Summer Research Fellowship Programme (SRFP)

IASc - INSA - NASI

Future Wireless Communications (FWC)
Hardware Programming and Data Handling
8 WEEK INTERNSHIP REPORT

Submitted by

Prasanna R
ENGS1417

B-Tech 2nd year
National Institute of Technology, Tiruchirappalli

Under the Guidance of

Dr. G.V.V. Sharma
Associate Professor
Department of Electrical Engineering
IIT Hyderabad

ACKNOWLEDGMENT

I take this opportunity to express my sincere gratitude to the Indian Academy of Sciences (IASc), Bengaluru, the Indian National Science Academy (INSA), New Delhi, and The National Academy of Sciences, India (NASI), Prayagraj, for offering me the prestigious Summer Research Fellowship Programme (SRFP). This program has been an enriching and transformative experience that allowed me to explore and contribute to real-world research in my area of interest.

I am immensely thankful to my research guide, Dr. G.V.V. Sharma, Department of Electrical Engineering, Indian Institute of Technology Hyderabad, for his invaluable mentorship, constant support, and insightful guidance throughout the project. His patient and motivating supervision not only helped me understand the subject in depth but also encouraged independent thinking and problem-solving.

I would like to extend my heartfelt thanks to the faculty, research scholars, and lab members at IIT Hyderabad for creating a conducive research environment and for their readiness to help whenever required. Their encouragement and collaboration played a vital role in the successful completion of my work.

Lastly, I am grateful to my family and friends for their unwavering support, encouragement, and constant belief in me. Their presence and motivation kept me going throughout the duration of this program.

1 Introduction

This report outlines the work undertaken during a four-week internship program focused on the Future of Wireless Communications (FWC) through hardware programming and data handling. The internship emphasized a hands-on approach to learning fundamental concepts in embedded systems, digital design, and low-level programming using various microcontrollers and development boards.

The primary objective of this internship was to bridge the gap between theoretical digital design concepts and their practical implementation. The program involved the use of embedded C, assembly language programming, and development platforms such as PlatformIO and Termux-based Linux environments.

Throughout the internship, various hardware components like the Arduino Uno, seven-segment displays, BCD-to-seven-segment decoders (7447 IC), and D flip-flop ICs (7474 IC) were interfaced and programmed. We gained experience in using different toolchains, including AVR-GCC for compiling embedded C code, and writing low-level register-manipulation code in assembly language for the Atmega328P microcontroller.

The internship placed a strong emphasis on understanding the internal architecture of microcontrollers and manipulating peripheral devices directly through port registers. This enabled a deeper grasp of how microcontrollers interact with external hardware, a skill critical for embedded systems development. The use of open-source tools like Termux and Debian-based environments on mobile platforms also encouraged adaptability and resource-efficient development practices.

Furthermore, by exploring digital logic design through finite state machines (FSMs), flip-flops, and decoder circuits, the internship helped build a solid foundation in sequential logic design. These skills are essential for careers in hardware-software co-design, embedded product development, and systems programming.

The experience also promoted analytical thinking and debugging proficiency, as engineers often had to troubleshoot hardware-software interaction errors. This strengthened their problem-solving abilities and made them more confident in working with real-world electronic systems.

In an era where IoT, automation, and smart embedded solutions are rapidly growing, the knowledge and skills developed through this internship align well with current industry trends. This program has not only reinforced core engineering principles but has also equipped students with practical exposure that will be valuable in both academic and professional endeavors.

The report details the platforms used, development tools configured, programming techniques applied, and the various digital circuits implemented, along with challenges faced and solutions developed. The following pages provide a detailed account of the technical knowledge gained and practical skills acquired during this internship.

1.1 Termux

Termux is a versatile terminal emulator and Linux environment application for Android that allows users to run a full-fledged Linux distribution without the need for root access. It provides a minimal base system and supports package management through APT, enabling the installation of a wide range of command-line tools, compilers, and programming languages such as Python, C, and Node.js. Termux can be used for software development, running scripts, accessing remote servers via SSH, and even setting up chroot or proot environments to simulate other Linux distributions. Its lightweight design and flexibility make it a powerful tool for developers who want to perform Linux-based tasks directly from their Android devices.

1.2 Debian

Debian is a widely used, stable, and free operating system based on the Linux kernel. Known for its reliability and large software repository, Debian serves as the foundation for many popular distributions, including Ubuntu and Raspberry Pi OS. It follows strict guidelines for open-source software and is maintained by a global community of developers. Debian supports a wide range of hardware architectures and provides thousands of precompiled packages, making it suitable for servers, desktops, and embedded systems alike. Its package management system, based on APT and .deb files, ensures easy installation, updating, and removal of software, contributing to its reputation as a dependable and secure Linux distribution.

Debian can be run within Termux using a proot or chroot environment, allowing users to simulate a full Linux system on an Android device. This setup provides access to the Debian package manager (apt), enabling installation of development tools, programming languages, and libraries just like on a regular desktop. By combining Termux with Debian, users can compile code, run servers, and manage Linux-based projects entirely on mobile. It eliminates the need for a separate PC, making it highly convenient for embedded systems and Linux-based development on the go. This setup is especially useful for educational or prototyping purposes where portability and accessibility are key.

All the codes developed during this internship were executed within the Debian environment installed in Termux. This setup provided a complete Linux workspace on an Android device, enabling compilation, testing, and debugging of Verilog, embedded C, and shell scripts. It allowed seamless use of tools like gcc, make, and yosys without the need for a traditional PC. The portability of Termux-Debian ensured continuous development even on the move.

2 Development platforms and tools used

During the internship, multiple development platforms and tools were used to support a wide range of embedded and low-level programming tasks. PlatformIO, running inside the Termux-Debian environment, was used for writing, building, and uploading firmware to microcontrollers like ESP32, offering features like flexible configuration. ArduinoDroid, an Android-based IDE, enabled on-device Arduino code development and uploading without a PC. AVR-GCC, the GNU compiler for AVR microcontrollers, was used to write and compile C code for boards like ATmega328. Assembly language was also used for direct hardware-level programming and optimization. Tools like make, gcc, and gdb were accessed through Debian in Termux for compiling and debugging. This diverse set of platforms allowed efficient cross-platform development directly from a mobile device, eliminating the need for a traditional desktop environment.

2.1 Platformio

PlatformIO is an open-source ecosystem for embedded development that supports a wide range of microcontroller platforms such as ESP32, STM32, AVR, and more. It integrates tools for code building, uploading, library management, and debugging, offering a streamlined workflow for embedded programmers. Unlike traditional IDEs like the Arduino IDE, PlatformIO provides advanced project structuring and supports multiple frameworks, including Arduino, ESP-IDF, and Zephyr.

One of the key features of PlatformIO is its cross-platform nature—it runs on Windows, Linux, macOS, and even within terminal-based environments like Termux-Debian on Android. It uses a command-line interface (pio) to build, upload, and monitor code, making it ideal for automation and scripting. PlatformIO's built-in library manager also simplifies the integration of third-party modules, allowing developers to quickly add and manage dependencies.

During this internship, PlatformIO was used extensively within the Termux-Debian setup to write, compile, and upload firmware to ESP32 boards. This allowed full development workflows to be carried out entirely from a mobile device. Its compatibility with VS Code (on desktop) and CLI-based environments (like Termux) made it a powerful and flexible tool for embedded systems development.

2.1.1 Installation

For the installation of Platformio, the following commands were executed in Termux :

```
apt update && apt install python3 python3-pip -y
pip3 install platformio
```

2.1.2 Compilation and execution

The following command was executed in Termux for compiling various codes:

```
pio run
```

The firmware.hex file, a machine-readable representation of the compiled program, is generated after its compilation, which is uploaded and flashed to the Arduino Uno through USB-OTG cable from ArduinoDroid, which is an Android application that allows users to write, compile, and upload Arduino sketches directly from their mobile devices without requiring a PC.

Various codes were tested, compiled, and executed using PlatformIO. These included fundamental embedded programs such as blinking an LED and displaying digits on a seven-segment display. Additionally, interfacing and functionality of digital ICs like the 7447 (BCD to seven-segment decoder) and 7474 (D-type flip-flop) were implemented. The Arduino Uno board was used as the primary development and testing platform. These experiments helped in understanding basic digital logic operations, hardware interfacing, and real-time behavior of ICs when integrated with microcontroller-based systems.

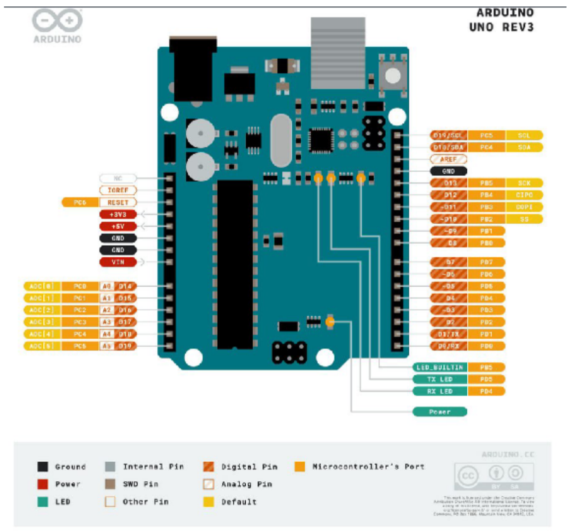


Figure 1: Arduino Uno pin configurations

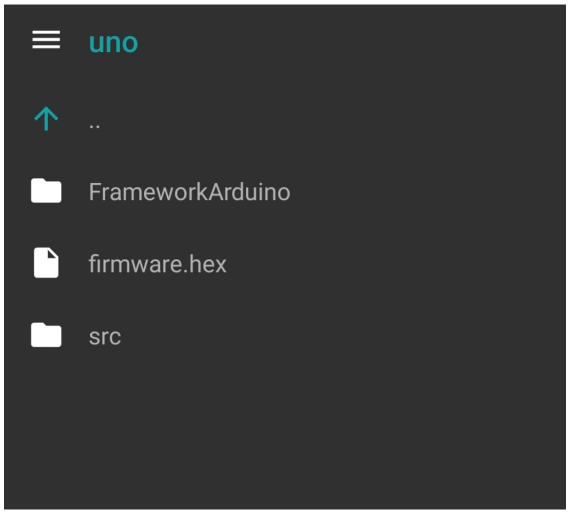


Figure 2: Arduino Droid interface for uploading

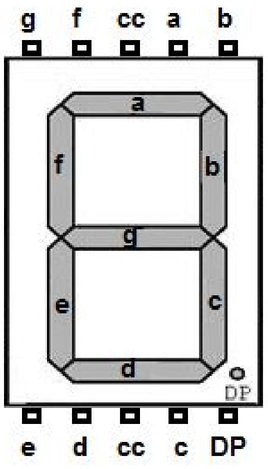


Figure 3: Common anode seven segment pin out

2.1.3 7447 IC logic

The Arduino code was written based on the logic given below:

```
a = BC'D' + A'B'C'D
b = BC'D + BCD'
c = B'CD'
d = AD + BCD + B'C'D + BC'D'
```

```
e = D + BC'
f = CD + A'B'D + A'B'C
g = A'B'C' + BCD
```

This logic implements a binary to 7-segment decoder (7447 IC logic). It reads 4-bit binary input from digital pins 12 to 9 (A–D), interprets the value, and computes which segments (a–g) of a 7-segment display should be turned on.

The results are output to digital pins of Arduino, each controlling a corresponding segment on the display, according to the logic.

2.1.4 7474 IC logic

The following is the code for the implementation of the 7474 IC, which is the D Flip-Flop. It was programmed to show the decade counter, and other sequential logic functions. The logic snippet from the code for decade counter is given below:

```
    if (count > 9) {
    count = 0;
  }

  // Convert count to individual BCD bits
  int A = count % 2;                    // bit 0
  int B = (count / 2) % 2;              // bit 1
  int C = (count / 4) % 2;              // bit 2
  int D = (count / 8) % 2;              // bit 3
```

The codes are compiled and uploaded through a USB cable to arduino.

2.1.5 FSM and K-Maps

Various finite state machines (FSMs), such as incrementing and decrementing counters, were implemented using the Arduino platform. The state transition logic was developed using standard FSM principles and encoded in embedded C. Digital inputs were used to trigger state changes, and outputs were monitored using LEDs. Testing was carried out on the Arduino Uno board to verify state progression and transitions.

The states s0 to s9 represent the sequence of counting from 0 to 9. The boolean expressions derived from state tables of this sequence were minimized using Karnaugh Map (K-map) techniques. This helped reduce logic complexity, gate count, and overall implementation effort. Simplified expressions were then used in the Arduino code for efficient control logic. The reduction also improved execution speed and made the code more readable. By applying K-map reduction, the digital logic design was optimized for both clarity and performance.

Overall, various combinational logic circuits were successfully implemented and tested using the PlatformIO environment. The logic was written in embedded C and deployed onto

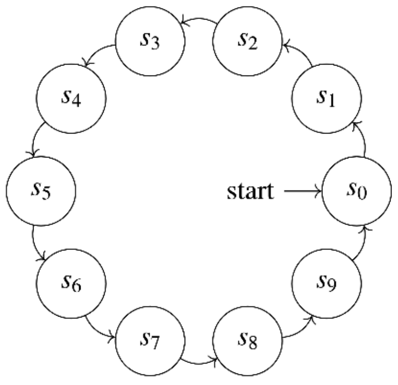


Figure 4: FSM for decade counter which was implemented after reducing with K-map

Arduino boards for real-time verification. Circuits such as decoders, multiplexers, adders, and segment drivers were developed and validated. PlatformIO provided an efficient and organized workflow for code management, compilation, and debugging. Pin-level testing using LEDs and input switches helped ensure accurate functional behavior. This hands-on implementation reinforced theoretical understanding through practical application. The PlatformIO environment thus proved to be a reliable tool for embedded combinational logic development.

2.2 Assembly programming

Assembly level programming is a low-level programming language that provides direct control over a computer's hardware. It uses symbolic representations of machine instructions, allowing programmers to write instructions specific to a processor's architecture. Unlike high-level languages, assembly enables precise manipulation of registers, memory locations, and I/O ports. It is often used in embedded systems, real-time applications, and situations requiring high performance or minimal resource usage. Assembly code is processor-dependent, meaning programs must be written for specific instruction sets like x86, ARM, or AVR. This type of programming offers better execution speed and efficient use of hardware, but requires deep hardware knowledge. Though more complex to write and debug, it is essential for tasks such as bootloaders, device drivers, and interrupt routines. Overall, assembly programming bridges the gap between hardware and software, offering critical control in system-level development.

2.2.1 Installations and setup

To begin assembly-level programming on AVR microcontrollers, the AVR toolchain must be installed. This includes avr-as, avr-ld, and avr-objcopy, along with the avr-libc library. On Linux systems, these can be installed using sudo apt install gcc-avr binutils-avr avr-libc. The avrdude tool is also required for uploading code to the microcontroller. A Makefile is used to automate the build and flashing process, improving efficiency.

In AVR assembly programming, .inc files are used to include predefined constants, register names, and bit definitions specific to a particular microcontroller.For example, in the internship, we had used the Atmega328p microcontroller for the arduino board. This has a m328Pdef.inc file, containing hardcoded registers and memory locations.

The pin mappings of the ATmega328P microcontroller was used as reference for writing the assembly codes for various logic

2.2.2 Compilation and execution

The follwing command is used to compile any assembly level code :

```
avra /path/of/file/filename.asm
```

Before executing any assembly level code, the m328Pdef.inc file must be added in the working path, to enable the compilation process

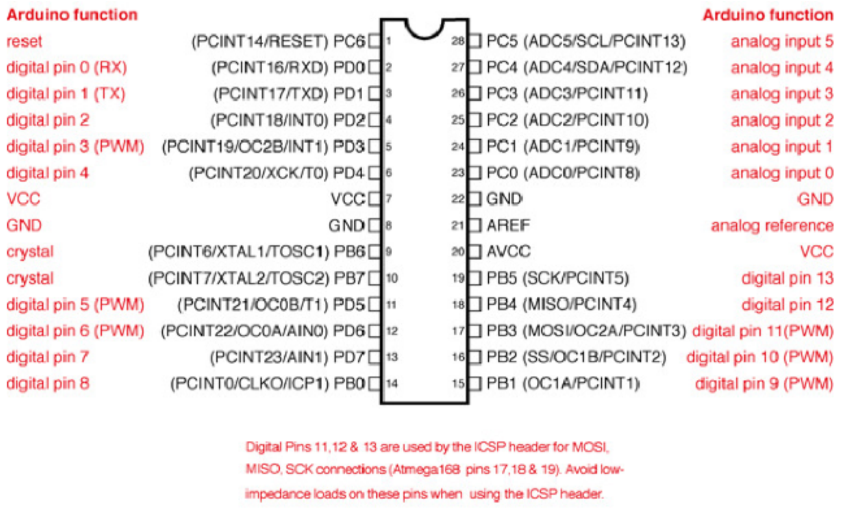


Figure 5: Pin mapping of ATmega328P microcontroller

The assembly code was compiled using AVR-specific assemblers such as avra, which converts the .asm source file into an Intel HEX format (.hex) suitable for microcontroller flashing. The generated HEX file contains machine-level instructions that can be uploaded to the microcontroller using avrdude and a compatible programmer like USBasp. During the upload process, the target device is specified using its microcontroller ID (e.g., m328p for ATmega328P). This workflow ensures that low-level instructions are correctly interpreted and executed by the hardware. The execution of the code was validated through peripheral outputs such as LEDs and GPIO responses.

The following are the commonly used assembly languuage instructions:

- **ldi rX, value** - Load Immediate: Loads a constant value into register rX (only valid for r16–r31).

- **out IO REG, rX** - Output to I/O: Writes the contents of register rX to the I/O register IO REG (e.g., PORTD, DDRB).
- **mov rX, rY** - Move: Copies the contents of register rY into register rX.
- **andi rX, value** - AND Immediate: Performs bitwise AND between rX and the constant value (used for masking bits).
- **eor rX, rY** - Exclusive OR (XOR): Performs bitwise XOR between rX and rY, stores result in rX.

The other instructions, such as in, and, or , etc follow the the same way of declaration

2.2.3 Seven segment display

Here the assembly code to control a seven segment display using the Atmega328P mmicrocontroller (Arduino) is given :

```
;using assembly language for
;displaying number on
;seven segment display

.include "/m328Pdef.inc"

;Configuring pins 2-7 (PD2-PD7) of Arduino
;as output
  ldi r16,0b11111100
  out DDRD,r16
;Configuring pin 8 (PB0) of Arduino
;as output
  ldi r16,0b00000001
  out DDRB,r16

;Writing the number 2 on the
;seven segment display
  ldi r17,0b11100000
  out PortD,r17

  ldi r17,0b00000001
  out PortB,r17
Start:
  rjmp Start
```

This assembly program is written for the ATmega328P microcontroller to display the digit ‘2’ on a common cathode seven-segment display. The code begins by including the device definition file m328Pdef.inc, which provides symbolic names for registers and ports. Pins PD2 to PD7 (Arduino digital pins 2–7) are configured as outputs to drive segments 'a' to 'g' of the display. Additionally, pin PB0 (Arduino pin 8) is configured as output to control the common cathode or enable line. The value 0b11100000 is loaded into PORTD to light up the required segments for displaying '2'. PB0 is set high to enable the display. The infinite loop at the end (rjmp Start) ensures the value remains latched continuously. This demonstrates basic GPIO control using AVR assembly language.

By varying the value in register r17, we can display other numbers. TBy changing values in r17, we turn on the required segments in the seven segnment display.

In the similar fashion, the assembly code for any combinational logic can be written using basic commands, such as the ldi, mov, eor, andi, and, or, etc.

To introduce delay, the assembly level code is modified to include the delay function, PAUSE, which can be used to give any delay in the circuit.

2.2.4 Conclusion

Assembly language offers precise and efficient control over microcontroller hardware, making it ideal for low-level programming tasks such as I/O manipulation and timer-based delays. Though it requires a thorough understanding of the architecture and is more complex to write and debug than high-level languages, it enables optimized performance with minimal resource usage. In applications where timing accuracy and direct hardware access are critical, assembly proves to be a powerful and reliable choice. Its deterministic behavior makes it suitable for real-time embedded systems. However, portability and scalability are limited compared to higher-level languages. Despite its complexity, learning assembly provides valuable insights into how hardware and software interact at the lowest level.

Many developers use assembly in conjunction with high-level languages like C to strike a balance between efficiency and maintainability. For instance, critical routines such as interrupt handlers or delay loops are often written in assembly, while the main application logic remains in C. Modern toolchains also support inline assembly, allowing specific low-level instructions to be embedded within high-level code. This hybrid approach leverages the strengths of both paradigms. Moreover, understanding assembly helps in debugging, as developers can interpret compiler-generated machine code and optimize performance bottlenecks. Overall, while not always necessary, assembly remains a vital tool in the embedded systems developer's skillset. This FWC course gives a basic foundation of using assembly level programming in very advanced applications.

2.3 AVR-GCC

AVR-GCC is a compiler toolchain used to write and compile C programs for AVR microcontrollers. It translates C code into machine code that can run on devices like the ATmega328P. The toolchain includes compiler (avr-gcc), assembler, linker, and uploader tools like avrdude. It allows easy integration of C libraries and hardware control while offering better readability than assembly.

2.3.1 Compilation and execution

To compile any AVR-GCC code, we include the Makefile in the project directory, to automate the compilation process. Another advantage of Makefile is that it handles dependencies well, reducing the errors. After including the Makefile the following command is executed to compile the sketch:

```
cd /path/to/project/directory
make
```

After the execution of this code, .hex file is built, which is flashed into the microcontroller. Similar method is employed to control the LEDs and seven segment displays.

3 Raspberry pi

The Raspberry Pi (RPI) is a low-cost, credit-card-sized single-board computer developed by the Raspberry Pi Foundation. It runs Linux-based operating systems like Raspberry Pi OS and supports programming in languages like Python, C/C++, and more. RPI has GPIO pins for hardware interfacing, making it ideal for IoT and robotics. It includes features like USB, HDMI, Wi-Fi, and Ethernet, and is widely used in prototyping.

3.1 Installation and setup

The raspberry os was flashed to raspberry pi through a micro sd card. After flashing it to rpi, the following command was used in termux to connect rpi to wifi:

```
ssh pi@ip.address
```

This command opens the rpi terminal to control the rpi and execute commands from rpi.

3.2 Compilation and other important commands

Below mentioned are some useful commands used for raspberry pi applications in this internship:

On termux, the following command is given to transfer the files from mobile to rpi:

```
scp /path/of/file pi@ip.address: /destination/path
```

This transfers the files from the specified directory in mobile to the specified directory in rpi.

4 Vaman

The VAMAN board is a compact development board based on the QuickLogic EOS S3 SoC, which integrates an Arm Cortex-M4F processor, embedded FPGA (eFPGA), and sensor hub functionality. It has the capabilities of ESP, FPGA and ARM. It is designed for low-power embedded AI/ML and IoT applications. The board includes onboard LEDs, buttons, and multiple I/O headers for interfacing with peripherals. It supports programming via JTAG or UART and is compatible with the QuickFeather and QORC SDK ecosystems. The eFPGA allows for custom digital logic to be integrated directly on-chip. VAMAN is ideal for rapid prototyping in embedded systems, edge AI, and hardware-accelerated applications.

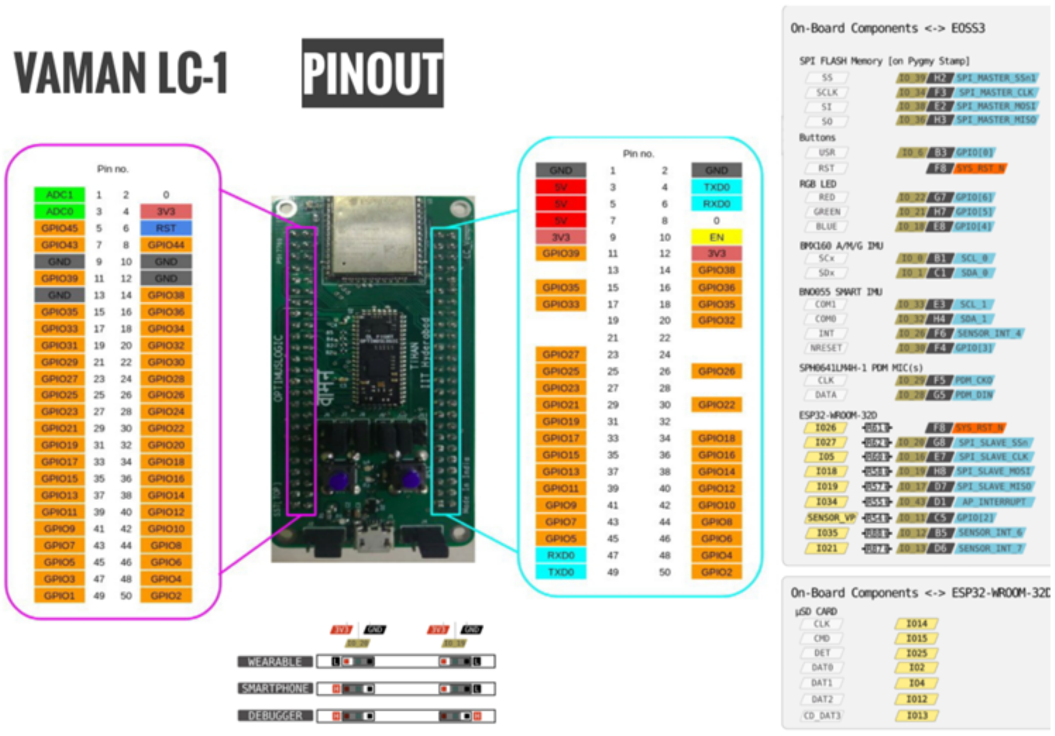


Figure 6: Pinout of Vaman Board

4.1 ESP

The VAMAN-ESP setup combines the VAMAN board with an ESP32 module to enable advanced processing and wireless communication. The VAMAN board handles sensor data processing and real-time control using its Cortex-M4F processor, while the ESP32 provides Wi-Fi and Bluetooth connectivity. Communication between the two is typically done using UART or SPI. This setup is ideal for IoT applications where local data processing is required before transmitting the results to the cloud or a mobile device.

4.1.1 Compilation and execution (Wired mode)

This subsection mentions the programming of ESP32 on vaman using arduino framework.

Initially, wired mode of communication was used to upload code to vaman ESP. The following command was used to compile the code:

```
pio run
```

An UART module, which refers to the Universal Asynchronous Receiver Transmitter, was used to transfer the compiled bin file to rpi. A separate module was used, because vaman doesnt have an inbuilt UART. In the rpi, the following command was executed in the project directory to upload the code to vaman ESP:

```
pio run -t nobuild -t upload
```

This uploads the code to vaman ESP through a JTAG cable.

4.1.2 OTA setup and compilation

The next part is the OTA (Over the Air) setup and commpilation. Initially, to setup the wifi communication, its essential to upload the OTA setup code in wired mode once to ESP. The Wifi network credentials are included in the code to establish the wifi connection.After uploading this code, in the future whenever we power the vaman board, the ESP automatically gets connected to wifi.

After its connected to wifi, we execute and uplaod the code through the following commands:

```
pio run
pio run -t upload --upload-port ip.address
```

Some of the codes such as blinking of LED and displaying a number on seven segment display were done using OTA.

4.2 FPGA

The VAMAN board features an embedded FPGA (Field-Programmable Gate Array) as part of the QuickLogic EOS S3 SoC. This FPGA allows users to implement custom digital logic circuits directly on the chip. It enables functions like PWM generation, signal processing, and hardware accelerators for specific tasks. The FPGA fabric is tightly integrated with the Cortex-M4F processor, allowing efficient interaction between hardware and software. Developers can use tools like Symbiflow or QORC SDK to design and program the logic. This makes VAMAN ideal for applications requiring hardware-level customization and real-time performance.

4.2.1 Verilog

Verilog is a hardware description language (HDL) used to model and design digital electronic systems like processors, memory, and controllers. It allows engineers to describe the structure and behavior of electronic circuits using code. Verilog supports both simulation (for testing logic) and synthesis (for converting code into actual hardware on FPGAs or ASICs). It includes constructs for combinational and sequential logic, such as always, assign, and if. Widely used in the semiconductor industry, Verilog helps in building and verifying complex digital designs efficiently.

4.2.2 Compilation and execution

The necessary packages were installed from from github using the following command:

```
wget https://raw.githubusercontent.com/gadepall/fwc-1/main/scripts/setup.sh
bash setup.sh
```

After the installations are done, any verilog code can be compiled using the following command:

```
source ~/.vamenv/bin/activate
ql_symbiflow -compile -src vaman/fpga/setup/codes/blink -d ql-eos-s3 -P
 PU64 -v filename.v -t modulename -p constraintsfile.pcf -dump binary
```

The source command activates the python virtual environment. It sets up environment variables so that tools like ql_symbiflow and related dependencies work in an isolated environment without affecting the system Python.

The next command compiles a Verilog design (helloworldfpga.v) for the QuickLogic EOS S3 FPGA using SymbiFlow. It uses the given source folder, part name (PU64), PCF constraints file (pygmy.pcf), and generates a binary bitstream for FPGA configuration.

```
source ~/.vamenv/bin/activate
 python3 TinyFPGA-ProgrammerApplication/tinyfpgaprogrammer-gui.py --
 port /dev/ttyACM0 --appfpga /home/pi/helloworldfpga.bin --mode fpga --
 reset
```

The vaman virtual environment is first activated and the next command runs the TinyFPGA Programmer GUI Python script to upload a bitstream file (helloworldfpga.bin) to the FPGA. It uses /dev/ttyACM0 as the serial port and programs the FPGA in application (FPGA) mode. The –reset option resets the board after programming.

Some of the codes such as the blinking of LED, controlling the seven segment display, etc were programmed and executed in the vaman pygmy.

4.3 ARM

The VAMAN board features an ARM Cortex-M4F processor as part of the QuickLogic EOS S3 SoC. This 32-bit microcontroller is optimized for low-power, high-performance embedded applications. It includes a floating-point unit (FPU) and DSP extensions, making it suitable for tasks like sensor data processing, audio handling, and edge AI inference. The ARM core executes the main application code, handles peripheral control, and interacts with the system's memory and I/O. Developers can write firmware in C/C++ using ARM toolchains, enabling real-time control and efficient processing. This makes the ARM core central to the VAMAN board's embedded functionality.

For uploading the code into vaman board, the bin file is transferred to rpi and the following command is executed in the rpi terminal:

4.3.1 Compilation and execution

To compile and run the ARM code several components are required. The ARM cross-compilation toolchain (arm-none-eabi-gcc) is used to compile the C code for the Cortex-M4F processor. A startup file and linker script are included to define the memory layout and initialize the system. The code consists of a main.c source file that configures. A Makefile is provided to automate the build process and generate a binary file (.bin). Finally, the binary is uploaded to the VAMAN board using the TinyFPGA Programmer tool via a USB interface.

The following command is used to compile the codes:

```
make -j4
```

The command make -j4 is used to compile a project using the make build system, with the -j4 option enabling parallel compilation. This means that up to four independent compilation tasks can run simultaneously, utilizing multiple CPU cores to speed up the build process. It is particularly useful for large projects with many source files, as it significantly reduces build time compared to running jobs sequentially. The number 4 represents the number of jobs or threads that make can execute in parallel, and it can be adjusted based on the number of processor cores available on the system.

4.3.2 Blinking of LED

The code for blinking a LED initializes the system hardware, configures specific GPIO pins as outputs, and then continuously toggles the LEDs on and off with delays in between. The program uses FreeRTOS and it includes support for debugging and command-line interaction. Low-level functions are defined to set GPIO direction, write values to output pins, and read input values, directly manipulating memory-mapped hardware registers. Here is the snippet of code for making LEDs blink:

```
PyHal Set GPIO(18,1);//blue
PyHal Set GPIO(21,1);//green
PyHal Set GPIO(22,1);//red
HAL DelayUSec(2000000);
PyHal Set GPIO(18,0);
PyHal Set GPIO(21,0);
PyHal Set GPIO(22,0)
```

This part of the code controls three onboard LEDs—blue (GPIO 18), green (GPIO 21), and red (GPIO 22)—on the VAMAN board. The lines first set all three GPIO pins to high , turning the LEDs ON simultaneously. The program then waits for 2 seconds using HAL_DelayUSec(2000000), which creates a delay in microseconds. After the delay, the three GPIO pins are set to low, turning off the LEDs. This sequence creates a visible blinking effect for the three LEDs together.

Similar to blinking, ARM codes were written to control seven-segment display and to test various combinational logic functions.

4.3.3 Conclusion

ARM codes are widely used in embedded systems, where efficiency and real-time control are essential, such as in consumer electronics, industrial machines, and medical devices. In the Internet of Things (IoT), ARM microcontrollers power smart home devices, environmental sensors, and wearable technology due to their low power consumption and performance. The automotive industry also relies on ARM-based systems for engine control units, infotainment systems, and advanced driver-assistance systems (ADAS). Additionally, ARM code is used in mobile and wearable devices for handling core functionality, sensors, and communication. This course serves as a base for development of various other advanced uses, such as the interchip communication.

5 Interchip Communication

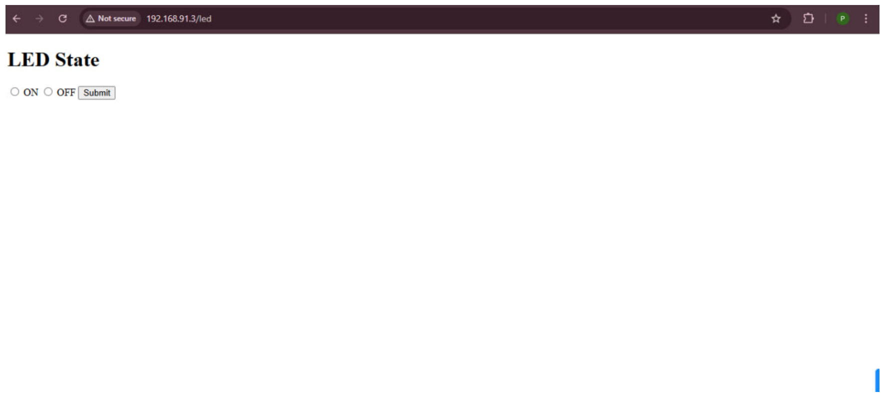
This section deals with the interchip communication between the ESP, FPGA and ARM chips/processors.

Interchip communication between ESP32, FPGA, and ARM Cortex-M4F in vaman board using SPI enables fast and efficient data exchange among the three devices, each playing a distinct role in an embedded system. The ESP32 typically handles wireless communication (Wi-Fi/Bluetooth), the FPGA manages custom logic or high-speed signal processing, and the ARM core (e.g., on the VAMAN board) serves as the central controller running application logic. SPI (Serial Peripheral Interface) is a synchronous, full-duplex protocol ideal for short-range, high-speed communication. It uses a master-slave architecture where one device (often ARM) controls the clock and data flow. This setup allows the ARM core to coordinate with the FPGA and ESP32, sending or receiving data in real time, making it suitable for tasks like sensor fusion, command processing, and data streaming in IoT or AI applications.

5.1 LED and seven segment display control

Here a LED is controlled using te interchip communication between ESP and FPGA, in which a web page is used to control the state of a LED. Controlling an LED through interchip communication between the ESP32 and FPGA involves sending control signals (e.g., ON/OFF commands) from the ESP32 to the FPGA via a protocol like SPI or UART. In this setup, the ESP32 acts as the master, transmitting data such as LED status or patterns. The FPGA acts as the slave, receiving these commands and driving the physical LED output pins accordingly. The FPGA logic decodes the received data and toggles specific GPIOs to control the LED. This allows the ESP32 to manage the LED remotely or wirelessly (e.g., via a web server or mobile app). The communication is synchronous and real-time, ensuring fast response. This setup is ideal in systems where the ESP32 handles wireless interfaces, and the FPGA handles low-level hardware control.

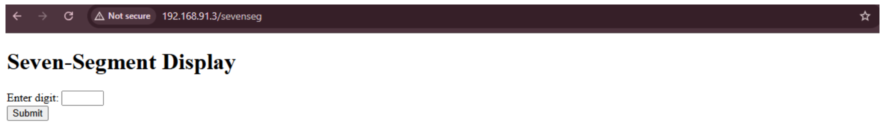
Similar to this, seven segmentdisplay is also controlled by manually entering the number that needs to be displayed.



Not secure 192.168.91.3/led

LED State

Figure 7: Controlling LED from webpage



Not secure 192.168.91.3/sevenseg

Seven-Segment Display

Enter digit:

Figure 8: Controlling seven segment from webpage

5.2 Conclusion

Inter-chip communication is widely used in embedded systems where multiple chips or processors need to exchange data efficiently. It is common in IoT devices, where a microcontroller (e.g., ARM) communicates with wireless modules (e.g., ESP32) or sensors via protocols like SPI, I^{2}C, or UART. In consumer electronics, interchip links connect processors with displays, memory, and peripheral controllers. Automotive systems use interchip communication to link ECUs, sensors, and actuators for real-time control. In AI edge devices, FPGAs and MCUs communicate to handle tasks like inference and data collection. It is also crucial in industrial automation and robotics for coordinating motion control, sensing, and wireless updates.

6 UGV

Unmanned Ground Vehicle's prototype was made to show its control using FPGA. An Unmanned Ground Vehicle (UGV) using an FPGA enables real-time control, fast data processing, and hardware-level customization for autonomous or remote operations. The FPGA can be programmed to generate PWM signals for motor control, handle sensor inputs, and implement logic for obstacle detection or path following. Its parallel processing capabilities allow the UGV to respond quickly to environmental changes. By integrating with other components like microcontrollers or wireless modules, the FPGA acts as the core controller for navigation and mobility. This makes FPGA-based UGVs ideal for applications in defense, agriculture, surveillance, and exploration.

6.1 Controlling of UGV

The connections to the L293 motor driver were made according to the pins declared in the verilog code. Connections were done from the declared GPIO pins to the pins A1, A2, B1, B2, Vcc and GND.

Initially, to demonstrate the basic control, manual inputs were applied. The 4 directions serves as 4 inputs for controlling the UGV movement, and 1 reset button for breaking. Whenever the inputs of the particular direction is high/low (depends on the active high/low as written in the code), the UGV moves in that particular direction.

Next, PWM control was demonstrated by defining duty cycles. 2 inputs were given to choose the direction and 2 more inputs to select the speed of UGV. By applying proper inputs, its possible to vary the speed of UGV based on the duty cycle.

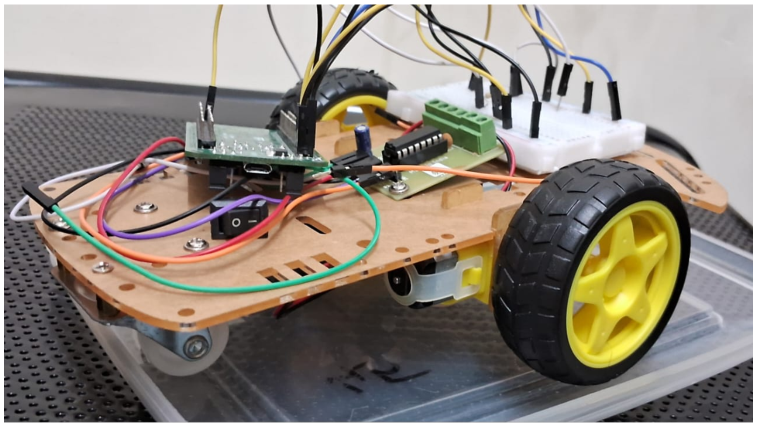


Figure 9: Controlling UGV through vaman

7 QuickLogic EOS S3

The QuickLogic EOS S3 is an ultra-low-power, multicore System-on-Chip (SoC) designed for IoT, wearable, and always-on sensing applications. It features a Cortex-M4F ARM processor for main application control and a dedicated Flexible Fusion Engine (FFE) for efficient sensor data processing with minimal power. The chip integrates a small, reconfigurable embedded FPGA (eFPGA), allowing developers to implement custom logic or accelerate tasks in hardware. It includes standard peripherals like SPI, I^{2}C, UART, PDM audio, ADC, and DMA controllers for real-time communication and control. The architecture is optimized for low-power operation, supporting dynamic clock and power gating across subsystems. Its compact footprint and versatility make it ideal for smart embedded systems where energy efficiency and adaptability are key.

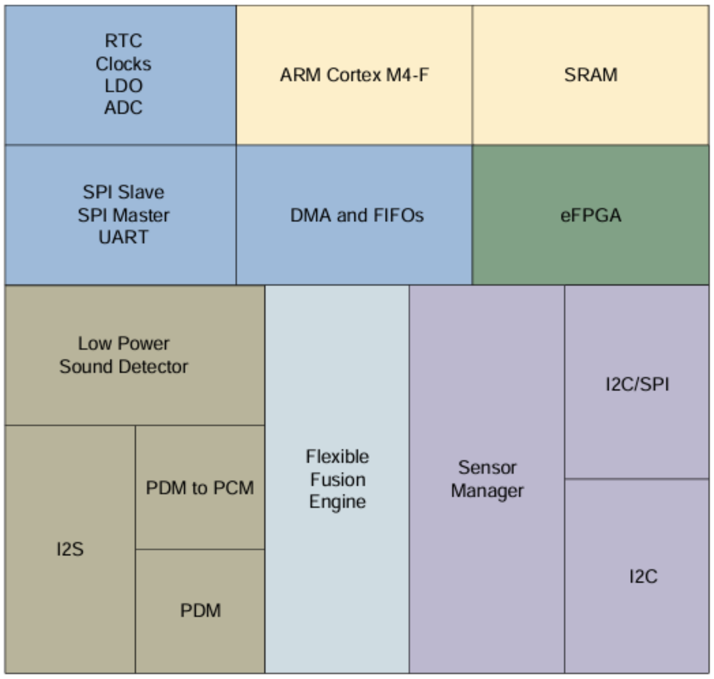


Figure 10: EOS S3 Ultra Low Power multicore MCU Platform Architecture

Some of its features are mentioned below:

- M4-F Subsystem
- Digital Microphone Support
- Power Management Unit
- On-chip Programmable Logic
- High Frequency Clock Source
- System DMA

There are many other excellent features and some of them are mentioned here. Each feature of this SoC is very useful in designing various embedded system applications.

QuickLogic EOS S3 chips are widely used in wearable devices and IoT edge applications where ultra-low power consumption and real-time processing are essential. They are ideal for always-on sensor fusion, voice recognition, and gesture detection in fitness trackers, smart home devices, and health monitors. The built-in FPGA allows for custom logic implementation, making it highly flexible for specialized tasks. In industrial systems, EOS S3 is used for real-time control and monitoring. Its compact design and low power architecture make it suitable for battery-operated and space-constrained applications.

8 Conclusion

Through this hands-on project, we explored embedded systems using tools like PlatformIO, Termux, and AVR-GCC, working directly with AVR assembly, ESP32, ARM, and FPGA platforms. We wrote low-level code to control small circuits such as LEDs, motors, and seven-segment displays, gaining real-time experience in GPIO manipulation and peripheral interfacing. Using SPI and UART protocols, we implemented inter-chip communication between ESP, ARM, and FPGA, reinforcing hardware-software coordination. These experiments helped build a strong foundation in embedded logic, timing control, and hardware-level debugging—all essential for real-world embedded design.

This foundational work serves as a stepping stone for advanced technologies like IoT, edge AI, autonomous systems, and next-gen wireless communication. The skills we practiced—efficient coding, protocol-level data handling, and hardware control—are directly applicable to modern applications in smart devices, robotics, and wireless sensor networks. With hardware becoming increasingly intelligent and connected, our experience with low-level programming and chip-level coordination prepares us to build scalable, adaptive systems for future innovations.

9 References

- https://github.com/gadepall/fwc-1
- https://github.com/gadepall/vaman Support
- https://github.com/gadepall/embedded-system/tree/main/inter-chip