resource management ensure optimal performance on limited hardware resources, a crucial

requirement in many embedded and industrial applications. Additionally, its extensive driver support allows seamless integration with various peripherals and hardware components, ensuring compatibility and ease of use.

Furthermore, Linux's open-source nature provides benefits such as security and reliability. The active development community continuously reviews and enhances the operating system, addressing vulnerabilities and improving stability. This collaborative effort leads to timely updates and patches, minimizing the risk of security breaches and ensuring long-term support for rugged board deployments.

In this report, we will delve into the internal workings of Linux, exploring its architecture, key components, and mechanisms that contribute to its efficiency and stability. We will also examine the specific considerations and advantages of using Linux on rugged boards, highlighting its suitability for challenging environments and outlining practical examples of its successful implementation.

By understanding the internal workings of Linux and its compatibility with rugged boards, we can appreciate the value it brings to various industries and gain insights into how to leverage its capabilities for future deployments.

Course Module Contents: - Linux Internals

Linux internals refer to the underlying architecture and components of the Linux operating system. It involves understanding how Linux interacts with hardware, manages system resources, and provides a platform for executing applications. To delve into Linux internals, we need to explore its key components and their roles in the overall system.

1. **Kernel:** The Linux kernel forms the core of the operating system. It is responsible for managing system resources, including memory, processes, input/output operations, and device drivers. The kernel acts as an interface between the hardware and software layers, enabling communication and coordination between them.
2. **System Calls:** System calls provide a means for applications to interact with the kernel and access its services. They are function calls that allow processes to request specific operations from the kernel, such as file system access, process management, network communication, and more.
3. **Processes and Threads:** Linux follows a multitasking model, allowing multiple processes to run concurrently. A process represents an executing program, while threads are lightweight units within a process that can be scheduled independently. The kernel manages the creation, scheduling, and termination of processes and threads, ensuring efficient utilization of system resources.
4. **Memory Management:** Linux employs a virtual memory system that enables processes to have their own isolated memory spaces. It uses techniques like demand paging, virtual memory mapping, and memory swapping to efficiently manage memory allocation and deallocation. Memory

management ensures the isolation and protection of processes, preventing them from interfering with each other's memory.

5. **File Systems:** Linux supports various file systems, including ext4, Btrfs, XFS, and more. File systems provide a structured way to store and organize data on storage devices. The kernel's file system layer handles file operations, such as reading, writing, creating, deleting, and managing file metadata.
6. **Networking:** Linux incorporates a comprehensive networking subsystem that facilitates network communication. It includes protocols, socket APIs, device drivers, and networking utilities. The kernel handles network-related tasks like packet routing, network stack implementation, and device driver management.
7. **Device Drivers:** Device drivers enable Linux to communicate with hardware devices such as graphics cards, network adapters, storage controllers, and input devices. They act as interfaces between the kernel and the hardware, abstracting the hardware details and providing a unified way for the kernel to access and control devices.
8. Understanding Linux internals is crucial for system administrators, developers, and anyone working with Linux-based systems. It allows them to optimize system performance, troubleshoot issues, develop device drivers, and create efficient applications that leverage the underlying capabilities of the operating system.

Linux Intro & Installation

Linux is a powerful and versatile open-source operating system that has gained widespread popularity due to its stability, security, and flexibility. It is based on the Unix operating system and offers a wide range of distributions, each with its own features and target audience. In this section, we will provide an overview of Linux and guide you through the installation process.

1. Advantages of Linux:

Open Source: Linux is distributed under open-source licenses, allowing users to access, modify, and distribute the source code freely.

Stability: Linux is known for its stability and reliability, with systems often running for extended periods without requiring reboots.

Security: Linux benefits from its open-source nature, as a vast community of developers actively reviews and enhances its security features.

Flexibility: Linux offers a high degree of customization, enabling users to tailor the operating system to their specific needs.

Wide Range of Software: Linux provides a vast ecosystem of free and open-source software, including productivity tools, development environments, and multimedia applications.

2. Linux Distributions:

Linux is available in various distributions, commonly referred to as distros. Some popular ones include Ubuntu, Fedora, Debian, CentOS, and Arch Linux.

Each distribution has its own package management system, software repositories, default desktop environment, and release cycles, catering to different user preferences and requirements.

The choice of distribution depends on factors such as ease of use, hardware compatibility, software availability, and specific use cases.

3. Linux Installation:

Before installing Linux, it is advisable to back up your important data and ensure compatibility with your hardware.

Choose a Linux distribution that suits your needs and download the installation ISO file from the official distribution website.

Create a bootable USB drive using tools like Rufus (Windows) or Etcher (Windows, macOS, Linux).

Insert the bootable USB drive into your computer and restart it.

Access the system's BIOS or UEFI settings and configure the boot order to prioritize the USB drive.

Boot from the USB drive and follow the installation wizard's instructions.

Select the language, time zone, keyboard layout, and partitioning scheme during the installation process.

Create a username and password for your user account and choose any additional software packages or features you want to install.

Once the installation is complete, restart your computer and remove the USB drive.

You can now log in to your Linux system and begin exploring its features and functionalities.

4. Post-Installation Steps:

After installing Linux, it is recommended to update your system to ensure you have the latest security patches and software updates.

Familiarize yourself with the package management system of your chosen distribution to install, update, and remove software packages.

Customize your desktop environment and explore the wide range of applications available for Linux.

Join online communities and forums to seek assistance, share knowledge, and learn from other Linux users.

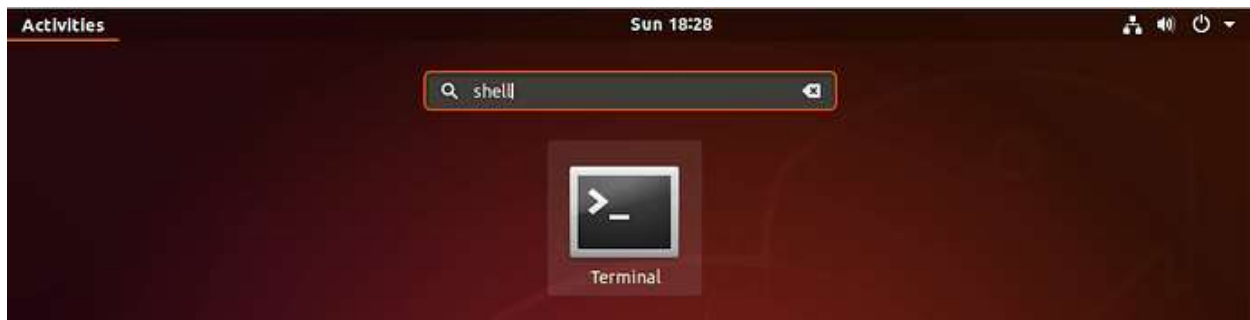
By understanding the advantages of Linux and following the installation process, you can embark on a journey with this powerful operating system, unleashing its potential and exploring its vast capabilities.

Linux Shell Commands

A shell is a special user program that provides an interface to the user to use operating system services. Shell accepts human-readable commands from the user and converts them into something which the kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.

➤ Opening a terminal:

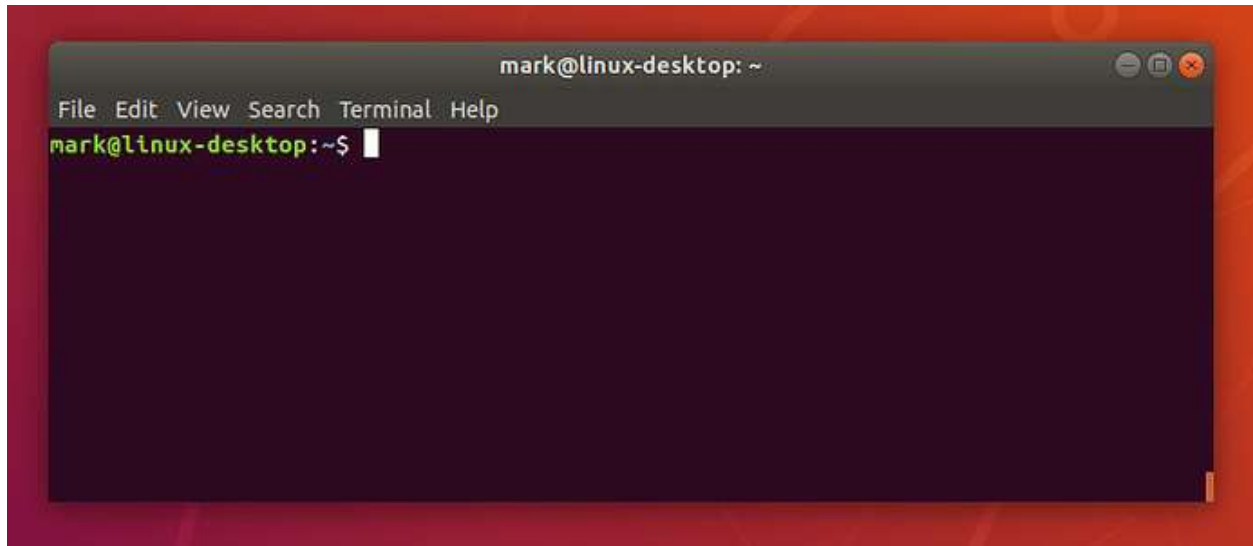
On an Ubuntu 22.04 system you can find a launcher for the terminal by clicking on the **Activities** item at the top left of the screen, then typing the first few letters of “terminal”, “command”, “prompt” or “shell”. Yes, the developers have set up the launcher with all the most common synonyms, so you should have no problems finding it.



Other versions of Linux, or other flavors of Ubuntu, will usually have a terminal launcher located in the same place as your other application launchers. It might be hidden away in a submenu, or you might have to search for it from within your launcher, but it’s likely to be there somewhere.

If you can’t find a launcher, or if you just want a faster way to bring up the terminal, most Linux systems use the same default keyboard shortcut to start it: Ctrl-Alt-T.

However, you launch your terminal, you should end up with a rather dull looking window with an odd bit of text at the top, much like the image below. Depending on your Linux system the colors may not be the same, and the text will likely say something different, but the general layout of a window with a large (mostly empty) text area should be similar.



1. **ls:** List files and directories in the current directory. Example: `ls`
2. **cd:** Change directory. Example: `cd Documents`
3. **pwd:** Print the current working directory. Example: `pwd`
4. **mkdir:** Create a new directory. Example: `mkdir NewFolder`
5. **rm:** Remove files and directories. Example: `rm file.txt`
6. **cp:** Copy files and directories. Example: `cp file.txt destination/`
7. **mv:** Move or rename files and directories. Example: `mv file.txt newfile.txt`
8. **cat:** Display the contents of a file. Example: `cat file.txt`
9. **grep:** Search for a pattern in files. Example: `grep "keyword" file.txt`
10. **find:** Search for files and directories based on various criteria. Example: `find /home -name "*.txt"`

- 11.**chmod:** Change permissions of files and directories. Example: chmod 755 file.txt
- 12.**chown:** Change ownership of files and directories. Example: chown user:group file.txt
- 13.**ps:** Display information about running processes. Example: ps aux
- 14.**kill:** Terminate a running process. Example: kill 1234 (where 1234 is the process ID)
- 15.**df:** Display disk space usage. Example: df -h
- 16.**du:** Estimate file and directory space usage. Example: du -sh directory/
- 17.**tar:** Archive files into a tarball. Example: tar -czvf archive.tar.gz files/
- 18.**unzip:** Extract files from a zip archive. Example: unzip archive.zip

Shell Scripting

Usually, shells are interactive, which means they accept commands as input from users and execute them. However, sometimes we want to execute a bunch of commands routinely, so we must type in all commands each time in the terminal.

As a shell can also take commands as input from file, we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with **‘.sh’** file extension e.g., myscript.sh.

A shell script has syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. It would be very easy to get started with it.

A shell script comprises the following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.

➤ Why do we need shell scripts?

There are many reasons to write shell scripts:

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups.
- System monitoring
- Adding new functionality to the shell etc.
-

➤ Some Advantages of shell scripts

- The command and syntax are the same as those directly entered in the command line, so programmers do not need to switch to entirely different syntax.
- Writing shell scripts are much quicker.
- Quick start
- Interactive debugging etc.

➤ Some Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful.
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc.

➤ Simple demo of shell scripting using Bash Shell

If you work on a terminal, something you traverse deep down in directories. Then for coming few directories up in path we have to execute a command like this as shown below to get to the “python” directory:

Example:

```
#!/bin/bash

# A simple bash script to move up to desired directory level directly

function jump()
{
    # original value of Internal Field Separator
    OLDIFS=$IFS

    # setting field separator to "/"
    IFS=/

    # converting working path into array of directories in path
    # eg. /my/path/is/like/this
    # into [, my, path, is, like, this]
    path_arr=($PWD)

    # setting IFS to original value
    IFS=$OLDIFS

    local pos=-1

    # ${path_arr[@]} gives all the values in path_arr
    for dir in "${path_arr[@]}"
```

```

do
    # find the number of directories to move up to
    # reach at target directory
    pos=$((pos+1))
    if [ "$1" = "$dir" ];then

        # length of the path_arr
        dir_in_path=${#path_arr[@]}

        #current working directory
        cwd=$PWD
        limit=$((dir_in_path-pos-1))
        for ((i=0; i<limit; i++))
        do
            cwd=$cwd/..
        done
        cd $cwd
        break
    fi
done
}

```

For now, we cannot execute our shell script because it does not have permissions. We have to make it executable by typing following command –

```
$ chmod +x path/to/our/file/jump.sh
```

Now to make this available on every terminal session, we have to put this in “.bashrc” file.

“.bashrc” is a shell script that Bash shell runs whenever it is started interactively. The purpose of a .bashrc file is to provide a place where you can set up variables, functions, and aliases, define our prompt, and define other settings that we want to use whenever we open a new terminal window.

Now open the terminal and type the following command:

```
$ echo "source ~/path/to/our/file/jump.sh">> ~/.bashrc
```

Now open your terminal and try out new “jump” functionality by typing following command-

```
$ jump dir_name
```

Process Management

Process management is an essential component of the Linux operating system. Linux is known for its robust and stable performance, and it owes a lot to its process management capabilities. In simple terms, a program in execution is considered a process. The code is available in a readable format in an executable file inside a program's address space. The elements of activity inside a process are called threads and every thread has a unique program counter. This subsystem is responsible for managing all the processes and task lists on a Linux system. This article will discuss the basics of process management, types of process management, and commands used to manage these processes in Linux.

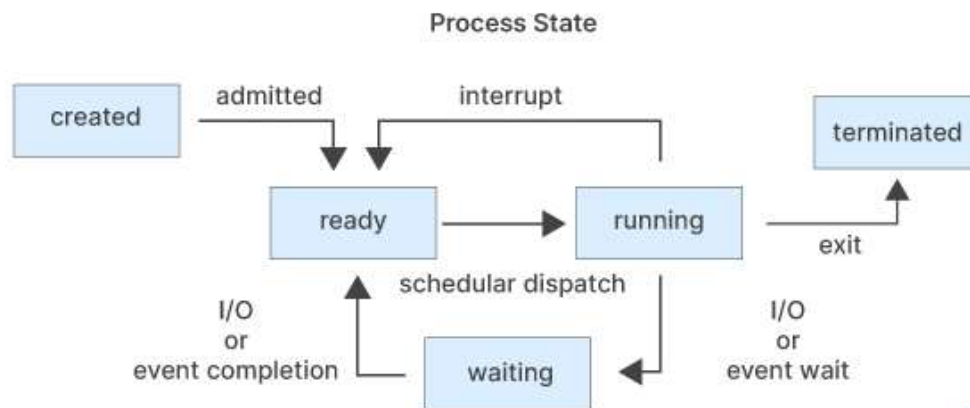
➤ Stages of a Process in Linux:

In Linux, a process goes through several stages during its lifetime. Understanding these stages and checking how they run in the background is important for process management and troubleshooting. The states of a process in Linux are as follows:

1. **Created:** A process is created when a program is executed. At this stage, the process is in a "created" state, and its data structures are initialized.
2. **Ready:** The process enters the "ready" state when it is waiting to be assigned to a processor by the Linux scheduler. At this stage, the process is waiting for its turn to execute.
3. **Running:** The process enters the "running" state when it is assigned to a processor and is actively executing its instructions.
4. **Waiting:** The process enters the "waiting" state when it is waiting for some event to occur, such as input/output completion, a signal, or a timer. At this stage, the process is not actively executing its instructions.

5. **Terminated:** The process enters the "terminated" state when it has completed its execution or has been terminated by a signal. At this stage, the process data structures are removed, and its resources are freed.
6. **Zombie:** A process enters the "zombie" state when it has completed its execution, but its parent process has not yet read its exit status. At this stage, the process details still have an entry in the process table, but it does not execute any instructions. The zombie process is removed from the process table when its parent process reads its exit status.

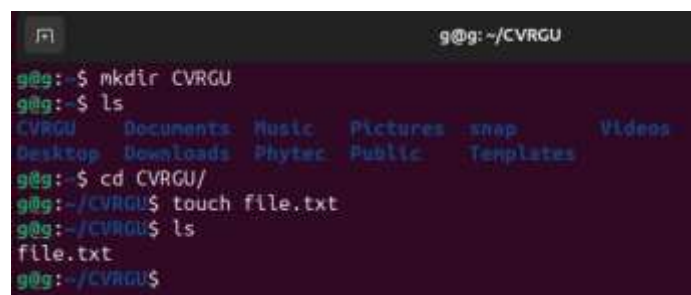
To view active processes and the current task lists in Linux we can use two popular commands i.e., ps and top command. The ps command displays information about the active programs or active processes on the system whereas the top command gives you a real-time view of the active processes in the running system.



File Operation

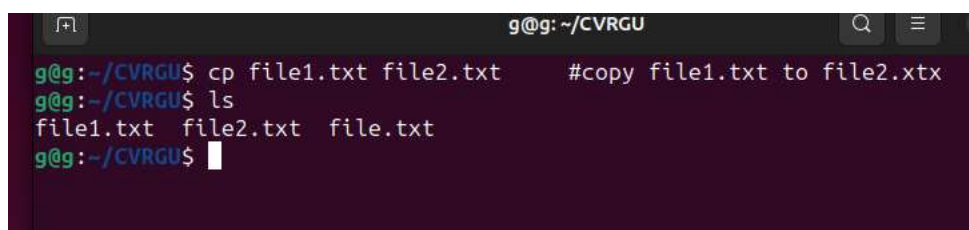
In Linux, file operations can be performed using various command-line tools and utilities. Here are some commonly used file operations in Linux:

1. **Creating Files:** The **touch** command is used to create empty files. For example:

A terminal window with a dark background and light green text. The prompt is 'g@g: ~/CVRGU'. The user enters 'mkdir CVRGU', then 'ls', which shows a list of directories: CVRGU, Documents, Music, Pictures, snap, Videos, Desktop, Downloads, Phytec, Public, and Templates. The user then enters 'cd CVRGU/', followed by 'touch file.txt', and finally 'ls', which shows 'file.txt'.

```
g@g:~/CVRGU
g@g:~$ mkdir CVRGU
g@g:~$ ls
CVRGU  Documents  Music  Pictures  snap  Videos
Desktop Downloads  Phytec  Public   Templates
g@g:~$ cd CVRGU/
g@g:~/CVRGU$ touch file.txt
g@g:~/CVRGU$ ls
file.txt
g@g:~/CVRGU$
```

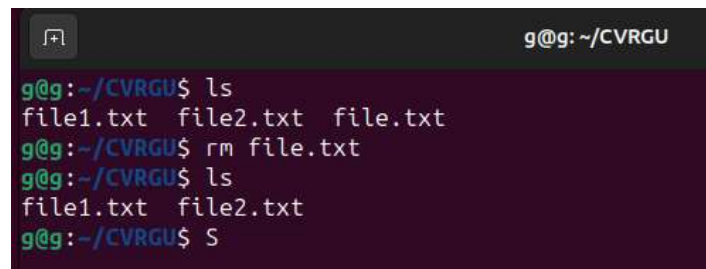
2. **Creating Directories:** The **mkdir** command is used to create directories. For example:
3. **Listing Files and Directories:** The **ls** command is used to list files and directories in a directory. For example: In Fig
4. **Copying Files and Directories:** The **cp** command is used to copy files and directories. For example:

A terminal window with a dark background and light green text. The prompt is 'g@g: ~/CVRGU'. The user enters 'cp file1.txt file2.txt', followed by a comment '#copy file1.txt to file2.txt'. Then the user enters 'ls', which shows 'file1.txt file2.txt file.txt'. The prompt is now 'g@g:~/CVRGU\$' with a cursor.

```
g@g:~/CVRGU$ cp file1.txt file2.txt      #copy file1.txt to file2.txt
g@g:~/CVRGU$ ls
file1.txt file2.txt file.txt
g@g:~/CVRGU$
```

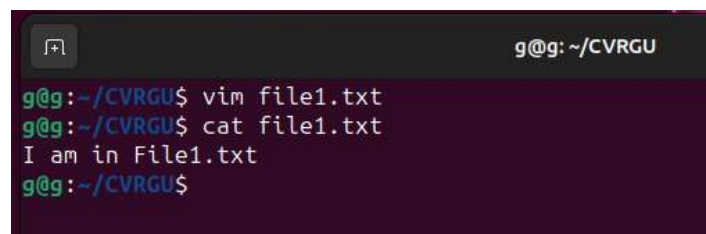

5. Moving/Renaming Files and Directories: The **mv** command is used to move or rename files and directories. For example: Fig

6. Deleting Files and Directories: The **rm** command is used to remove/delete files and directories. For example:

A terminal window with a dark purple background. The prompt is 'g@g: ~/CVRGU'. The user enters 'ls', showing 'file1.txt file2.txt file.txt'. Then they enter 'rm file.txt'. Another 'ls' command shows 'file1.txt file2.txt'. Finally, they enter 'S' and the prompt returns.

```
g@g: ~/CVRGU$ ls
file1.txt file2.txt file.txt
g@g: ~/CVRGU$ rm file.txt
g@g: ~/CVRGU$ ls
file1.txt file2.txt
g@g: ~/CVRGU$ S
```

7. Viewing File Contents: The **cat** command is used to display the contents of a file. For example:

A terminal window with a dark purple background. The prompt is 'g@g: ~/CVRGU'. The user enters 'vim file1.txt'. Then they enter 'cat file1.txt', which displays 'I am in File1.txt'. Finally, they enter '\$' to exit vim, and the prompt returns.

```
g@g: ~/CVRGU$ vim file1.txt
g@g: ~/CVRGU$ cat file1.txt
I am in File1.txt
g@g: ~/CVRGU$
```

8. Editing Files: You can use text editors like **nano**, **vim**, or **emacs** to edit files directly from the command line. For example: Fig

9. Changing File Permissions: The **chmod** command is used to change the permissions of files and directories. For example:

chmod **+x** script.sh # Add execute permission to script.sh

chmod **644** file.txt # Set read and write permission for the owner, and read permission for others on file.txt

number	read	write	execute
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 1

These are just a few examples of common file operations in Linux. There are many more commands and options available for working with files and directories in Linux. You can refer to the manual pages (**man** command) for each command to learn more about their usage and options.

Signals

A *signal* is an event generated by the UNIX and Linux systems in response to some condition, upon receipt of which a process may in turn take some action. We use the term *raise* to indicate the generation of a signal, and the term *catch* to indicate the receipt of a signal. Signals are raised by some error conditions, such as memory segment violations, floating-point processor errors, or illegal instructions. They are generated by the shell and terminal handlers to cause interrupts and can also be explicitly sent from one process to another as a way of passing information or modifying behaviour. In all these cases, the programming interface is the same. Signals can be raised, caught and acted upon, or (for some at least) ignored.

Signal names are defined by including the header file **signal.h**. They all begin with SIG and include those listed in the following table.

Signal Name	Description
SIGALRM	Alarm clock
SIGHUP	Hangup
SIGINT	Terminal interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader.
SIGQUIT	Terminal quit

SIGCHLD can be useful for managing child processes. It's ignored by default. The remaining signals cause the process receiving them to stop, except for SIGCONT, which causes the process to resume. They are used by shell programs for job control and are rarely used by user programs.

Linux Scheduler & Memory Management

In Linux, the scheduler and memory management are critical components of the operating system that handle task scheduling and memory allocation for optimal system performance. Here's an overview of Linux scheduler and memory management:

1. **Scheduler:** The scheduler in Linux is responsible for determining which tasks or processes should run on the CPU and for how long. Linux uses a priority-based scheduler called the Completely Fair Scheduler (CFS) as the default scheduler. It strives to provide fairness and responsiveness to tasks.

The CFS assigns a dynamic priority, known as the "nice value," to each process based on its priority level. Lower nice values indicate higher priority. The scheduler uses a red-black tree data structure to keep track of the processes and their priorities.

Additionally, Linux provides different scheduling policies such as `SCHED_FIFO`, `SCHED_RR`, and `SCHED_BATCH`, which allow users to prioritize and manage processes according to specific requirements.

2. **Memory Management:** Memory management in Linux involves managing the system's physical and virtual memory resources efficiently. The key components of Linux memory management include:

- **Virtual Memory:** Linux employs a demand-paging virtual memory system. Each process has its virtual address space, which is larger than the physical memory available. The kernel handles the translation of virtual addresses to physical addresses using page tables.
- **Paging and Swapping:** Linux uses a technique called paging to manage memory. It divides the virtual address space and physical memory into fixed-size blocks called pages. When a process accesses a page that is not present in physical memory, a page fault occurs, and the required page is fetched from disk into memory. If the system is low on memory, the least recently used (LRU) pages are selected for swapping out to disk.

- **Page Cache:** Linux utilizes the page cache to cache file data in memory, enhancing file I/O performance. The page cache keeps frequently accessed file data in memory, reducing disk I/O operations.
- **Memory Allocation:** The Linux kernel provides functions such as `malloc()` and `free()` to manage dynamic memory allocation within user-space programs. Internally, the kernel manages memory allocation using data structures like the slab allocator and the buddy system allocator.
- **Swappiness:** Swappiness is a parameter that determines the tendency of the Linux kernel to swap out pages from physical memory to disk. Administrators can adjust the swappiness value to optimize the balance between using physical memory and swap space.
- **Memory Management Unit (MMU):** The MMU is a hardware component that works in conjunction with the Linux kernel to handle memory management tasks, such as address translation and protection.

Linux provides various tools and commands like `free`, `top`, and `vmstat` to monitor memory usage and performance.

Both the scheduler and memory management subsystems in Linux play crucial roles in ensuring efficient resource utilization, responsiveness, and stability of the operating system.

Linux Multi-Threading Programming

In Linux, multi-threaded programming involves creating and managing multiple threads within a single process. Threads are lightweight execution units that can run concurrently within a process, sharing the same memory space. They allow for parallel execution and efficient utilization of system resources. Here's an overview of multi-threaded programming in Linux:

1. **Thread Creation:** Linux provides various libraries and APIs for creating threads. The most used library is the POSIX Threads (**pthread**s) library, which conforms to the POSIX thread standards. The **pthread**s library offers functions such as **pthread_create()** to create threads and **pthread_join()** to synchronize threads.

To create a thread using **pthread**s, you need to define a function that will be executed by the thread. Here's an **example**:

```
#include <pthread.h>

#include <stdio.h>

void* threadFunction(void* arg) {
    // Thread logic here

    return NULL;
}

int main() {
    pthread_t thread;

    pthread_create(&thread, NULL, threadFunction, NULL);

    // Main thread continues execution while the new thread runs concurrently

    pthread_join(thread, NULL);

    return 0;
}
```

1. **Synchronization:** When multiple threads access shared resources simultaneously, synchronization mechanisms are necessary to prevent race conditions and ensure data consistency. Linux provides synchronization primitives such as mutexes, condition variables, and semaphores.

Mutexes (**pthread_mutex_t**) can be used to protect critical sections of code by allowing only one thread to access them at a time. Condition variables (**pthread_cond_t**) facilitate thread synchronization based on certain conditions. Semaphores (**sem_t**) are used for controlling access to a shared resource with a limited capacity.

Proper use of these synchronization mechanisms is crucial to avoid issues like data corruption and deadlocks.

2. **Thread Communication:** Threads often need to communicate with each other to exchange data or coordinate their activities. Linux provides various mechanisms for inter-thread communication, such as pipes, message queues, shared memory, and sockets.

Pipes (**pipe**) provide a unidirectional communication channel between two related threads or processes. Message queues (**msgget**, **msgsnd**, **msgrcv**) enable message-based communication between threads or processes. Shared memory (**shmget**, **shmat**) allows multiple threads to access a shared memory region for efficient data sharing. Sockets (**socket**, **send**, **recv**) provide network-based communication between threads running on different machines.

3. **Thread Management:** Linux provides several functions to manage threads, including thread-specific data, thread attributes, and thread cancellation. The **pthread** library offers functions like **pthread_key_create()** and **pthread_setspecific()** to manage thread-specific data. Thread attributes (**pthread_attr_t**) allow you to control various properties of threads, such as stack size and scheduling policy. Thread cancellation functions (**pthread_cancel**, **pthread_setcancelstate**, **pthread_setcanceltype**) enable the cancellation of threads.

It's essential to properly manage thread resources to avoid memory leaks and ensure efficient execution.

4. **Debugging and Analysis:** Debugging multi-threaded programs can be challenging due to issues like race conditions and deadlocks. Tools like gdb (GNU Debugger) and valgrind (memory debugging and profiling tool) can assist in debugging and analyzing multi-threaded programs.
5. **Thread Safety and Atomic Operations:** In multi-threaded programming, thread safety is crucial to ensure that data accessed by multiple threads remains consistent. Linux provides atomic operations (such as atomic variables and atomic instructions) that guarantee thread-safe access to shared data without the need for explicit locking.
6. **Libraries and Frameworks:** In addition to the pthreads library, Linux offers various high-level libraries and frameworks that simplify multi-threaded programming.

Inter Process Communication

What Is a Pipe?

We use the term *pipe* to mean connecting a data flow from one process to another. Generally, you attach, or pipe, the output of one process to the input of another. Most Linux users will already be familiar with the idea of a pipeline, linking shell commands together so that the output of one process is fed straight to the input of another. For shell commands, this is done using the pipe character to join the commands, such as

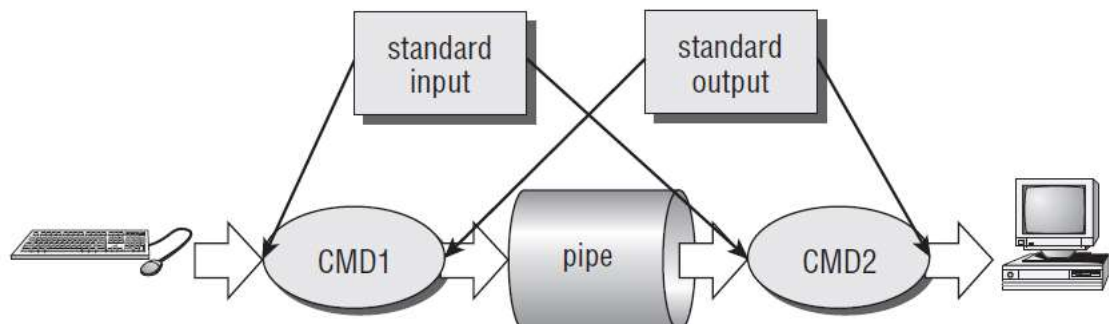
cmd1 | cmd2

The shell arranges the standard input and output of the two commands, so that

- ☐ The standard input to **cmd1** comes from the terminal keyboard.
- ☐ The standard output from **cmd1** is fed to **cmd2** as its standard input.
- ☐ The standard output from **cmd2** is connected to the terminal screen.

What the shell has done, in effect, is reconnect the standard input and output streams so that data flows

from the keyboard input through the two commands and is then output to the screen. See Figure 1 for a visual representation of this process.



Process Pipes

Perhaps the simplest way of passing data between two programs is with the `popen` and `pclose` functions. These have the following prototypes:

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *open_mode);
```

```
int pclose(FILE *stream_to_close);
```

Parent and Child Processes

The next logical step in our investigation of the pipe call is to allow the child process to be a different program from its parent, rather than just a different process running the same program. You do this using the `exec` call. One difficulty is that the new **execed** process needs to know which file descriptor to access. In the previous example, this wasn't a problem because the child had access to its copy of the **file_pipes** data. After an `exec` call, this will no longer be the case, because the old process has been replaced by the new child process. You can get around this by passing the file descriptor (which is, after all, just a number) as a parameter to the newly execed program.

To show how this works, you need two programs. The first is the data producer. It creates the pipe and then invokes the child, the data consumer.

1. For the first program, you adapt `pipe2.c` to `pipe3.c`. The changed lines are shown shaded:

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{
int data_processed;
int file_pipes[2];
const char some_data[] = "123";
char buffer[BUFSIZ + 1];
pid_t fork_result;
memset(buffer, '\0', sizeof(buffer));
if (pipe(file_pipes) == 0) {
fork_result = fork();
if (fork_result == (pid_t)-1) {
fprintf(stderr, "Fork failure");
exit(EXIT_FAILURE);
}
if (fork_result == 0) {
sprintf(buffer, "%d", file_pipes[0]);
(void)execl("pipe4", "pipe4", buffer, (char *)0);
exit(EXIT_FAILURE);
}
else {
data_processed = write(file_pipes[1], some_data,
strlen(some_data));
printf("%d - wrote %d bytes\n", getpid(), data_processed);
}
}
exit(EXIT_SUCCESS);
}

```

Network Programming in Linux

Network programming in Linux involves developing applications that communicate over the network using various protocols and APIs. Linux provides a rich set of networking capabilities and libraries to facilitate network programming. Here's an overview of network programming in Linux:

1. **Socket Programming:** Sockets are the fundamental building blocks of network programming. Linux offers the Berkeley sockets API, which is widely used for network communication. Socket programming allows you to create client-server applications that can send and receive data over the network.

The socket API provides functions such as **socket()**, **bind()**, **listen()**, **accept()**, **connect()**, **send()**, and **recv()** to create, configure, and communicate over sockets. These functions enable the establishment of network connections, sending and receiving data, and handling errors.

2. **Internet Protocols:** Linux supports various internet protocols, including TCP/IP (Transmission Control Protocol/Internet Protocol) and UDP (User Datagram Protocol). TCP provides reliable, connection-oriented communication, while UDP offers lightweight, connectionless communication.

By utilizing the socket API, you can develop applications that communicate using TCP or UDP protocols. TCP sockets are commonly used for applications that require reliable, ordered, and error-checked delivery of data. UDP sockets are suitable for applications that prioritize low latency and do not require reliable data transmission.

3. **Network Addressing:** In Linux network programming, IP addresses and port numbers are used to identify network endpoints. IP addresses are represented as IPv4 (e.g., 192.168.0.1) or IPv6 (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334) formats. Port numbers range from 0 to 65535 and specify the application or service running on a particular host.

The struct `sockaddr_in` structure is commonly used to represent IP addresses and port numbers. It contains fields such as `sin_family` (address family), `sin_addr` (IP address), and `sin_port` (port number).

4. **Network Libraries:** Linux offers several network libraries and frameworks that simplify network programming. Some popular libraries include:

- **libpcap:** Provides packet capture and network analysis capabilities.
- **libcurl:** Enables easy implementation of HTTP, FTP, SMTP, and other network protocols.
- **OpenSSL:** Offers secure network communication using SSL/TLS protocols.
- **ZeroMQ:** Provides lightweight messaging for distributed systems.
- **Boost.Asio:** A C++ library for network and low-level I/O programming.

These libraries abstract the complexities of network programming and provide higher-level interfaces and functionalities.

5. **Network Configuration and Administration:** Linux provides a range of tools and utilities for network configuration and administration. Some commonly used tools include **ifconfig** for network interface configuration, **iptables** for firewall configuration, **route** for routing table manipulation, and **netstat** for network statistics and monitoring.

These tools can be used to configure network interfaces, manage network connections, monitor network traffic, and perform network diagnostics.

6. **Network Protocols and APIs:** Linux supports a wide range of network protocols and APIs. Besides TCP and UDP, it provides APIs for protocols such as ICMP (Internet Control Message Protocol), DNS (Domain Name System), HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), and more.

Developers can utilize these protocols and APIs to implement specific network functionalities within their applications.

Network programming in Linux offers extensive capabilities for building robust and scalable networked applications. It's important to understand the underlying network protocols, socket programming concepts, and available libraries to leverage the power of Linux in developing network applications.

Rugged Board: - Introduction

PhyCORE-A5D2x-SOM is a compact form-factor multi-layer (6-layer PCB Board) with 35x35mm dimension with 0.8mm Pitch with Edge Cartelization/ Edge Plated Holes based on the highperformance SAMA5D2 SIP series ultra-low-power 289 Pin BGA Module.



Fig 1

PhyCORE-A5D2x-SOM available in three different variants with same footprint pin-to-pin compatibility.

SOM Variants	A5D2x-SOM-“Premium”	A5D2x-SOM- “Professional”	A5D2x-SOM-“Standard”
SIP Module	ATSAMA5D27C-D5M-CU	ATSAMA5D27C-D1G-CU	ATSAMA5D28C-D1G-CU
Architecture	ARM Cortex-A5 CPU	ARM Cortex-A5 CPU	ARM Cortex-A5 CPU
Clock Frequency	500 MHz	500 MHz	500 MHz
DDR2-SDRAM (Part of SIP)	512Mbit (64MByte)	1Gbit (128MByte)	1Gbit (128MByte)
Flash Memory (NOR Type 32Mbit or 64Mbit)	SST26VF032B-104I/SM or SST26VF064BT-104I/SM	SST26VF064BT-104I/SM	SST26VF064BT-104I/SM
EEPROM memory with EUI-48™ Node Identity	24AA02E48T-I/OT	24AA02E48T-I/OT	24AA02E48T-I/OT
PMIC IC	MIC2800-G1JJYML	MIC2800-G1JJYML	MIC2800-G1JJYML
Ethernet PHY	KSZ8081RNAIA-TR	KSZ8081RNAIA-TR	KSZ8081RNAIA-TR
Operating voltage:	3.3V ± 5%	3.3V ± 5%	3.3V ± 5%
Temperature Grade /Range	Industrial Grade (-40°C ~ 85°C)	Industrial Grade (-40°C ~ 85°C)	- Industrial Grade (-40°C ~ 85°C)
Board thickness	1.2mm, Standard FR4	1.2mm, Standard FR4	1.2mm, Standard FR4
Package	QFN 160 Pins + 20 EPAD (GND): :Half Edge Plated Holes	QFN 160 Pins + 20 EPAD (GND): :Half Edge Plated Holes	QFN 160 Pins + 20 EPAD (GND): :Half Edge Plated Holes
Module Pitch	0.8mm Pitch	0.8mm Pitch	0.8mm Pitch
Dimension	35x35mm QFN, DSC	35x35mm QFN, DSC	35x35mm QFN, DSC
Variant Code	PhyCORE-PICM-A0_001	PhyCORE-PICM-A0_002	PhyCORE-PICM-A0_002

Table 1

The PhyCORE-A5D2x-SOM offers an extensive peripheral set, including High-speed USB Host and Device, HSIC Interface, 10Base-T/100Base-TX Ethernet Interface, system control and up to 100+ I/O's featuring with 3.3V IO level:

- Up to 4 UARTS
- Up to 4 Flexcoms
- Up to 6 Capacitive Touch lines for up to 9 touch buttons
- Up to 4 ADC Inputs
- Up to 2 CAN
- Up to 7 Tamper Pins
- Serial Interfaces such as SPI, TWI, QSPI, SSC and I²S
- SD/MMC, eMMC, SDIO Interfaces
- Up to 24-bit LCD RGB Interface
- CMOS Camera Interface
- Mono PDMIC and Full-Bridge Class-D Stereo
- Up to 6 Capacitive Touch Lines

Note:

Each I/O of the PhyCORE-A5D2x-SOM is configurable, as either a general-purpose I/O line only, or as an I/O line multiplexed with up to six peripheral I/O's. As the multiplexing is hardware defined, the hardware designer and programmer must carefully determine the configuration of the PIO Controllers required by their application.

PhyCORE_A5D2x SOM Architecture

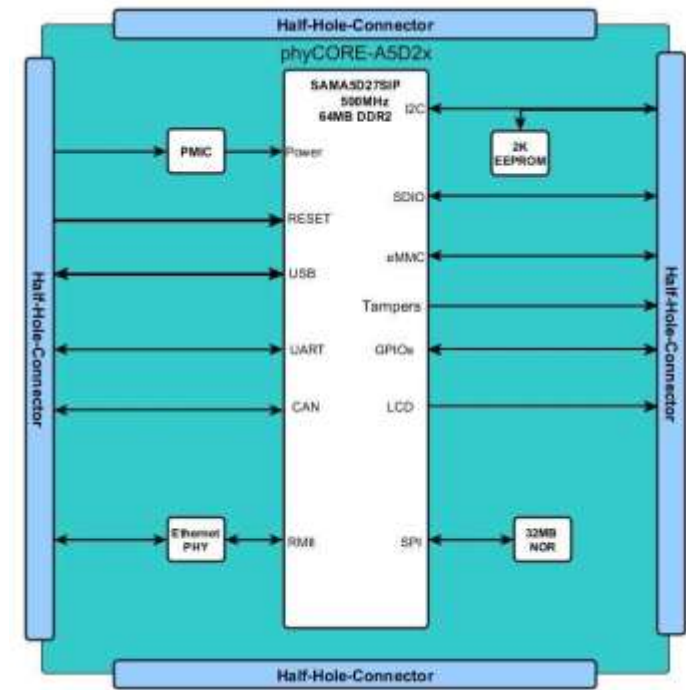


Fig 2

PhyCORE_A5D2x Components Placement Diagram

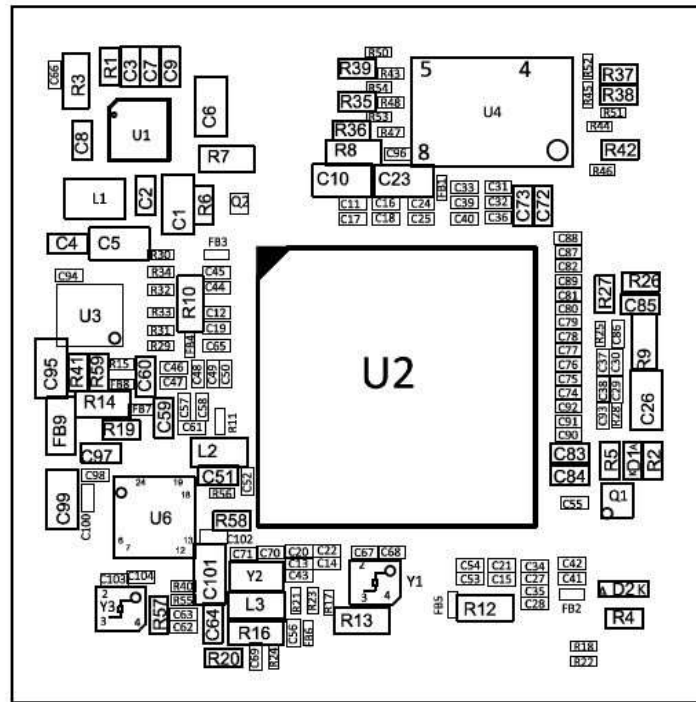


Fig 3

PhyCORE_A5D2x SOM: Power Source Types

1,2,159,160	VDDIN_3V3 (3V3_IN)	Main-power input Signal to SOM	Main 3.3V Supply inputs. Used for Peripheral I/O lines and MIC2800 supplies.	Power
158	VDDBU	Power to internal RTC	Input supply for Slow Clock Oscillator, internal 32 kHz RC Oscillator and a part of the System Controller	Power
3V3	VDDSDHC	Power to SD Card	SDMMC I/O lines supply input (N8 SOM BGA)	Power
3,4	VDDISC	Power to Sensors	Image Sensor I/O lines supply input	Power
34,37,41,80,81,118-121,157,161-180	GND	Ground	Ground connections (Must be connected together)	Power

Table 2

Important Note:

1. Signal = 'PIO' if GPIO; Dir = Direction; PU = Pull-up; PD = Pull-down; HiZ = High impedance; ST = Schmitt Trigger.
2. Refer to the DDR2-SDRAM datasheet for DDRM_VDDQ and DDRM_VDDL definitions. $\text{DDRM_VDDQ/DDRM_VDDL} = 1.8\text{V} \pm 0.1\text{V}$.

Types of Signals

Different types of signals are brought out at the **phyCore SOM**. The following table lists the abbreviations used to specify the type of signals.

Signal Type	Description	Abbreviation
Power	Main Supply Input Voltage	3V3_IN
Power	RTC backup Supply Voltage	VDDBU
Power	Image Sensor I/O lines supply input	VDDISC
Power	SDMMC I/O lines supply input	VDDSDHC
Input	Digital Input	I
Output	Digital Output	O
IO	Bidirectional Input/Output	IO
Ethernet Signals	Differential pair 100 ohm impedance	ETH

Table 3

Functional Description

SAMA5D2x System-In-Package

The SAMA5D2x System-In-Package (SIP) integrates the ARM Cortex-A5 processor-based SAMA5D2 MPU with upto 1 Gbit DDR2-SDRAM in a single package.

By combining the high-performance, ultra-low-power SAMA5D2 with DDR2-SDRAM in a single package, PCB routing complexity, area and number of layers is reduced. This makes board design easier and lowers the overall cost of bill of materials.

The SAMA5D27C-D1G-CU is available in a 289-ball TFBGA package.

DDR2-SDRAM Features

- Power Supply: VDD, VDDQ = 1.8 V \pm 0.1 V
- Double Data Rate architecture: two data transfers per clock cycle
- CAS Latency: 3
- Burst Length: 8
- Bi-directional, differential data strobes (DQS and DQSN) are transmitted/received with data
- Edge-aligned with Read data and center-aligned with Write data
- DLL aligns DQ and DQS transitions with clock
- Differential clock inputs (CLK and CLKN)
- Data masks (DM) for write data
- Commands entered on each positive CLK edge, data and data mask are referenced to both edges of DQS
- Auto-refresh and Self-refresh modes
- Precharged Power down and Active Power down
- Write Data Mask

- Write Latency = Read Latency - 1 (WL = RL - 1)
- Interface: SSTL_18

PhyCORE_A5D2x Primary “System Power (VDD_3V3)”

The PhyCORE_A5D2x SOM is supplied by an external 3.3V, ~2A rated current and generates its own internal supplies by interfacing with the Microchip MIC2800-G1JJYML -TR power management unit.

The MIC2800 is a high-performance power management IC (PMIC), providing three output voltages with maximum efficiency and is optimized to respect the MPU power up and down cycles. Integrating a 2 MHz DC/DC converter with an LDO post regulator, the MIC2800 gives two high-efficiency outputs with a second, 300mA LDO for maximum flexibility. The DC-to-DC converter uses small values of L and C to reduce board space while still retaining efficiency over 90% at load currents up to 600mA.

The three outputs supply the following internal nodes:

- DCDC set at 1.8V supplies PhyCORE_A5D2x DDR2 pads and device.
- LDO1 set at 1.25V supplies PhyCORE_A5D2x Core.
- LDO2 set at 2.5V supplies PhyCORE_A5D2x VDDFUSE pad.

The MIC2800 is a μ Cap design, operating with very small ceramic output capacitors and inductors for stability.

The SAMA5D2x requires a power source in order to permanently power the backup part of the SAMA5D2x device (refer to SAMA5D2x Series datasheet). The super capacitor / VRTC sustains such permanent power to VDDBU when all system power sources are off “VDDBU”.

For Proper operation of the PhyCORE-A5D2x SOM module must be supplied with a voltage source of $3.3V \pm 5\%$ at the VCC pins of the SOM QFN pads.

Connect all 3.3V VCC input pins to your power supply and at least the matching number of GND pins and some of the GND pins located underneath the Module.

PhyCORE_A5D2x SOM Current Consumption Table

System Control

The PhyCORE_A5D2x SOM provides global system Reset (NRST) and Shutdown (SHDN) pins to the application board.

- The NRST pin is an output pin generated by the internal Power Management Unit (MIC2800G1JJYML) in respect with power sequence timing. It can be forced externally in case of a system crash and must be connected as described in the example schematic below.
- The SHDN pin is an output pin and is managed by the software application. It switches the Main 3.3V Supply ON or OFF.

Ethernet PHY (10BASE-T/100BASE-TX)

The Microchip PhyCORE_A5D2x SOM embeds a single-supply 10BASE-T/100BASE-TX Ethernet physical layer transceiver for transmission and reception of data over standard CAT-5 unshielded twisted pair (UTP) cable.

The KSZ8081RNAIA is a highly-integrated PHY solution. The KSZ8081RNAIA offers the Reduced Media Independent Interface (RMII) for direct connection to RMII-compliant MACs in Ethernet processors. The

board supports RMII interface modes. The Ethernet interface consists of two pairs of lowvoltage

differential pair signals. These signals can be used to connect to a 10/100 Base-T RJ45 connector integrated on the Carrier Board.

Additionally, for monitoring and control purposes, LED functionality is carried on the RJ45 connectors to indicate activity, link, and speed status information.

For more information about the Ethernet controller device, refer to the Microchip KSZ8081RNAIA controller.

QSPI Memory

The PhyCORE_A5D2x SOM embeds the SST26VF064BT-104I/MF, a 64Mbit Serial Quad I/O Flash memory. The SST26VF032B-104I/SM (32Mbit) SQI features a six-wire, 4-bit I/O interface that allows for low-power, high-performance operation in a low pin-count package.

The **SST26VF032B-104I/SM** is available in 8-SOIC (0.209", 5.30mm Width) package.

EEPROM Memory

The PhyCORE_A5D2x SOM embeds the 24AA02E48T-I/OT, a 1Kb Serial EEPROM with preprogrammed EUI-48 MAC address.

The device is organized as one block of 128 x 8-bit memory with a 2-wire serial interface. The second block is reserved for MAC Address storage.

The 24AA02E48T-I/OT also has a page write capability for up to 8 bytes of data.

The 24AA02E48T-I/OT is available in the standard 5-lead SOT-23 package.

Important: If the 2-Wire serial interface is used externally, the device connected must have a different I²C address than the embedded EEPROM. For more details, refer to the device datasheet and more no of slave devices connected long distance means route I2C signals via a I2C buffer IC.

Power Supply Connections:

The PhyCORE_A5D2x SOM can be supplied in different ways depending on application needs.

Four power domains must be supplied and can be connected differently. The four different power connections are described below:

- **Power Configuration #1:** All supplies are connected to the Main 3.3V Supply.
- **Power Configuration #2:** Backup domain is connected to a coin-cell and the rest to the Main 3.3V Supply.

Conclusion

The immersion into Linux internals has provided a deep understanding of the operating system's inner workings, including the scheduler, memory management, and file operations. These insights into Linux internals are invaluable for optimizing system performance and efficiency.

The exploration of network programming in Linux has equipped us with the skills to develop applications that communicate over the network using sockets and various protocols. The understanding of socket programming, internet protocols, and network addressing enables us to create robust and efficient networked applications.

The collaborative nature of the internship, working alongside experienced professionals and receiving guidance from a dedicated supervisor, has accelerated our growth and learning. The support and knowledge shared by the host organization, technical staff, and fellow interns have been instrumental in broadening our perspectives and honing our skills.

This internship has laid a solid foundation for future endeavors in the field of Linux internals and rugged board technology. The knowledge gained and experiences gathered during this training period will undoubtedly contribute to our professional growth and enable us to make meaningful contributions in these domains.

In conclusion in this internship training on Linux internals and rugged board technology has provided a comprehensive understanding of Linux internals, network programming, and rugged board systems. This knowledge and experience will serve as a strong foundation for our future endeavors in these exciting fields.

REFERENCE

[1] GEEKS FOR GEEKS, “Linux Kernel”, GEEKS FOR GEEKS. [Online]. Available: <https://www.geeksforgeeks.org/the-linux-kernel/>

[2] GEEKS FOR GEEKS, “CPU Scheduling in Operating Systems”, GEEKS FOR GEEKS. [Online]. Available: <https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/>

[3] DEV, “File Operation with Linux Command”, DEV. [Online]. Available: <https://dev.to/nadirbasalamah/file-operation-with-linux-command-31a9>

[4] Ubuntu, “Manual Installation”, Ubuntu. [Online]. Available: <https://help.ubuntu.com/kubuntu/desktopguide/C/manual-install.html>

[5] Kaiwan N Billimoria October (2018). Hands-On System Programming with Linux.

[6] Dennis Yurichev October (2018). Reverse Engineering for Beginners.