Custom ESP32-C3 Mini Development Board

Ritvik Garg

July 21, 2025

1 Overview

This project involves the development of a compact, microcontroller-based communication module centered around the **ESP32-C3 microcontroller**, featuring USB-to-serial conversion via the **CH340C chip** and CAN bus communication using an external **CAN transceiver**. The primary objective is to create a **reliable interface** that allows seamless **programming, debugging, and robust communication** for embedded systems operating in electrically noisy or distributed environments such as robotics, automotive, or industrial networks.

1.1 Analogy

The **ESP32-C3** acts like the **"brain of the robot"** - it runs the robot's software, makes decisions, processes sensor data, and sends commands to motors or other actuators. The **CAN bus transceiver** helps the brain of the robot communicate with its other parts i.e., it is a **communication network** inside the robot. The **JTAG debug header** allows for **X-ray vision into the "brain"**. This special connector helps in **debugging** the microcontroller. The **USB-to-serial interface** is a **gateway for programming and communication**, allowing the user to send instructions to the robot and get logs and sensor data back from the robot. The **GPIO headers** connect the brain to its **eyes and ears i.e. sensors**. So, the mini-development board can be though of as the **core nervous system** of a robot.

2 Component Selection

The table below outlines the key components used in the design:

Component	Part Number	Manufacturer
Microcontroller	ESP32-C3 Mini-1	Espressif Systems
USB-to-serial interface	CH340C	WCH(Jiangsu Qin Heng)
CAN bus transceiver	SN65HVD230	Texas Instruments
Voltage regulator	AP2112	Diodes Inc.
USB Connector	USB-B10	XUNPU

Table 1: Key Components

Other components : Tactile switch (push button), capacitors, resistors, male headers

3 Circuit Schematic

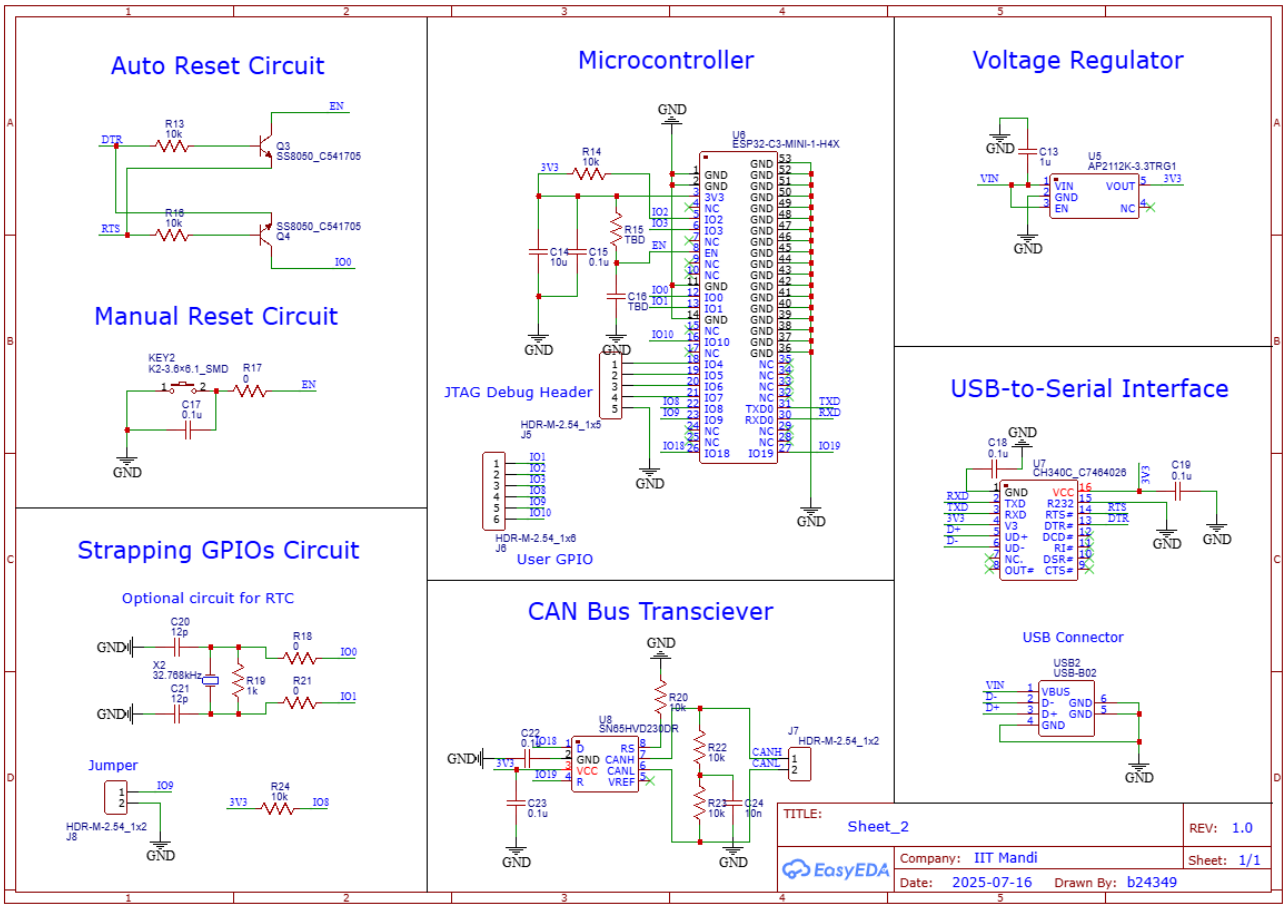


Figure 1: Circuit Schematic

3.1 CAN Bus Connection

3.1.1 What is CAN?

- CAN (Controller Area Network) is a robust, serial communication protocol which enables multiple microcontrollers or devices (known as **nodes**) to communicate over a shared bus without the need for a central host.
- Key features of CAN include :
  - **Differential signaling**, which improves noise immunity and enables long-distance communication.
  - **Multi-master capability**, allowing any node to initiate communication.
  - **Message prioritization and arbitration**, ensuring efficient use of the bus without collisions.
  - **Error detection and fault confinement**, which enhance reliability in harsh environments.
- Originally developed for vehicles, CAN is now also widely used in automation systems, medical equipment, robotics, and other embedded applications that demand reliable, real-time communication.

3.1.2 Data Transmission in CAN

- CAN transmits data over two wires :

- **CANH** (CAN High)
- **CANL** (CAN Low)

- These two lines carry inverted versions of the signal, forming a differential pair. Instead of interpreting logic levels based on voltage to ground, the receiver measures the voltage difference between CANH and CANL.

- This differential nature makes CAN highly resistant to electromagnetic interference (EMI), which is crucial in environments like vehicles or industrial machinery.

3.1.3 Implementation with ESP32-C3

- The **ESP32-C3** features a built-in **TWAI (Two-Wire Automotive Interface)** controller, which is Espressif's name for its CAN 2.0 controller. However, this is only the **controller layer** - it needs an **external CAN transceiver** to drive the differential bus.

- Any of the **GPIOs** can be chosen to support the TWAI. This design uses **GPIO18** and **GPIO19** for that purpose

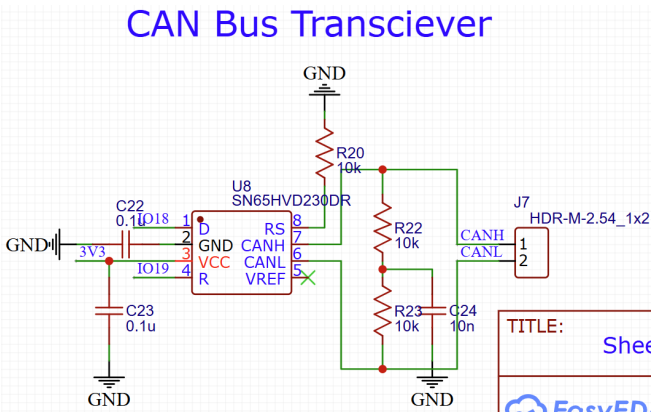


Figure 2: Circuit Schematic of SN65HVD230

- To prevent signal reflections on the CAN bus (which can corrupt data), termination resistance of 120Ω is used. However, the datasheet recommends a **split termination** layout to improve **common-mode noise performance** (common-mode noise refers to unwanted electrical signals that appear equally on both lines of a differential pair).

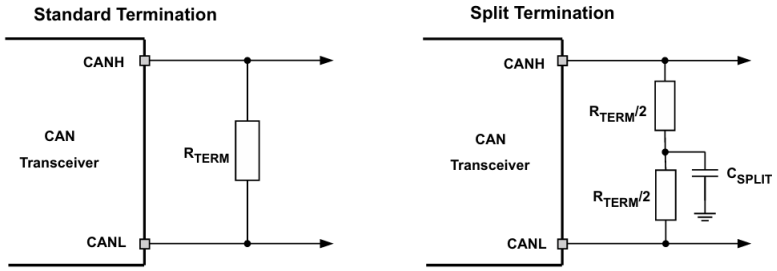


Figure 3: CAN Bus Termination Concept

3.2 JTAG Interface

3.2.1 What is JTAG?

- **JTAG (Joint Test Action Group)** is an **industry-standard debug and boundary scan interface** used in microcontrollers, FPGAs, and other digital ICs.

- It provides developers with **low-level access** to the **internal workings of a chip**, including:

- Program memory
- Register states
- Peripheral configuration
- Real-time debugging

3.2.2 JTAG Signals

Signal	Full Name	Function
TDI	Test Data In	Serial input to shift instructions and data into the JTAG chain
TDO	Test Data Out	Serial output from the chip (data shifted out)
TCK	Test Clock	Clock signal that controls the shifting of data/instructions
TMS	Test Mode Select	Determines the JTAG state machine's transitions
GND	Ground	Electrical reference for the interface

Table 2: JTAG Signal Explanation

The TMS and TCK lines drive the internal finite state machine of the JTAG logic, while TDI and TDO shift data into and out of the target device.

3.2.3 Implementation with ESP32-C3

This design uses the default GPIO mapping for JTAG on the ESP32-C3:

ESP32-C3 GPIO	JTAG Signal	Header Pin	Description
IO5	TDI	1	Input data line
IO7	TDO	2	Output data line
IO6	TCK	3	Clock input
IO4	TMS	4	Mode select input

Table 3: ESP32-C3 JTAG Pin Mapping

These are connected to a **5-pin 2.54 mm pitch header**, allowing compatibility with standard JTAG cables and adapters.

3.3 Boot Modes of ESP32-C3

3.3.1 Flashing Interface

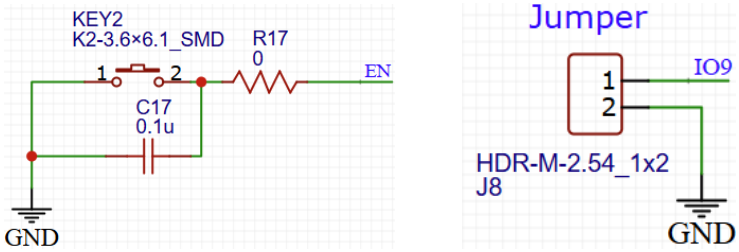
- **Flash memory** is a type of **non-volatile** storage in the ESP32-C3. This means it retains data even after the power is removed. The flash memory is where the firmware (user compiled code) is stored along with any configuration data.
- **Flashing** refers to the process of **writing compiled firmware** into the microcontroller's flash memory. This is typically done over a USB or UART interface using software tools like : esptool.py, Arduino IDE and ESP-IDF.
- The interface used for this is called the **flashing interface**. The designed board provides this through :
  1. The CH340C USB-to-UART chip
  2. Control lines (EN and BOOT) to trigger flash mode

3.3.2 Types of Boot Modes

- When a microcontroller powers up or is reset, it goes through a **boot process**. "To boot" means to start-up.
- The ESP32-C3 can boot in two modes:
  1. **Boot from flash** : When the ESP32-C3 boots normally, it reads user firmware (the program written by the user) from flash memory, loads it into RAM, and starts executing it.
  2. **Boot into download mode (flash mode)** : In this case, the ESP32-C3 does not run user program. Instead, it runs a small built-in program in ROM (read-only memory), which listens on the USB/UART for a new firmware image. This is used only during development, firmware update, or initial programming.

3.3.3 Manual and Automatic Flashing

- The user can manually enter each boot mode :
  1. To enter **normal boot**, press EN (reset) button alone. Ensure that the GPIO9 (BOOT) is pulled LOW.
  2. To enter **download mode**, pull GPIO9 (BOOT) LOW. Then, press and release the EN button. Now, release BOOT. The chip will wait for new firmware via UART. This method is fine for prototyping but tedious.



(a) Circuit for EN signal (b) Circuit for GPIO9 signal

Figure 4: Manual Reset Circuit

- To **automate** the method of entering the download mode, the development board uses a **CH340C chip**. Not only does it transmits serial data to the ESP32-C3 but also has two control lines wired to :
  - **DTR**, which is further connected to **GPIO9 (BOOT)**
  - **RTS**, which is further connected to **EN (reset)**
- **Flashing tools** like esptools.py send signals on DTR and RTS in a specific pattern making the flashing automatic i.e., no need to press any buttons.

3.3.4 Implementation of auto reset in circuit

- **What is DTR and RTS?** : **DTR (data terminal ready)** is used to indicate that a terminal (eg. PC) is ready to communicate. **RTS (request to send)** is used to tell the device (eg. microcontroller) that the computer wants to send data. These are **hardware-level control lines**, meaning they're separate from the actual serial TX/RX lines used to send data. Instead, they toggle HIGH or LOW to signal events or control devices.
- To flash new firmware, the flashing tool toggles RTS and DTR like this:

Time	RTS (EN)	DTR (GPIO9)	What Happens
1	LOW	LOW	Reset held low, BOOT held low
2	HIGH	LOW	Reset released, BOOT still low → **Download mode**
3	HIGH	HIGH	Normal operation resumes (after flashing)

Table 4: ESP32-C3 Boot Mode Control Using RTS and DTR

- However, the DTR and RTS signals cannot be directly connected to the ESP32-C3's BOOT and EN pins because these are digital signals **not designed to drive other IC pins** directly. Therefore, **NPN transistors** are used as **switches**.

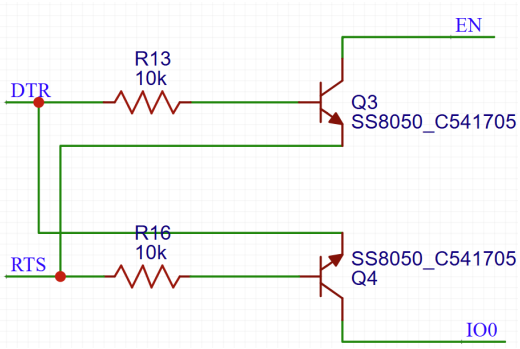


Figure 5: Auto Reset Circuit

- When RTS is LOW, the transistor turns ON. Consequently, EN is pulled low and ESP32 resets. Similarly, when DTR is LOW, the transistor turns ON and BOOT is pulled LOW.

4 PCB Design

4.1 Power Regulation

4.1.1 Power Input

The board is powered via **USB** using the **CH340C USB-to-UART converter**. USB supplies a nominal **5V** through the **VBUS line**.

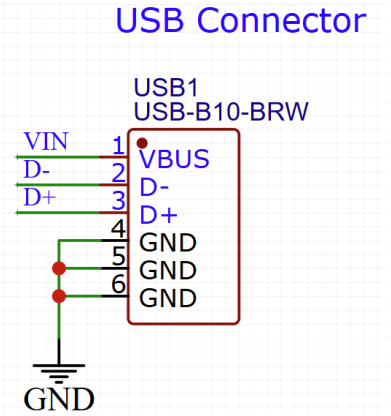


Figure 6: Circuit of USB Connector

4.1.2 Onboard Voltage Regulation

Since the ESP32-C3 operates at **3.3 V** and not 5 V, a **Low Dropout Regulator (LDO)** is used to step down 5V to 3.3V. Instead of the recommended **AMS1117-3.3**, the PCB uses a **APK2112K-3.3**

voltage regulator primarily for the reason that the former requires a voltage greater than 5.5 V. On a USB supply of 5 V, the dropout will be too high. On the other hand, the APK2112K-3.3 is specifically used for USB powered boards.

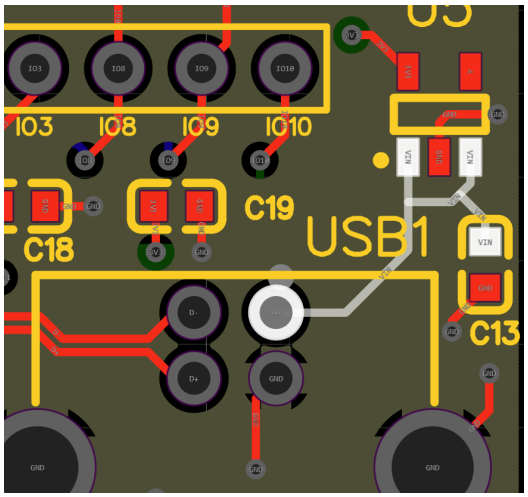


Figure 7: VIN connection highlighted on PCB

4.2 Trace Width Calculation

- In this project, all the connected components - including the ESP32-C3 Mini, CH340C USB-to-Serial converter, and the SN65HVD230 CAN transceiver - are low-power digital ICs with current consumption in microamps or milliamps.
- According to the standard **IPC-2221 guidelines** for external layers with **1 oz/ft²** copper

  > "A 10 mil wide trace can safely carry approximately 1 A of current with a temperature rise of 10°C."

  Since none of the traces in this design are expected to carry more than **500 mA**, the default trace width offers ample margin.

4.3 Component Value Decisions

4.3.1 ESP32-C3 Mini

The datasheet mentions the typical application circuit of the module connected with peripheral components.

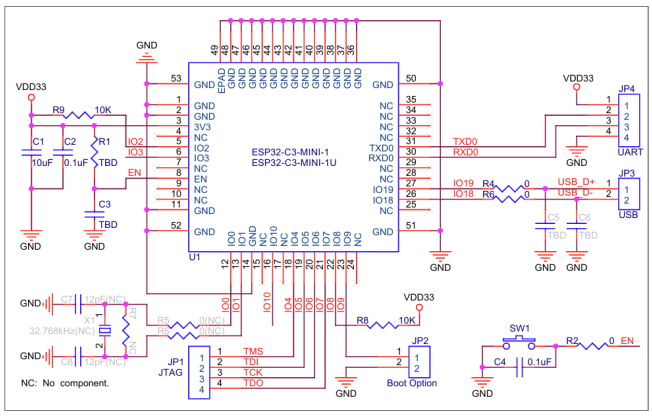


Figure 8: Peripheral Schematic of ESP32-C3 Mini

4.3.2 APK1112K-3.3

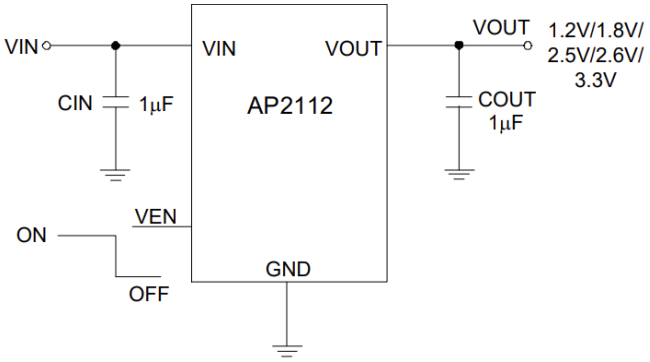


Figure 9: Application Circuit for APK1112K-3.3

4.3.3 CH340C

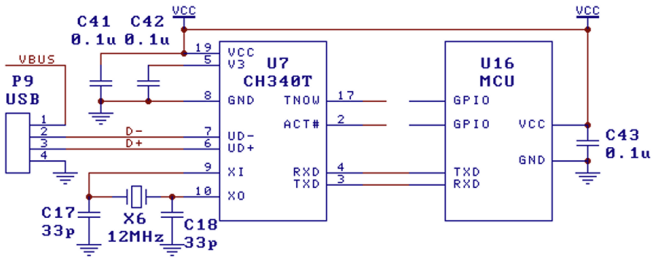


Figure 10: Recommended Circuit for CH340C Chip

Note : Since the **CH340C** has an internal crystal oscillator, there is no need for external crystal oscillator **X6** and capacitors **C17, C18**.

4.3.4 SN65HVD230D

The datasheet provides a board layout, and mentions the values of the components elsewhere in the datasheet.

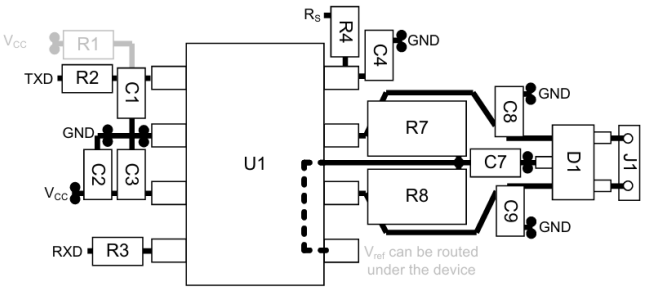


Figure 11: SN65HVD230D Board Layout

5 Conclusion

This project not only reinforces foundational principles in embedded hardware design but also provides a **robust and flexible platform** for further development and experimentation with the **ESP32-C3 SoC**.