

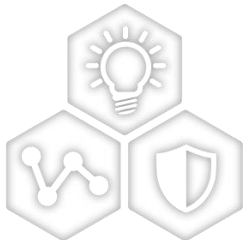
SAM9X60 ARM926EJ-S

Microprocessor & Embedded Linux

LNx1001-Application v1.10



A Leading Provider of Smart, Connected and Secure Embedded Control Solutions



SMART | CONNECTED | SECURE

Kevin Lu
CAE Taiwan
Microchip Technology Inc.
September 10, 2025

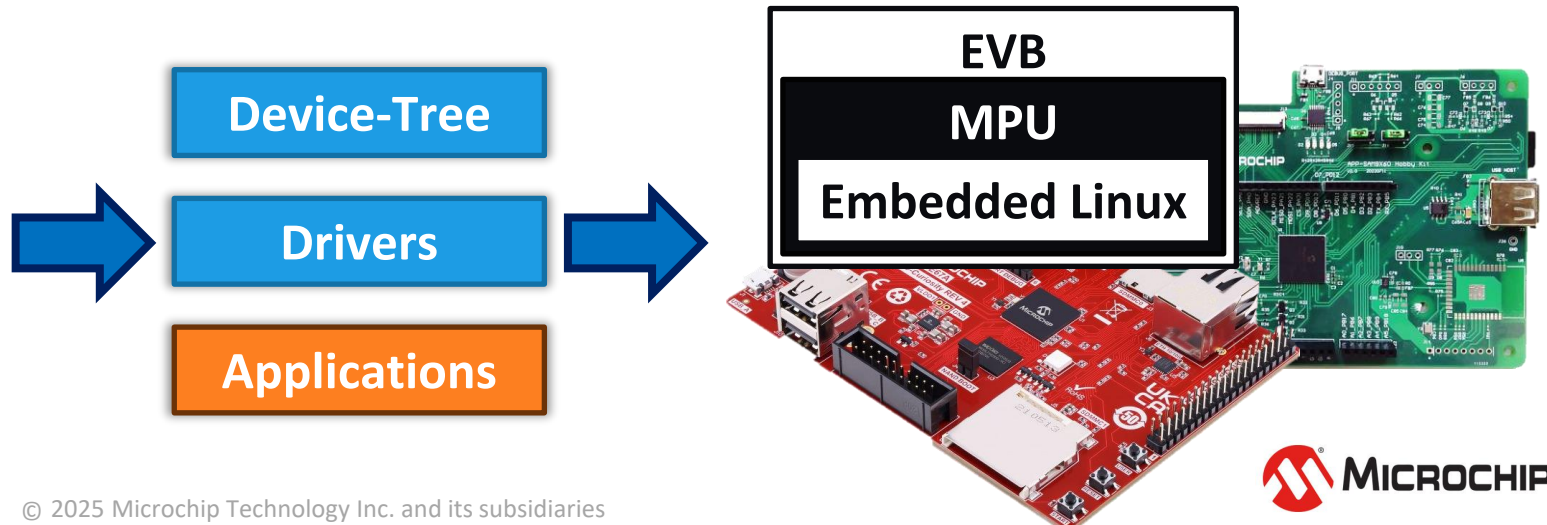
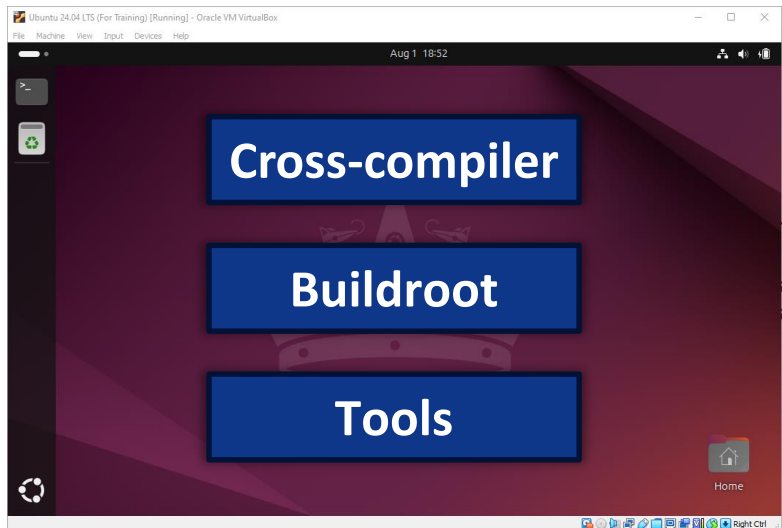
Course Outline

- **Develop Embedded Linux on MPU**
 - Embedded Linux System Components
 - Develop Application for Embedded Linux System
- **Labs - Bash Script**
 - Lab1 – Prepare Development Tools
 - Lab2 – Use Script to Control LEDs
 - Device-Tree and Device-Driver for Peripherals
 - Lab3 – Use Script to Read ADC Value and Control PWM
- **Labs - Use Tools and C Language to Control Peripherals**
 - Lab4 – Use Tools and Application to Detect Button Events
 - Lab5 – Use Tools and C Language to Control UART
 - Lab6 – Use Tools and C Language to Control I2C
 - Lab7 – Use Tools and C Language to Control SPI
- **Appendix - Device Tree and Device Driver**

Develop Embedded Linux on MPU

Develop Embedded Linux on MPU

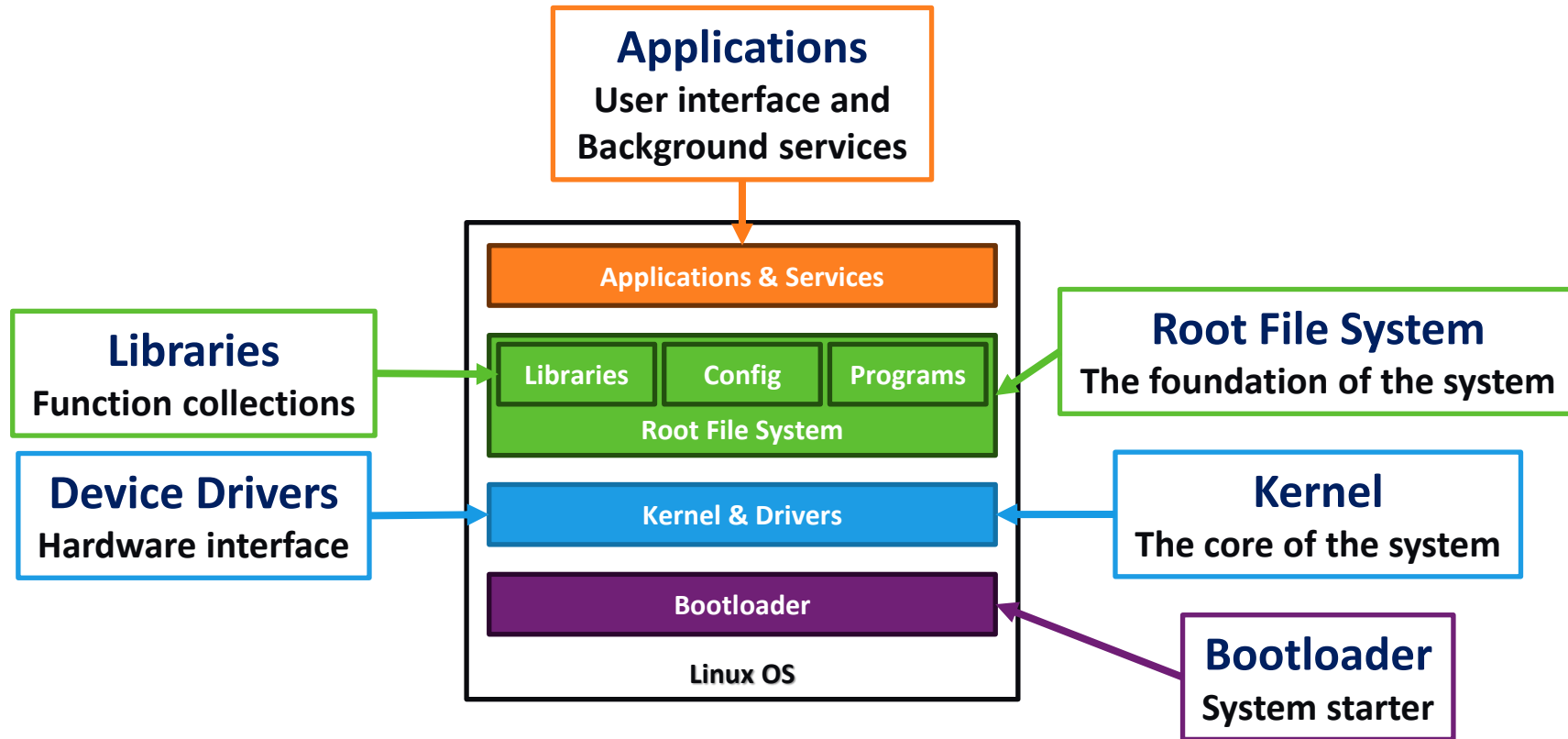
- Before developing the Embedded Linux, we need to understand the composition and architecture of the system and prepare some tools.
- For developing on Embedded Linux OS, we need to **activate the required peripherals and matching drivers**, we will achieve this by **modify the configurations of Build System and Device-Tree**, and finally access and control these peripherals in the User-Space.
- We will execute the Embedded Linux OS on Microchip's MPU, and design applications to do something in the system. Before that, we need to prepare the required **PC base Linux development platform**, including the specified **cross-compiler** and other tools.



Embedded Linux System Components

Embedded Linux System Components

- An Embedded Linux system components are more **streamlined** and **customized** compared to traditional desktop Linux, the composition is as follows :



Kernel

- The Linux Kernel is **the heart of OS**, responsible for allocating resources and coordinating the operation of various software and hardware.
- **Functions of Embedded Linux Kernels:**
 - **Processor Management**
Allocates processor time, handles interrupts, and performs context switching.
 - **Memory Management**
Allocates and reclaims memory, manages virtual memory.
 - **Device Drivers**
Provides interfaces for interacting with various hardware devices.
 - **File Systems**
Manages file systems and provides file access functionality.
 - **Network Stack**
Enables network communication.
 - **Process Management**
Creates, terminates, and manages communication between processes.

Root File System

- **Root File System (rootfs)** is the first file system mounted when a Linux system boots up. It serves as the foundation for the entire operating system, and contains the basic files required for system operation.
- **Components of Root File System :**
 - **/bin** : user-level commands. (e.g., ls, cp, cat, etc.)
 - **/sbin**: system-level commands. (e.g., fdisk, mount, etc.)
 - **/lib** : libraries and dynamically loadable modules. (like device drivers)
 - **/etc** : system configuration files.
 - **/dev** : device special files, which represent hardware devices.
 - **/var** : store variable data (e.g., log files, temporary files, etc.)
 - **/proc**: virtual file system.
 - **/sys** : another virtual file system.
 - **/mnt**: mount point directory.
 - **Others**

File System

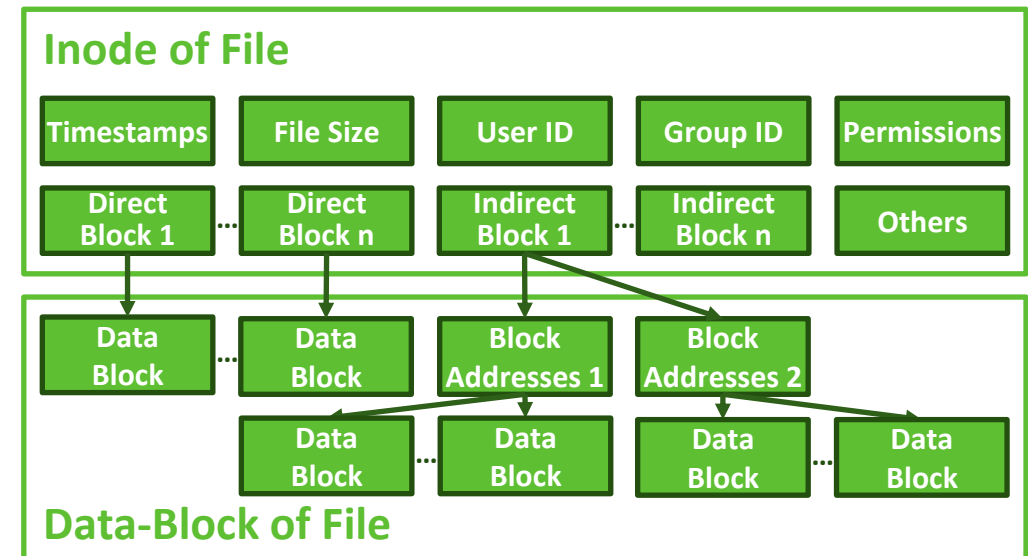
- In the Linux OS, **everything is a file**, includes regular files, processes, directories, links, hardware devices, and so on.
- The File System is **an organized way of storing and retrieving data on a storage device**, it as a library system for digital files, where each file is assigned a specific location and name for easy access, and allows users to access, modify, and delete these files.
- The file system is usually divided into **Data-Block** and **Inode** (index node).

- **Data-Block**

is a contiguous region on a physically stored, is the **fundamental unit** of storage in file system.

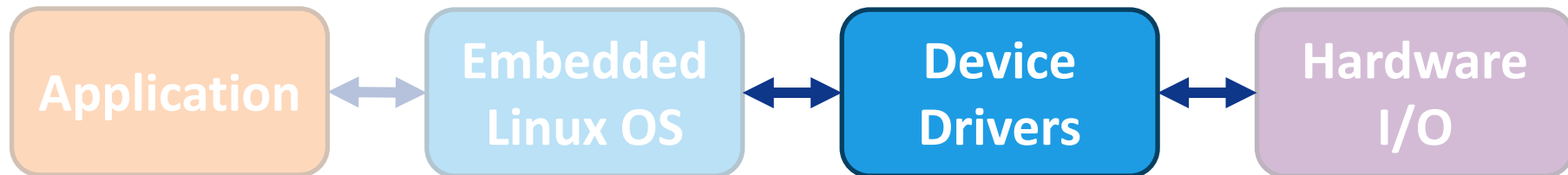
- **Inode**

is a file data structure that stores **information** about any file except its name and data.



Device Drivers

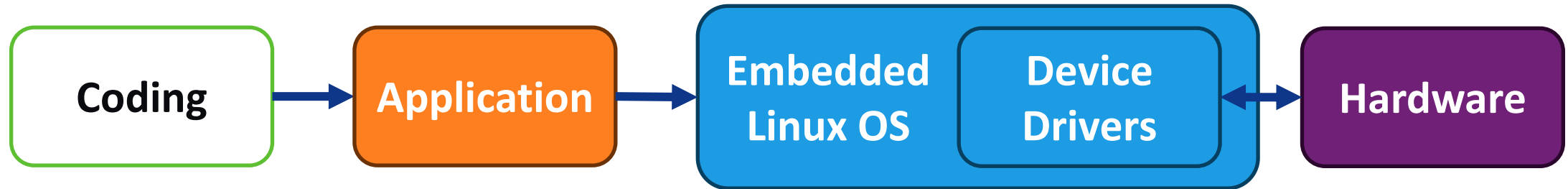
- The Device Driver is the **bridge between hardware and the OS**. It translates low-level hardware operations into high-level commands that the OS can understand. This allows applications to interact with hardware devices using standard **System-Calls**.
- Common types of Device Driver :
 - Character Device Driver
 - Block Device Driver
 - Network Device Driver
 - Others



Develop Application for Embedded Linux System

Develop Application for Embedded Linux System

- Developing applications for Embedded Linux system has greater hardware interactivity and resource constraints than PC.
- This means that developers need to understanding of the Linux system, device drivers, and Hardware architecture.



Kernel-Space and User-Space of Embedded Linux

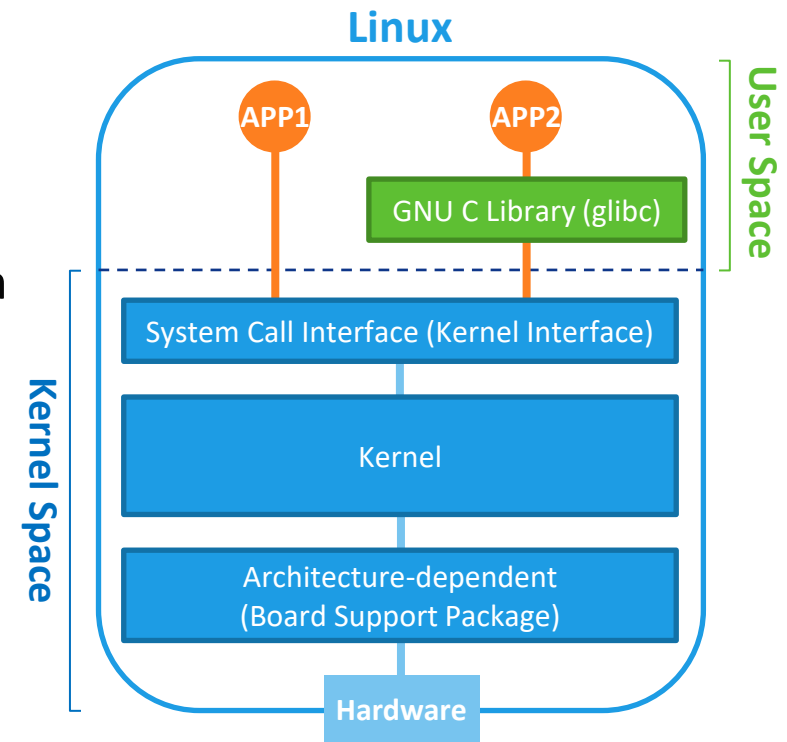
- Linux OS uses virtual memory to provide separate address spaces, divided into two main parts – Kernel-Space and User-Space. This mechanism provides memory protection and hardware protection from malicious or errant software behavior.

- Kernel-Space**

is the kernel privileged area and a critical part of the OS. Responsible for directly interacting with the hardware, controlling memory and managing system services. It requires a secure and isolated space.

- User-Space**

is a non-privileged area where applications operate. Each application has its own private memory space, ensuring that it cannot interfere with others or the kernel. If the application needs to access the hardware, it needs to be controlled through System-Calls (resemble library APIs), to request services from the Kernel that require higher privileges to run.



Control Hardware in User-Space

- In Embedded Linux system, user-space programs cannot directly access hardware peripherals to protect system stability and security. Typically, user-space programs request the kernel to perform hardware operations through system calls.
- Primary ways to control hardware in User-Space :
 - **System Calls**

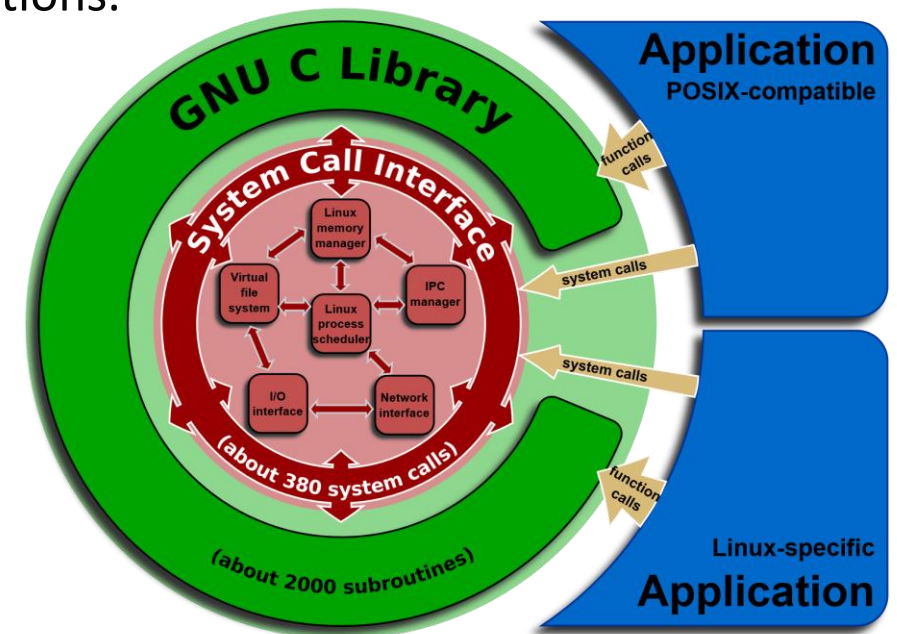
User-space programs make specific system calls to pass hardware operation requests to the kernel. The kernel verifies the legitimacy of the request and calls the corresponding driver to complete the hardware operation.
 - **Device Files**

Each hardware device corresponds to a device file in the Linux system, typically located in the `/dev` directory. User-space programs can read and write to device files as if they were ordinary files.
 - **Library Functions**

Some common hardware operations, such as network operations and file system operations, have been encapsulated into library functions. User-space programs can directly call these library functions to complete the corresponding operations.

System Call

- In an Embedded Linux system, system-call is a special function that serves as a bridge between user-space programs and the kernel.
- User-space programs cannot directly access system hardware resources, they rely on system-calls to request the kernel to perform specific operations.
- **Functions of System Calls**
 - Provides an interface between user space and the kernel.
 - Memory management, Process control and File system operations.
 - Operating hardware devices.
- **Examples of System Calls**
 - **open()** : Open a file.
 - **read()** : Read data from a file.
 - **write()** : Write data to a file.
 - **close()** : Close a file.
 - **fork()** : Create a child process.
 - **exit()** : Terminate a process.
 - **execve()** : Load and execute a program.



Devices

- In Embedded Linux system, a device refers to **any hardware component connected to the system that performs a specific function.**
- These devices can be broadly categorized based on their function and interaction with the system :
 - **Character Devices** (e.g., Keyboard)
 - Data is transferred in bytes.
 - Unbuffered, data transfer is usually real-time.
 - No fixed block size.
 - **Block Devices** (e.g., SD card)
 - Data is transferred in fixed-size blocks.
 - Buffered for improved I/O efficiency.
 - Supports random access.
 - **Network Devices** (e.g., Network interface card)
 - Responsible for transmitting and receiving network data.
 - **Other Devices**
 - Sensors, Touchscreen, Audio devices, Video devices, etc.

Libraries

- The Libraries in Linux are pre-built blocks of code that provide specific functionalities, they allow developers to **quickly assemble complex applications**.
- **Types and Functions of Libraries :**
 - Core System Libraries
 - Network Communication Libraries
 - Graphical User Interface Libraries
 - Multimedia Libraries
 - Database Libraries
 - Other Libraries

Labs - Use Bash Script to Control Peripherals

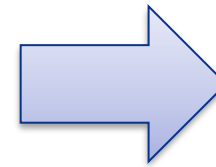
Use Bash Script to Control Peripherals

- The course designed some Labs to show how to use **Bash Script** to control MPU's peripherals. We will try to access and configure the LED, ADC and PWM corresponding device drivers through the files listed in **sysfs**.

```
#!/bin/bash
```

```
...
```

bash_script.sh



Show
Hello World !

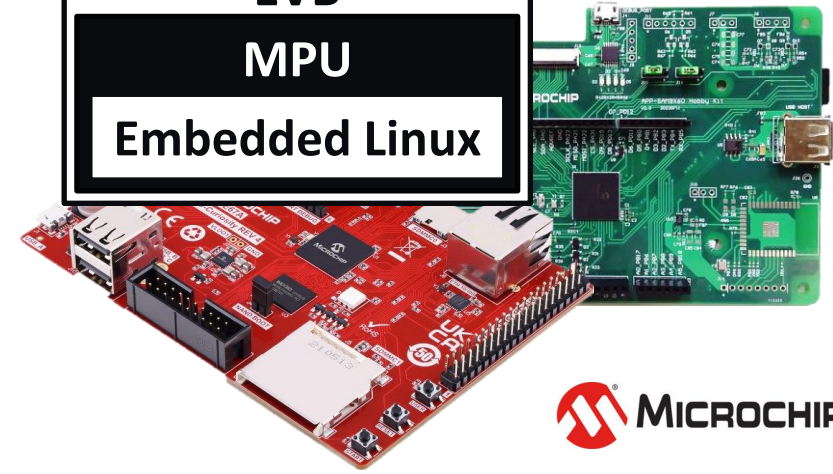
LED Blink

Control
ADC & PWM

EVB

MPU

Embedded Linux



Bash Script

- Bash script is a script written in the Bash shell language that can **automate the execution of a series of commands**. In embedded Linux systems, we can use Bash scripts to control various hardware components such as GPIO, I2C, SPI, and more.
- Bash is the **standard shell** in Linux systems, it can be used on different embedded platforms, and relatively simple and easy to learn. You can combine various system commands and script logic.

```
# ./bash_script.sh  
Hello World !
```

```
#!/bin/bash
```

```
echo "Hello World !"
```

```
bash_script.sh
```

Sysfs

- Some hardware devices have corresponding device drivers, it will be responsible for managing the hardware and mapping hardware registers, interrupts, etc. to the kernel space, and creates files or directories in sysfs to represent the device's attributes.
- **Sysfs is a pseudo(virtual) file system**, that **provides a way to access and configure device drivers** in the Linux kernel, user-space programs can control the hardware by accessing these files in **/sys** directory.
- Files and directories within sysfs represent devices, buses, and drivers in the system. By reading and writing to these files, user-space programs can get or set device attributes.

Lab1 – Prepare Development Tools

Linux common commands

Hints

- **ls** : List all files in the current directory.
- **cd** : Enter directory.
- **pwd** : Show current path.
- **mkdir** : Create folder.
- **rm** : Delete Files, add -r to delete the folder.
- **cp** : Copy files.
- **mv** : Move files.
- **sudo** : Temporarily grant users privileged access to system.

- **nano** : A small editor for on the terminal.
- **microcom** : A text-based serial port communications program.

Lab1 – Prepare Development Tools

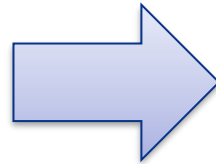
- We will prepare the Terminal on the computer to control and develop the EVB, and then run a script in the Embedded Linux OS.
- In addition, we will check the WSL on PC in this Lab to ensure it is available when needed for later courses.
- Follow the steps below to complete the above requirements :
 1. Prepare a WSL2 with a Linux OS installed on the computer and Launch it.
 2. Prepare a SD Card with Embedded Linux OS and mount it to EVB.
 3. Prepare a Terminal on the computer and Launch it.
 4. Connect the Debug port of EVB to the VM and login the Embedded Linux OS.
 5. Create a script in the Embedded Linux OS and execute it.
 6. Show "**Hello World !**" in the Embedded Linux OS.

Let's go!

Lab1 – Prepare Development Tools

Step 1 (WSL will be used later)

- Launch Ubuntu on the WSL.
([Refer the document to install Linux on Windows with WSL](#))

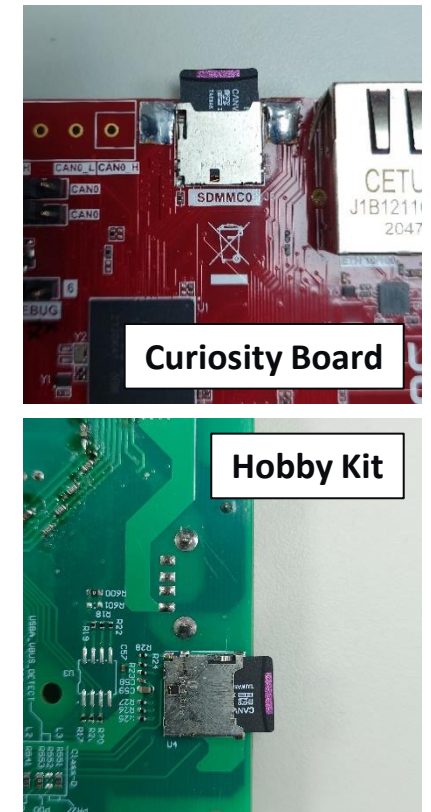
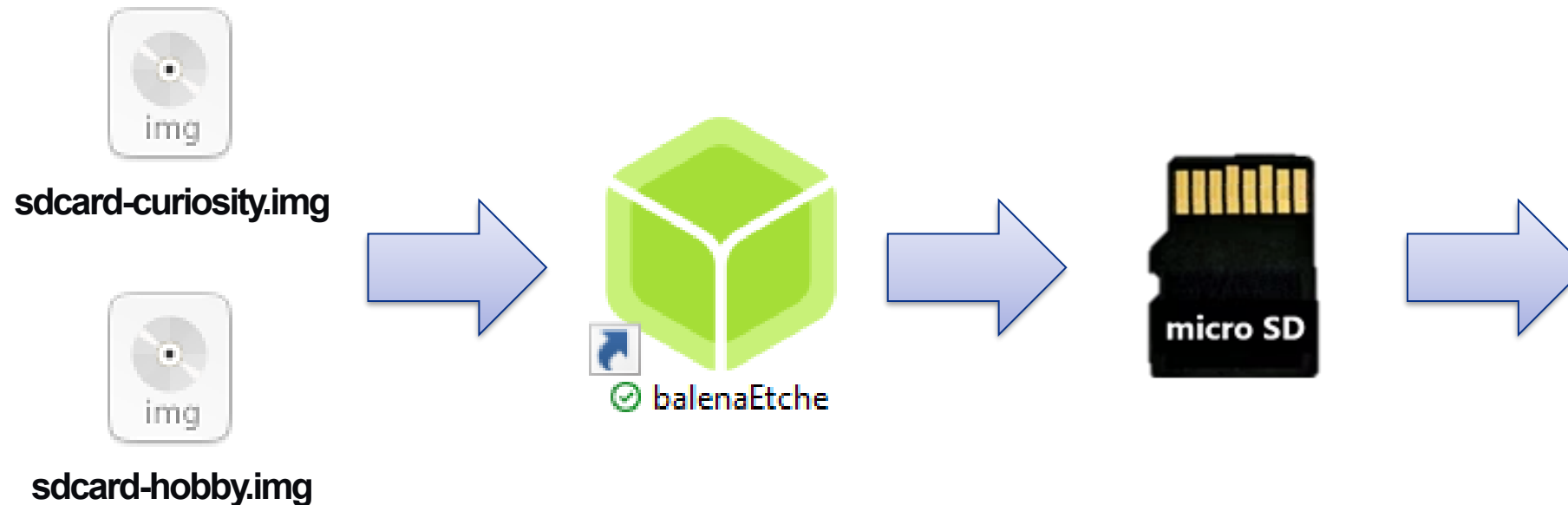


```
user@LNX1001: ~  
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)  
  
* Documentation:  https://help.ubuntu.com  
* Management:    https://landscape.canonical.com  
* Support:        https://ubuntu.com/pro  
  
System information as of Wed Aug  6 14:37:06 CST 2025  
  
System load:  0.0      Processes:            79  
Usage of /:   3.2% of 1006.85GB  Users logged in:     0  
Memory usage: 2%  
Swap usage:   0%  
  
* Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s  
  just raised the bar for easy, resilient and secure K8s cluster deployment.  
  
https://ubuntu.com/engage/secure-kubernetes-at-the-edge  
  
This message is shown once a day. To disable it please create the  
/home/user/.hushlogin file.  
user@LNX1001:~$
```

Lab1 – Prepare Development Tools

Step 2

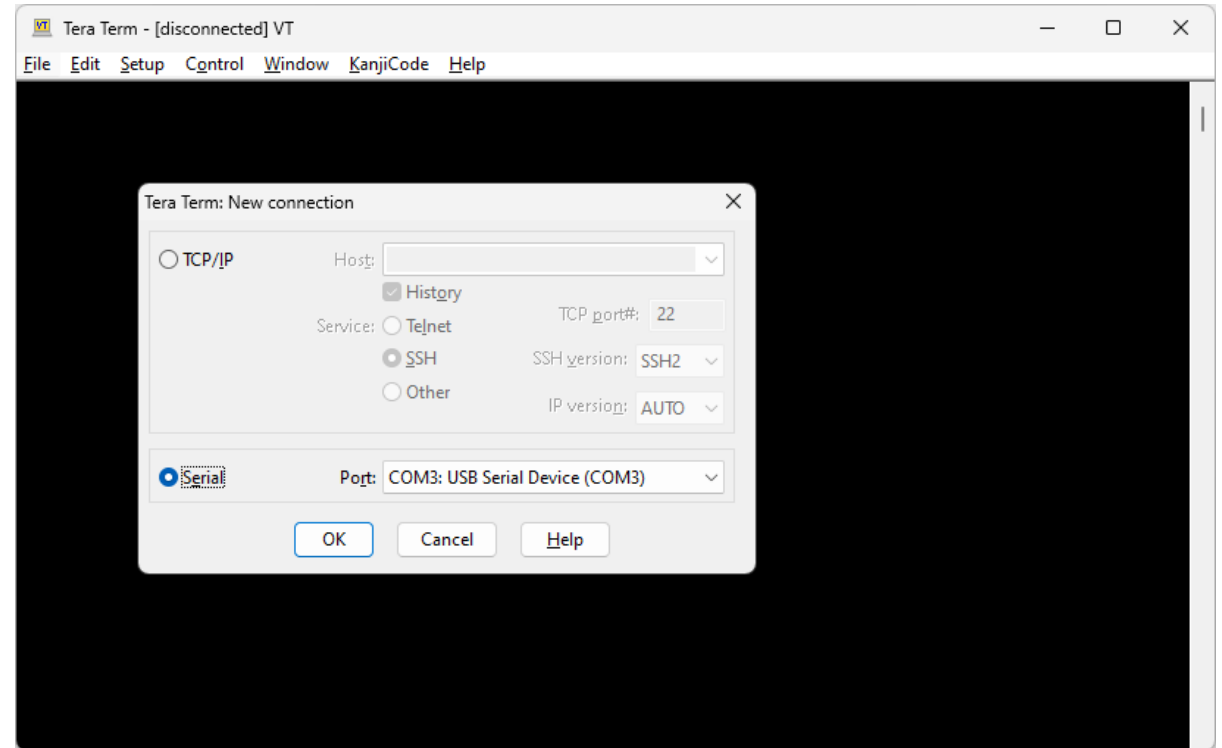
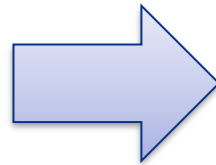
- Prepare a SD Card with Embedded Linux OS and mount it to EVB.
(Refer to the appendix to complete the Flash SD Card)



Lab1 – Prepare Development Tools

Step 3

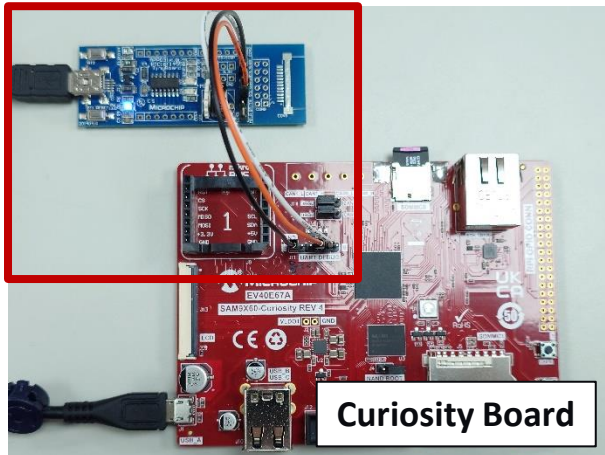
- Launch Tera Term on the PC.
([Refer the document to install Tera Term on Windows](#))



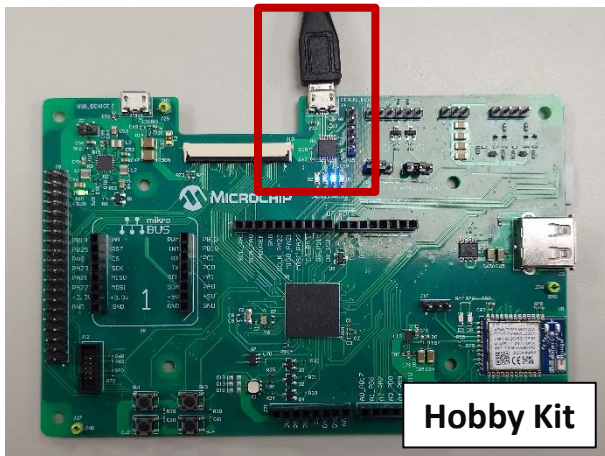
Lab1 – Prepare Development Tools

Step 4

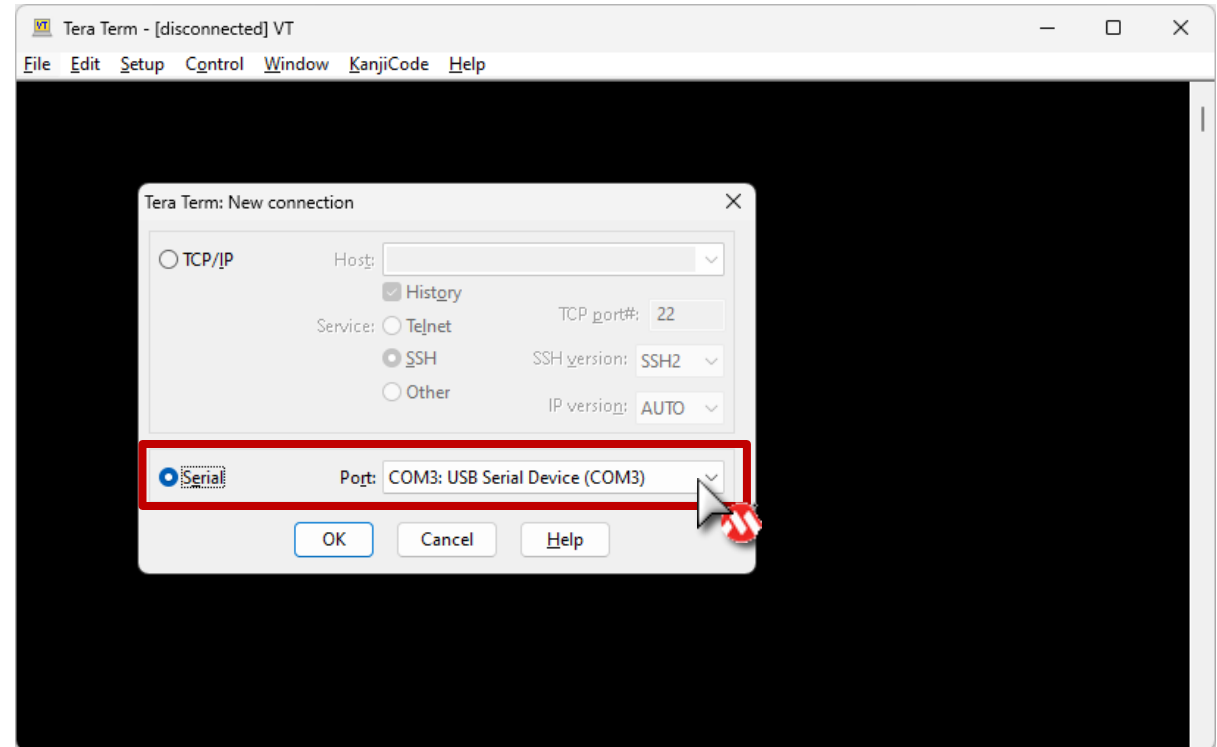
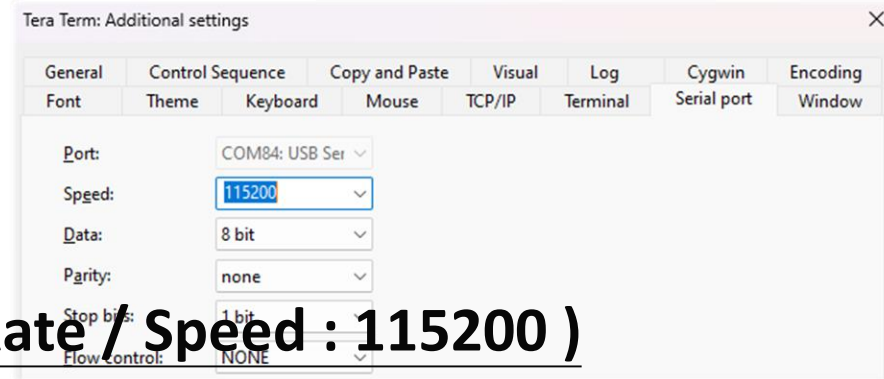
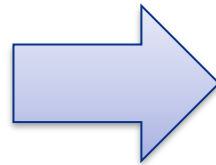
- **Connect the Debug port** of EVB and Tera Term. (**Baud Rate / Speed : 115200**)
(Select the **COM Port** that appears after the connection)



Curiosity Board



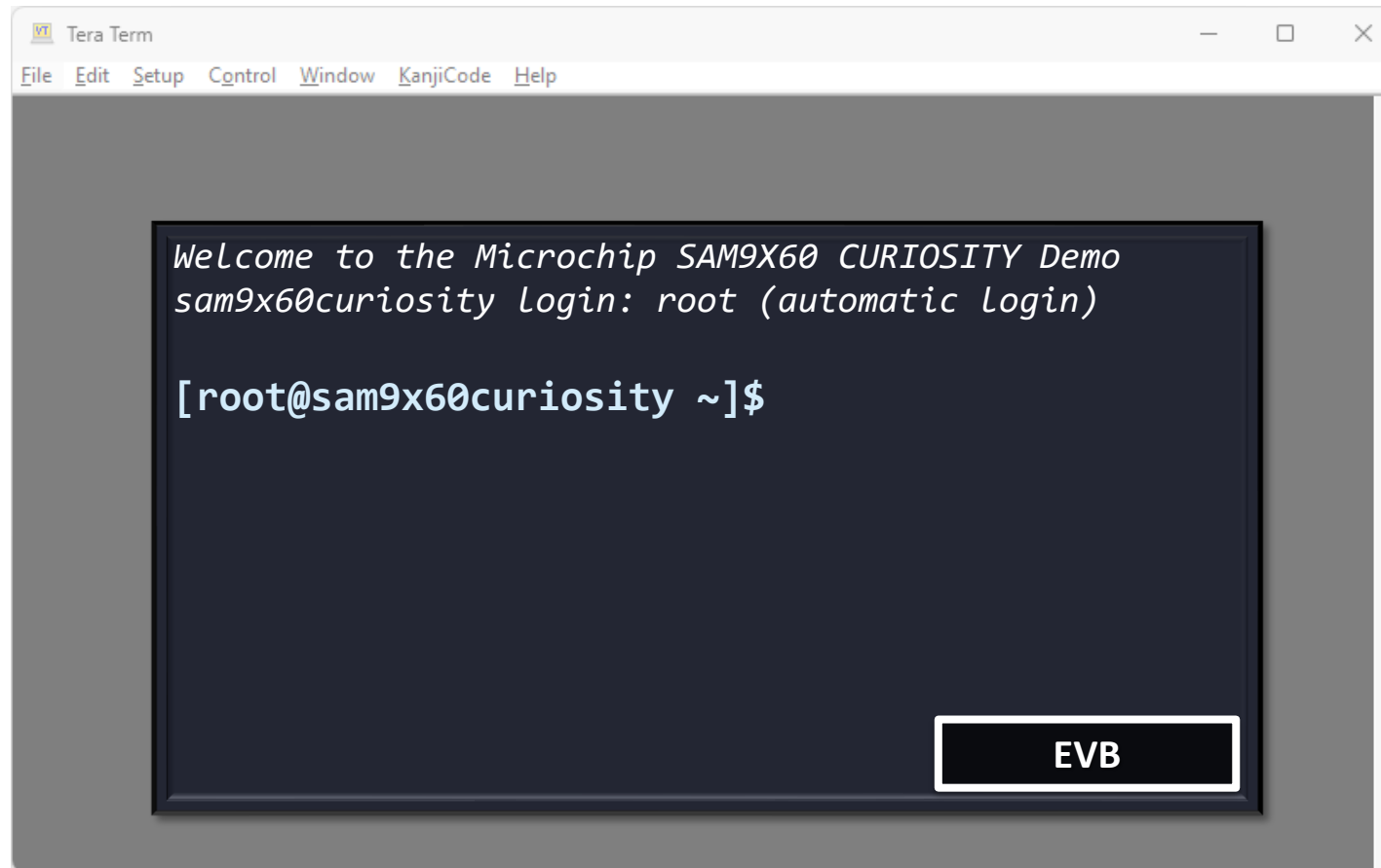
Hobby Kit



Lab1 – Prepare Development Tools

Step 5

- Use a Terminal (Tera Term) to **login** the Embedded Linux OS of EVB.

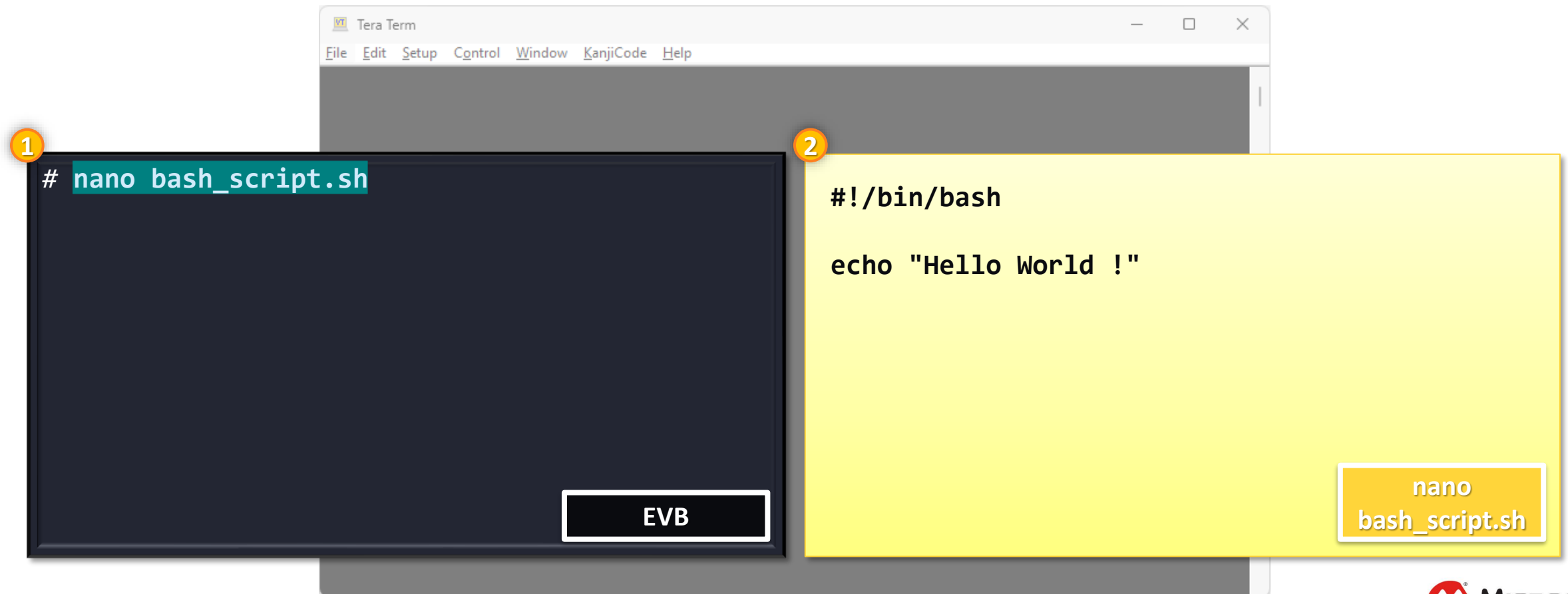


The image shows a screenshot of a Tera Term terminal window. The window has a title bar that says "Tera Term" and a menu bar with options: File, Edit, Setup, Control, Window, KanjiCode, and Help. The terminal output displays a welcome message: "Welcome to the Microchip SAM9X60 CURIOSITY Demo" followed by "sam9x60curiosity login: root (automatic login)". Below this, the prompt "[root@sam9x60curiosity ~]" is shown with a dollar sign "\$" at the end. In the bottom right corner of the terminal window, there is a button labeled "EVB".

Lab1 – Prepare Development Tools

Step 5

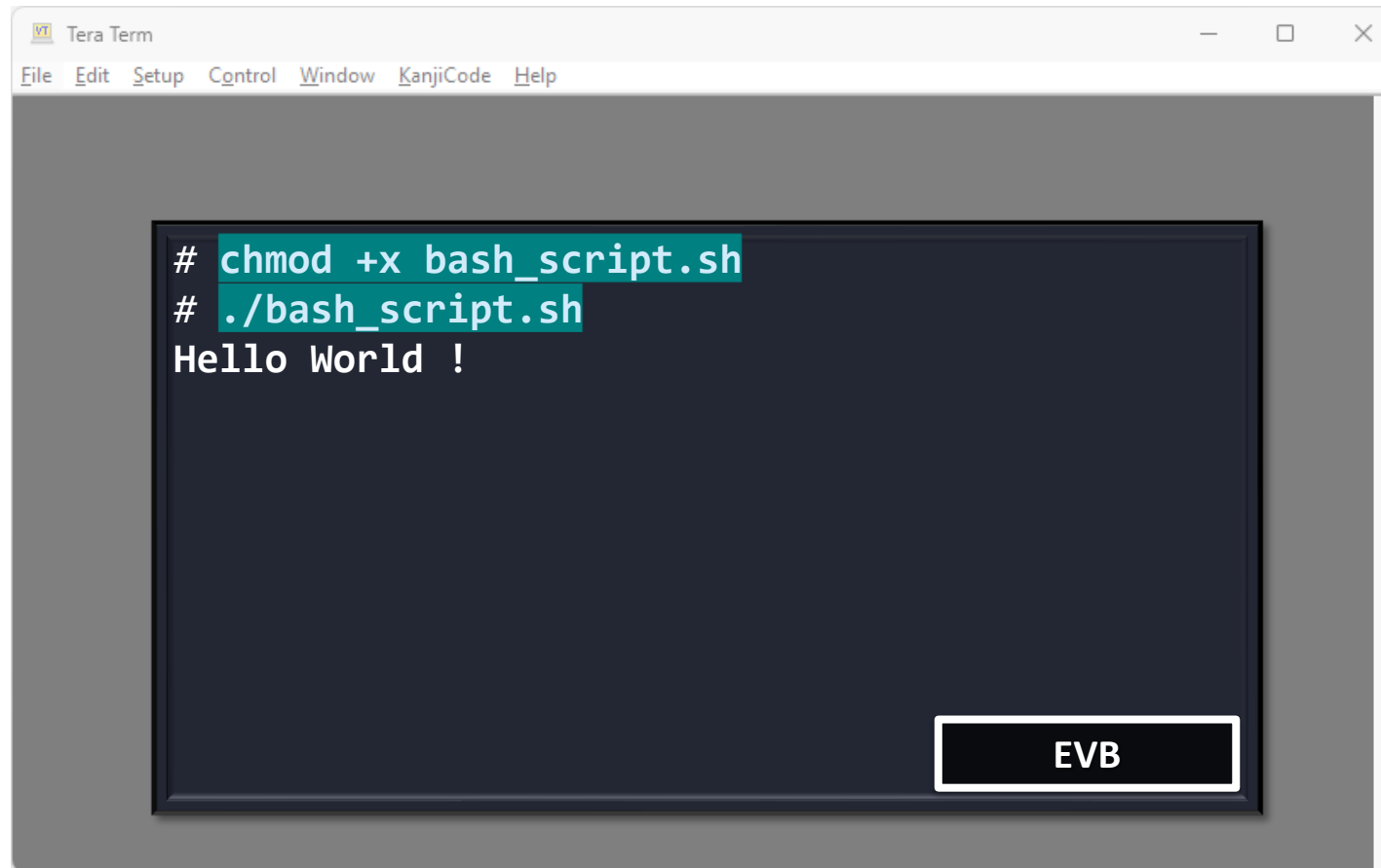
- **Try create a Bash script** in the Embedded Linux OS.



Lab1 – Prepare Development Tools

Step 6

- **Execute the Bash script** in the Embedded Linux OS.

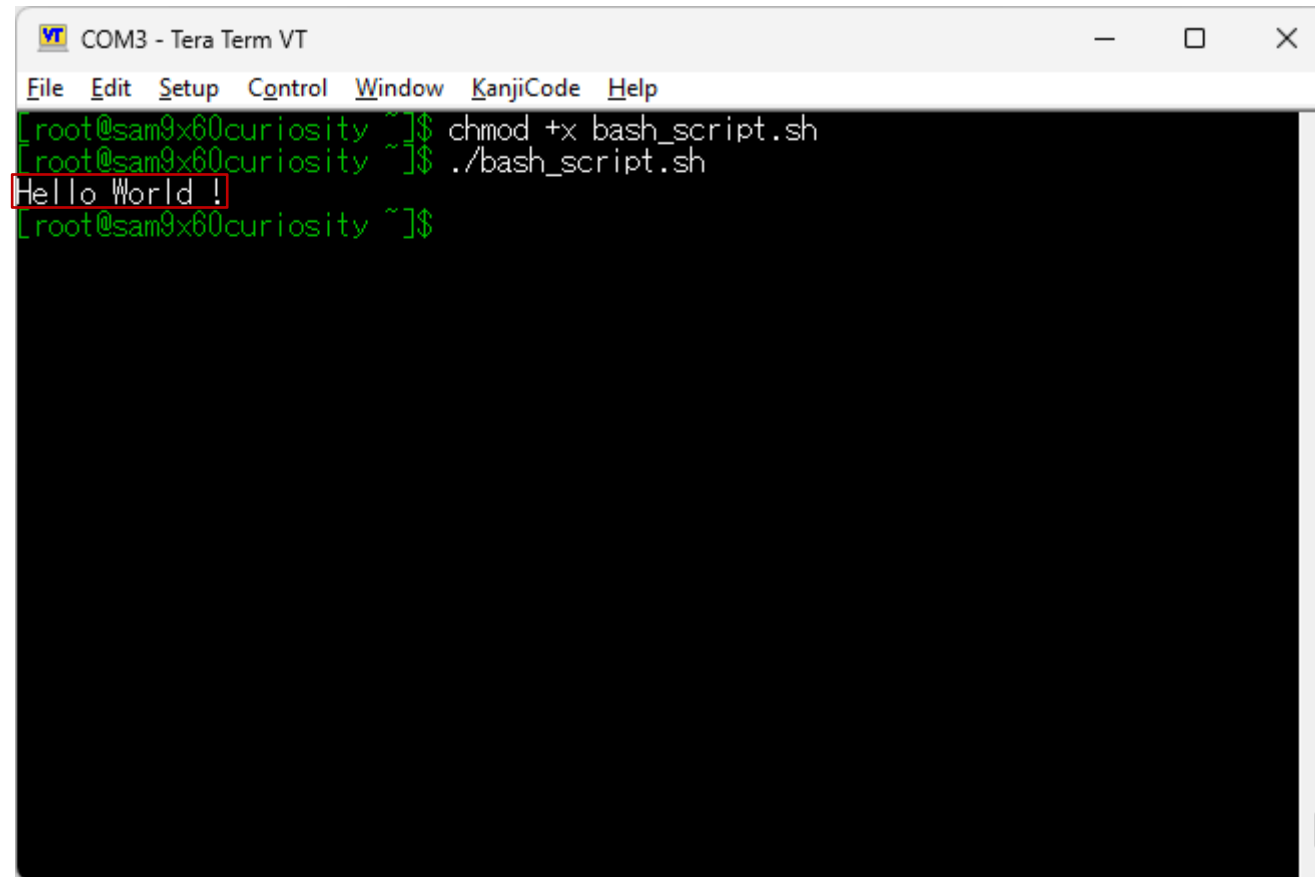
A screenshot of a Tera Term terminal window. The window has a title bar with 'Tera Term' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Setup', 'Control', 'Window', 'KanjiCode', and 'Help'. The main area is a dark terminal with white text. It shows two commands being entered: '# chmod +x bash_script.sh' and '# ./bash_script.sh'. The output of the second command is 'Hello World !'. In the bottom right corner of the terminal area, there is a white rectangular button with the text 'EVB' in black.

```
# chmod +x bash_script.sh
# ./bash_script.sh
Hello World !
```

EVB

Lab1 – Prepare Development Tools

Result



A screenshot of a Tera Term VT terminal window. The window title is "COM3 - Tera Term VT". The menu bar includes "File", "Edit", "Setup", "Control", "Window", "KanjiCode", and "Help". The terminal shows the following commands and output:

```
[root@sam9x60curiosity ~]$ chmod +x bash_script.sh
[root@sam9x60curiosity ~]$ ./bash_script.sh
Hello World !
[root@sam9x60curiosity ~]$
```

The output "Hello World !" is highlighted with a red rectangular box.

Lab2 – Use Script to Control LEDs

Lab2 – Use Script to Control LEDs

- In the Embedded Linux provided from the course, the default will be to set three colors of Red, Green and Blue LEDs in the Device-Tree, and enable the corresponding device driver. It will list the three colors of LEDs in **sysfs** respectively, we can control the LED by writing the file corresponding to the LEDs.
- Follow the steps below to complete the above requirements :
 1. Create a script in the Embedded Linux OS.
 2. Execute the script and alternate the three colors of the LED turn on and off.

Let's go!

Lab2 – Use Script to Control LEDs

Step 1

- **Create a Bash script** in the Embedded Linux OS.

1

```
# nano LEDs_blink.sh
```

2

```
#!/bin/bash
while true
do
    echo 1 > /sys/class/leds/red/brightness
    echo 0 > /sys/class/leds/green/brightness
    echo 0 > /sys/class/leds/blue/brightness
    sleep 1
    echo 0 > /sys/class/leds/red/brightness
    echo 1 > /sys/class/leds/green/brightness
    sleep 1
    echo 0 > /sys/class/leds/green/brightness
    echo 1 > /sys/class/leds/blue/brightness
    sleep 1
done
```

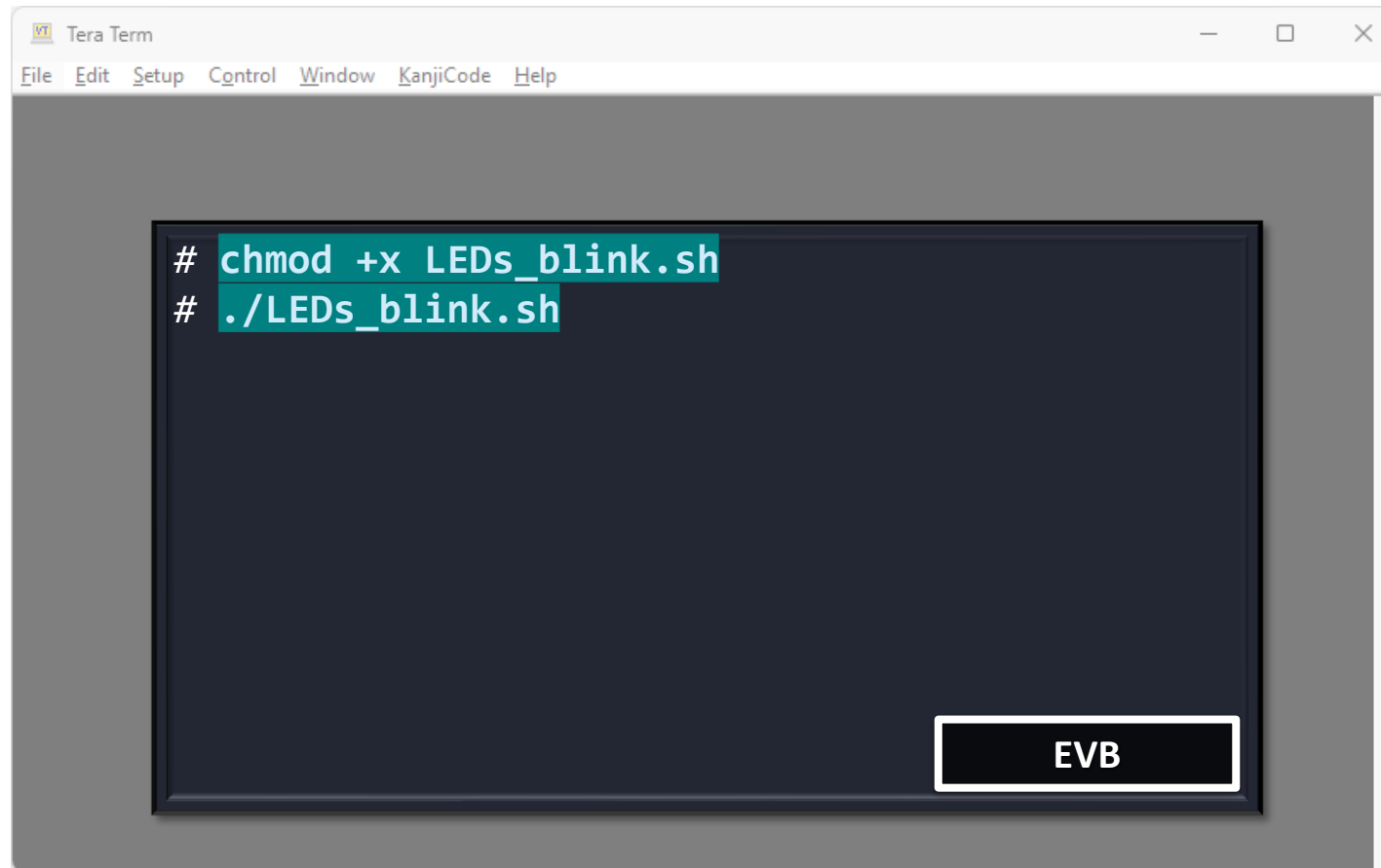
EVB

nano
LEDs_blink.sh

Lab2 – Use Script to Control LEDs

Step 2

- **Execute the Bash script** in the Embedded Linux OS.



The screenshot shows a Tera Term terminal window with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help) and a dark blue terminal area. Two lines of text are visible, both highlighted in teal: `# chmod +x LEDs_blink.sh` and `# ./LEDs_blink.sh`. A white rectangular box with the text "EVB" is positioned in the bottom right corner of the terminal area.

Lab2 – Use Script to Control LEDs

Result

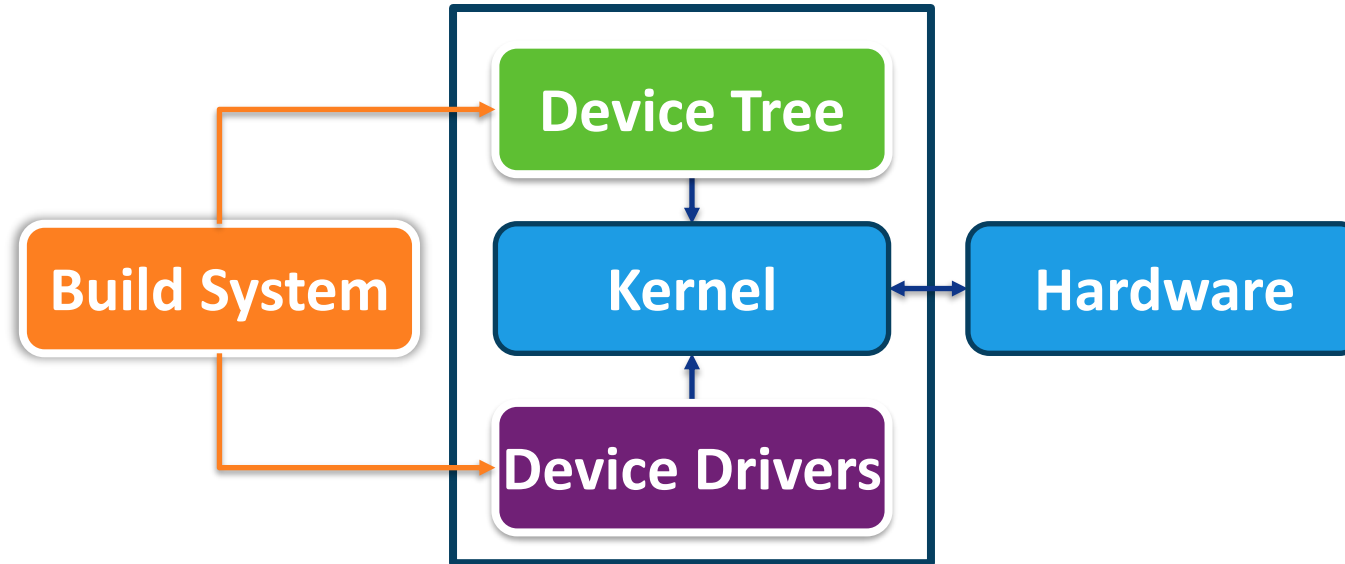


Device-Tree and Device-Driver for Peripherals

PIO, Pin Multiplexing, ADC, PWM and FLEXCOM

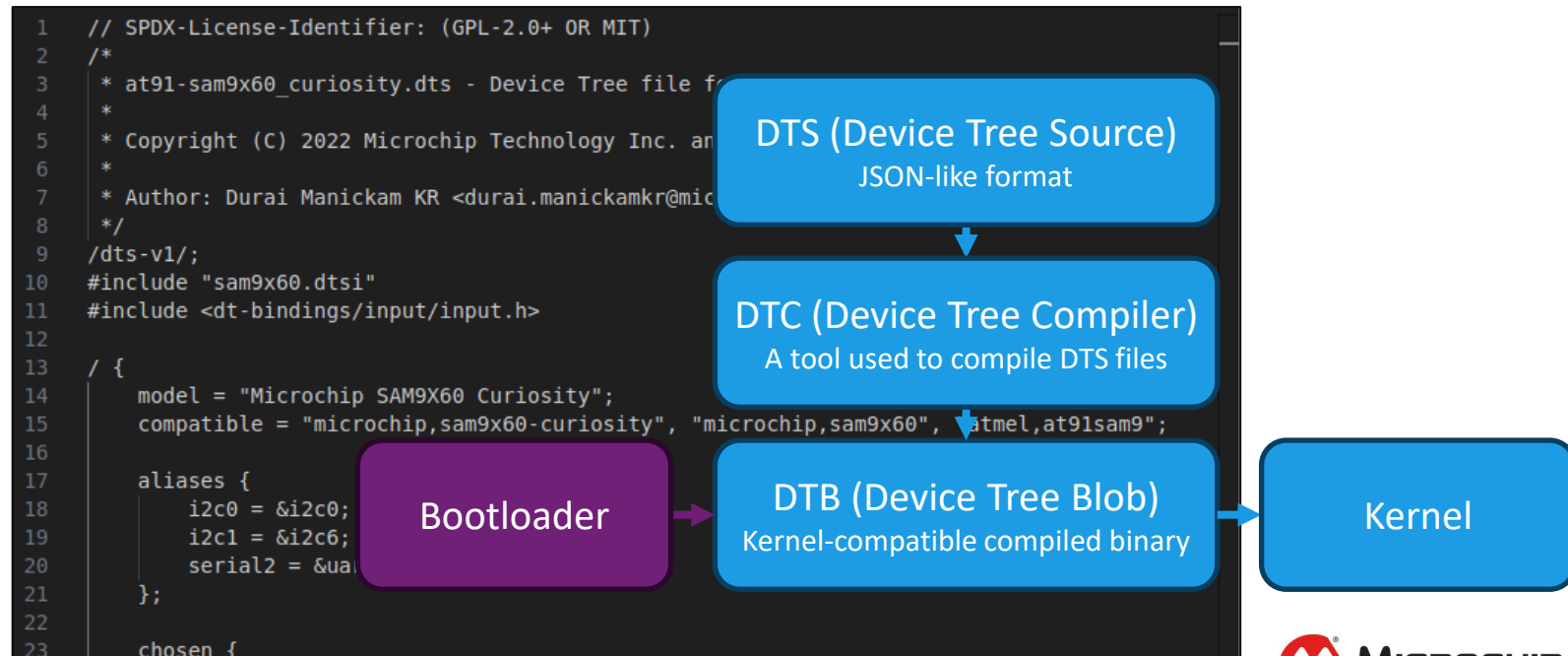
Device-Drivers Configuration

- Since the Embedded Linux OS run on various MPU and EVB, we need to customized the configuration, and select the corresponding device drivers for different hardware.
- In the modern Linux system, we can describe the required peripherals through the **Device-Tree**. In addition, we also need device drivers to control these hardware, we can use the **Build System** to manage the drivers and compile anything required resource.



Device-Tree

- Device-Tree is a data structure used to describe the hardware configuration of an Embedded Linux system. It represents the various hardware devices in the system, along with their connections, in a tree-like structure.
- Device-Tree separates hardware configuration information from the kernel and stores it in an independent file, it allows the Linux kernel to no longer include a large amount of platform-specific code, simplifying the kernel's structure.
- Functions of Device Tree :
 - Describes hardware
 - Creates device nodes
 - Configures drivers
- Advantages of Device Tree :
 - High flexibility
 - Creates device nodes
 - Configures drivers



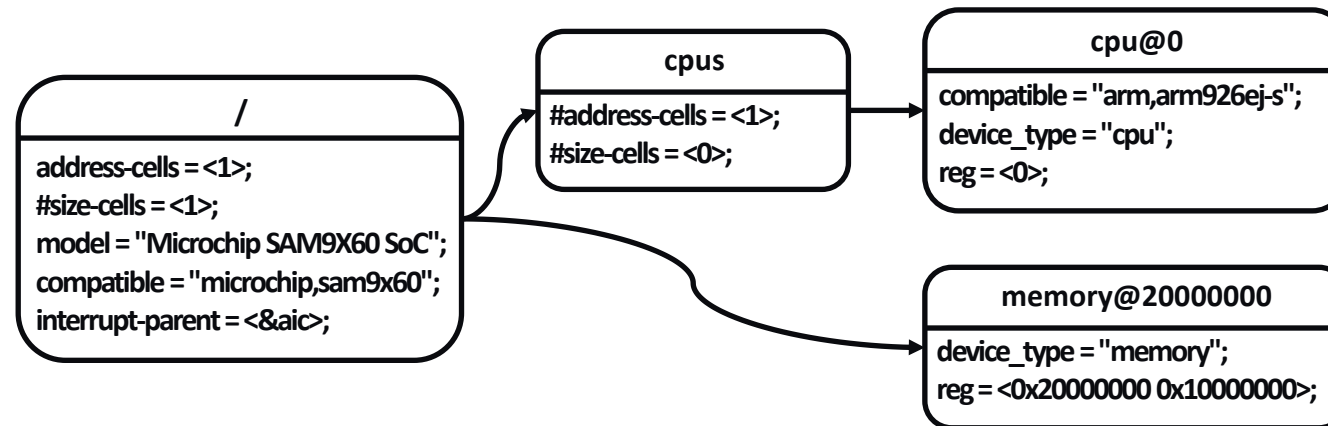
Device-Tree-Source Files

- In Embedded Linux System, Device-Tree files are mainly divided into two types: **Device-Tree-Source-Include (.dtsi)** and **Device-Tree-Source (.dts)** files. These two files work together to describe the hardware configuration of the system.
- Usually, the manufacturer provides a **Device-Tree-Source-Include (.dtsi)** file containing the information of the MPU, and based on it to **add or modifies the difference part** in the **Device-Tree-Source (.dts)** for the design requirements of the evaluation board.
 - **Device Tree Source Include (.dtsi)** : They contain common hardware descriptions that can be shared across multiple platforms.
 - **Device Tree Source (.dts)** : They describe the hardware configuration for a specific platform.



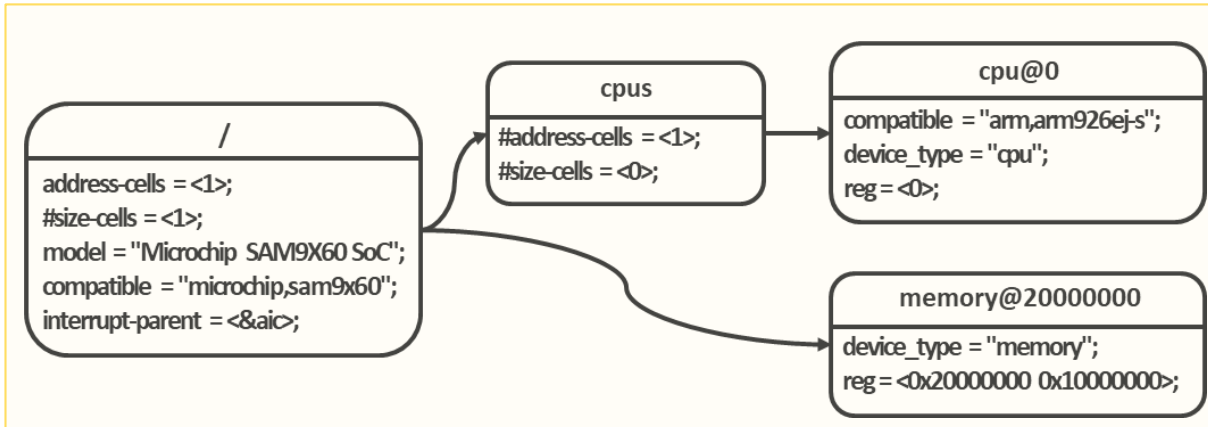
Description Structure of Device-Tree

- Device-Tree consists of a series of **nodes**, which are organized in a tree-like hierarchy. Each node represents a hardware device, such as a CPU, memory, bus, or peripheral.
- **Basic Structure of a Device Tree**
 - **Root Node** : Every device tree has a unique root node, representing the entire system.
 - **Child Nodes** : Child nodes are sub-devices of a parent node and inherit some properties from their parent.
 - **Properties** : Each node can have multiple properties that describe specific information about that node.



Device-Tree-Source Syntax

- Device-Tree Source example for CPU and memory.



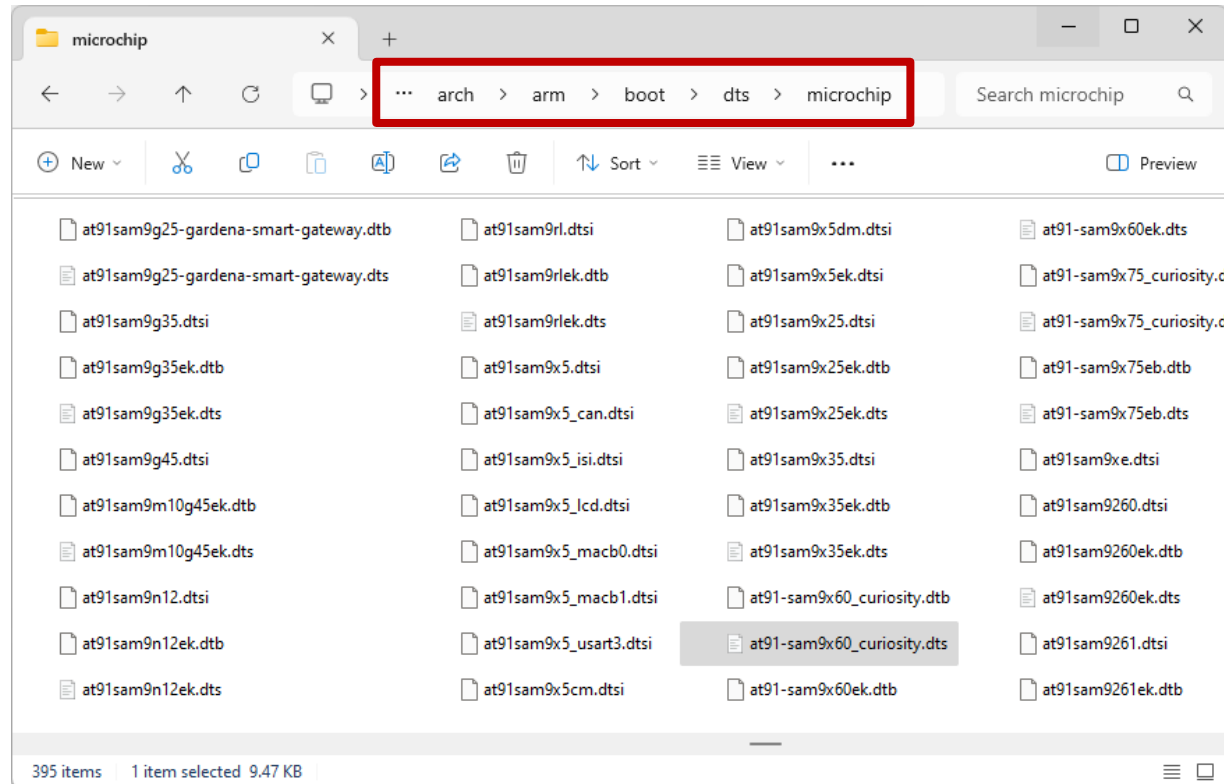
```
/ {  
    cpus {  
        ...  
        cpu@0 {  
            compatible = "arm,arm926ej-s";  
            device_type = "cpu";  
            reg = <0>;  
        };  
    };  
    memory@20000000 {  
        device_type = "memory";  
        reg = <0x20000000 0x10000000>;  
    };  
};
```

Root Node
Nodes for CPUs
CPU0 Node
Properties for CPU0
Memory Node
Properties for Memory

[linux/arch/arm/boot/dts/microchip/sam9x60.dtsi](#)

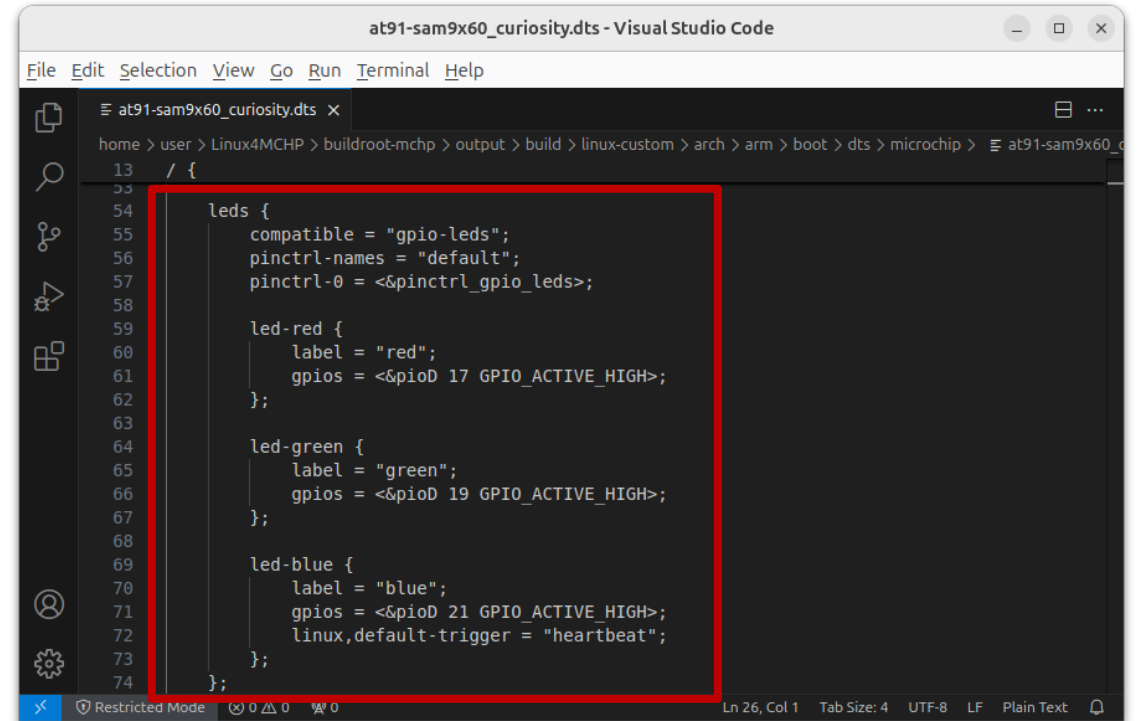
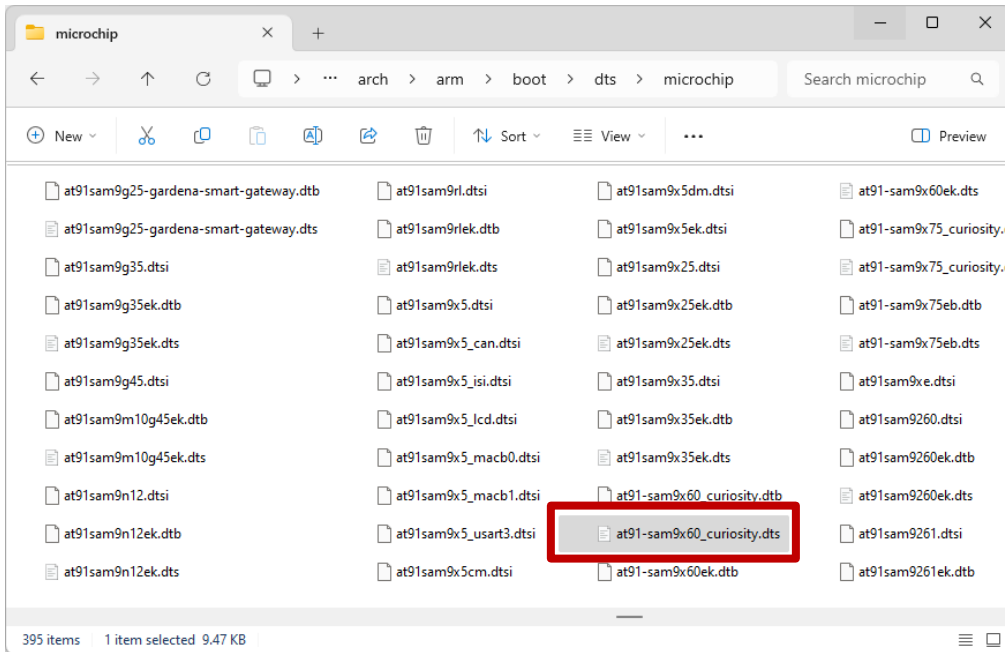
Device-Tree-Source Location in Embedded Linux

- We can find Device-Tree-Source for different MPUs or Evaluation Boards in the source code of Embedded Linux.
- In this course, we can be found the Device-Tree-Source files corresponding to Microchip's MPUs or Evaluation Boards in the "[linux/arch/arm/boot/dts/microchip](#)" folder, the actual path depends on the version of Linux for you.



Device-Tree-Source for LEDs

- **Check** the Device-Tree configuration for the LEDs .
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Tree and Device-Driver

- The Device-Tree provides a description of the system hardware, while the Device-Driver is responsible for controlling and managing these hardware devices.
- The two work closely together to ensure the normal operation of the system.
 - Device-Tree provides configuration information for drivers.
 - Drivers configure hardware based on the Device-Tree.



Device-Driver for LEDs

- Check the required device drivers.

1

```
user@LNX1001: ~  
user@LNX1001:~$  
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp  
user@VM:~$ make linux-menuconfig
```

VM

2

3

4

5

Device-Tree for Peripherals

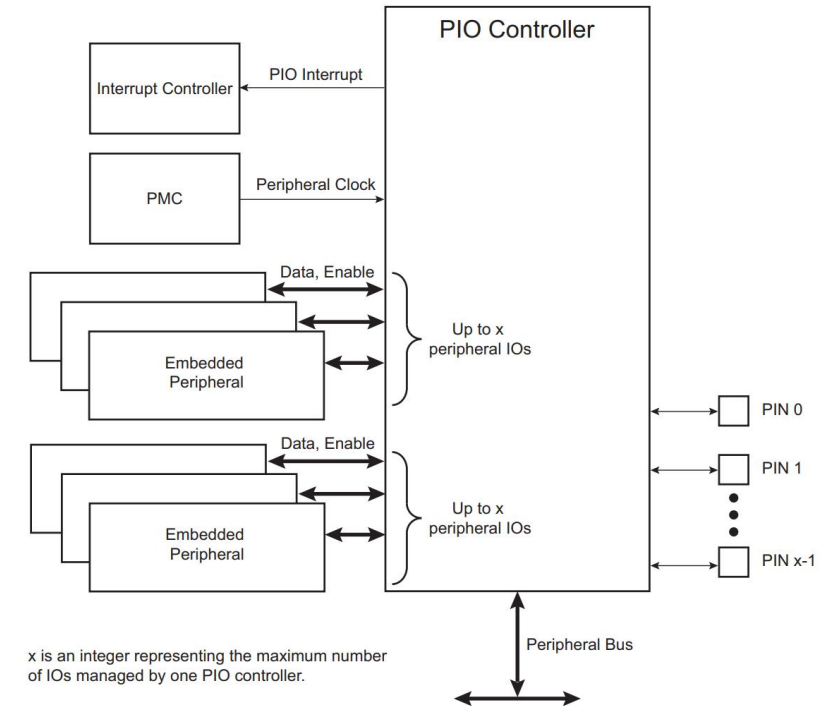
- **This course will mainly focus on the mikroBUS pins of the SAM9X60 curiosity board and Hobby Kit to introduce the Device-Tree configuration of peripherals.**
 - Parallel Input/Output Controller (PIO)
 - Pulse Width Modulation Controller (PWM)
 - Analog-to-Digital Controller (ADC)
 - Flexible Serial Communication Controller (FLEXCOM)
 - Universal Synchronous Asynchronous Receiver Transceiver (USART)
 - Serial Peripheral Interface (SPI)
 - Two-wire Interface (TWI)

PIO and Pin Multiplexing

Parallel Input/Output Controller (PIO)

PIO Introduction

- The **Parallel Input/Output Controller (PIO)** manages up to 32 fully programmable **input/output lines** with synchronous output, providing up to 32 bits of data output in a single write operation.
- Each I/O line may be dedicated as a **general-purpose I/O** or be assigned to a **function of an embedded peripheral**. This ensures effective optimization of the pins of the product.
- Each I/O line of the PIO Controller features the following :
 - An input change interrupt enabling level change detection
 - Additional Interrupt modes enabling rising edge, falling edge, low-level or high-level detection
 - Glitch filter and Debouncing filter
 - Control of the I/O line pullup and pulldown or open drain
 - Multiplexing of Four Peripheral Functions per I/O Line
 - Input visibility and output control

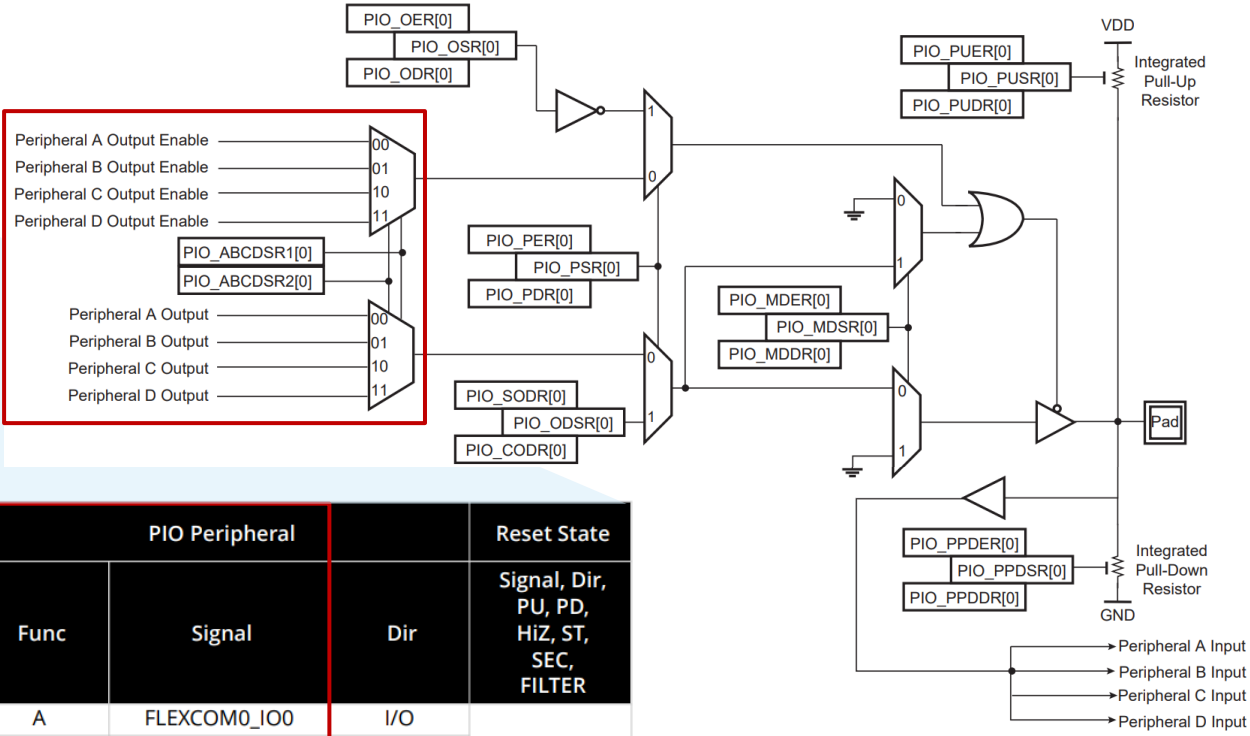


Pin Multiplexing

- Each pin is configurable, depending on the product, as either a general-purpose I/O line only, or as an I/O line multiplexed with peripheral I/O.
- The PIO Controller provides **multiplexing of up to four peripheral functions (A~D)** on a single pin, and the corresponding functions can be found in the Pin Description Table.

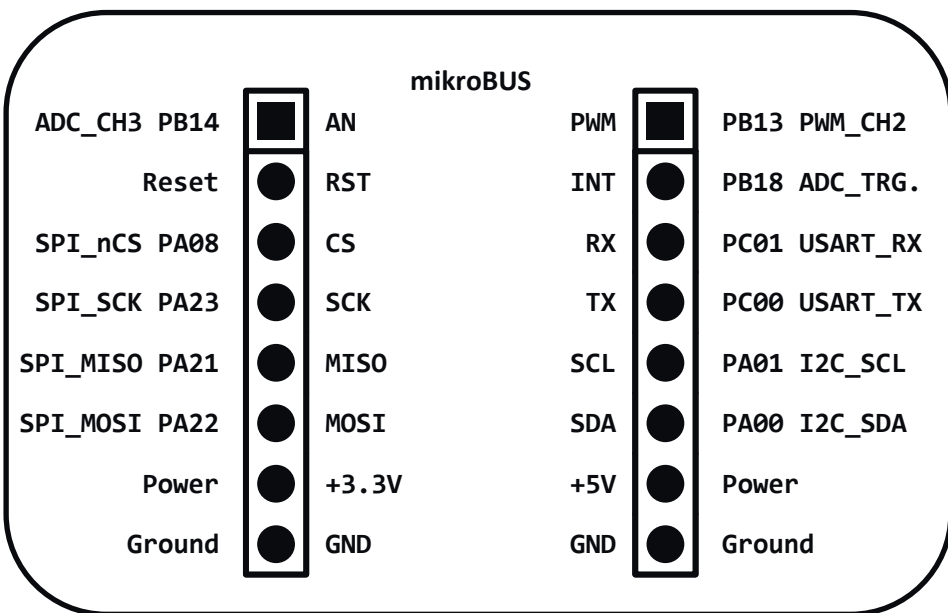
228-pin BGA	Power Rail	I/O Type	Primary		Alternate		PIO Peripheral			Reset State
			Signal	Dir	Signal	Dir	Func	Signal	Dir	
P2	VDDIOP0	GPIO	PA0	I/O	—	—	A B C	FLEXCOM0_IO0 FLEXCOM5_IO4 FLEXCOM4_IO4	I/O O O	PIO, I, PU, ST
M3	VDDIOP0	GPIO	PA1	I/O	—	—	A B C	FLEXCOM0_IO1 FLEXCOM4_IO5 FLEXCOM0_IO4	I/O O O	PIO, I, PU, ST
P1	VDDIOP0	GPIO	PA2	I/O	WKUP1	—	A B C	FLEXCOM0_IO4 SDMMC1_DAT1 E0_TX0	I/O I/O O	PIO, I, PU, ST
L3	VDDIOP0	GPIO	PA3	I/O	—	—	A B C	FLEXCOM0_IO3 SDMMC1_DAT2 E0_TX1	I/O I/O O	PIO, I, PU, ST
N1	VDDIOP0	GPIO	PA4	I/O	—	—	A B C	FLEXCOM0_IO2 SDMMC1_DAT3 E0_TXER	I/O I/O O	PIO, I, PU, ST
L4	VDDIOP0	GPIO	PA5	I/O	—	—	A B C	FLEXCOM1_IO0 CANTX1 FLEXCOM1_IO1	I/O O I/O	PIO, I, PU, ST
M2	VDDIOP0	GPIO	PA6	I/O	—	—	A B C	FLEXCOM1_IO1 CANTX1 FLEXCOM1_IO0	I/O O I/O	PIO, I, PU, ST

228-pin BGA	Power Rail	I/O Type	Primary		Alternate		PIO Peripheral			Reset State
			Signal	Dir	Signal	Dir	Func	Signal	Dir	
P2	VDDIOP0	GPIO	PA0	I/O	—	—	A B C	FLEXCOM0_IO0 FLEXCOM5_IO4 FLEXCOM4_IO4	I/O O O	PIO, I, PU, ST



Pin Multiplexing

- The course will use the mikroBUS socket on the evaluation board to introduce how the peripherals of SAM9X60 MPU is used in the Embedded Linux system.
- The following is the pin definition of the mikroBUS socket on the evaluation board, we can set the pins to different peripheral functions by modifying the Device-Tree.

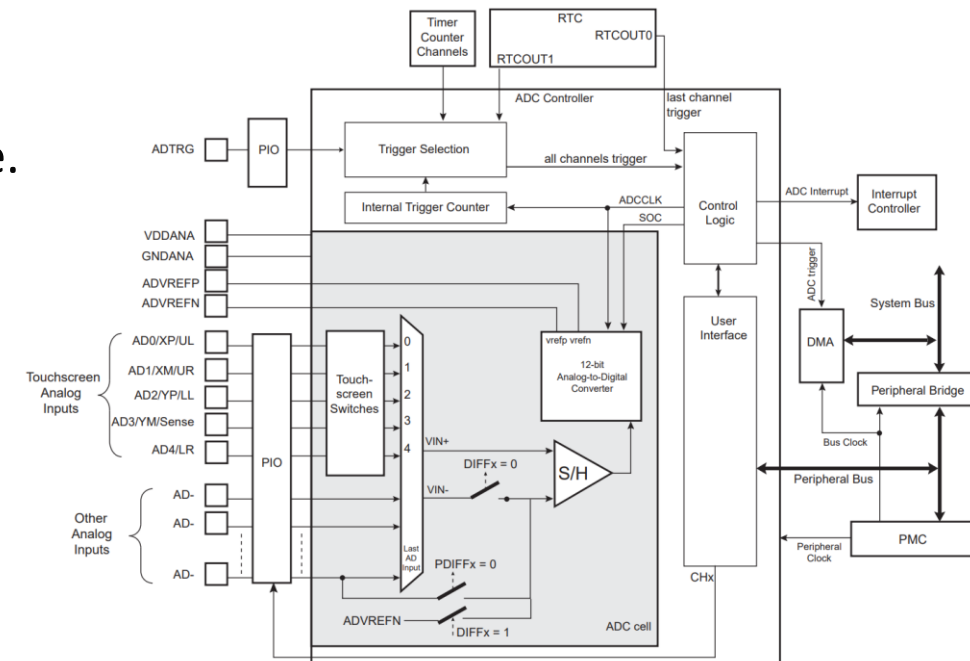


mikroBUS Function	Primary Signal	Alternate Signal	PIO Peripheral	
I2C_SCL	PA00	-	A	FLEXCOM0_IO0
I2C_SDA	PA01	-	A	FLEXCOM0_IO1
SPI_nCS	PA08	-	B	FLEXCOM5_IO3
SPI_MISO	PA21	-	B	FLEXCOM5_IO1
SPI_MOSI	PA22	-	B	FLEXCOM5_IO0
SPI_SCK	PA23	-	B	FLEXCOM5_IO2
PWM_CH2	PB13	-	B	PWM2
ADC_CH3	PB14	AD3	N/A	N/A
ADC_TRG.	PB18	WKUP7	B	ADTRG
USART_TX	PC00	-	C	FLEXCOM7_IO0
USART_RX	PC01	-	C	FLEXCOM7_IO1

Analog-to-Digital Controller (ADC)

ADC Introduction

- The **Analog-to-Digital Controller (ADC)** converts a continuous analog signal (voltage) into a discrete digital signal (numbers).
- When ADC channels are selected as inputs, the associated I/O is **automatically turned in Analog mode**.
- **Characteristics :**
 - 12-bit Resolution with Enhanced Mode up to 16 bits.
 - 1 MSps Conversion Rate.
 - Single-Ended, Pseudo-Differential or Differential Input Voltage.
 - Digital correction of offset and gain errors.
 - Resistive 4-wire and 5-wire Touchscreen Controller.
 - DMA Support.
 - Two Sleep Modes (Automatic Wake-Up on Trigger).
 - Channel Sequence Customizing.
 - Automatic Window Comparison of Converted Values.



Device-Tree for ADC

```
/ {  
  ahb {  
    apb {  
      adc: adc@f804c000 {  
        compatible = "microchip,sam9x60-adc", "atmel,sama5d2-adc";  
        reg = <0xf804c000 0x100>;  
        interrupts = <19 IRQ_TYPE_LEVEL_HIGH 7>;  
        clocks = <&pmc PMC_TYPE_PERIPHERAL 19>;  
        clock-names = "adc_clk";  
        ...  
        atmel,trigger-edge-type = <IRQ_TYPE_EDGE_RISING>;  
        #io-channel-cells = <1>;  
        vddana-supply = <&vdd1_3v3>;  
        vref-supply = <&vdd1_3v3>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_adc_default &pinctrl_adtrg_default>;  
        status = "okay";  
      };  
    };  
  };  
};
```

ADC Node

specifying compatibility between the device and the kernel driver

VDDANA power supply for the ADC

VREF power supply for the ADC

input pins and trigger pin of the ADC

Enable ADC

[sam9x60.dtsi & at91-sam9x60_curiosity.dts](#)

Device-Tree for ADC

```
&pinctrl {
```

Modify the pinctrl controller node

```
    pinctrl_adc_default: adc-default {  
        atmel,pins = <AT91_PIOB 14 AT91_PERIPH_A AT91_PINCTRL_NONE>;  
    };
```

configure the input
pin of the ADC

```
    pinctrl_adtrg_default: adtrg-default {  
        atmel,pins = <AT91_PIOB 18 AT91_PERIPH_B AT91_PINCTRL_PULL_UP>;  
    };
```

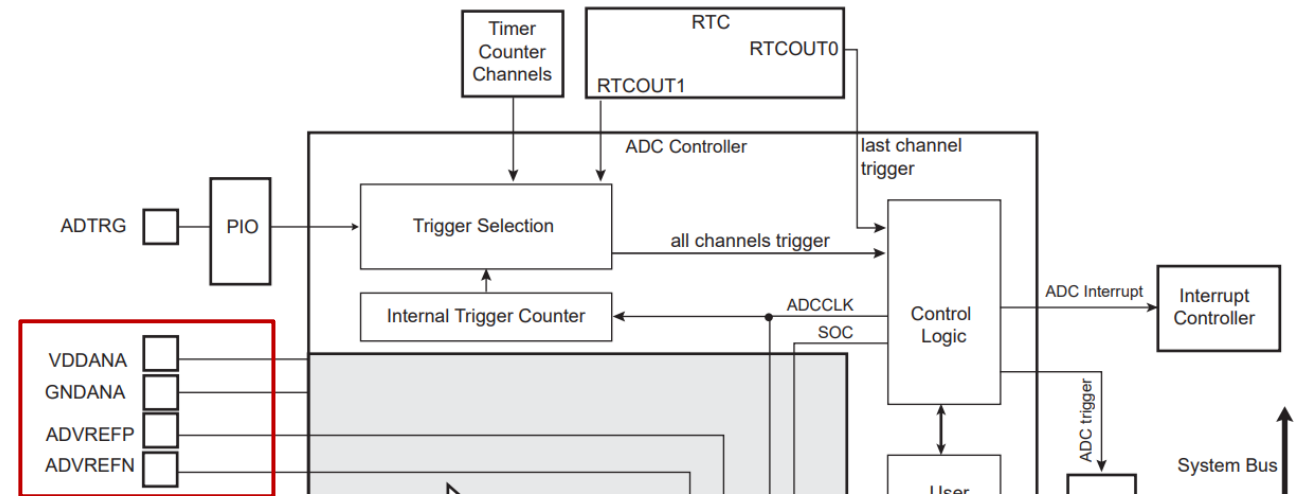
configure the trigger
pin of the ADC

```
    ...  
};
```

[at91-sam9x60_curiosity.dts](#)

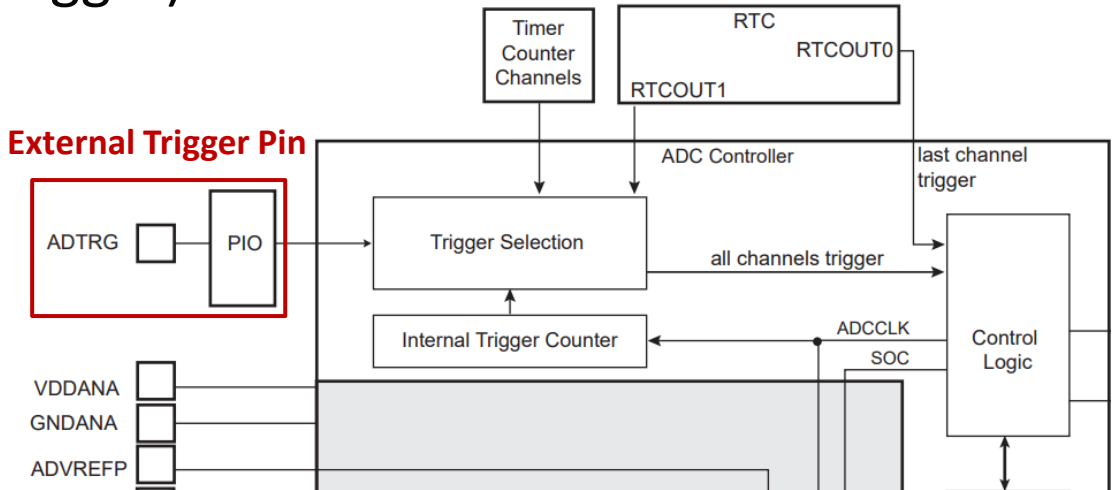
ADC Reference Voltage

- The voltage reference input of the ADC is the ADVREFP pin and the negative reference voltage is ADVREFN.
- **ADVREFP** input pin voltage range is 2.4 to V_{DDANA} (Typical 3.3V), default connection to **3.3V** on SAM9X60 evaluation board.
- **ADVREFN** pin must be connected to the PCB **Ground (GND)** plane.



ADC Trigger Mode

- ADC Trigger from:
 - Software trigger (ADC_CR. START)
 - External trigger pin (ADTRG pin)
 - Timer counter outputs (corresponding TIOA trigger)
 - ADC internal trigger counter
 - Trigger on pen contact detection
 - PWM event line

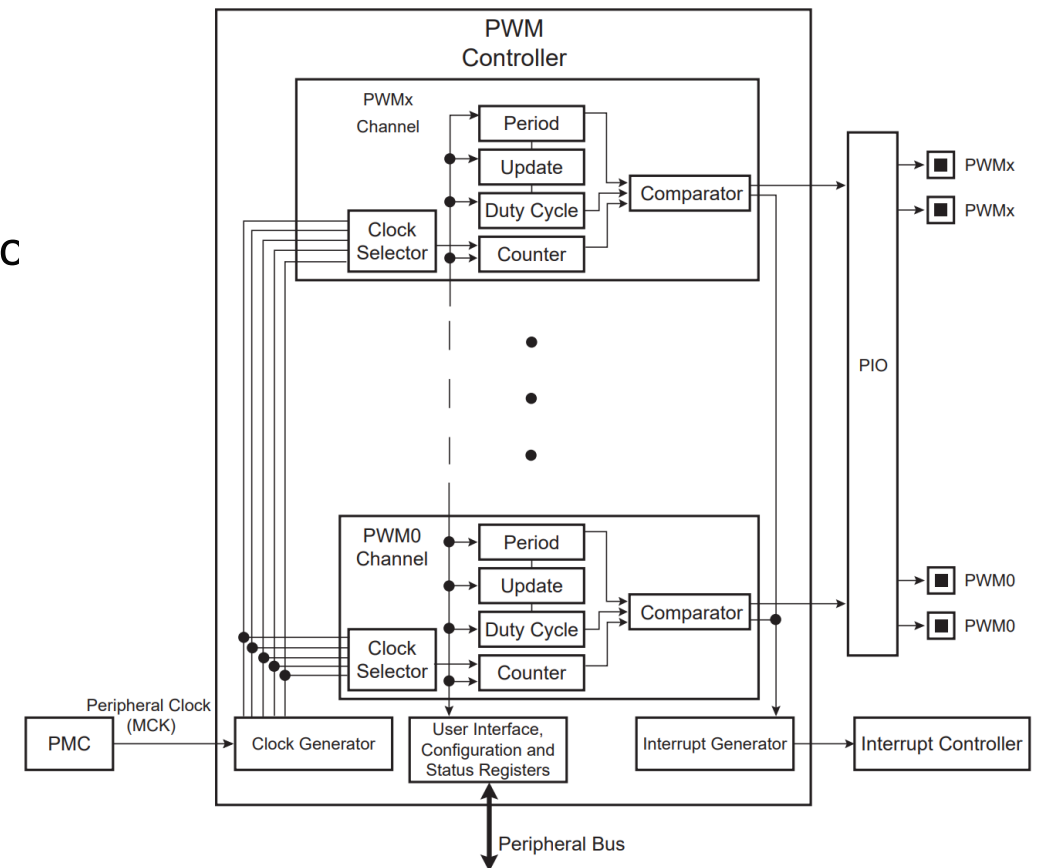


228-pin BGA	Power Rail	I/O Type	Primary		Alternate		PIO Peripheral			Reset State
			Signal	Dir	Signal	Dir	Func	Signal	Dir	Signal, Dir, PU, PD, HiZ, ST, SEC, FILTER
E4	VDDANA	GPIO	PB18	I/O	WKUP7	—	A	IRQ	I	PIO, I, PU, ST
							B	ADTRG	I	

Pulse Width Modulation Controller (PWM)

PWM Introduction

- The **Pulse Width Modulation Controller (PWM)** can be configured to control the characteristics of the output waveform (such as **period**, **duty cycle**, and **polarity**), each channel can be controlled independently.
- **Characteristics :**
 - 4 Channels
 - One 16-bit Counter Per Channel
 - Common Clock Generator Providing Thirteen Different Clc
 - Independent Channels
 - Independent Enable Disable Command
 - Independent Clock Selection
 - Independent Period and Duty Cycle
 - Double Buffering of Period or Duty Cycle
 - Programmable Selection of The Output Waveform Polarity
 - Programmable Center or Left Aligned Output Waveform



Device-Tree for PWM

```
/ {  
  ahb {  
    apb {  
      pwm0: pwm@f8034000 {  
        compatible = "microchip,sam9x60-pwm";  
        reg = <0xf8034000 0x300>;  
        interrupts = <18 IRQ_TYPE_LEVEL_HIGH 4>;  
        clocks = <&pmc PMC_TYPE_PERIPHERAL 18>;  
        #pwm-cells = <3>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_pwm0_0 &pinctrl_pwm0_1 &pinctrl_pwm0_2>;  
        status = "okay";  
      };  
    };  
  };  
};
```

PWM Node

specifying compatibility between the device and the kernel driver

input pins of the PWM

Enable PWM

[sam9x60.dtsi](#) & [at91-sam9x60_curiosity.dts](#)

Device-Tree for PWM

```
&pinctrl {
```

Modify the pinctrl controller node

```
    pwm0 {
```

```
        pinctrl_pwm0_0: pwm0-0 {
```

```
            atmel,pins = <AT91_PIOB 12 AT91_PERIPH_B AT91_PINCTRL_NONE>;
```

```
        };
```

```
        pinctrl_pwm0_1: pwm0-1 {
```

```
            atmel,pins = <AT91_PIOB 13 AT91_PERIPH_B AT91_PINCTRL_NONE>;
```

```
        };
```

```
        pinctrl_pwm0_2: pwm0-2 {
```

```
            atmel,pins = <AT91_PIOD 16 AT91_PERIPH_B AT91_PINCTRL_NONE>;
```

```
        };
```

```
    };
```

```
    ...
```

```
};
```

configure the input
pin of the PWM

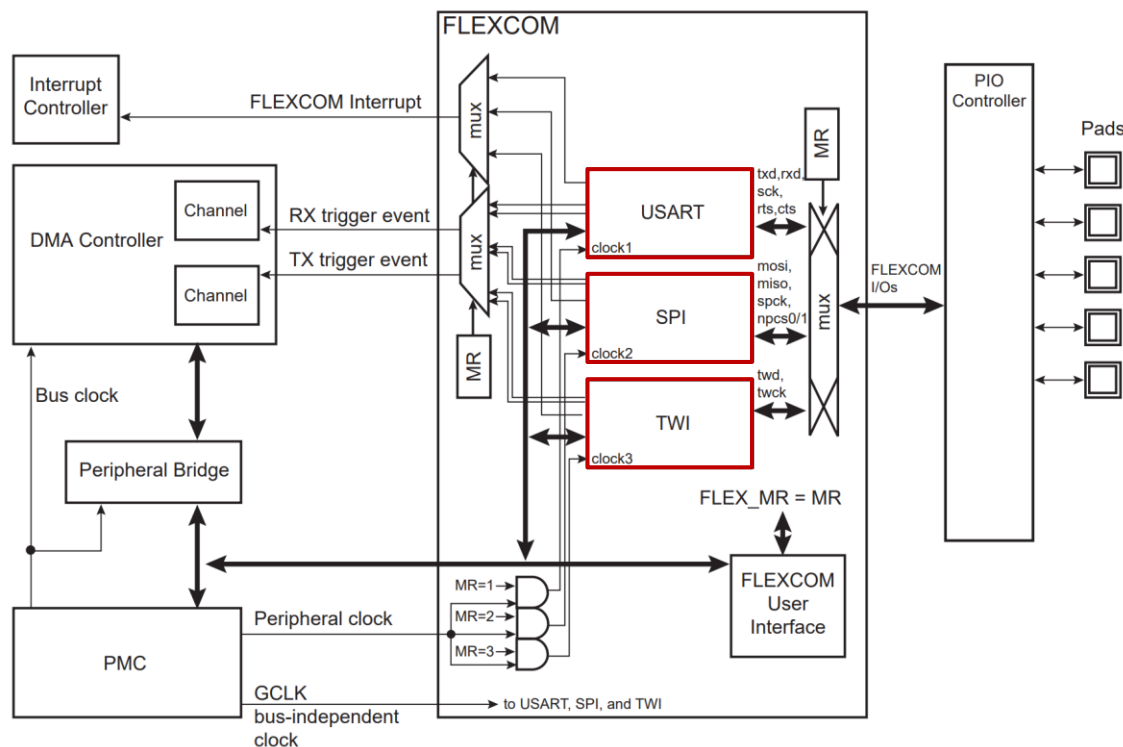
at91-sam9x60_curiosity.dts

Flexible Serial Communication Controller (FLEXCOM)

USART, SPI and TWI (I²C)

FLEXCOM Introduction

- The **Flexible Serial Communication Controller (FLEXCOM)** offers several serial communication protocols that are managed by the three submodules : **USART**, **SPI**, and **TWI (I2C)**.



Functions and Features	FLEXCOM Instance												
	0	1	2	3	4	5	6	7	8	9	10	11	12
TWI Function	X	X	X	X	X	X	X	X	X	X	X	X	X
Normal/fast (400 kbit/s) / FM+ (1 Mbit/s)	X	X	X	X	X	X	X	X	X	X	X	X	X
Alternate command	X	X	X	X	X	X	X	X	X	X	X	X	X
Three-client ADDR	X	X	X	X	X	X	X	X	X	X	X	X	X
High speed (3.4 Mbit/s)	X	X	X	X	X	X	X	X	X	X	X	X	X
Sniffer	X	X	X	X	X	X	X	X	X	X	X	X	X
TWI FIFO size	16 bytes												
UART / USART Function	X	X	X	X	X	X	X	X	X	X	X	X	X
Two-wire UART	X	X	X	X	X	X	X	X	X	X	X	X	X
Five-wire USART	X	X	X	X	X	X	-	-	-	-	-	-	-
Hardware handshaking/RS485	X	X	X	X	X	X	-	-	-	-	-	-	-
ISO7816	X	X	X	X	X	X	-	-	-	-	-	-	-
IrDA	X	X	X	X	X	X	-	-	-	-	-	-	-
LIN	X	X	X	X	X	X	-	-	-	-	-	-	-
LON	X	X	X	X	-	-	-	-	-	-	-	-	-
Manchester	X	X	X	X	X	X	-	-	-	-	-	-	-
USART FIFO Size	16 bytes												
SPI Function	X	X	X	X	X	X	-	-	-	-	-	-	-
1CS	X	X	X	X	X	X	-	-	-	-	-	-	-
4CS	-	-	-	-	X	X	-	-	-	-	-	-	-
SPI FIFO size	16 bytes						-	-	-	-	-	-	-

FLEXCOM Introduction

- The **Flexible Serial Communication Controller (FLEXCOM)** offers several serial communication protocols that are managed by the three submodules :

- **USART**

The Universal Synchronous Asynchronous Receiver Transceiver (USART) provides one full-duplex universal synchronous asynchronous serial link.

- **SPI**

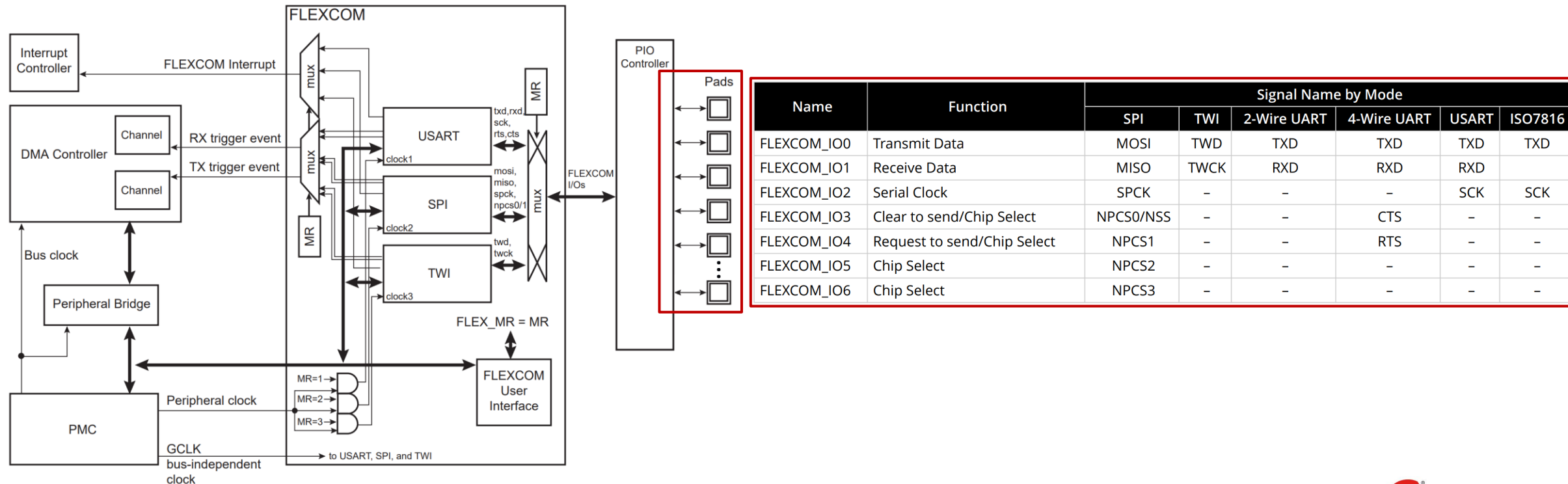
The Serial Peripheral Interface (SPI) is essentially a shift register that serially transmits data bits to other SPIs. During a data transfer, one SPI system acts as the “host” which controls the data flow, while the other devices act as “clients” which have data shifted into and out by the host.

- **TWI (I²C)**

The Two-wire Interface (TWI) interconnects components on a unique two-wire bus, made up of one clock line and one data line based on a byte-oriented transfer format. It can be used with any Two-wire Interface bus Serial EEPROM and I2C-compatible devices

FLEXCOM I/O Lines

- The pins used for interfacing the FLEXCOM are multiplexed with the PIO lines.
- The programmer must first program the PIO controller to assign the desired FLEXCOM pins to their peripheral function. If I/O lines of the FLEXCOM are not used by the application, they can be used for other purposes by the PIO Controller.



FLEXCOM - USART

Device Tree

Device-Tree for FLEXCOM (UART)

```
/ {  
    ahb {  
        apb {  
            flx7: flexcom@f8014000 {  
                compatible = "atmel,sama5d2-flexcom";  
                reg = <0xf8014000 0x200>;  
                clocks = <&pmc PMC_TYPE_PERIPHERAL 10>;  
                #address-cells = <1>;  
                #size-cells = <1>;  
                ranges = <0x0 0xf8014000 0x800>;  
                atmel,flexcom-mode = <ATMEL_FLEXCOM_MODE_USART>;  
                status = "okay";  
  
                uart7: serial@200 {  
                    ...  
                };  
                ...  
            };  
        };  
    };  
};
```

Flexcom 7 Node

specifying compatibility between the device and the kernel driver

Set to UART mode

Enable Flexcom 7

UART Node of Flexcom 7

Merge sam9x60.dtsi & at91-sam9x60_curiosity.dts

Device-Tree for FLEXCOM (UART)

```
...  
flx7: flexcom@f8014000 {
```

```
    ...  
    uart7: serial@200 {
```

UART 7 Node

```
        compatible = "microchip,sam9x60-usart", "atmel,at91sam9260-usart";
```

specifying compatibility between the device and the kernel driver

```
        reg = <0x200 0x200>;
```

```
        interrupts = <10 IRQ_TYPE_LEVEL_HIGH 7>;
```

```
        ...
```

```
        clocks = <&pmc PMC_TYPE_PERIPHERAL 10>;
```

```
        clock-names = "usart";
```

```
        atmel,use-dma-rx;
```

```
        atmel,use-dma-tx;
```

```
        atmel,fifo-size = <16>;
```

```
        pinctrl-names = "default";
```

```
        pinctrl-0 = <&pinctrl_flx7_default>;
```

configure the pin of the UART 7

```
        status = "okay";
```

Enable UART 7

```
    };
```

```
    ...
```

```
};  
...  
...
```

Merge sam9x60.dtsi & at91-sam9x60_curiosity.dts

Device-Tree for FLEXCOM (UART)

```
/ {  
    aliases {  
        ...  
        serial2 = &uart7; Set alias for UART 7 (ttyS2)  
    };  
};
```

```
&pinctrl { Modify the pinctrl controller node  
    ...  
    flexcom {  
        pinctrl_flx7_default: flx7-usart {  
            atmel,pins =  
                <AT91_PIOC 0 AT91_PERIPH_C AT91_PINCTRL_NONE  
                  AT91_PIOC 1 AT91_PERIPH_C AT91_PINCTRL_NONE>;  
        };  
    };  
    ...  
};
```

Name	Function	Signal Name by Mode					
		SPI	TWI	2-Wire UART	4-Wire UART	USART	ISO7816
FLEXCOM_IO0	Transmit Data	MOSI	TWD	TXD	TXD	TXD	TXD
FLEXCOM_IO1	Receive Data	MISO	TWCK	RXD	RXD	RXD	
FLEXCOM_IO2	Serial Clock	SPCK	-	-	-	SCK	SCK
FLEXCOM_IO3	Clear to send/Chip Select	NPCS0/NSS	-	-	CTS	-	-
FLEXCOM_IO4	Request to send/Chip Select	NPCS1	-	-	RTS	-	-
FLEXCOM_IO5	Chip Select	NPCS2	-	-	-	-	-
FLEXCOM_IO6	Chip Select	NPCS3	-	-	-	-	-

configure the pin of the Flexcom 7 Pad 0 (TXD)

configure the pin of the Flexcom 7 Pad 1 (RXD)

[at91-sam9x60_curiosity.dts](#)

FLEXCOM - TWI (I²C)

Device Tree

Device-Tree for FLEXCOM (I²C)

```
/ {
  ahb {
    apb {
      flx0: flexcom@f801c000 {
        compatible = "atmel,sama5d2-flexcom";
        reg = <0xf801c000 0x200>;
        clocks = <&pmc PMC_TYPE_PERIPHERAL 5>;
        #address-cells = <1>;
        #size-cells = <1>;
        ranges = <0x0 0xf801c000 0x800>;
        atmel,flexcom-mode = <ATMEL_FLEXCOM_MODE_TWI>;
        status = "okay";

        i2c0: i2c@600 {
          ...
        };
        ...
      };
    };
  };
};
```

Flexcom 0 Node

specifying compatibility between the device and the kernel driver

Set to I²C (TWI) mode

Enable Flexcom 0

I²C Node of Flexcom 0

Merge sam9x60.dtsi & at91-sam9x60_curiosity.dts

Device-Tree for FLEXCOM (I²C)

```
...  
flx0: flexcom@f801c000 {
```

```
...  
i2c0: i2c@600 {
```

I²C 0 Node

```
compatible = "microchip,sam9x60-i2c";
```

specifying compatibility between
the device and the kernel driver

```
...
```

```
dmab = <0>, <0>;
```

Disable DMA

```
dma-names = "tx", "rx";
```

```
atmel,fifo-size = <16>;
```

```
pinctrl-names = "default";
```

```
pinctrl-0 = <&pinctrl_flx0_default>;
```

configure the pin of the I²C 0

```
#address-cells = <1>;
```

define how address is represented for I2C devices

```
#size-cells = <0>;
```

define how size is represented for I2C devices

```
i2c-analog-filter;
```

```
i2c-digital-filter;
```

```
i2c-digital-filter-width-ns = <35>;
```

```
status = "okay";
```

Enable I²C 0

```
};
```

```
...
```

```
};
```

Merge sam9x60.dtsi & at91-sam9x60_curiosity.dts

Device-Tree for FLEXCOM (I²C)

```
/ {  
  aliases {
```

```
    ...  
    i2c0 = &i2c0;  
  };  
};
```

Set alias for I²C 0 (I2C-0)

```
&pinctrl {
```

Modify the pinctrl controller node

```
    ...  
    flexcom {  
      pinctrl_flx0_default: flx0-twi {  
        atmel,pins =  
          <AT91_PIOA 0 AT91_PERIPH_A AT91_PINCTRL_PULL_UP  
            AT91_PIOA 1 AT91_PERIPH_A AT91_PINCTRL_PULL_UP>;  
      };  
    };  
    ...  
};
```

configure the pin of the Flexcom 0 Pad 0 (TWD, SDA)

configure the pin of the Flexcom 0 Pad 1 (TWCK, SCK)

Name	Function	Signal Name by Mode					
		SPI	TWI	2-Wire UART	4-Wire UART	USART	ISO7816
FLEXCOM_IO0	Transmit Data	MOSI	TWD	TXD	TXD	TXD	TXD
FLEXCOM_IO1	Receive Data	MISO	TWCK	RXD	RXD	RXD	
FLEXCOM_IO2	Serial Clock	SPCK	-	-	-	SCK	SCK
FLEXCOM_IO3	Clear to send/Chip Select	NPCS0/NSS	-	-	CTS	-	-
FLEXCOM_IO4	Request to send/Chip Select	NPCS1	-	-	RTS	-	-
FLEXCOM_IO5	Chip Select	NPCS2	-	-	-	-	-
FLEXCOM_IO6	Chip Select	NPCS3	-	-	-	-	-

at91-sam9x60_curiosity.dts

FLEXCOM - SPI

Device Tree

Device-Tree for FLEXCOM (SPI)

```
/ {  
    ahb {  
        apb {  
            flx5: flexcom@f0004000 {  
                compatible = "atmel,sama5d2-flexcom";  
                reg = <0xf0004000 0x200>;  
                clocks = <&pmc PMC_TYPE_PERIPHERAL 14>;  
                #address-cells = <1>;  
                #size-cells = <1>;  
                ranges = <0x0 0xf0004000 0x800>;  
                atmel,flexcom-mode = <ATMEL_FLEXCOM_MODE_SPI>;  
                status = "okay";  
  
                spi5: spi@400 {  
                    ...  
                };  
                ...  
            };  
        };  
    };  
};
```

Flexcom 5 Node

specifying compatibility between the device and the kernel driver

Set to SPI mode

Enable Flexcom 5

SPI Node of Flexcom 5

Merge sam9x60.dtsi with added SPI node

Device-Tree for FLEXCOM (SPI)

```
...  
flx5: flexcom@f0004000 {  
    ...  
    spi5: spi@400 {
```

SPI 5 Node

```
        compatible = "microchip,sam9x60-spi", "atmel,at91rm9200-spi";  
        ...  
        dmas = <0>, <0>;  
        dma-names = "tx", "rx";  
        atmel,fifo-size = <16>;  
        pinctrl-names = "default";  
        pinctrl-0 = <&pinctrl_flx5_default>;  
        status = "okay";  
        spidev@0 {
```

specifying compatibility between the device and the kernel driver

Disable DMA

```
        compatible = "rohm,dh2228fv";  
        reg = <0>;  
        spi-max-frequency = <1000000>;  
    };  
};  
...  
};  
...
```

configure the pin of the SPI 5

Enable SPI 5

SPI Device 0 Node

declares compatibility with the basic SPI device

Set the memory offset of SPI

specifies the maximum supported frequency for the SPI device

Merge sam9x60.dtsi with added SPI node

Device-Tree for FLEXCOM (SPI)

&pinctrl {

Modify the pinctrl controller node

```
...
flexcom {
    pinctrl_flx5_default: flx5_spi {
        atmel,pins =
        <AT91_PIOA 22 AT91_PERIPH_B AT91_PINCTRL_PULL_UP
        AT91_PIOA 21 AT91_PERIPH_B AT91_PINCTRL_PULL_UP
        AT91_PIOA 23 AT91_PERIPH_B AT91_PINCTRL_PULL_UP
        AT91_PIOA 8 AT91_PERIPH_B AT91_PINCTRL_PULL_UP>;
    };
};
```

configure the pin of the Flexcom 5 Pad 0 (MOSI)

configure the pin of the Flexcom 5 Pad 1 (MISO)

configure the pin of the Flexcom 5 Pad 2 (SPCK)

configure the pin of the Flexcom 5 Pad 3 (NPCS0)

```
...
};
```

at91-sam9x60_curiosity.dts

Lab3 – Use Script to Read ADC Value and Control PWM

Lab3 – Use Script to Read ADC Value and Control PWM

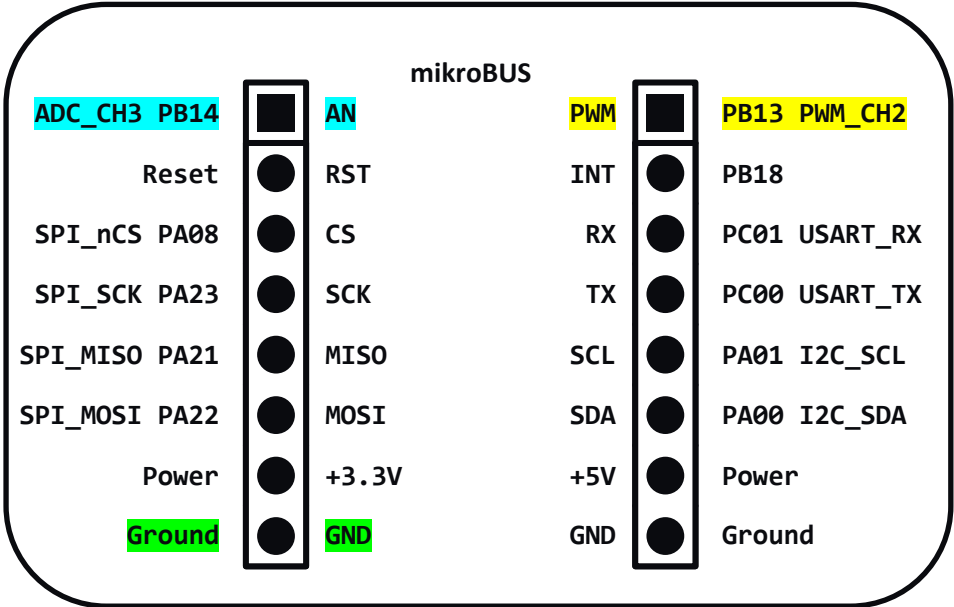
- In the Embedded Linux provided from the course, the default will be to config the ADC and PWM in the Device-Tree, and enable the corresponding device driver. It will list them in **sysfs**, we can control them by reading and writing the .
- Follow the steps below to complete the above requirements :
 1. Connect the ADC of **mikroBUS AN Pin (PB14, ADC Ch. 3)** to the external potentiometer (0 ~ 3.3V).
 2. Connect the PWM of **mikroBUS PWM pin (PB13, PHM Ch. 2)** to the external LED.
 3. Create a script in the Embedded Linux OS.
 4. Execute the script to setup the PWM period to **1kHz**, then read the ADC value periodically and use it to control the PWM Duty between **0 ~ 100%**.

Let's go!

Lab3 – Use Script to Read ADC Value and Control PWM

Preparation

- Connect the ADC of mikroBUS AN Pin (PB14, ADC Ch. 3) to the external potentiometer.
- Connect the PWM of mikroBUS PWM pin (PB13, PWM Ch. 2) to the external LED.



mikroBUS PIN	External
AN	Potentiometer, VR1 (0 ~ 3.3V)
PWM	LED, LED1
GND	GND

Lab3 – Use Script to Read ADC Value and Control PWM

Step 1

- **Create a Bash script** for setup PWM.

The image shows a Tera Term terminal window with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help). A dark-themed terminal window is overlaid on the left, and a yellow callout box on the right displays the script content.

1 `# nano PWM_setup.sh`

2

```
#!/bin/bash
cd /sys/class/pwm/pwmchip0/
if [ ! -e "./pwm2" ]; then
    echo 2 > export
fi
echo 10000 > pwm2/period
echo 0 > pwm2/duty_cycle
echo 1 > pwm2/enable
```

EVB

nano PWM_setup.sh

Lab3 – Use Script to Read ADC Value and Control PWM

Step 2

- **Create a Bash script** for read ADC value and control PWM.

1

nano ADC_PWM.sh

EVB

2

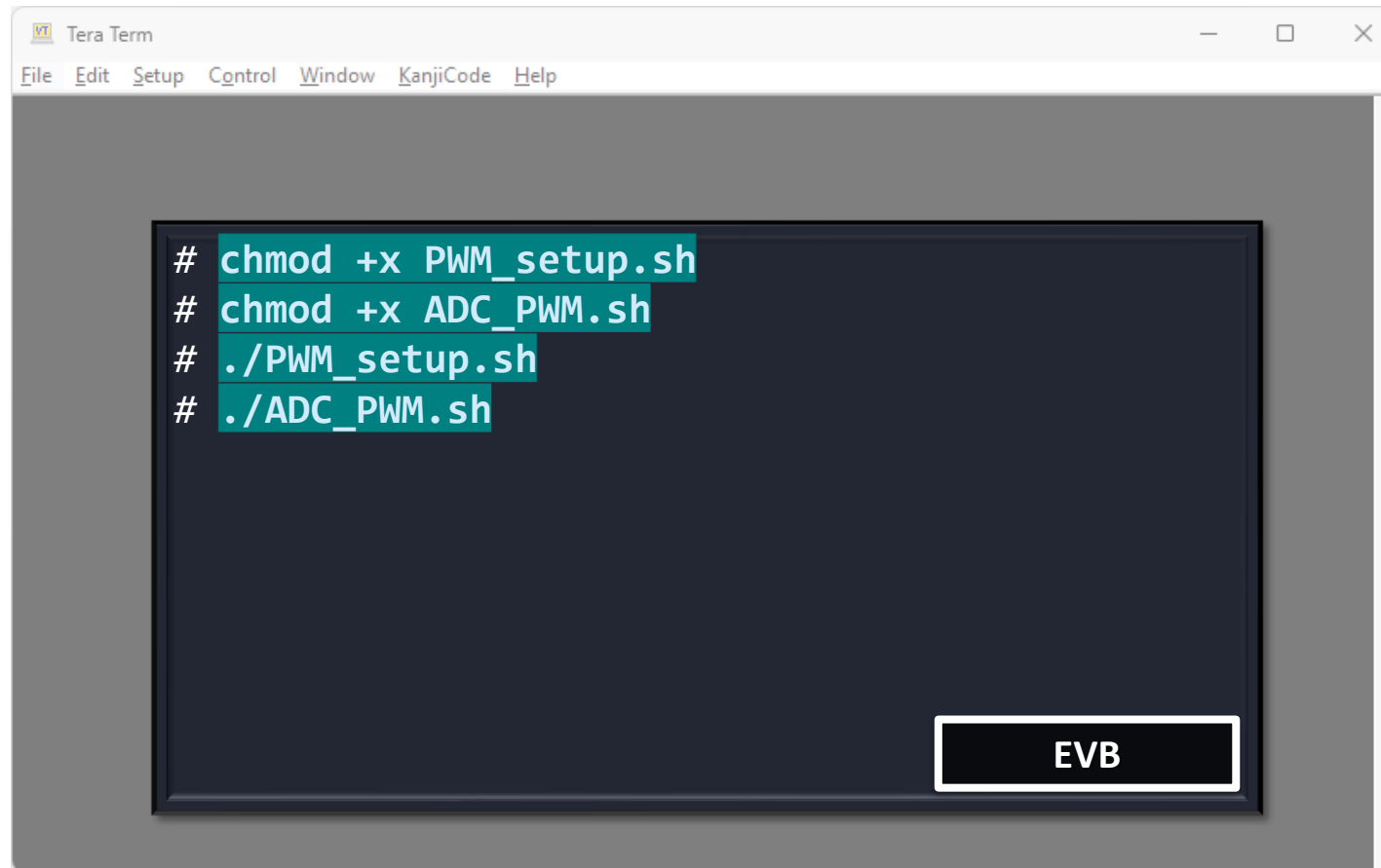
```
#!/bin/bash
adc_dir="/sys/bus/iio/devices/iio:device0"
pwm_dir="/sys/class/pwm/pwmchip0"
echo 1 > $adc_dir"/scan_elements/in_voltage3_en"
pwm_period=$(cat $pwm_dir"/pwm2/period")
while true
do
    adc_value=$(cat $adc_dir"/in_voltage3_raw")
    pwm_duty=$((pwm_period*adc_value/16380))
    echo $pwm_duty > $pwm_dir"/pwm2/duty_cycle"
    echo "PWM duty is "$[pwm_duty*100/pwm_period]"%"
    sleep 0.01
done
```

nano
ADC_PWM.sh

Lab3 – Use Script to Read ADC Value and Control PWM

Step 3

- **Execute the Bash script** in the Embedded Linux OS.

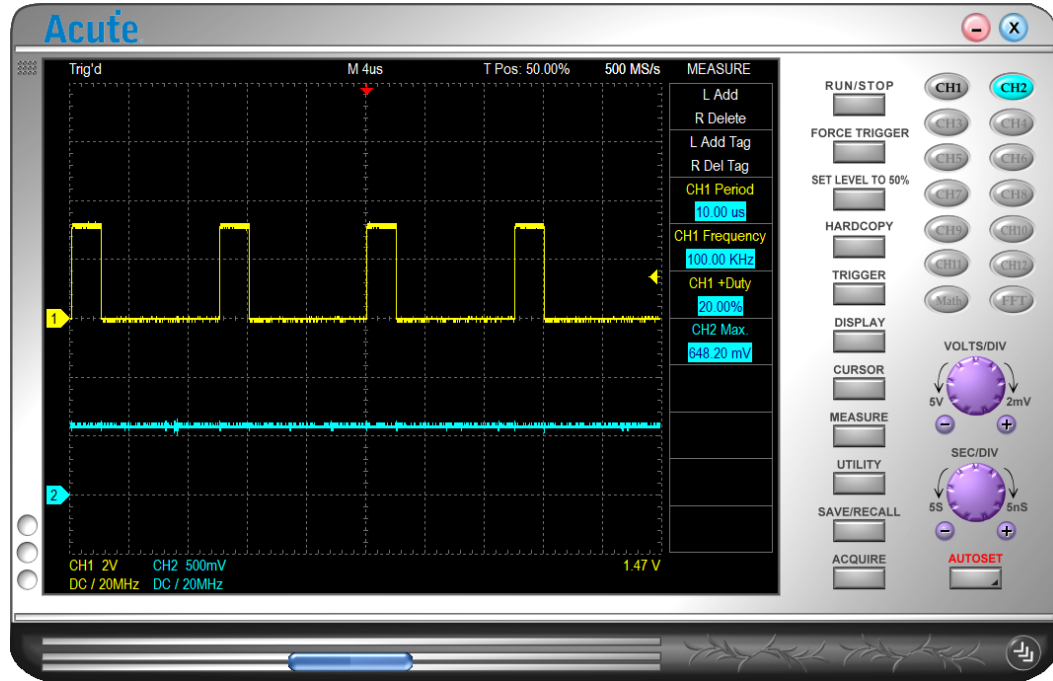


```
# chmod +x PWM_setup.sh
# chmod +x ADC_PWM.sh
# ./PWM_setup.sh
# ./ADC_PWM.sh
```

EVB

Lab3 – Use Script to Read ADC Value and Control PWM

Result



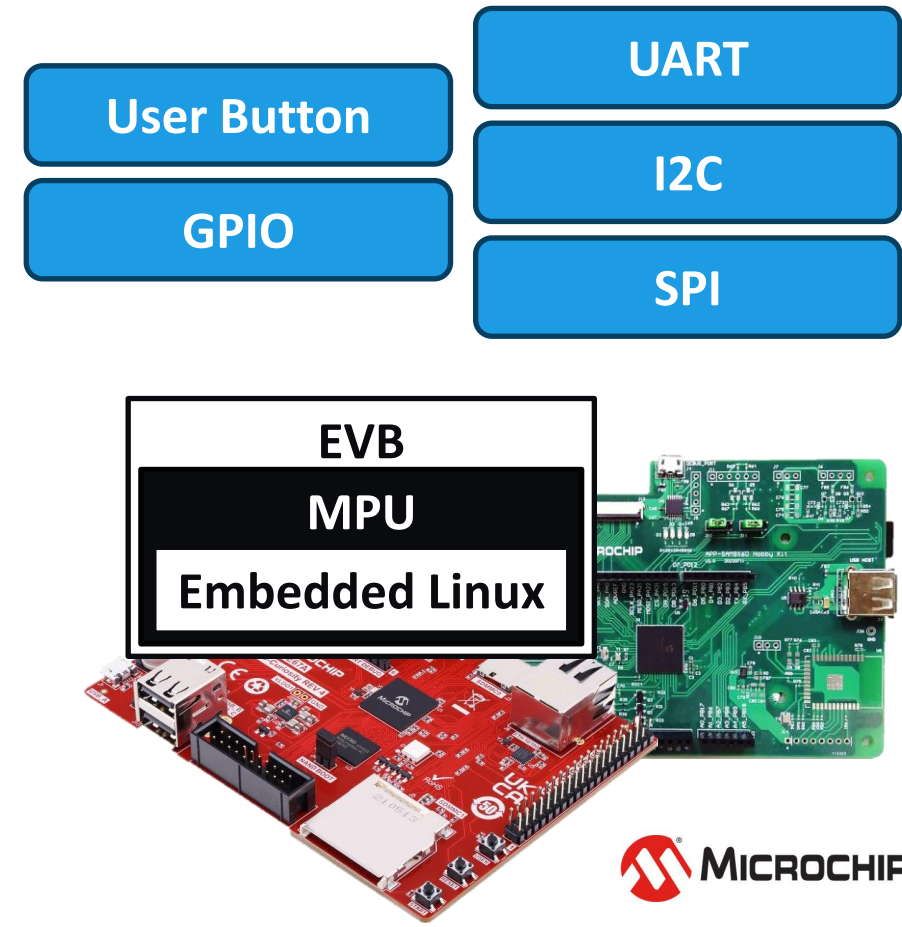
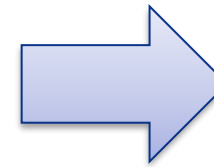
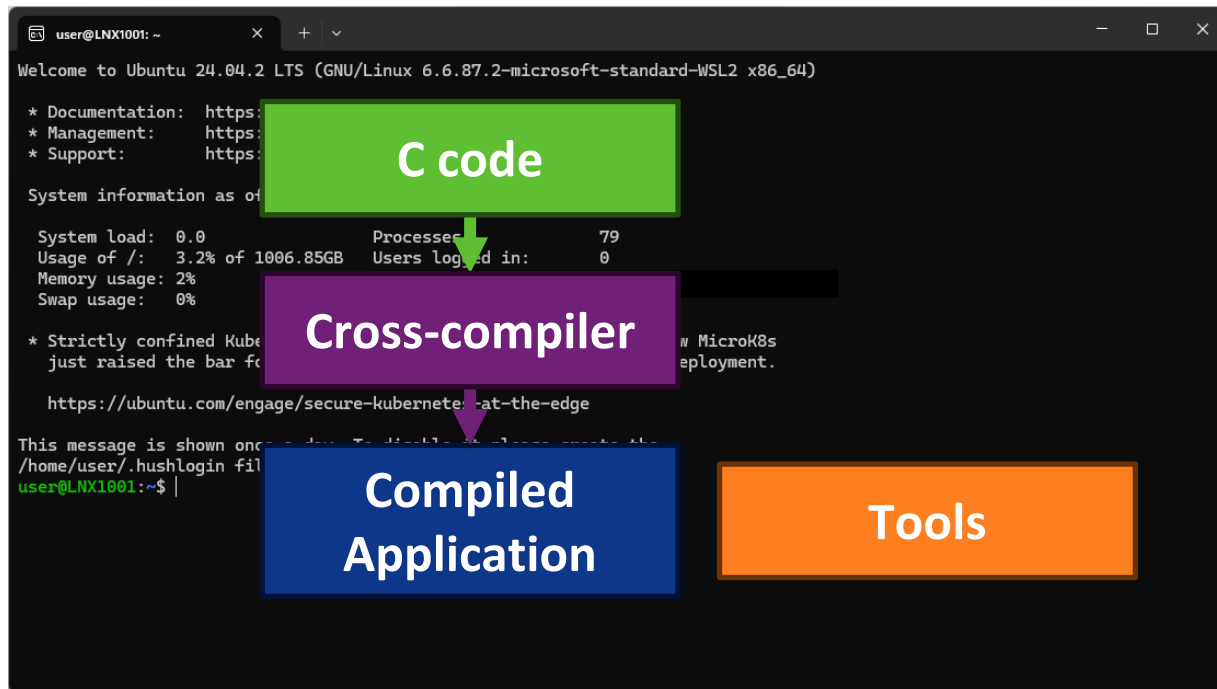
```
user@VM: ~  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 19%  
PWM duty is 20%  
PWM duty is 19%  
PWM duty is 20%  
PWM duty is 19%  
PWM duty is 20%  
PWM duty is 19%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 19%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 20%  
PWM duty is 19%  
P
```

mikroBUS PIN	External
AN	Potentiometer, VR1 (0 ~ 3.3V)
PWM	LED, LED1
GND	GND

Labs - Use Tools and C Language to Control Peripherals

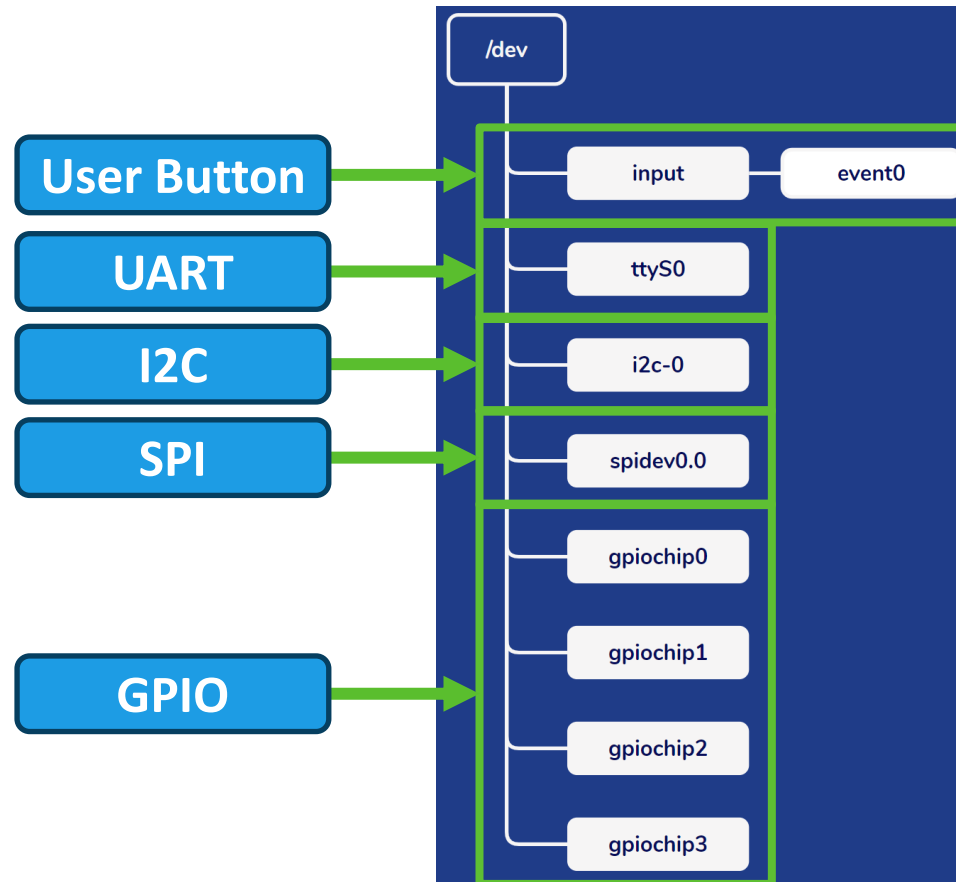
Use Tools and C Language to Control Peripherals

- The course designed some Labs to show how to use **Tools** and **C-Language-Applications** to control MPU's peripherals. We will try to control Buttons or control the external hardware devices through UART, I2C, SPI, User Button and GPIO by execute tools or compiled applications.



Device and /dev Directory

- In embedded Linux systems, the **/dev** directory providing a standardized interface for interacting with various hardware devices. This means that **users can operate hardware devices like common files**, such as reading, writing, opening, closing, etc.



Tools and Utilities for Embedded Linux System

- Embedded Linux System provide many specialized tools to assist developers in interacting with hardware, configuring the system, and debugging.

The following are some common tools :

- **UART Tools**

- **Microcom** : A simple serial terminal emulator for communicating with devices over serial ports.

- **I2C Tools**

- **i2cdetect** : Detects devices on the I2C bus.
 - **i2cdump** : Reads data from an I2C device.
 - **i2cget** : Reads a value from an I2C device.
 - **i2cset** : Reads a value from an I2C device.
 - **i2ctransfer** : For more complex I2C operations.

- **SPI Tools**

- **spi-config** : Sets up the SPI bus (speed, data bits, etc.).
 - **spi-pipe** : Sends and receives data over the SPI bus.

Tools and Utilities for Embedded Linux System

- **Event Test Tools**

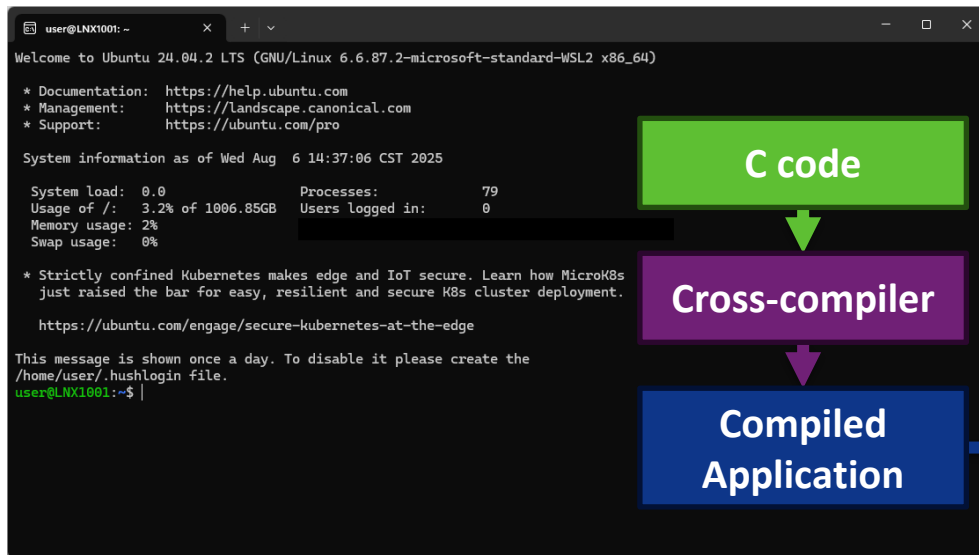
- **evtest** : Used to test input devices like keyboards, mice, and touchscreens. It can display events generated by these devices.

- **GPIO Tools (libgpiod Tools)**

- **gpiodetect** : List all gpiochips present on the system, their names, labels, and number of GPIO lines.
- **gpioinfo** : List all lines of specified gpiochips, their names, consumers, direction, active state, and additional flags.
- **gpioget** : Read values of specified GPIO lines.
- **gpioset** : Set values of specified GPIO lines.
- **gpiofind** : Find the gpiochip name and line offset given the line name.
- **gpiomon** : Wait for events on GPIO lines.

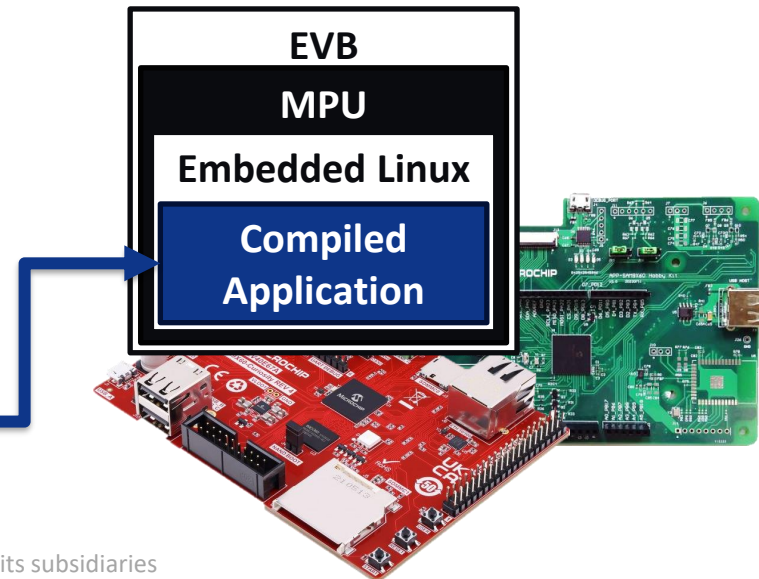
C Language and Cross-Compiler

- C language is the most commonly used programming language for developing embedded Linux systems, the majority of Linux system components are coded in C.
- In Embedded Linux, we develop code on a PC (host) and compile it for a different target architecture (e.g., ARM) using a cross-compiler.
- Cross-compilers are compilers that generate executable code for a platform different from the one on which the compiler is running, we can use this tool to generate applications suitable for different platforms.



```
user@LNX1001: ~  
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)  
* Documentation:  https://help.ubuntu.com  
* Management:    https://landscape.canonical.com  
* Support:        https://ubuntu.com/pro  
  
System information as of Wed Aug 6 14:37:06 CST 2025  
  
System load:  0.0      Processes:      79  
Usage of /:   3.2% of 1006.85GB  Users logged in:  0  
Memory usage: 2%  
Swap usage:  0%  
  
* Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s  
just raised the bar for easy, resilient and secure K8s cluster deployment.  
  
https://ubuntu.com/engage/secure-kubernetes-at-the-edge  
  
This message is shown once a day. To disable it please create the  
/home/user/.hushlogin file.  
user@LNX1001:~$
```

The terminal window is overlaid with a flow diagram showing the compilation process: **C code** (green box) → **Cross-compiler** (purple box) → **Compiled Application** (blue box). A blue arrow points from the 'Compiled Application' box to the EVB hardware diagram on the right.



Lab4 – Use Tools and Application to Detect Button Events

Lab4 – Use Tools and Application to Detect Button Events

- In the Embedded Linux provided from the course, the default will be to config the User Button (GPIO-Key) in the Device-Tree, and enable the corresponding device driver. It will list them in `/dev/input/eventx`, we can use a tool or application to access events for different keys, and check the key is pressed or released.
- Follow the steps below to complete the above requirements :
 1. Try using the Tools for Event-Test to access the events of User Button.
 2. Create a C language application and compile it.
 3. Execute the application and show the events message of User Button.

Let's go!

Lab4 – Use Tools and Application to Detect Button Events

Step 1.1

- Try using the Tools for **Event-Test** to access the events of GPIO-Key (User Button).

```

Tera Term
File Edit Setup Control Window KanjiCode Help

# evtest /dev/input/event0
No device specified, trying to scan all of
/dev/input/event*
Available devices:
/dev/input/event0:      gpio-keys
Select the device event number [0-0]: 0

```

Key in "0"

EVB

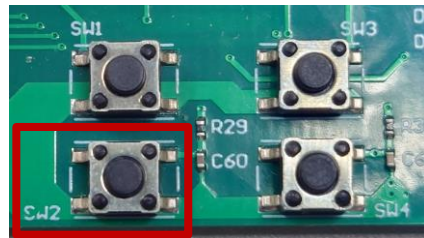
Lab4 – Use Tools and Application to detect Button Events

Step 1.2

- Pressing or Releasing the User-Button will see different events.



Curiosity Board



Hobby Kit

```
user@VM: ~  
# evtest  
No device specified, trying to scan all of /dev/input/event*  
Available devices:  
/dev/input/event0:      gpio-keys  
Select the device event number [0-0]: 0  
Input driver version is 1.0.1  
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100  
Input device name: "gpio-keys"  
Supported events:  
  Event type 0 (EV_SYN)  
  Event type 1 (EV_KEY)  
    Event code 148 (KEY_PROG1)  
Properties:  
Testing ... (interrupt to exit)  
Event: time 1706136438.383577, type 1 (EV_KEY), code 148 (KEY_PROG1), value 1  
Event: time 1706136438.383577, ----- SYN_REPORT -----  
Event: time 1706136438.489493, type 1 (EV_KEY), code 148 (KEY_PROG1), value 0  
Event: time 1706136438.489493, ----- SYN_REPORT -----  
Event: time 1706136439.271354, type 1 (EV_KEY), code 148 (KEY_PROG1), value 1  
Event: time 1706136439.271354, ----- SYN_REPORT -----  
Event: time 1706136439.348726, type 1 (EV_KEY), code 148 (KEY_PROG1), value 0  
Event: time 1706136439.348726, ----- SYN_REPORT -----
```

Key in "0"

Press Button

Release Button

Press Button

Release Button

Lab4 – Use Tools and Application to Detect Button Events

Step 2.1

- **Create a C language application** for Embedded Linux OS.
(Please copy the code from the CopyPaste files to the Text Editor !!)

The image shows a virtual machine (VM) environment with a terminal window and a VS Code editor. The terminal window, labeled '1', shows the following commands being executed:

```
user@VM:~$ mkdir -p ~/Linux4MCHP/Projects
user@VM:~$ cd ~/Linux4MCHP/Projects
user@VM:~$ code button.c
```

The VS Code editor, labeled '2', shows the following C code for `button.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/input.h>

#define DEV "/dev/input/event0"

int main(int argc, char *argv[])
{
    int fd, ret;
    struct input_event event;

    fd = open(DEV, O_RDONLY);
    if(fd < 0) {
        printf("ERROR: open %s fd=%d\n", DEV, fd);
        return -1;
    }
    ...
}
```

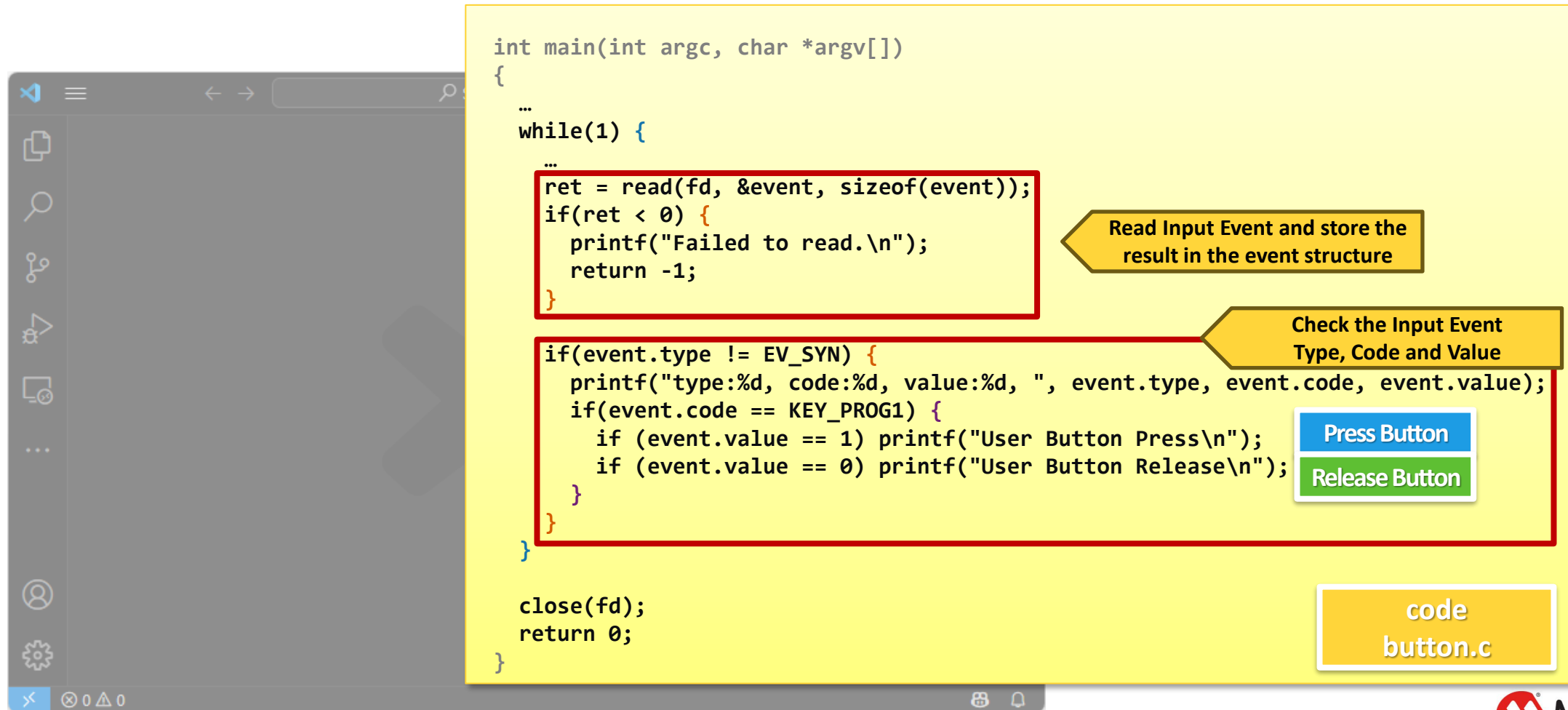
Annotations in the VS Code editor highlight the following parts of the code:

- GNU C Library & Input Event Library**: Points to the include statements for `<stdio.h>`, `<unistd.h>`, `<fcntl.h>`, and `<linux/input.h>`.
- Input Event for Device**: Points to the `#define DEV "/dev/input/event0"` line.
- Declare variables for File Descriptor, Return, and Input Event**: Points to the variable declarations `int fd, ret;` and `struct input_event event;`.
- Get the File Descriptor of the device**: Points to the `fd = open(DEV, O_RDONLY);` line.
- VS Code button.c**: Points to the filename `button.c` in the editor's title bar.

Lab4 – Use Tools and Application to Detect Button Events

Step 2.2

- **Create a C language application** for Embedded Linux OS.



```
int main(int argc, char *argv[])
{
    ...
    while(1) {
        ...
        ret = read(fd, &event, sizeof(event));
        if(ret < 0) {
            printf("Failed to read.\n");
            return -1;
        }

        if(event.type != EV_SYN) {
            printf("type:%d, code:%d, value:%d, ", event.type, event.code, event.value);
            if(event.code == KEY_PROG1) {
                if (event.value == 1) printf("User Button Press\n");
                if (event.value == 0) printf("User Button Release\n");
            }
        }

        close(fd);
        return 0;
    }
}
```

Read Input Event and store the result in the event structure

Check the Input Event Type, Code and Value

Press Button

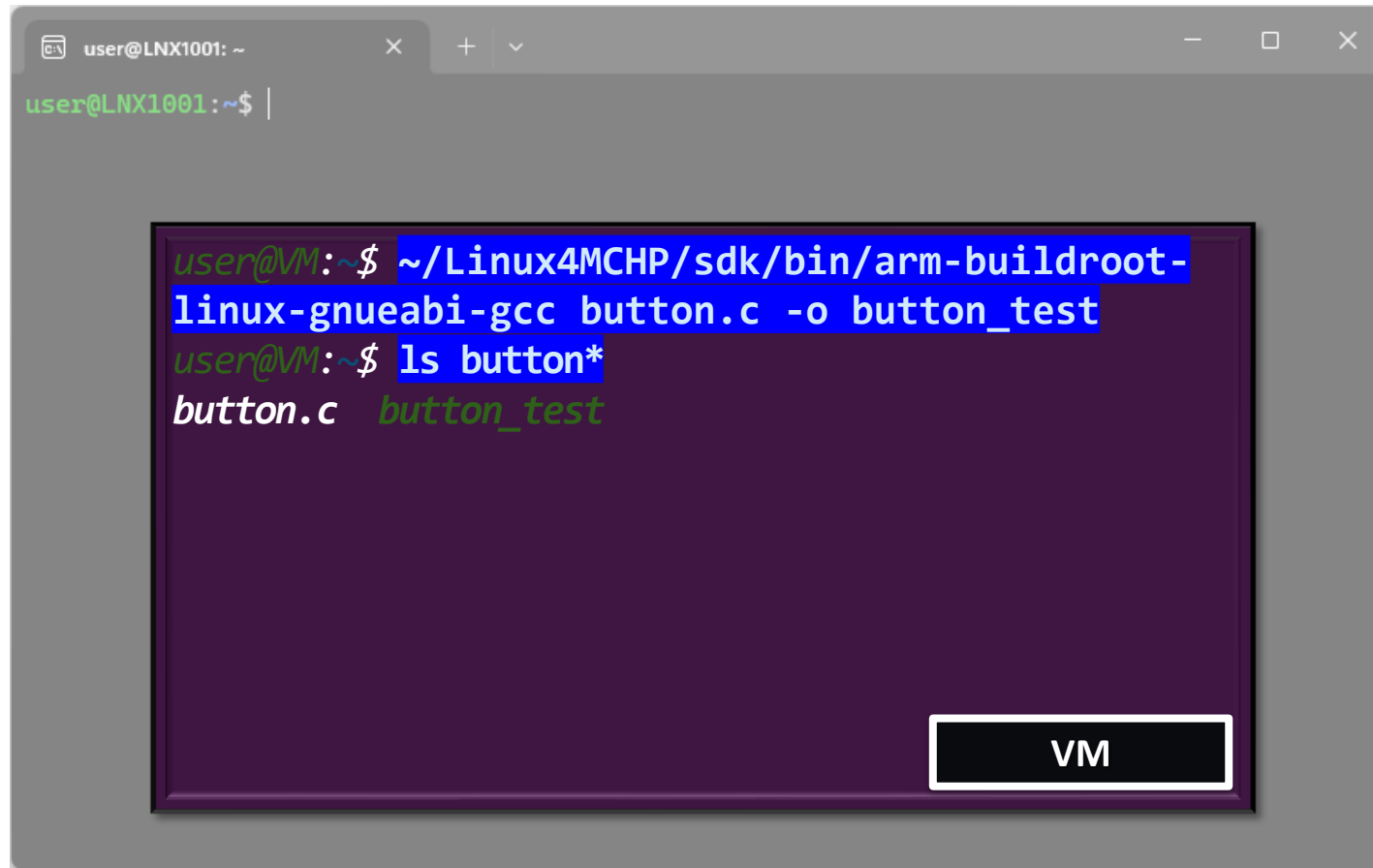
Release Button

code
button.c

Lab4 – Use Tools and Application to Detect Button Events

Step 3

- **Compile the C language application** for Embedded Linux OS.

A terminal window titled 'user@LNX1001: ~' with standard window controls. It contains a nested terminal window titled 'user@VM: ~\$'. The nested terminal shows the compilation of 'button.c' using 'arm-buildroot-linux-gnueabi-gcc' and the subsequent 'ls' command showing 'button.c' and 'button_test'. A 'VM' button is at the bottom right of the nested terminal.

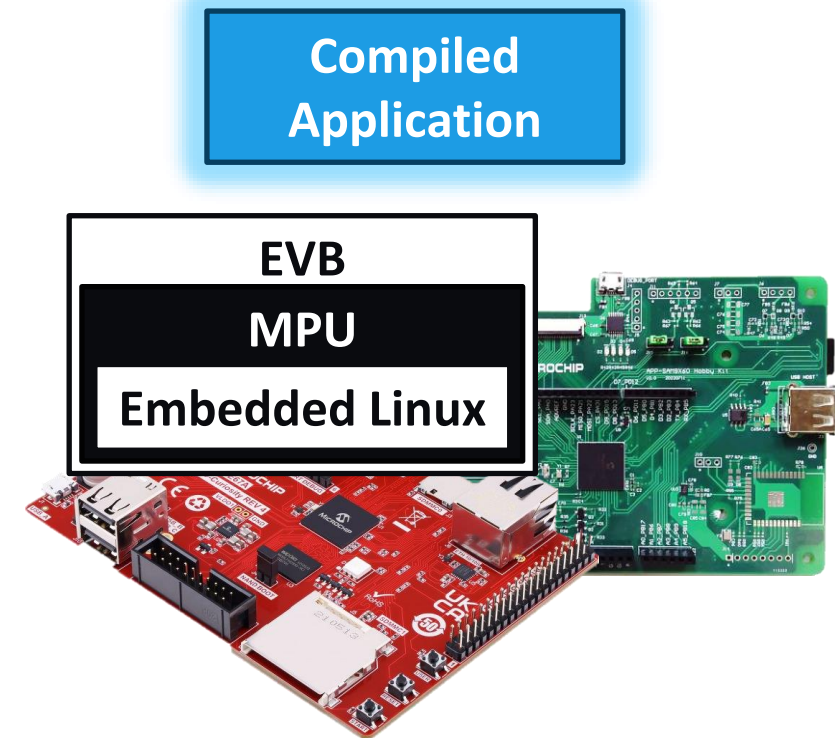
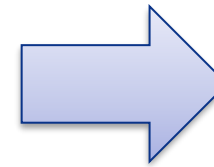
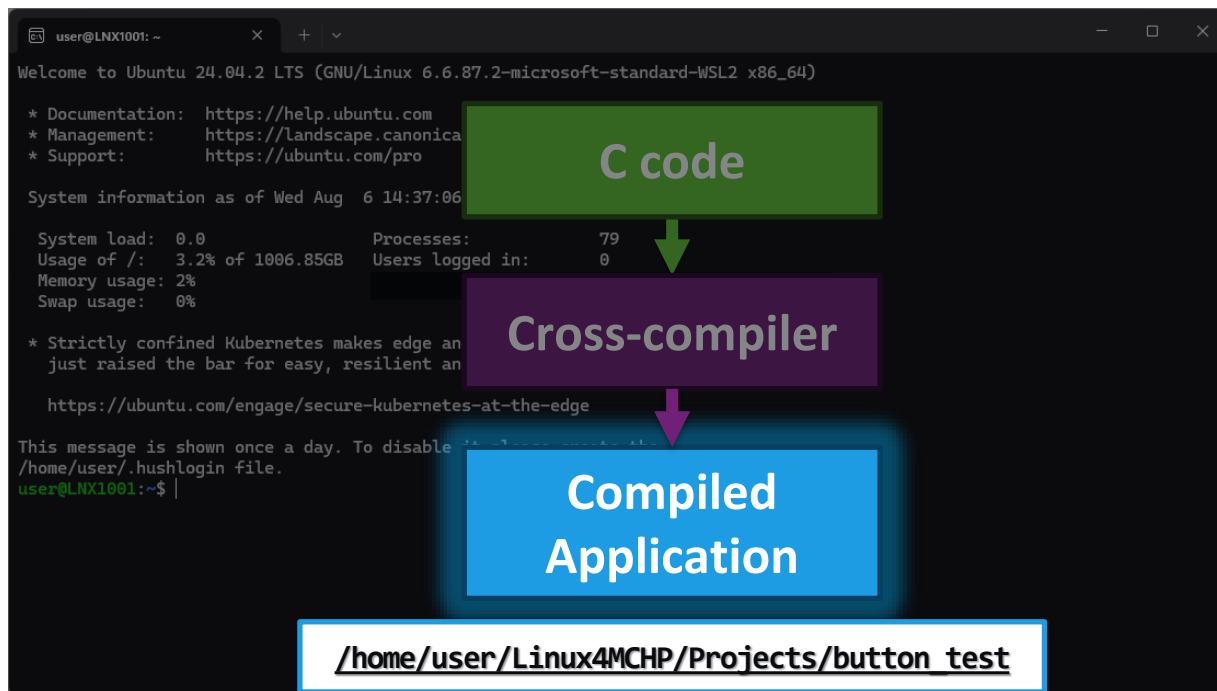
```
user@LNX1001: ~  
user@LNX1001:~$  
  
user@VM:~$ ~/Linux4MCHP/sdk/bin/arm-buildroot-  
linux-gnueabi-gcc button.c -o button_test  
user@VM:~$ ls button*  
button.c  button_test
```

VM

Lab4 – Use Tools and Application to Detect Button Events

Step 4

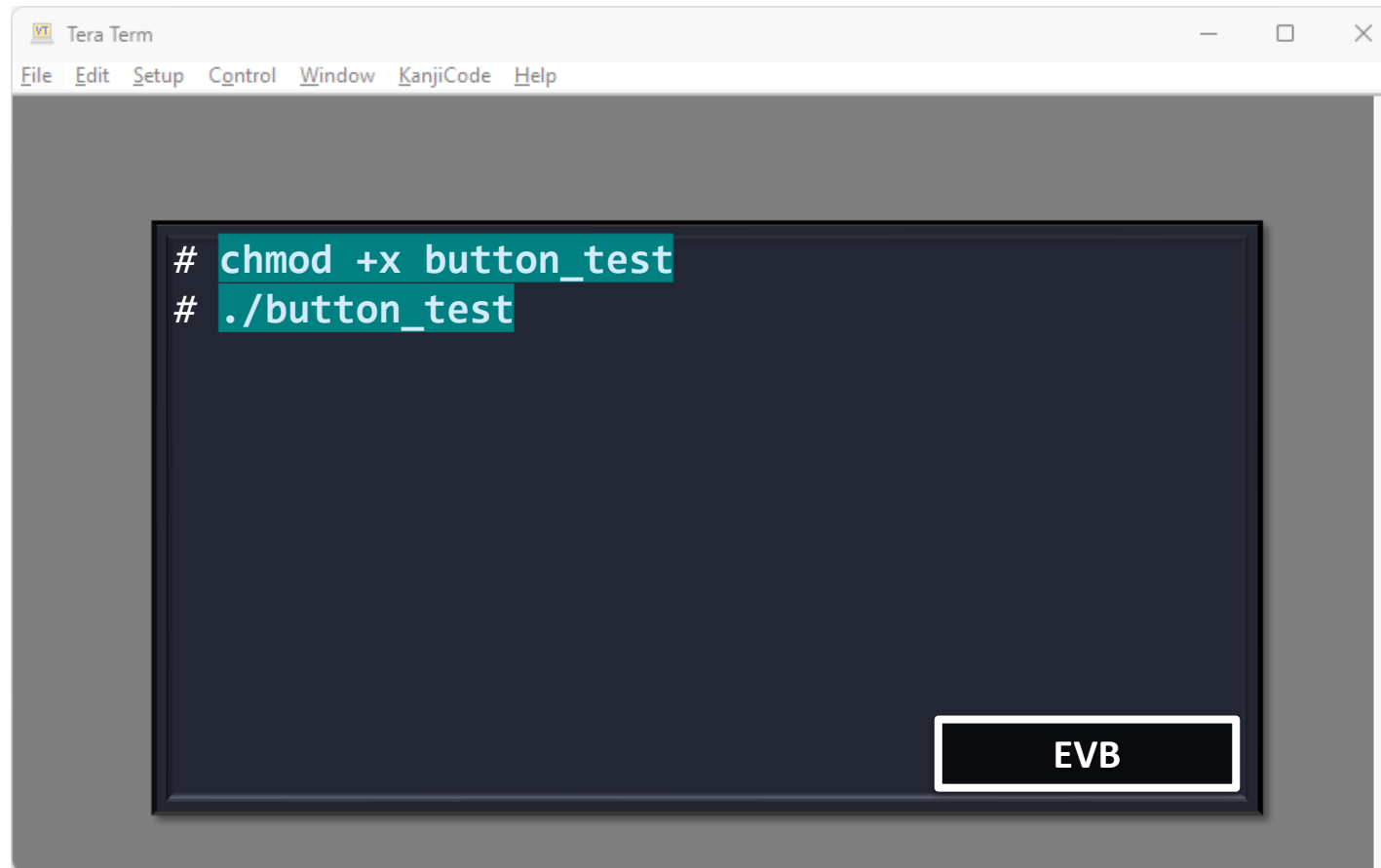
- Send the application to Embedded Linux OS.
(Refer to the appendix to send files to Embedded Linux System)



Lab4 – Use Tools and Application to Detect Button Events

Step 5

- **Execute the application** for Embedded Linux OS.

A screenshot of a Tera Term terminal window. The window has a title bar with 'Tera Term' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Setup', 'Control', 'Window', 'KanjiCode', and 'Help'. The main area is a dark terminal with two lines of text: '# chmod +x button_test' and '# ./button_test', both highlighted in teal. In the bottom right corner of the terminal area, there is a white rectangular button with the text 'EVB' in black.

```
# chmod +x button_test
# ./button_test
```

EVB

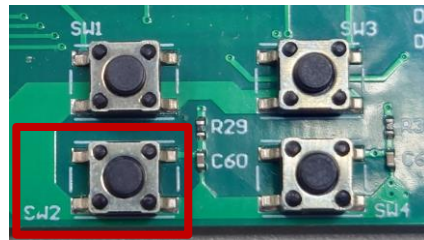
Lab4 – Use Tools and Application to Detect Button Events

Result

- Pressing or Releasing the User-Button and show the events message of User-Button.



Curiosity Board



Hobby Kit

```
user@VM: ~  
# chmod +x button_test  
# ./button_test  
type:1, code:148, value:1, User Button Press  
type:1, code:148, value:0, User Button Release  
type:1, code:148, value:1, User Button Press  
type:1, code:148, value:0, User Button Release
```

Lab5 – Use Tools and C Language to Control UART

Lab5 – Use Tools and C Language to Control UART

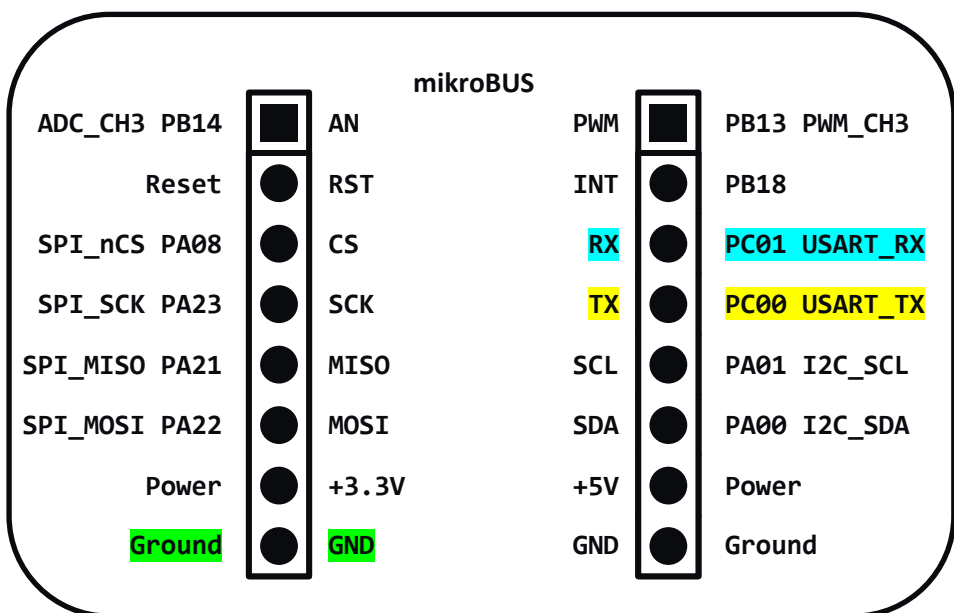
- In the Embedded Linux provided from the course, the default will be to config the UART in the Device-Tree, and enable the corresponding device driver. It will list them in `/dev/ttySx`, we can use a tool or application to access it, and transmit and receive messages from UART.
- Follow the steps below to complete the above requirements :
 1. Connect the UART of **mikroBUS Pin** to the external USB CDC Bridge.
 2. Try using the Tool to access the UART of MikroBUS.
 3. Create a C language application and compile it.
 4. Execute the application and access the UART of MikroBUS.

Let's go!

Lab5 – Use Tools and C Language to Control UART

Preparation

- Connect the UART of mikroBUS **TX Pin (PC00)** and **RX Pin (PC01)** to the External UART to USB CDC Bridge.

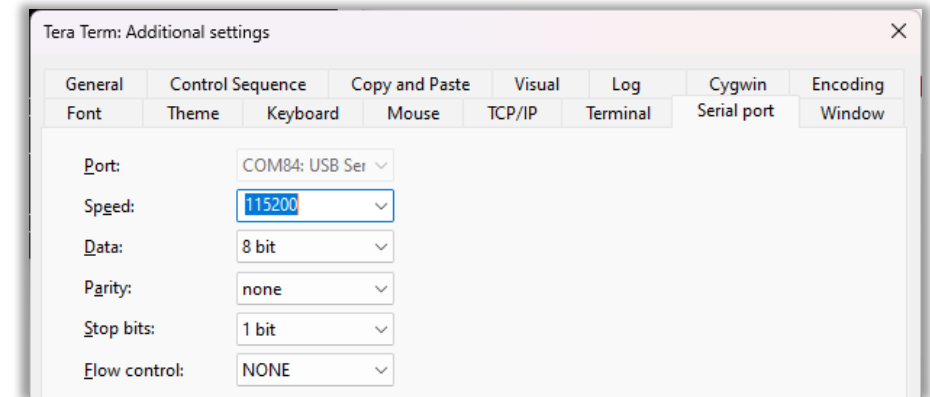
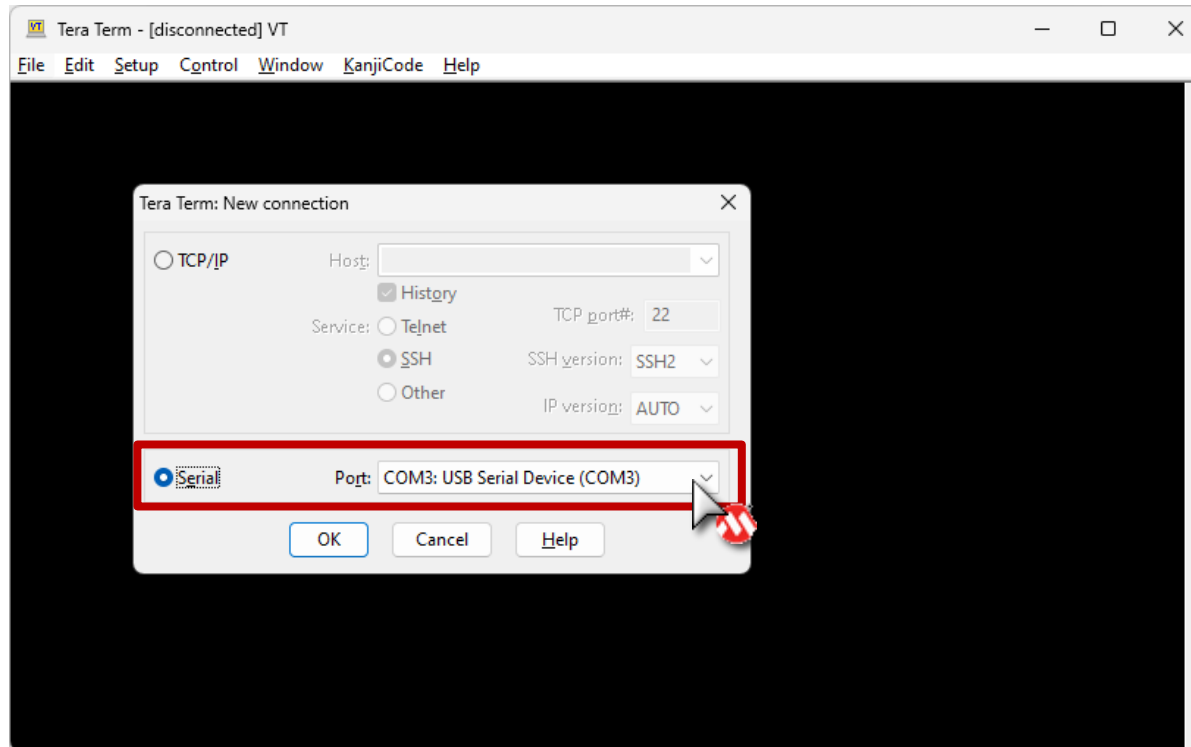


mikroBUS PIN	External
RX	USB CDC Bridge, TX
TX	USB CDC Bridge, RX
GND	GND

Lab5 – Use Tools and C Language to Control UART

Preparation

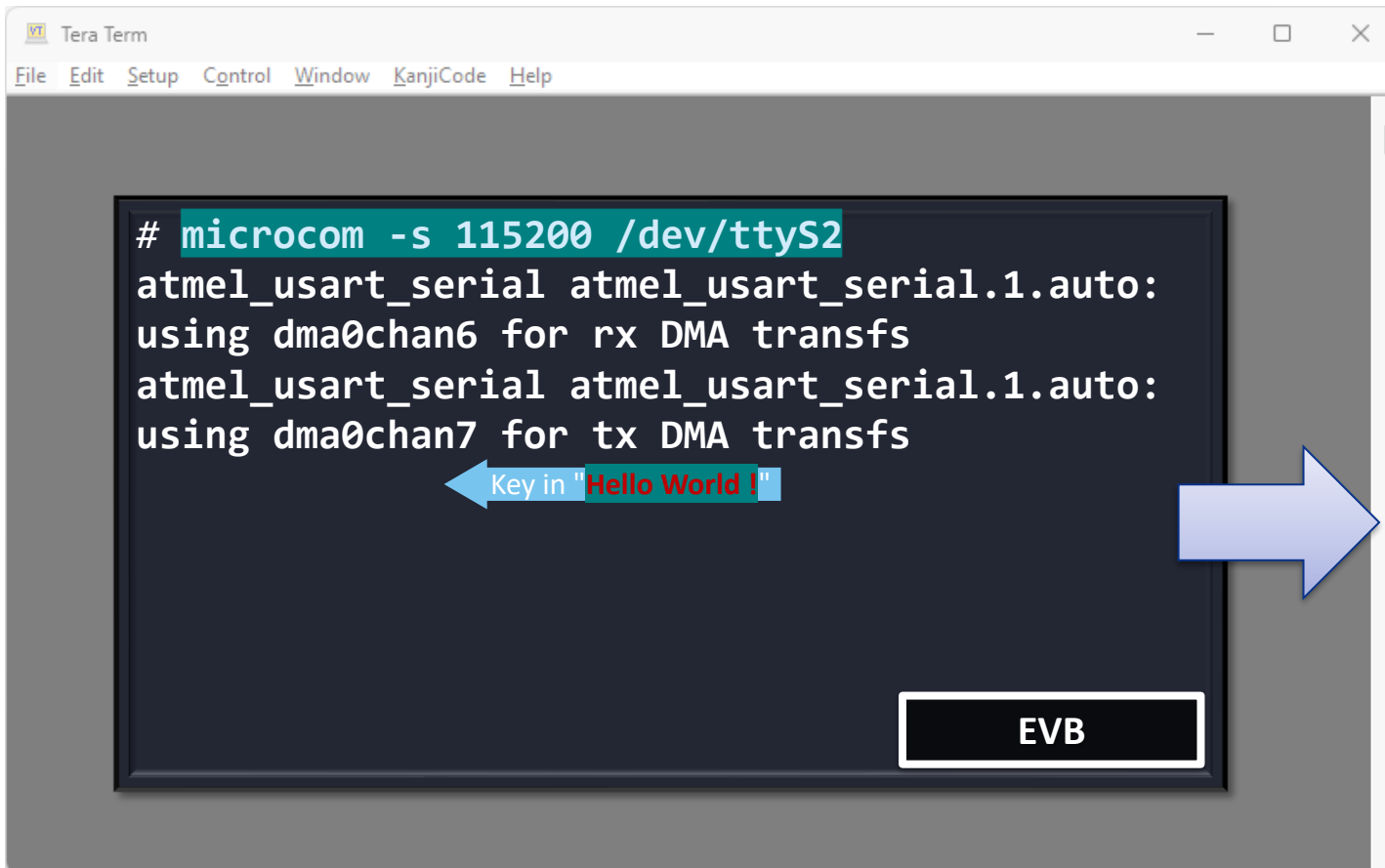
- **Use Tera Term to open the another serial port** of the External UART to USB CDC Bridge. (Select the COM Port that appears after the connection, Baud Rate / Speed : 115200)



Lab5 – Use Tools and C Language to Control UART

Step 1

- Try using the Tools for **Microcom** to access the UART. (CTRL + X to exit Microcom)

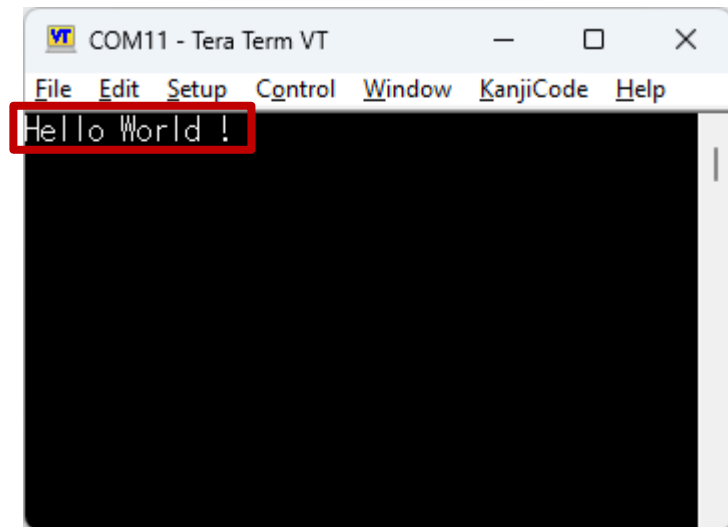


The screenshot shows a Tera Term window with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help). The terminal text is as follows:

```
# microcom -s 115200 /dev/ttyS2
atmel_usart_serial atmel_usart_serial.1.auto:
using dma0chan6 for rx DMA transfs
atmel_usart_serial atmel_usart_serial.1.auto:
using dma0chan7 for tx DMA transfs
```

A blue arrow points from the text "Key in 'Hello World!'" to the terminal. A large blue arrow points from the terminal to the right window.

EVB



The screenshot shows a Tera Term window titled "COM11 - Tera Term VT" with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help). The terminal displays "Hello World!" which is highlighted with a red rectangular box.

Lab5 – Use Tools and C Language to Control UART

Step 2.1

- **Create a C language application** for Embedded Linux OS.
(Please copy the code from the CopyPaste files to the Text Editor !!)

The image shows a Linux Virtual Machine (VM) environment with a terminal window and a VS Code editor. The terminal window, labeled '1', shows the following commands being executed:

```
user@VM:~$ mkdir -p ~/Linux4MCHP/Projects
user@VM:~$ cd ~/Linux4MCHP/Projects
user@VM:~$ code uart.c
```

The VS Code editor, labeled '2', shows the contents of the file `uart.c`. The code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <termios.h>

#define DEV "/dev/ttyS2"

void init_UART(int fd, speed_t baudrate, unsigned char timeout_x100ms) { ... }

int read_UART(int fd, void *ptr_data, unsigned int data_len) { ... }

int write_UART(int fd, void *ptr_data) { ... }

int main(int argc, char *argv[])
{
    ...
}
```

Annotations in the image identify the code sections:

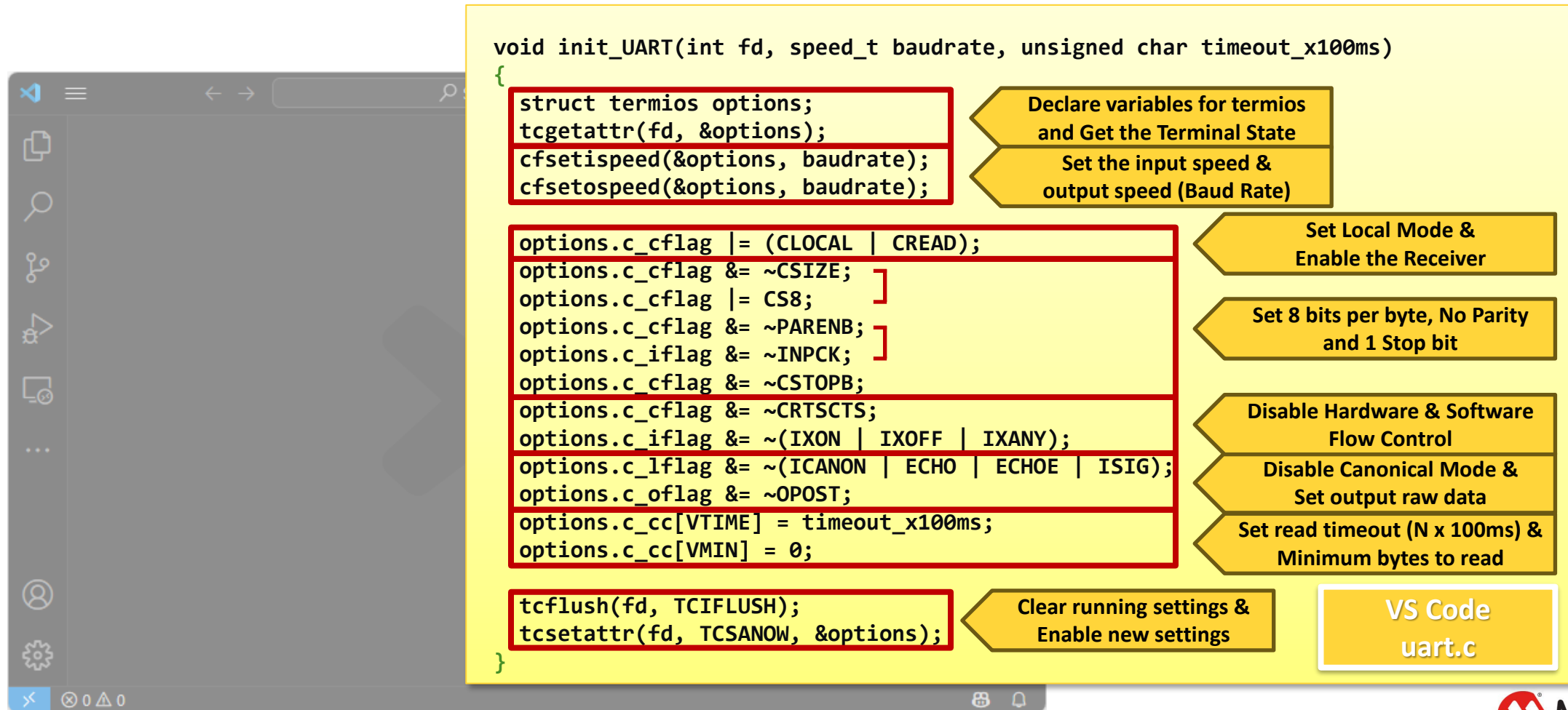
- GNU C Library & Termios Library**: Points to the include statements at the top of the file.
- UART(Serial2) for Device**: Points to the `DEV` macro definition.
- UART functions**: Points to the `init_UART`, `read_UART`, and `write_UART` function declarations.
- VS Code uart.c**: Points to the `main` function.

A label 'VM' is placed at the bottom of the terminal window.

Lab5 – Use Tools and C Language to Control UART

Step 2.2

- **Create a C language application** for Embedded Linux OS.



The image shows a code editor window with the following C code for `init_UART`. The code is annotated with yellow callouts explaining each section:

```
void init_UART(int fd, speed_t baudrate, unsigned char timeout_x100ms)
{
    struct termios options;
    tcgetattr(fd, &options);
    cfsetispeed(&options, baudrate);
    cfsetospeed(&options, baudrate);

    options.c_cflag |= (CLOCAL | CREAD);
    options.c_cflag &= ~CSIZE;
    options.c_cflag |= CS8;
    options.c_cflag &= ~PARENB;
    options.c_iflag &= ~INPCK;
    options.c_cflag &= ~CSTOPB;
    options.c_cflag &= ~CRTSCTS;
    options.c_iflag &= ~(IXON | IXOFF | IXANY);
    options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
    options.c_oflag &= ~OPOST;
    options.c_cc[VTIME] = timeout_x100ms;
    options.c_cc[VMIN] = 0;

    tcflush(fd, TCIFLUSH);
    tcsetattr(fd, TCSANOW, &options);
}
```

Annotations:

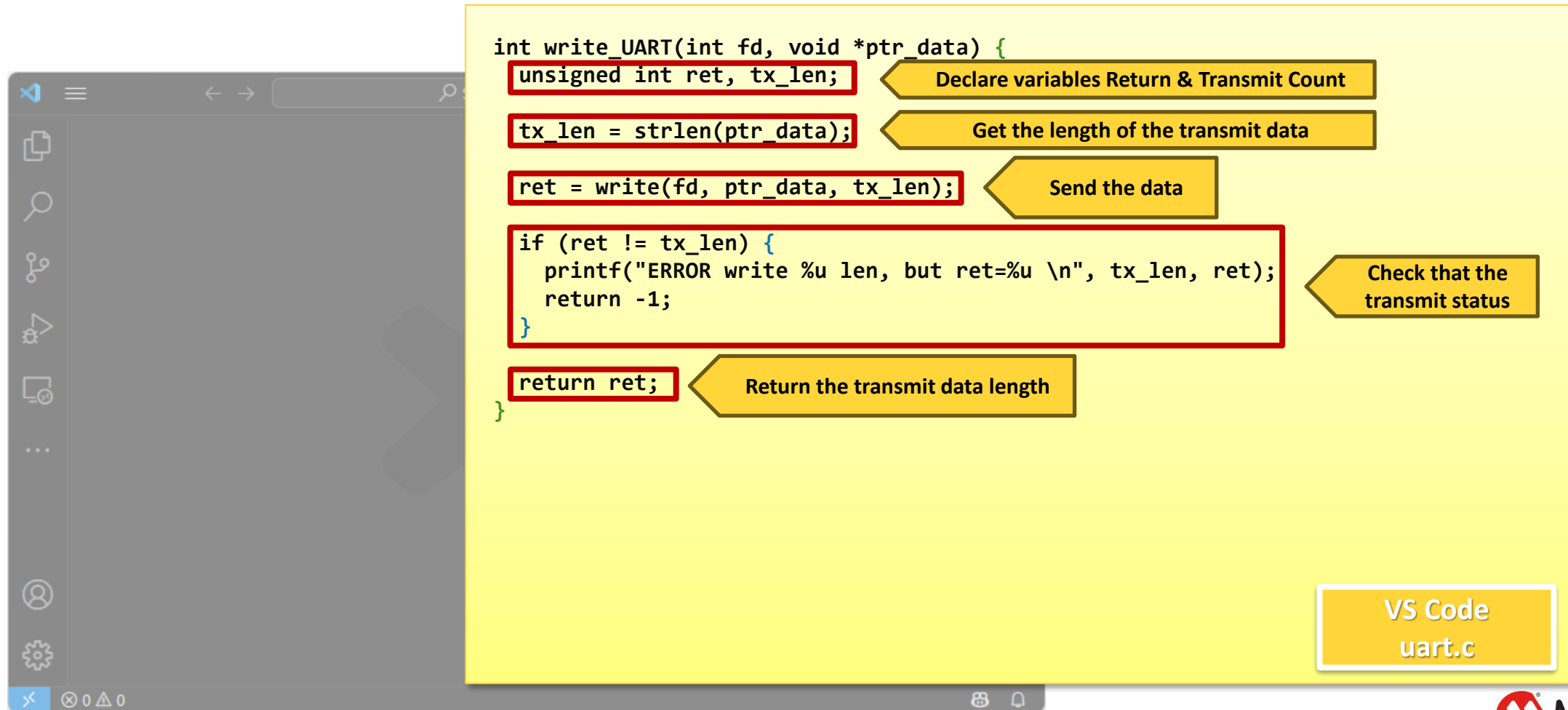
- Declare variables for termios and Get the Terminal State** (points to `struct termios options; tcgetattr(fd, &options);`)
- Set the input speed & output speed (Baud Rate)** (points to `cfsetispeed(&options, baudrate); cfsetospeed(&options, baudrate);`)
- Set Local Mode & Enable the Receiver** (points to `options.c_cflag |= (CLOCAL | CREAD);`)
- Set 8 bits per byte, No Parity and 1 Stop bit** (points to `options.c_cflag &= ~CSIZE; options.c_cflag |= CS8;`)
- Disable Hardware & Software Flow Control** (points to `options.c_cflag &= ~PARENB; options.c_iflag &= ~INPCK;`)
- Disable Canonical Mode & Set output raw data** (points to `options.c_cflag &= ~CSTOPB; options.c_cflag &= ~CRTSCTS;`)
- Set read timeout (N x 100ms) & Minimum bytes to read** (points to `options.c_iflag &= ~(IXON | IXOFF | IXANY); options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); options.c_oflag &= ~OPOST;`)
- Clear running settings & Enable new settings** (points to `options.c_cc[VTIME] = timeout_x100ms; options.c_cc[VMIN] = 0; tcflush(fd, TCIFLUSH); tcsetattr(fd, TCSANOW, &options);`)

VS Code uart.c

Lab5 – Use Tools and C Language to Control UART

Step 2.3

- **Create a C language application** for Embedded Linux OS.



The image shows a screenshot of the Visual Studio Code (VS Code) editor interface. On the left, the sidebar displays icons for Explorer, Search, Source Control, Run and Debug, and Settings. The main editor area shows a C program for UART control. The code is as follows:

```
int write_UART(int fd, void *ptr_data) {  
    unsigned int ret, tx_len;  
    tx_len = strlen(ptr_data);  
    ret = write(fd, ptr_data, tx_len);  
    if (ret != tx_len) {  
        printf("ERROR write %u len, but ret=%u \n", tx_len, ret);  
        return -1;  
    }  
    return ret;  
}
```

Annotations are provided for each line of code:

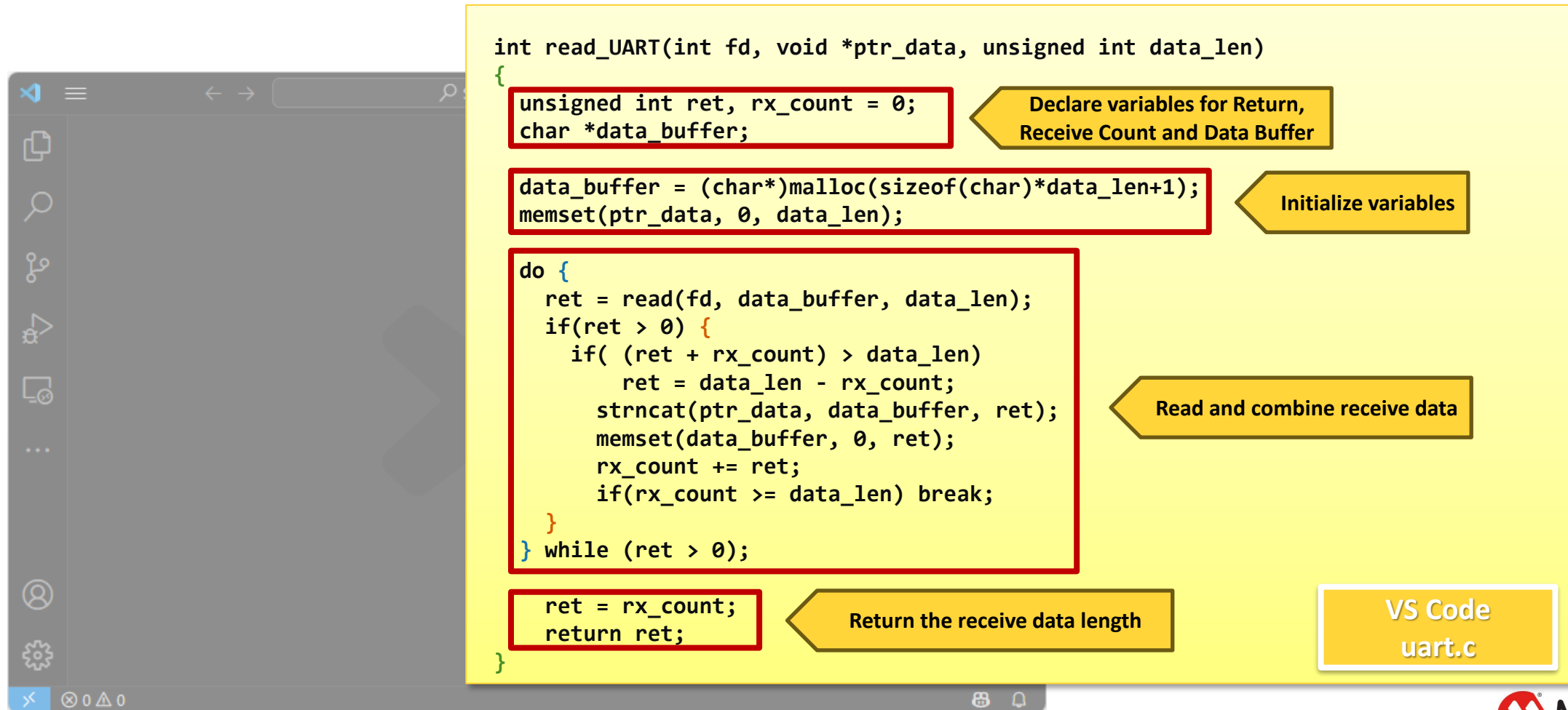
- Declare variables Return & Transmit Count** (points to `unsigned int ret, tx_len;`)
- Get the length of the transmit data** (points to `tx_len = strlen(ptr_data);`)
- Send the data** (points to `ret = write(fd, ptr_data, tx_len);`)
- Check that the transmit status** (points to the `if (ret != tx_len) { ... }` block)
- Return the transmit data length** (points to `return ret;`)

A yellow box in the bottom right corner of the editor area contains the text: **VS Code** and **uart.c**.

Lab5 – Use Tools and C Language to Control UART

Step 2.4

- **Create a C language application** for Embedded Linux OS.



```
int read_UART(int fd, void *ptr_data, unsigned int data_len)
{
    unsigned int ret, rx_count = 0;
    char *data_buffer;

    data_buffer = (char*)malloc(sizeof(char)*data_len+1);
    memset(ptr_data, 0, data_len);

    do {
        ret = read(fd, data_buffer, data_len);
        if(ret > 0) {
            if( (ret + rx_count) > data_len)
                ret = data_len - rx_count;
            strncat(ptr_data, data_buffer, ret);
            memset(data_buffer, 0, ret);
            rx_count += ret;
            if(rx_count >= data_len) break;
        }
    } while (ret > 0);

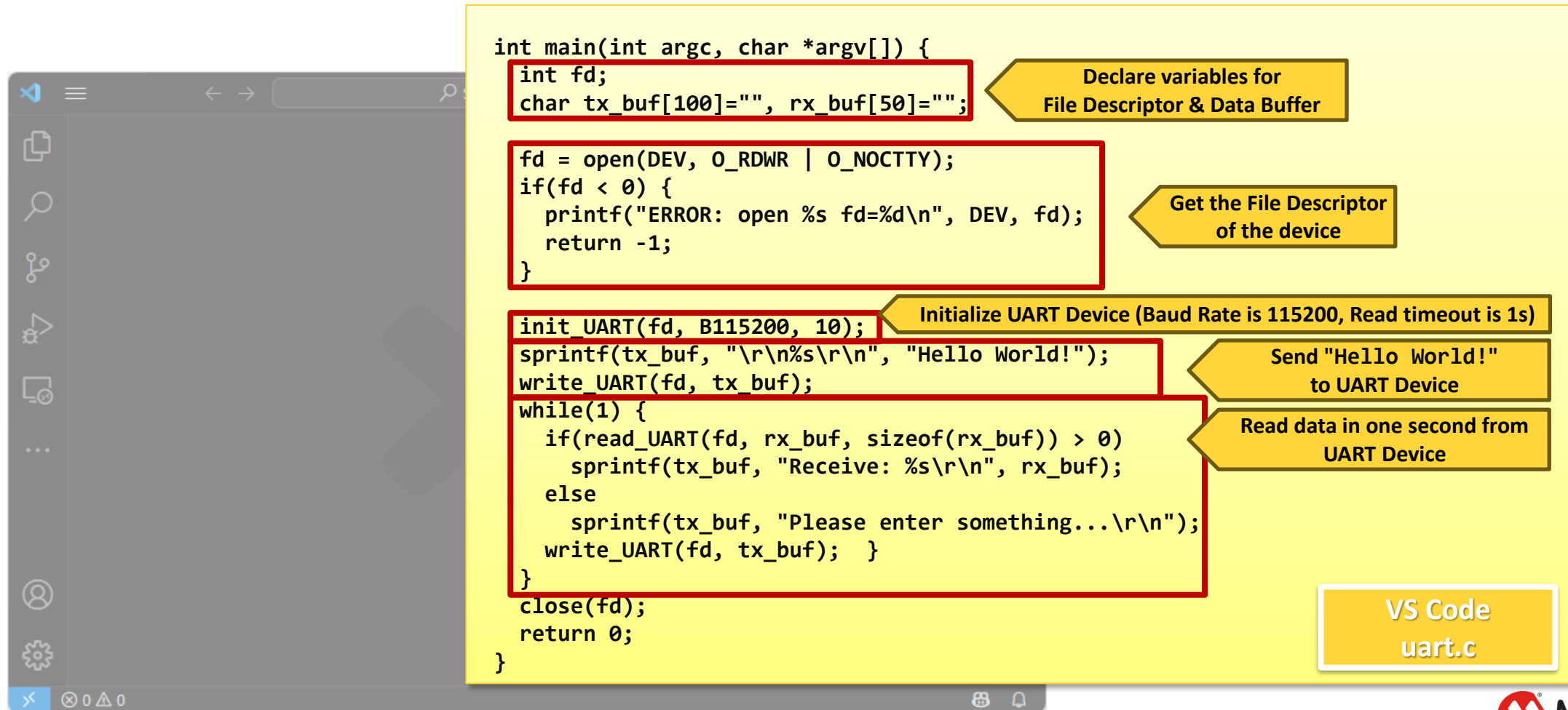
    ret = rx_count;
    return ret;
}
```

VS Code
uart.c

Lab5 – Use Tools and C Language to Control UART

Step 2.5

- **Create a C language application** for Embedded Linux OS.



The image shows a VS Code editor window with a C program for UART control. The code is annotated with yellow callouts explaining key parts of the program.

```
int main(int argc, char *argv[]) {  
    int fd;  
    char tx_buf[100]="", rx_buf[50]="";  
  
    fd = open(DEV, O_RDWR | O_NOCTTY);  
    if(fd < 0) {  
        printf("ERROR: open %s fd=%d\n", DEV, fd);  
        return -1;  
    }  
  
    init_UART(fd, B115200, 10);  
    sprintf(tx_buf, "\r\n%s\r\n", "Hello World!");  
    write_UART(fd, tx_buf);  
    while(1) {  
        if(read_UART(fd, rx_buf, sizeof(rx_buf)) > 0)  
            sprintf(tx_buf, "Receive: %s\r\n", rx_buf);  
        else  
            sprintf(tx_buf, "Please enter something...\r\n");  
        write_UART(fd, tx_buf);  
    }  
    close(fd);  
    return 0;  
}
```

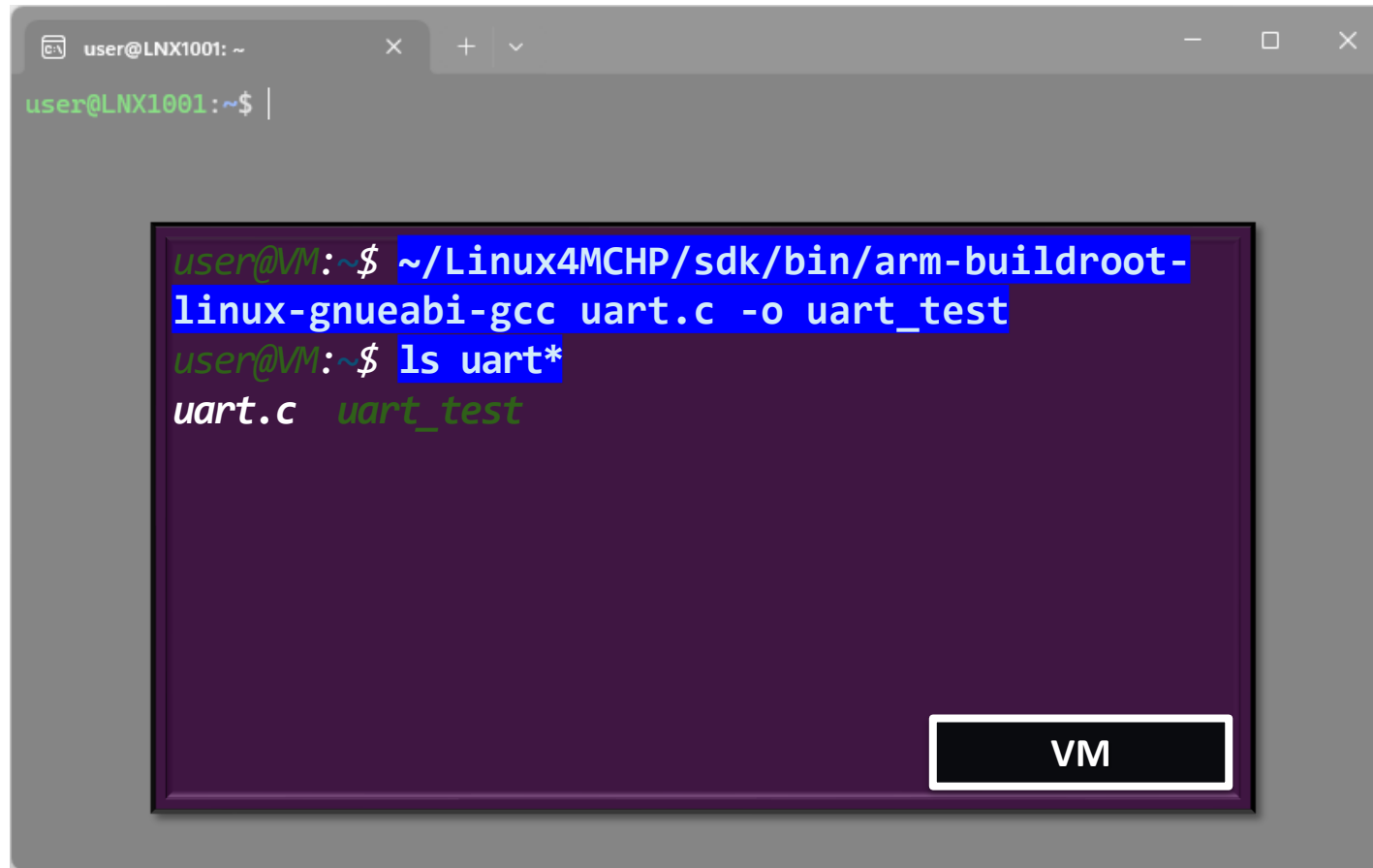
Annotations:

- Declare variables for File Descriptor & Data Buffer**: Points to the declaration of `int fd;` and `char tx_buf[100]="", rx_buf[50]="";`.
- Get the File Descriptor of the device**: Points to the `fd = open(DEV, O_RDWR | O_NOCTTY);` line.
- Initialize UART Device (Baud Rate is 115200, Read timeout is 1s)**: Points to the `init_UART(fd, B115200, 10);` line.
- Send "Hello World!" to UART Device**: Points to the `sprintf(tx_buf, "\r\n%s\r\n", "Hello World!");` and `write_UART(fd, tx_buf);` lines.
- Read data in one second from UART Device**: Points to the `while(1)` loop.
- VS Code uart.c**: A button in the bottom right corner of the code editor.

Lab5 – Use Tools and C Language to Control UART

Step 3

- **Compile the C language application** for Embedded Linux OS.



The screenshot shows a terminal window with a title bar that reads "user@LNX1001: ~". The terminal prompt is "user@LNX1001:~\$". A smaller, semi-transparent terminal window is overlaid on top, showing the following commands and output:

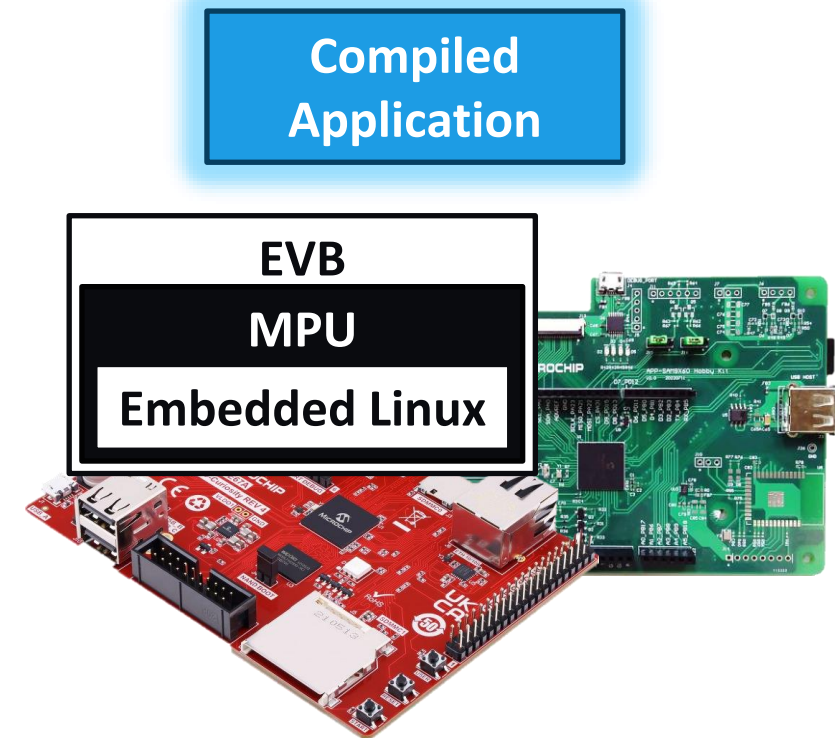
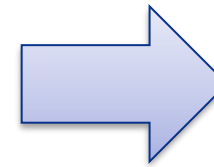
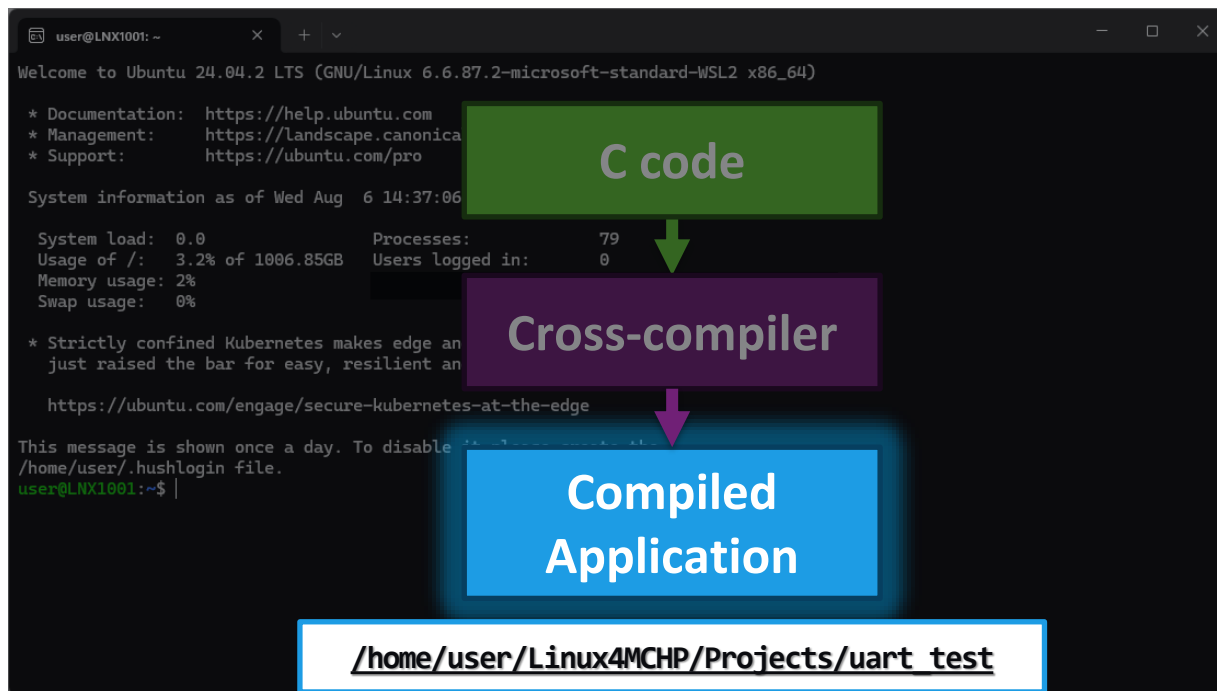
```
user@VM:~$ ~/Linux4MCHP/sdk/bin/arm-buildroot-  
linux-gnueabi-gcc uart.c -o uart_test  
user@VM:~$ ls uart*  
uart.c  uart_test
```

A button labeled "VM" is located in the bottom right corner of the overlaid terminal window.

Lab5 – Use Tools and C Language to Control UART

Step 4

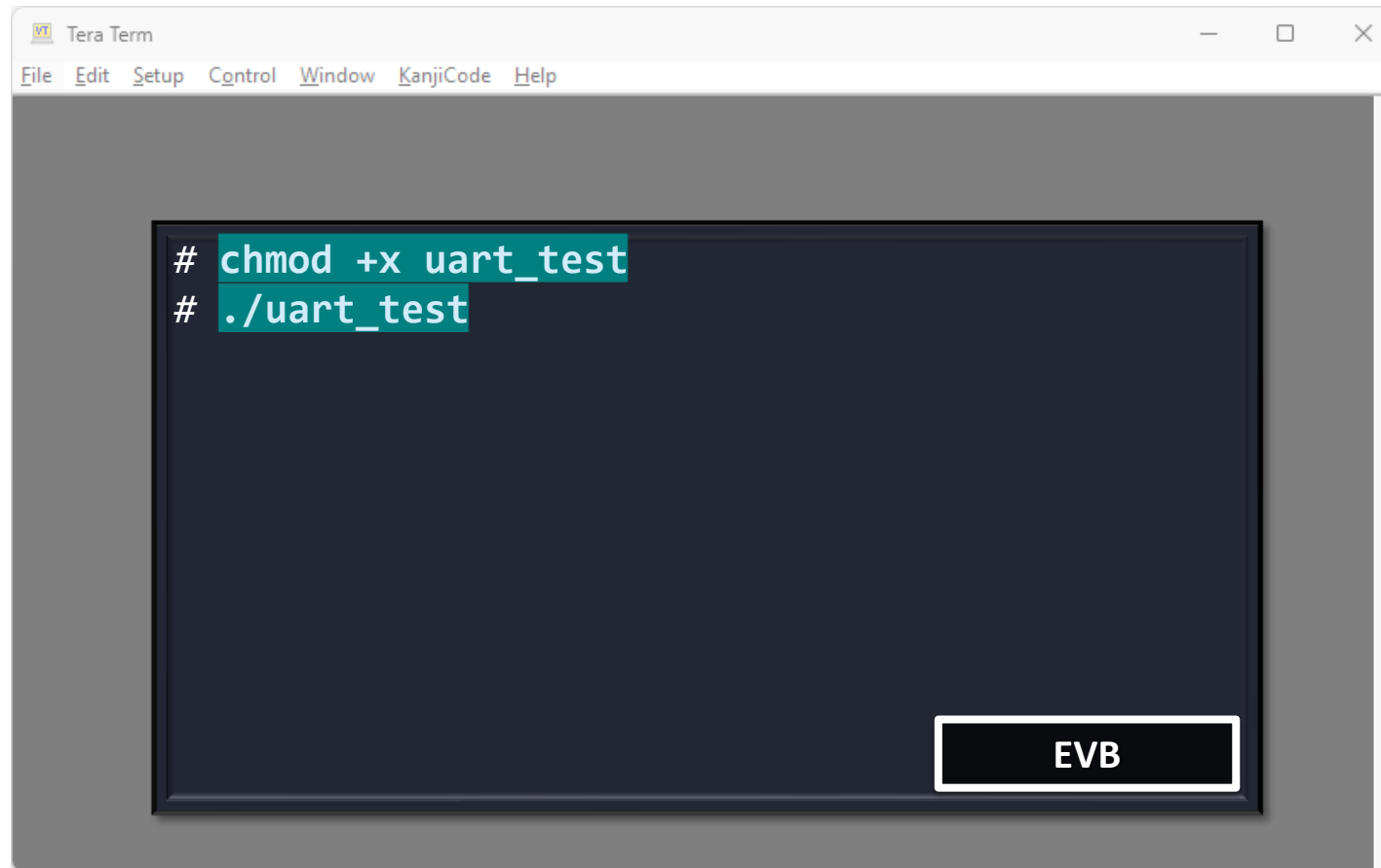
- Send the application to Embedded Linux OS.
(Refer to the appendix to send files to Embedded Linux System)



Lab5 – Use Tools and C Language to Control UART

Step 5

- **Execute the application** for Embedded Linux OS.



The screenshot shows a Tera Term terminal window. The title bar reads "Tera Term". The menu bar includes "File", "Edit", "Setup", "Control", "Window", "KanjiCode", and "Help". The terminal area has a dark background and displays two lines of text: "# chmod +x uart_test" and "# ./uart_test", both highlighted in green. A white rectangular button with the text "EVB" is located in the bottom right corner of the terminal area.

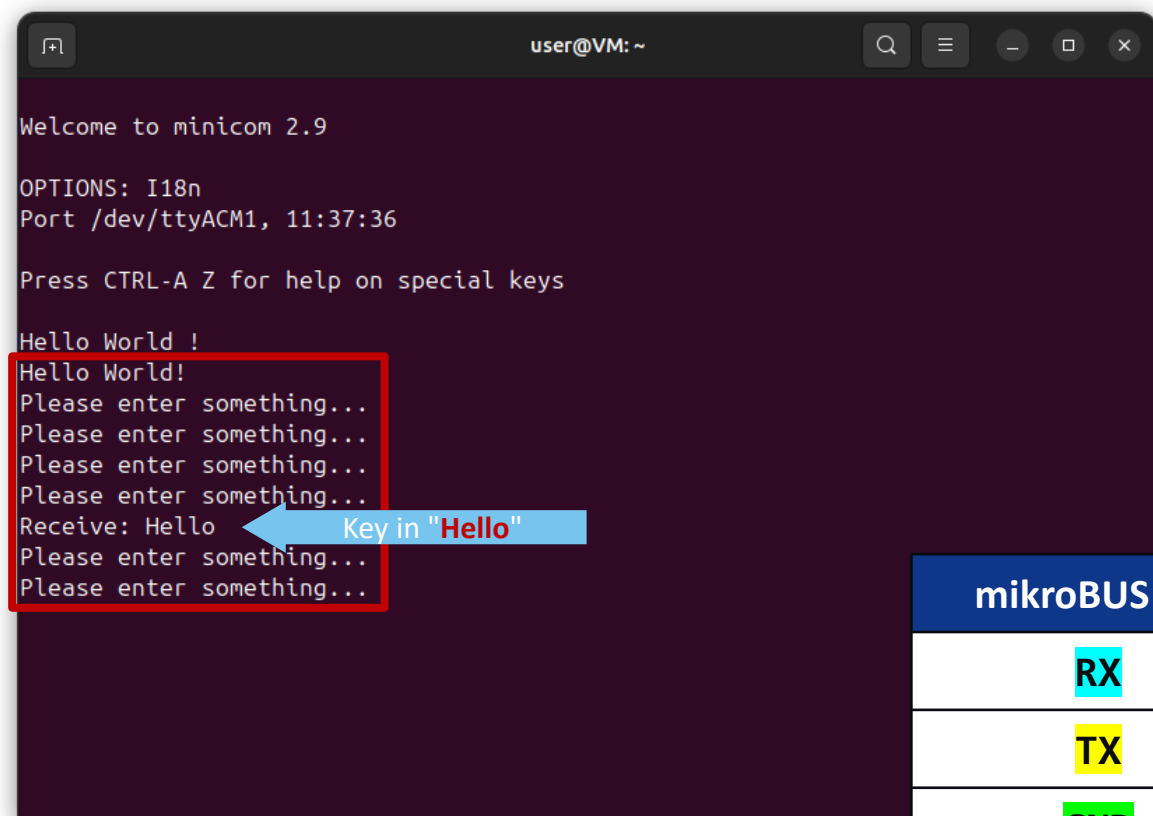
```
# chmod +x uart_test
# ./uart_test
```

EVB

Lab5 – Use Tools and C Language to Control UART

Result

- The application will continuously send messages to the minicom (/dev/ttyACM1) and return the messages Key in by the user.



```
user@VM: ~
Welcome to minicom 2.9

OPTIONS: I18n
Port /dev/ttyACM1, 11:37:36

Press CTRL-A Z for help on special keys

Hello World !
Hello World!
Please enter something...
Please enter something...
Please enter something...
Please enter something...
Receive: Hello
Please enter something...
Please enter something...
```

mikroBUS PIN	External
RX	USB CDC Bridge, TX
TX	USB CDC Bridge, RX
GND	GND

Lab6 – Use Tools and C Language to Control I²C

Lab6 – Use Tools and C Language to Control I²C

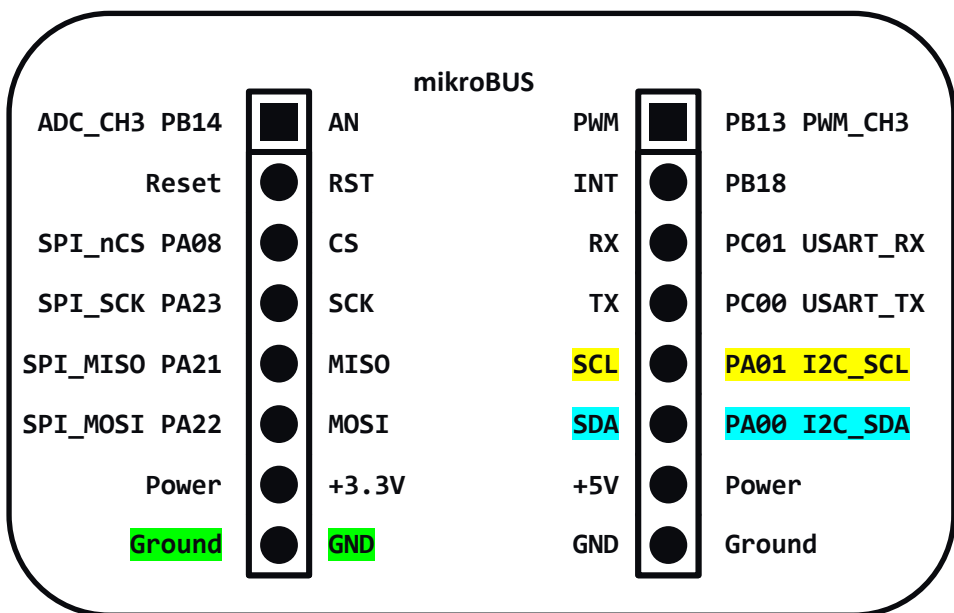
- In the Embedded Linux provided from the course, the default will be to config the I²C in the Device-Tree, and enable the corresponding device driver. It will list them in [/dev/i2c-x](#), we can use a tool or application to access it, and transmit and receive messages from I²C.
- Follow the steps below to complete the above requirements :
 1. Connect the I²C of **mikroBUS Pin** to the external I²C Device (MCP9800-A5).
 2. Try using the Tools to access the I²C of MikroBUS.
 3. Create a C language application and compile it.
 4. Execute the application and access the I²C of MikroBUS.

Let's go!

Lab6 – Use Tools and C Language to Control I²C

Preparation

- Connect the I²C of mikroBUS **SCL Pin (PA01)** and **SDA Pin (PA00)** to the External I²C Device.

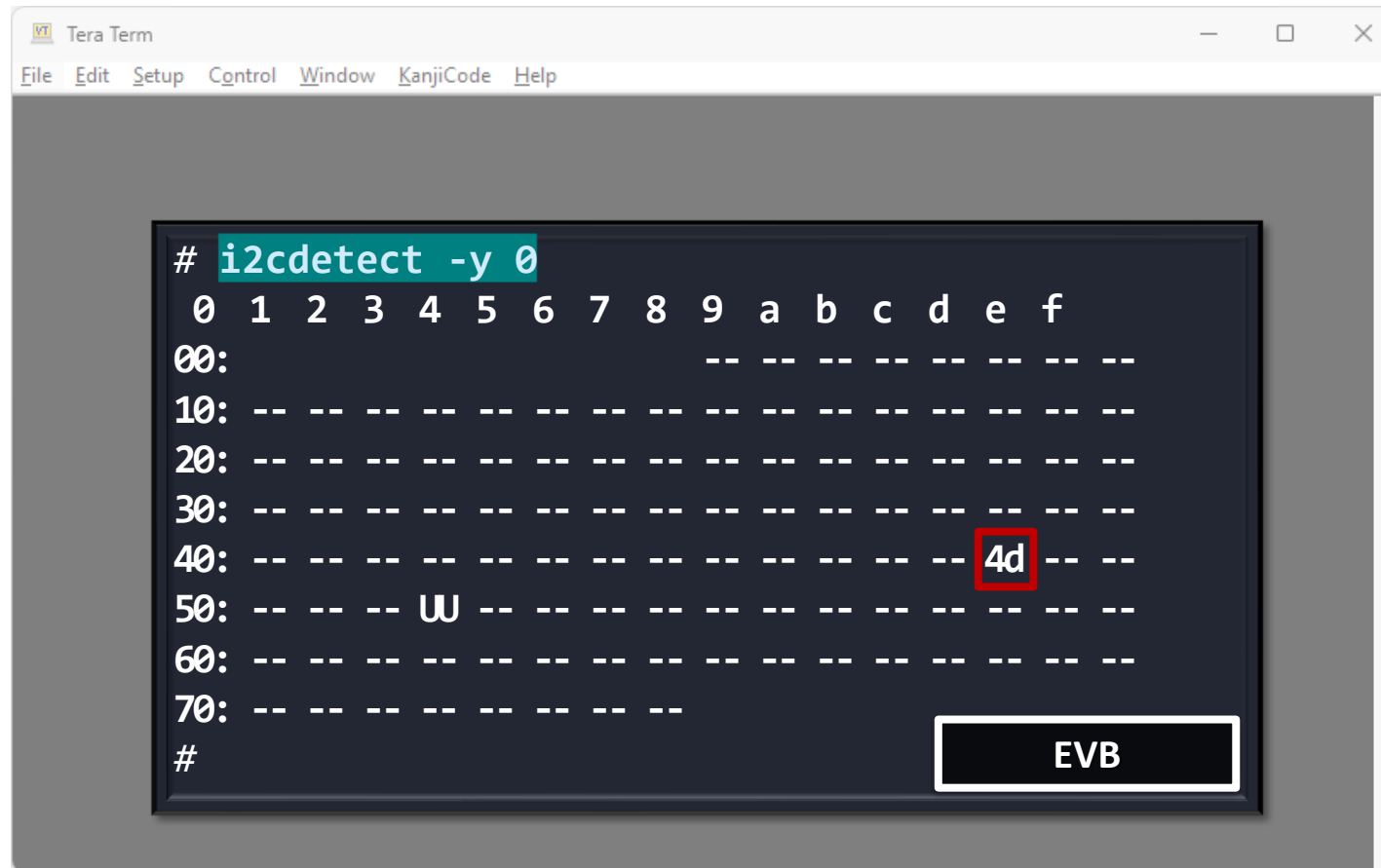


mikroBUS PIN	External
SCL	I ² C Temp. (MCP9800-A5), SCL
SDA	I ² C Temp. (MCP9800-A5), SDA
GND	GND

Lab6 – Use Tools and C Language to Control I²C

Step 1

- Try using the Tools for **I2C Tools** to access the I²C.



The screenshot shows a Tera Term terminal window with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help) and a dark-themed command prompt. The command `# i2cdetect -y 0` has been entered. Below the command, a grid of I2C addresses (00 to 7F) is displayed. The address 4D is highlighted with a red box. The address 50 contains the character 'U'. A button labeled 'EVB' is visible in the bottom right corner of the terminal window.

```
# i2cdetect -y 0
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                                     -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- 4d -- -- --
50: -- -- -- U -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
#
```

EVB

Lab6 – Use Tools and C Language to Control I²C

Step 2.1

- **Create a C language application** for Embedded Linux OS.
(Please copy the code from the CopyPaste files to the Text Editor !!)

The image shows a virtual machine (VM) environment with a terminal window and a VS Code editor. The terminal window, labeled '1', shows the following commands being executed:

```
user@VM:~$ mkdir -p ~/Linux4MCHP/Projects
user@VM:~$ cd ~/Linux4MCHP/Projects
user@VM:~$ code i2c.c
```

The VS Code editor, labeled '2', shows the contents of the `i2c.c` file. The code is as follows:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>

#define DEV "/dev/i2c-0"
#define DEVICE_ADDR 0x4d

int write_I2C(int fd, unsigned char addr, void *ptr_data, unsigned int len)
{ ... }

int read_I2C(int fd, unsigned char addr, void *ptr_data, unsigned int len)
{ ... }

int main(int argc, char *argv[])
{ ... }
```

Annotations in the image provide context for the code:

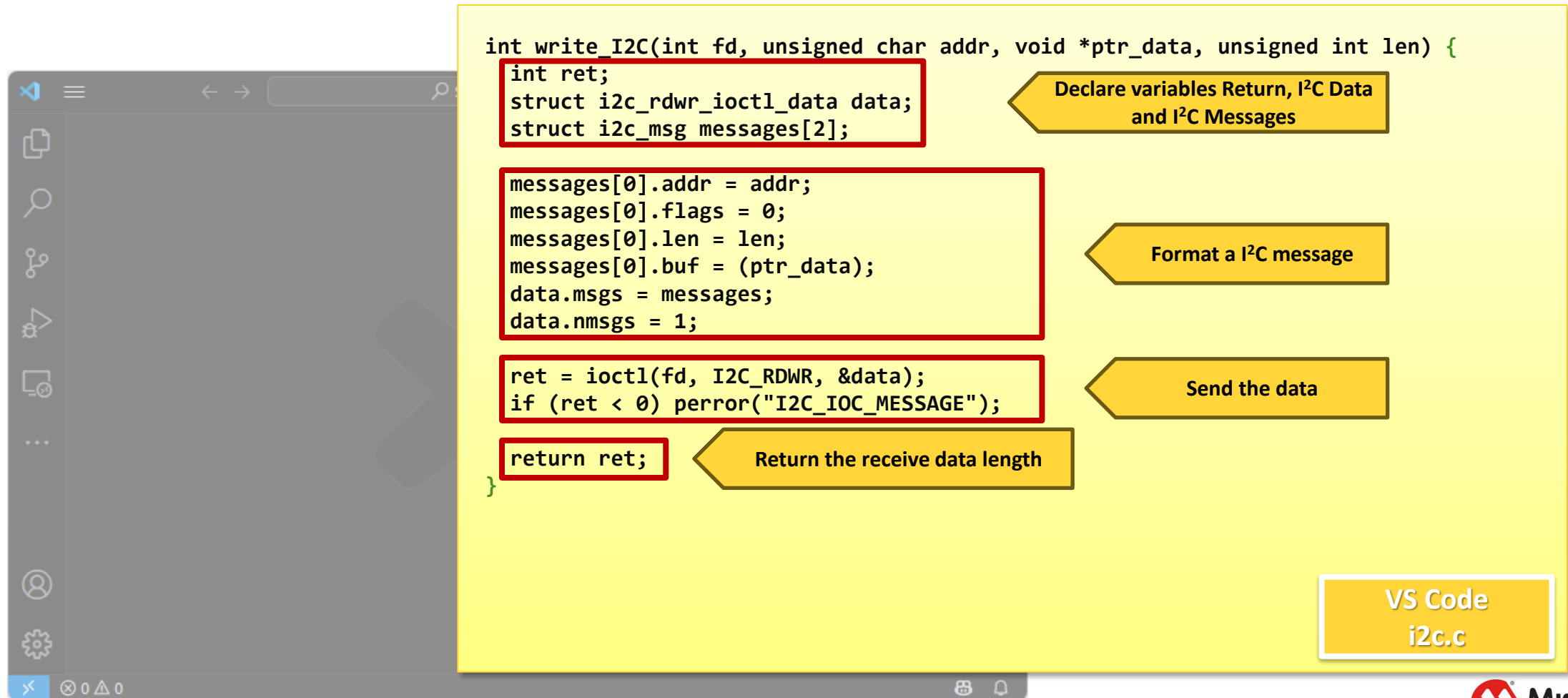
- GNU C Library & I2C Tools Library**: Points to the include statements for `<stdio.h>`, `<unistd.h>`, `<fcntl.h>`, `<string.h>`, `<sys/ioctl.h>`, `<linux/i2c.h>`, and `<linux/i2c-dev.h>`.
- I2C(Bus0) for Device**: Points to the `DEV` macro definition.
- I2C Client Device Address, modify to your device**: Points to the `DEVICE_ADDR` macro definition.
- I2C functions**: Points to the `write_I2C` and `read_I2C` function definitions.
- VS Code i2c.c**: Points to the `main` function.

A label 'VM' is placed at the bottom of the terminal window.

Lab6 – Use Tools and C Language to Control I²C

Step 2.2

- **Create a C language application** for Embedded Linux OS.



The image shows a VS Code editor window with a C program for writing to an I²C device. The code is annotated with yellow callouts explaining each section:

```
int write_I2C(int fd, unsigned char addr, void *ptr_data, unsigned int len) {  
    int ret;  
    struct i2c_rdwr_ioctl_data data;  
    struct i2c_msg messages[2];  
  
    messages[0].addr = addr;  
    messages[0].flags = 0;  
    messages[0].len = len;  
    messages[0].buf = (ptr_data);  
    data.msgs = messages;  
    data.nmsgs = 1;  
  
    ret = ioctl(fd, I2C_RDWR, &data);  
    if (ret < 0) perror("I2C_IOC_MESSAGE");  
  
    return ret;  
}
```

Annotations:

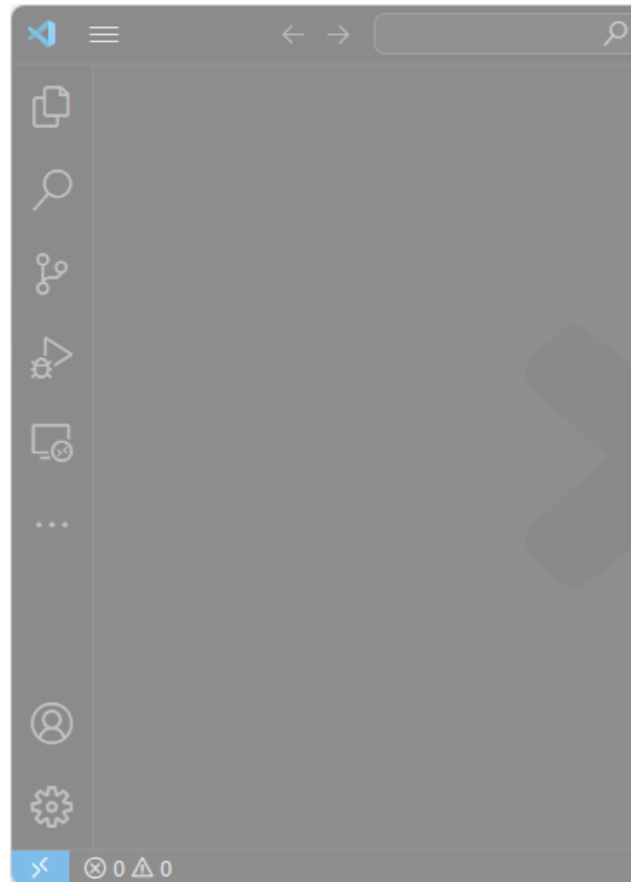
- Declare variables Return, I²C Data and I²C Messages**: Points to the variable declarations at the top of the function.
- Format a I²C message**: Points to the initialization of the `messages` array and the `data` struct.
- Send the data**: Points to the `ioctl` call and the error handling.
- Return the receive data length**: Points to the `return ret;` statement.

VS Code i2c.c

Lab6 – Use Tools and C Language to Control I²C

Step 2.3

- **Create a C language application** for Embedded Linux OS.



```
int read_I2C(int fd, unsigned char addr, void *ptr_data, unsigned int len) {  
    int ret;  
    struct i2c_rdwr_ioctl_data data;  
    struct i2c_msg messages[2];  
  
    messages[0].addr = addr;  
    messages[0].flags = I2C_M_RD;  
    messages[0].len = len;  
    messages[0].buf = (ptr_data);  
    memset(messages[0].buf, 0, len);  
    data.msgs = messages;  
    data.nmsgs = 1;  
  
    ret = ioctl(fd, I2C_RDWR, &data);  
    if (ret < 0) perror("I2C_IOC_MESSAGE");  
  
    return ret;  
}
```

Declare variables Return, I²C Data
and I²C Messages

Format a I²C message

Send the data

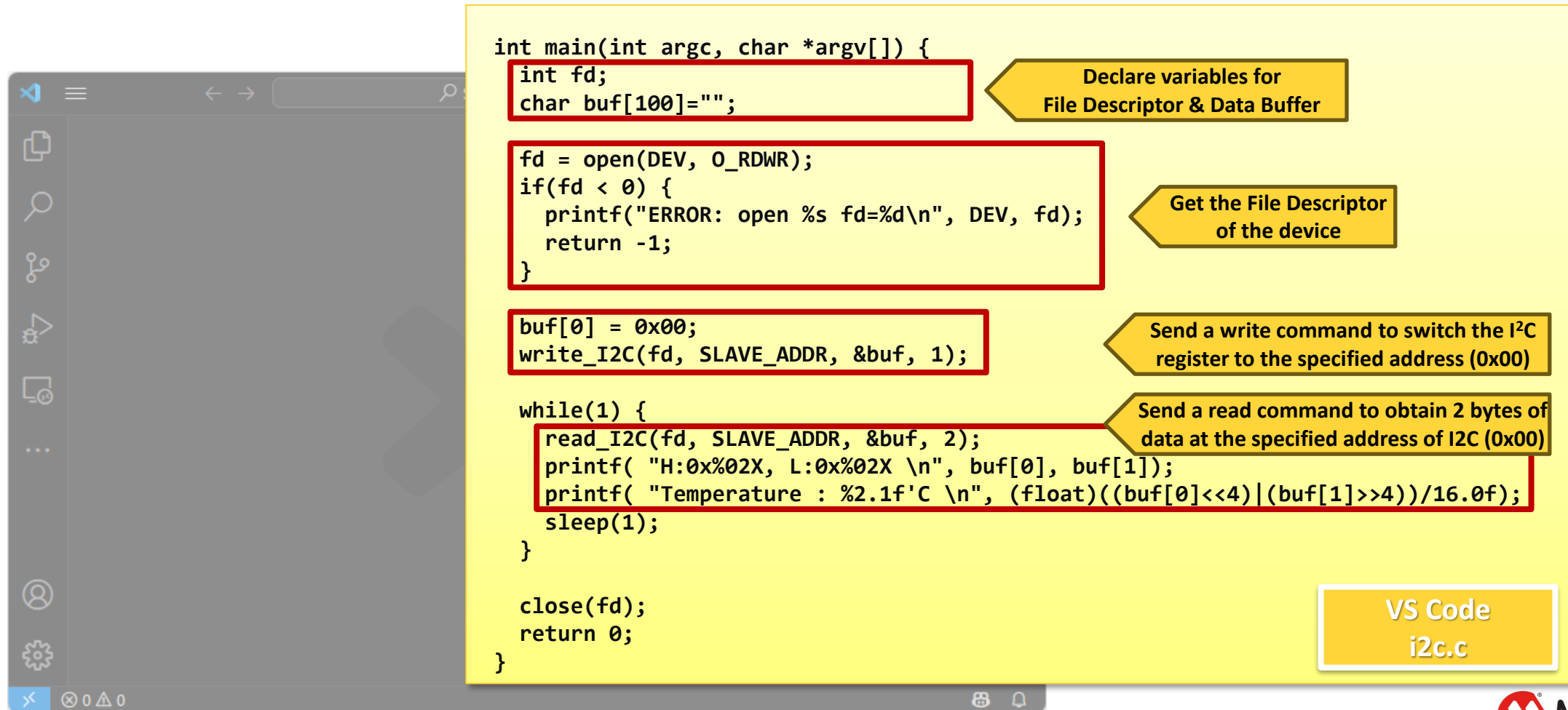
Return the receive data length

VS Code
i2c.c

Lab6 – Use Tools and C Language to Control I²C

Step 2.4

- **Create a C language application** for Embedded Linux OS.



```
int main(int argc, char *argv[]) {  
    int fd;  
    char buf[100]="";  
  
    fd = open(DEV, O_RDWR);  
    if(fd < 0) {  
        printf("ERROR: open %s fd=%d\n", DEV, fd);  
        return -1;  
    }  
  
    buf[0] = 0x00;  
    write_I2C(fd, SLAVE_ADDR, &buf, 1);  
  
    while(1) {  
        read_I2C(fd, SLAVE_ADDR, &buf, 2);  
        printf( "H:0x%02X, L:0x%02X \n", buf[0], buf[1]);  
        printf( "Temperature : %2.1f'C \n", (float)((buf[0]<<4)|(buf[1]>>4))/16.0f);  
        sleep(1);  
    }  
  
    close(fd);  
    return 0;  
}
```

Declare variables for File Descriptor & Data Buffer

Get the File Descriptor of the device

Send a write command to switch the I²C register to the specified address (0x00)

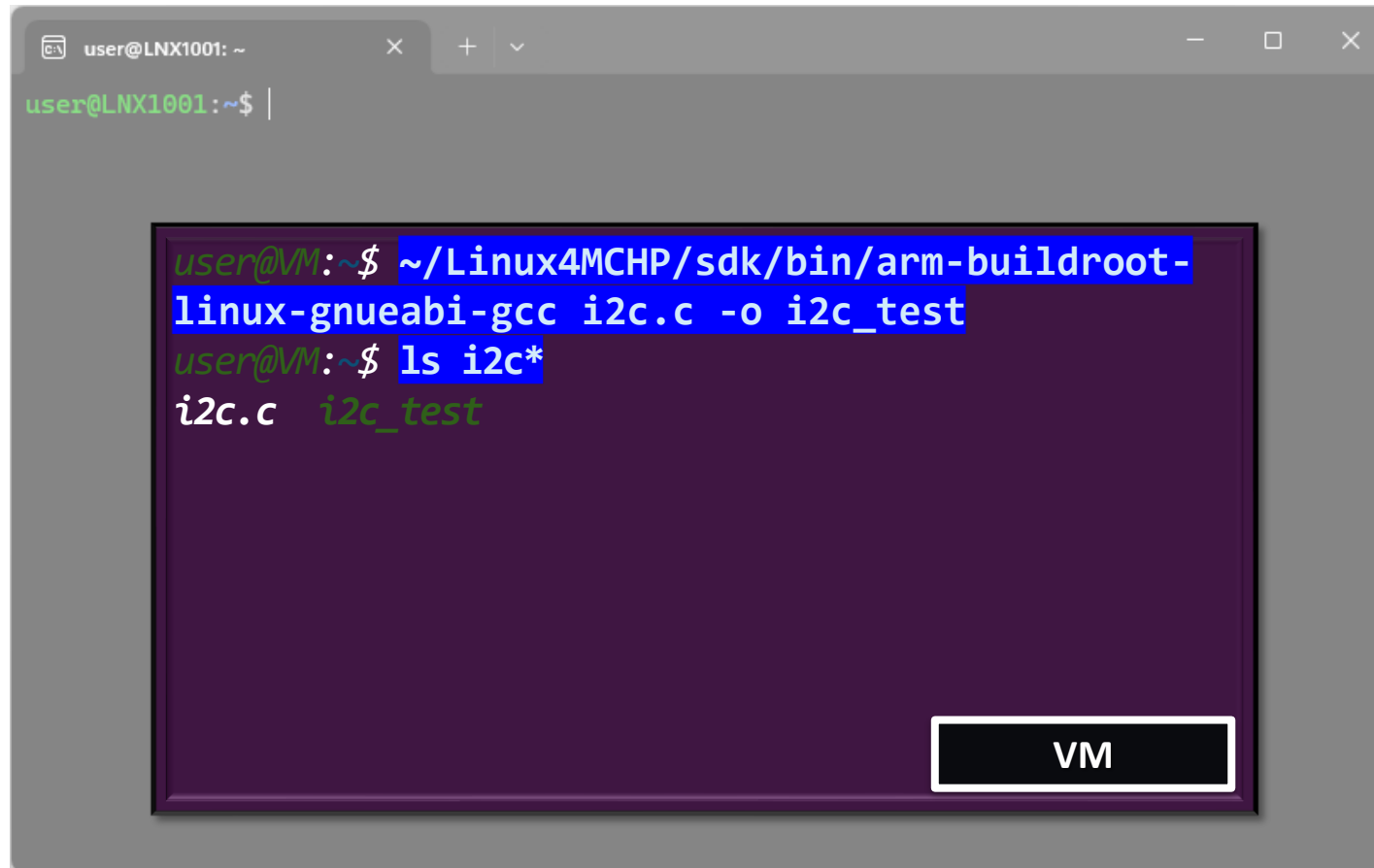
Send a read command to obtain 2 bytes of data at the specified address of I2C (0x00)

VS Code
i2c.c

Lab6 – Use Tools and C Language to Control I²C

Step 3

- **Compile the C language application** for Embedded Linux OS.

A terminal window titled 'user@LNX1001: ~' with standard window controls. It contains a nested terminal window titled 'user@VM: ~\$'. The nested terminal shows the compilation of 'i2c.c' using 'arm-buildroot-linux-gnueabi-gcc' and a subsequent 'ls' command listing the files.

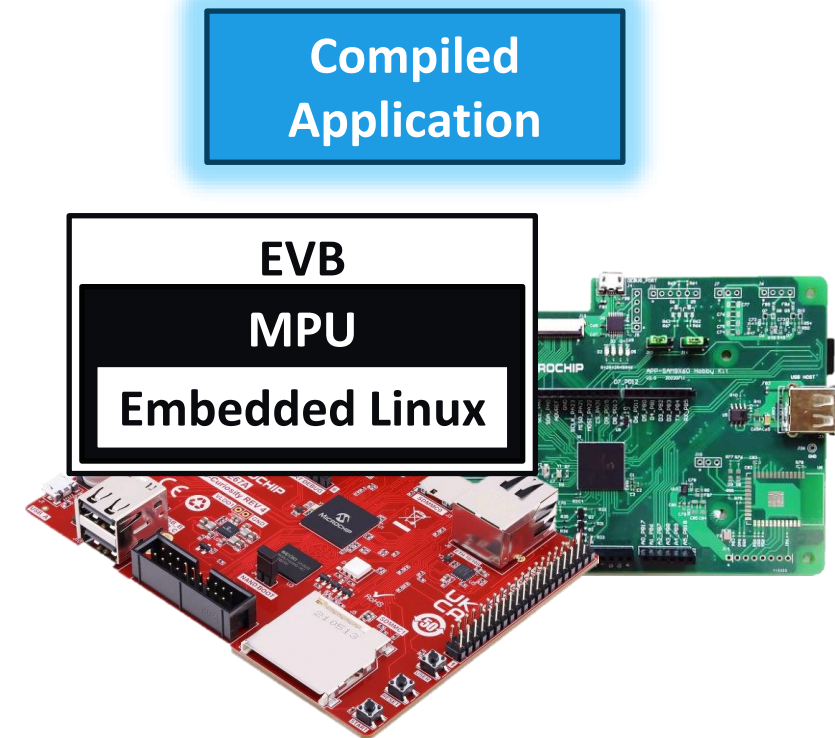
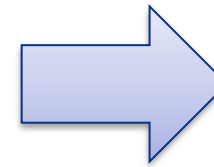
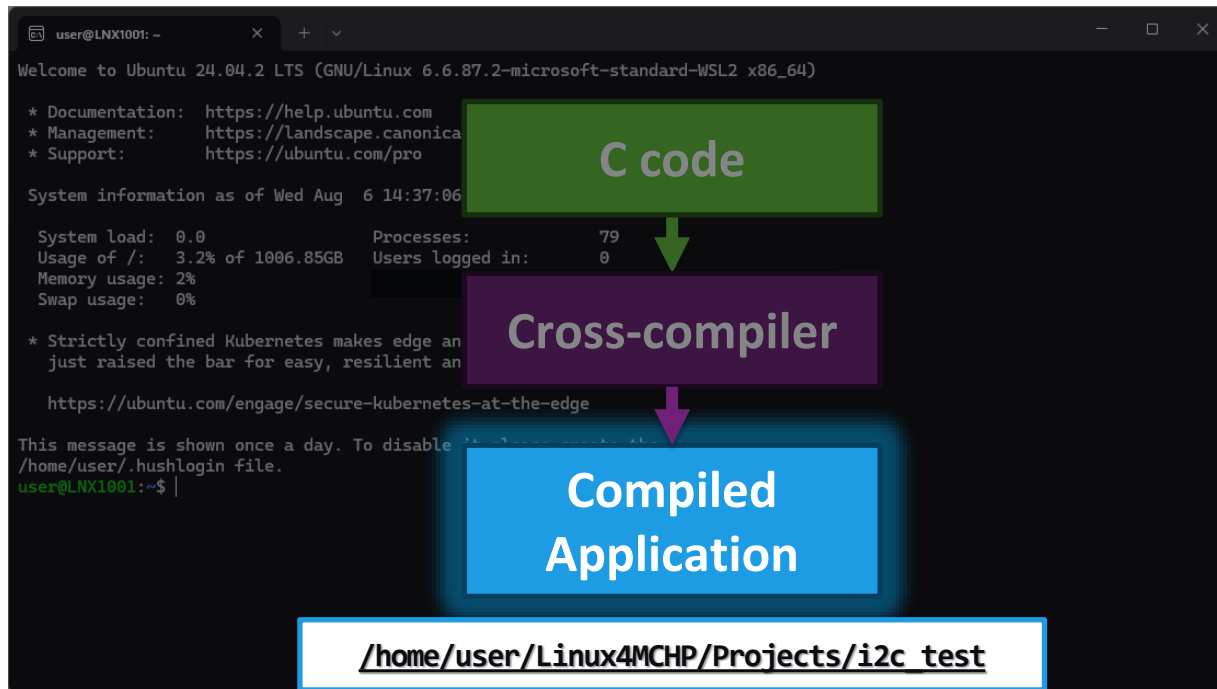
```
user@LNX1001: ~  
user@LNX1001:~$  
  
user@VM:~$ ~/Linux4MCHP/sdk/bin/arm-buildroot-  
linux-gnueabi-gcc i2c.c -o i2c_test  
user@VM:~$ ls i2c*  
i2c.c  i2c_test
```

VM

Lab6 – Use Tools and C Language to Control I²C

Step 4

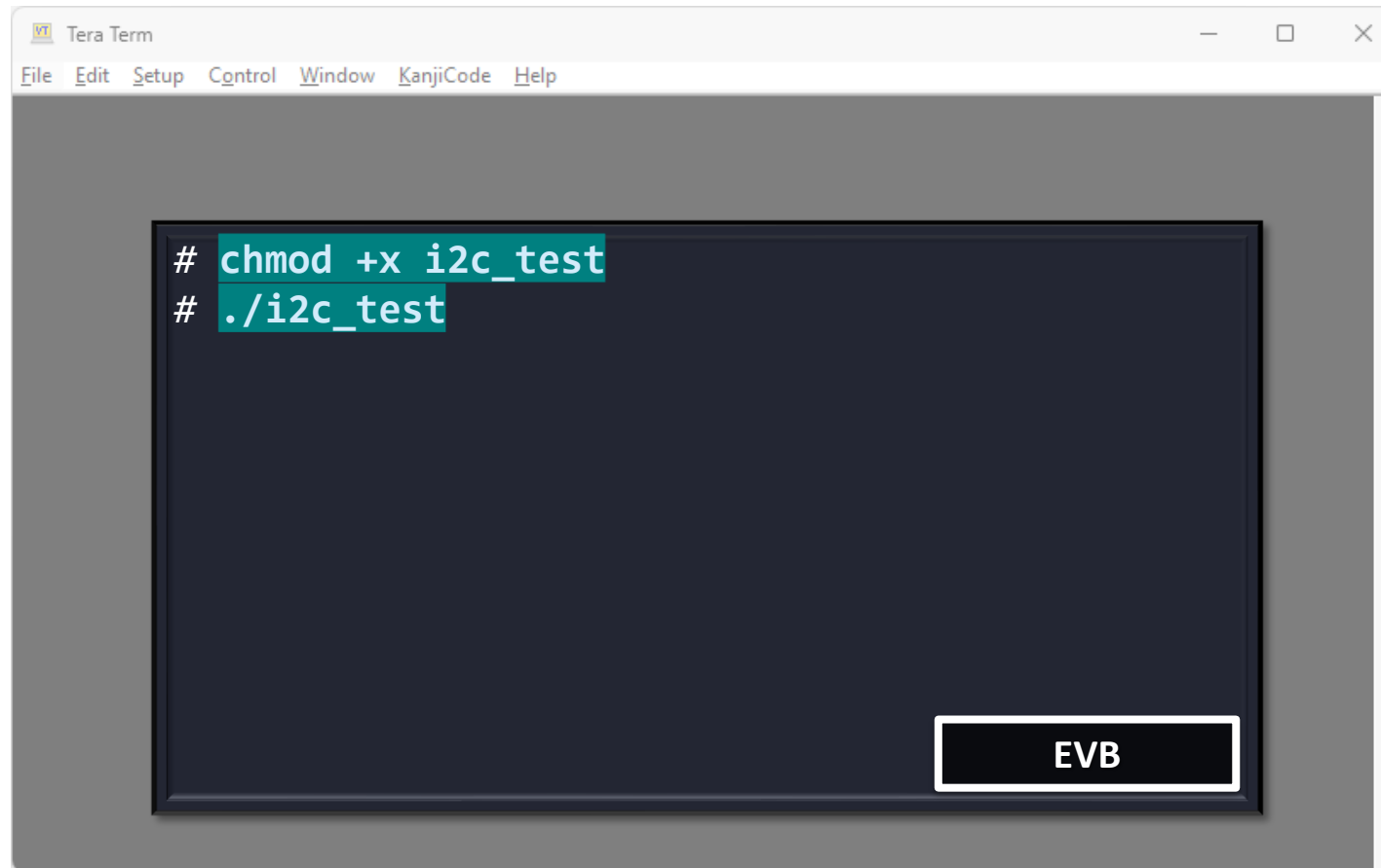
- Send the application to Embedded Linux OS.
(Refer to the appendix to send files to Embedded Linux System)



Lab6 – Use Tools and C Language to Control I²C

Step 5

- **Execute the application** for Embedded Linux OS.



The image shows a screenshot of a Tera Term terminal window. The window has a title bar with the text "Tera Term" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with the following items: File, Edit, Setup, Control, Window, KanjiCode, and Help. The main area of the terminal is dark gray and contains two lines of text, both highlighted in a light blue color: "# chmod +x i2c_test" and "# ./i2c_test". In the bottom right corner of the terminal window, there is a small white rectangular button with the text "EVB" in black.

```
# chmod +x i2c_test
# ./i2c_test
```

EVB

Lab6 – Use Tools and C Language to Control I²C

Result

- The application will continuously send messages to the I²C Bus (/dev/i2c-0) and return the temperature data from the I²C Device (MCP9800).

```
user@VM: ~  
# chmod +x i2c_test  
# ./i2c_test  
H:0x19, L:0x80  
Temperature : 25.5'C  
H:0x19, L:0x80  
Temperature : 25.5'C  
H:0x19, L:0x80  
Temperature : 25.5'C  
H:0x1A, L:0x80  
Temperature : 26.5'C  
H:0x1C, L:0x00  
Temperature : 28.0'C  
H:0x1C, L:0x80  
Temperature : 28.5'C  
H:0x1D, L:0x00  
Temperature : 29.0'C  
H:0x1B, L:0x80  
Temperature : 27.5'C  
█
```

mikroBUS PIN	External
SCL	I ² C Temp. (MCP9800-A5), SCL
SDA	I ² C Temp. (MCP9800-A5), SDA
GND	GND

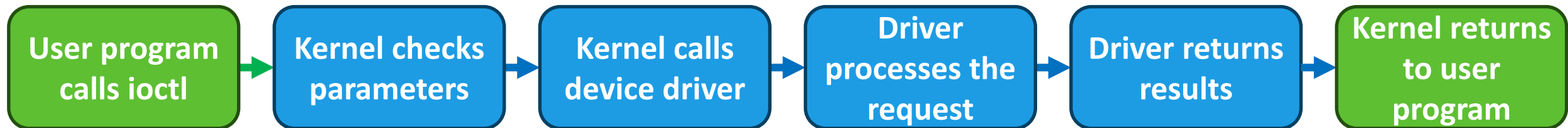
Input/Output Control (ioctl)

Input/Output Control (ioctl)

- In Embedded Linux system, **ioctl (Input/Output Control)** is **a system call that enables low-level control of device drivers**. It provides a mechanism for user-space programs to interact directly with device drivers to set or query specific parameters or states of the device.
- We can use ioctl to implement various device control functions and directly interact with device hardware, but we need to understand the details of the device driver to use ioctl correctly, which is **highly complex**.
- How to use ioctl :

```
# int ioctl(int fd, unsigned long request, ...);
```

fd :	<u>file descriptor that points to the device.</u>
request :	<u>the command to be executed.</u>
... :	<u>variable arguments passed to the driver.</u>



Lab7 – Use Tools and C Language to Control SPI

Lab7 – Use Tools and C Language to Control SPI

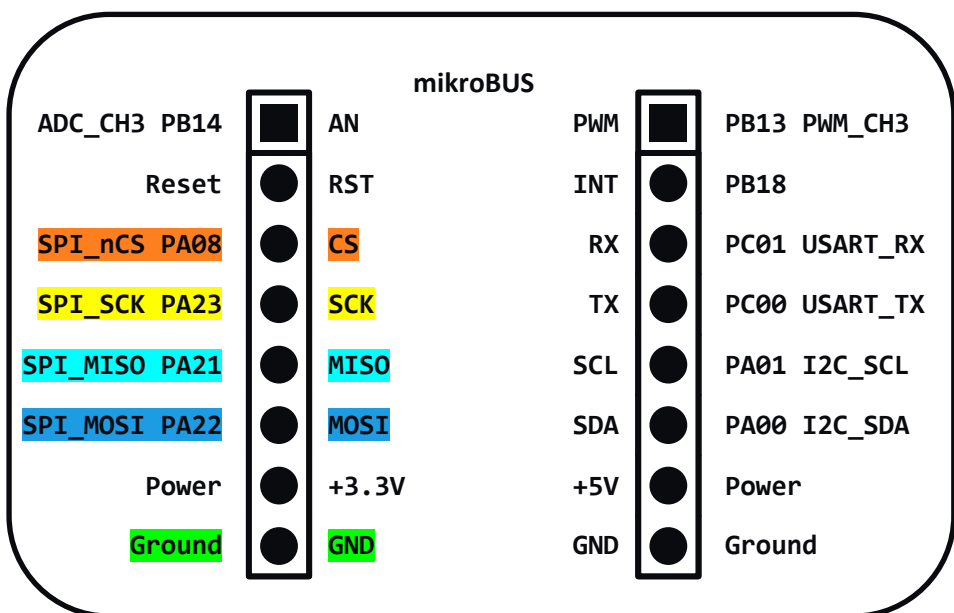
- In the default OS Image from Linux4SAM website, since the EVB does not have SPI devices, the Device-Tree does not define the SPI Bus.
- We provide an OS Image that has been added with the required SPI Bus in the Device-Tree, and enable the corresponding device driver. It will list them in [/dev/spidev-x.x](#), we can use a tool or application to access it, and transmit and receive messages from SPI.
- Follow the steps below to complete the above requirements :
 1. Connect the SPI of [mikroBUS Pin](#) to the external SPI Device (25LC256).
 2. Try using the Tools to access the SPI of MikroBUS.
 3. Create a C language application and compile it.
 4. Execute the application and access the SPI of MikroBUS.

Let's go!

Lab7 – Use Tools and C Language to Control SPI

Preparation

- Connect the SPI of mikroBUS **CS Pin (PA08)**, **SCK Pin (PA23)**, **MISO Pin (PA21)** and **MOSI Pin (PA22)** to the External SPI Device.

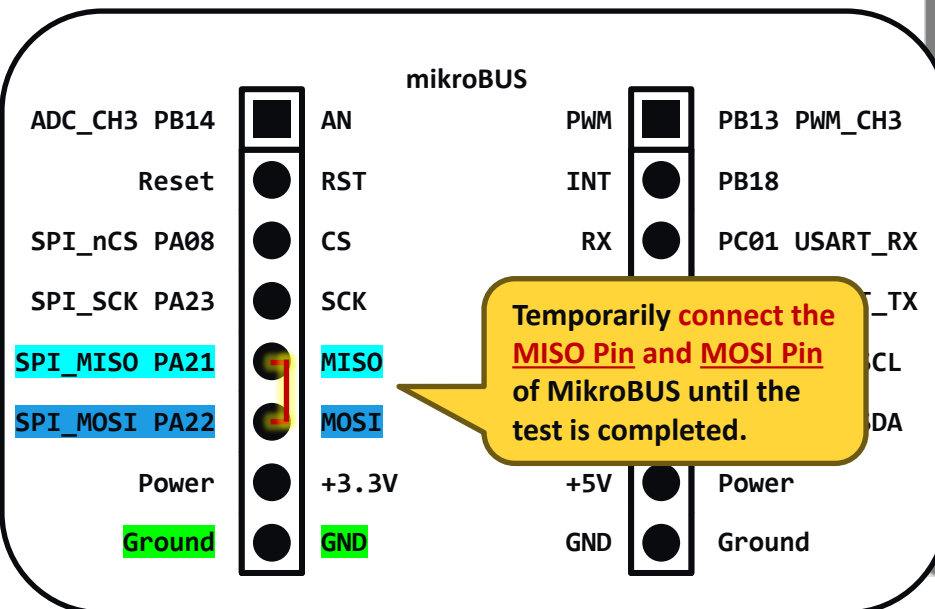


mikroBUS PIN	External
SCK	SPI EEPROM (25LC256), SCK
MISO	SPI EEPROM (25LC256), MISO
MOSI	SPI EEPROM (25LC256), MOSI
CS	SPI EEPROM (25LC256), CS
GND	GND

Lab7 – Use Tools and C Language to Control SPI

Step 1

- Try using the Tools for **SPI Tools** to access the SPI.



```
Tera Term
File Edit Setup Control Window KanjiCode Help

# spi-config -d /dev/spidev0.0 -q
/dev/spidev0.0: mode=0, lsb=0, bits=8,
speed=1000000, spiready=0
# printf 'Hello\n' | spi-pipe -d /dev/spidev0.0
Hello
#
```

EVB

(Please restore the original connect after testing !!)

Lab7 – Use Tools and C Language to Control SPI

Step 2.1

- **Create a C language application** for Embedded Linux OS.
(Please copy the code from the CopyPaste files to the Text Editor !!)

The image shows a virtual machine (VM) environment with a terminal window and a VS Code editor. The terminal window, labeled '1', shows the following commands being executed:

```
user@VM:~$ mkdir -p ~/Linux4MCHP/Projects
user@VM:~$ cd ~/Linux4MCHP/Projects
user@VM:~$ code spi.c
```

The VS Code editor, labeled '2', shows the contents of the `spi.c` file. The code is as follows:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>

#define DEV "/dev/spidev0.0"

...

int write_SPI(int fd, void *ptr_data, unsigned int len)
{ ... }

int read_SPI(int fd, void *ptr_data, unsigned int len)
{ ... }

int main(int argc, char *argv[])
{
    ...
}
```

Annotations in the image point to specific parts of the code:

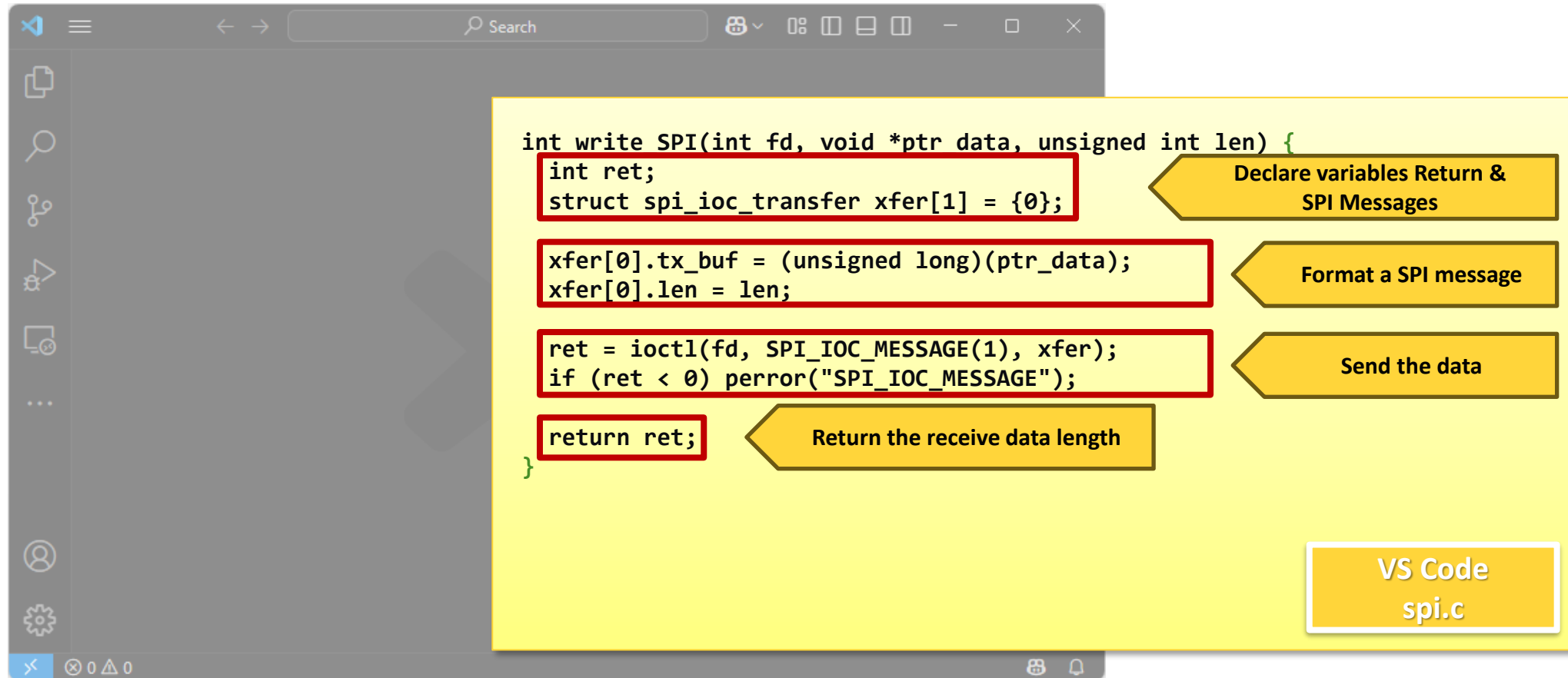
- GNU C Library & SPI Tools Library** points to the include statements.
- SPI for Device** points to the `DEV` definition.
- SPI functions** points to the `write_SPI` and `read_SPI` functions.
- VS Code spi.c** points to the editor window.

A label **VM** is placed at the bottom of the terminal window.

Lab7 – Use Tools and C Language to Control SPI

Step 2.2

- **Create a C language application** for Embedded Linux OS.



The image shows a screenshot of the Visual Studio Code (VS Code) editor interface. The main editor window displays a C program for SPI control. The code is as follows:

```
int write SPI(int fd, void *ptr data, unsigned int len) {  
    int ret;  
    struct spi_ioc_transfer xfer[1] = {0};  
  
    xfer[0].tx_buf = (unsigned long)(ptr_data);  
    xfer[0].len = len;  
  
    ret = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);  
    if (ret < 0) perror("SPI_IOC_MESSAGE");  
  
    return ret;  
}
```

Annotations are provided for each line of code:

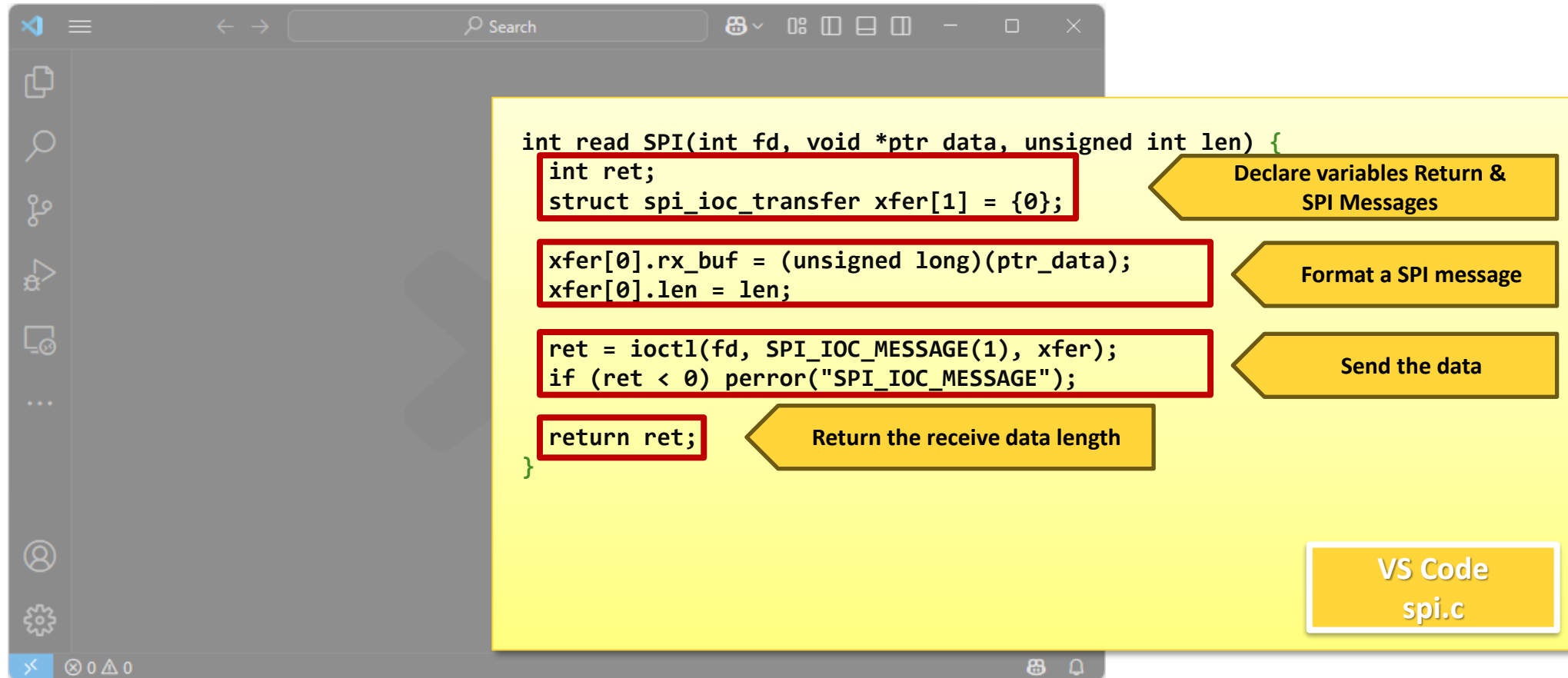
- Declare variables Return & SPI Messages**: Points to the declaration of `int ret;` and `struct spi_ioc_transfer xfer[1] = {0};`.
- Format a SPI message**: Points to the assignment of `xfer[0].tx_buf` and `xfer[0].len`.
- Send the data**: Points to the `ioctl` call and the `perror` statement.
- Return the receive data length**: Points to the `return ret;` statement.

A button labeled **VS Code spi.c** is located in the bottom right corner of the code editor area.

Lab7 – Use Tools and C Language to Control SPI

Step 2.3

- **Create a C language application** for Embedded Linux OS.



The image shows a VS Code editor window with a C program for SPI read operation. The code is annotated with yellow callouts explaining each step:

```
int read SPI(int fd, void *ptr data, unsigned int len) {  
    int ret;  
    struct spi_ioc_transfer xfer[1] = {0};  
  
    xfer[0].rx_buf = (unsigned long)(ptr_data);  
    xfer[0].len = len;  
  
    ret = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);  
    if (ret < 0) perror("SPI_IOC_MESSAGE");  
  
    return ret;  
}
```

Annotations:

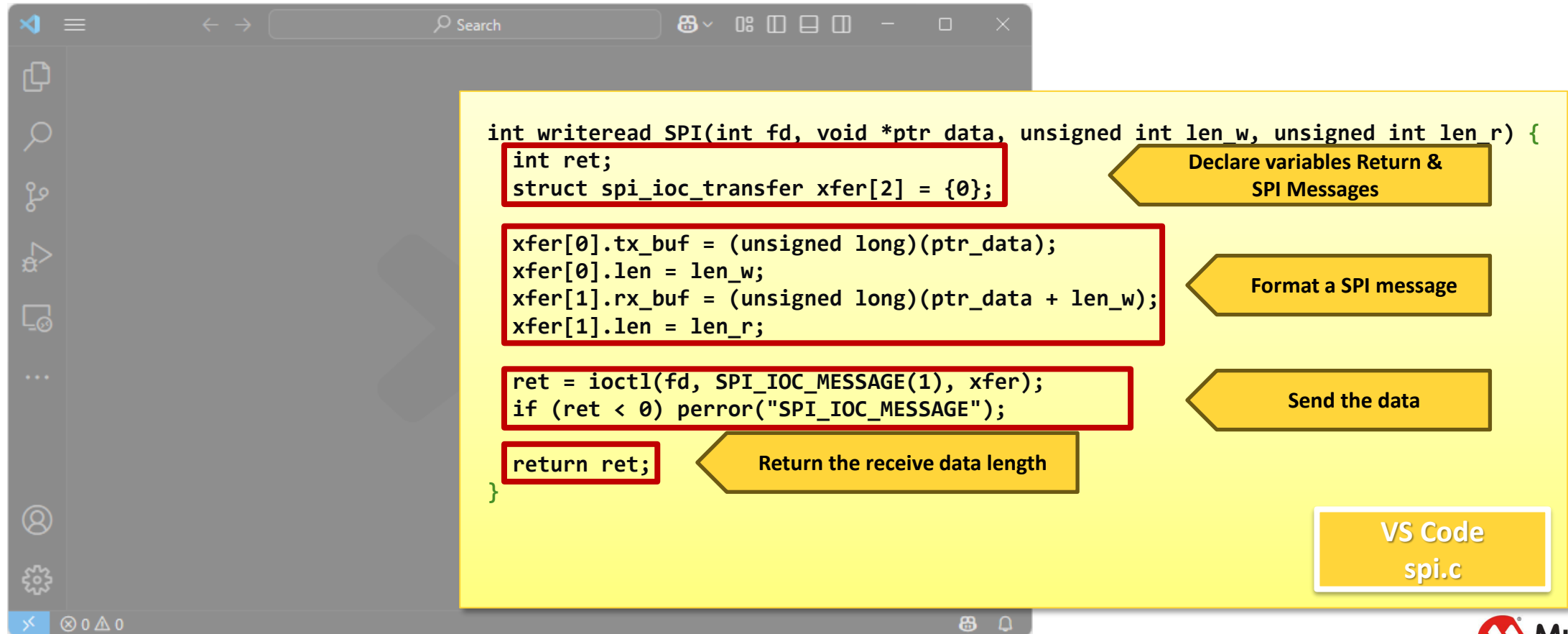
- Declare variables Return & SPI Messages**: Points to the declaration of `int ret;` and `struct spi_ioc_transfer xfer[1] = {0};`.
- Format a SPI message**: Points to the assignment of `xfer[0].rx_buf` and `xfer[0].len`.
- Send the data**: Points to the `ioctl` call and the `perror` check.
- Return the receive data length**: Points to the `return ret;` statement.

VS Code spi.c

Lab7 – Use Tools and C Language to Control SPI

Step 2.4

- **Create a C language application** for Embedded Linux OS.



The image shows a screenshot of the Visual Studio Code (VS Code) editor interface. The main window displays the contents of a file named `spi.c`. The code is a C function named `writeread SPI` that takes four arguments: `int fd`, `void *ptr data`, `unsigned int len_w`, and `unsigned int len_r`. The function is annotated with yellow callouts explaining its steps:

```
int writeread SPI(int fd, void *ptr data, unsigned int len_w, unsigned int len_r) {  
    int ret;  
    struct spi_ioc_transfer xfer[2] = {0};  
  
    xfer[0].tx_buf = (unsigned long)(ptr_data);  
    xfer[0].len = len_w;  
    xfer[1].rx_buf = (unsigned long)(ptr_data + len_w);  
    xfer[1].len = len_r;  
  
    ret = ioctl(fd, SPI_IOC_MESSAGE(1), xfer);  
    if (ret < 0) perror("SPI_IOC_MESSAGE");  
  
    return ret;  
}
```

The annotations are as follows:

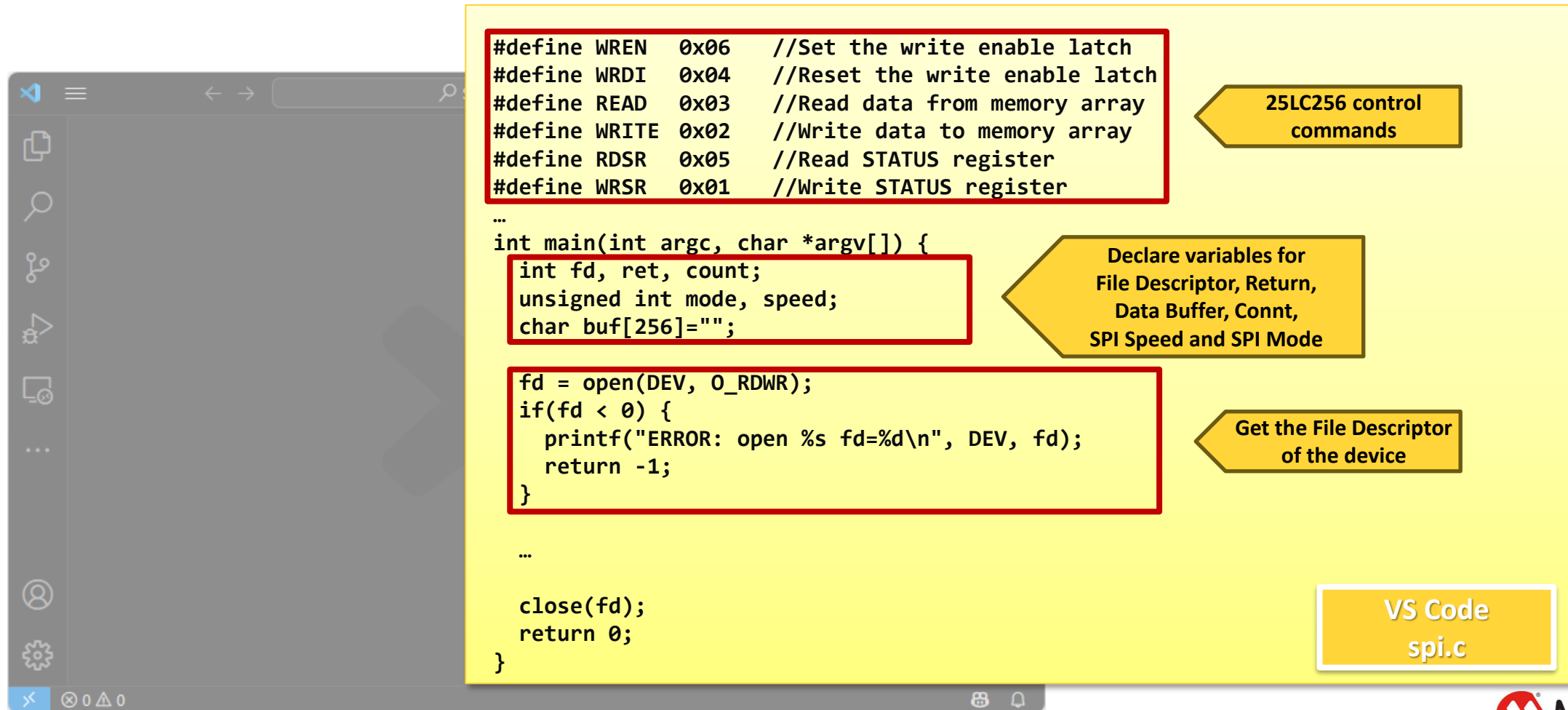
- Declare variables Return & SPI Messages**: Points to the declaration of `int ret;` and `struct spi_ioc_transfer xfer[2] = {0};`.
- Format a SPI message**: Points to the assignment of `xfer[0]` and `xfer[1]` fields.
- Send the data**: Points to the `ioctl` call and the `perror` check.
- Return the receive data length**: Points to the `return ret;` statement.

The VS Code interface includes a search bar at the top, a sidebar on the left with icons for Explorer, Search, Source Control, Run and Debug, and Settings, and a status bar at the bottom showing the file name `spi.c` and the current line and column numbers (0, 0, 0).

Lab7 – Use Tools and C Language to Control SPI

Step 2.5

- **Create a C language application** for Embedded Linux OS.



```
#define WREN  0x06  //Set the write enable latch
#define WRDI  0x04  //Reset the write enable latch
#define READ  0x03  //Read data from memory array
#define WRITE 0x02  //Write data to memory array
#define RDSR  0x05  //Read STATUS register
#define WRSR  0x01  //Write STATUS register

...

int main(int argc, char *argv[]) {
    int fd, ret, count;
    unsigned int mode, speed;
    char buf[256]="";

    fd = open(DEV, O_RDWR);
    if(fd < 0) {
        printf("ERROR: open %s fd=%d\n", DEV, fd);
        return -1;
    }

    ...

    close(fd);
    return 0;
}
```

25LC256 control commands

Declare variables for File Descriptor, Return, Data Buffer, Connt, SPI Speed and SPI Mode

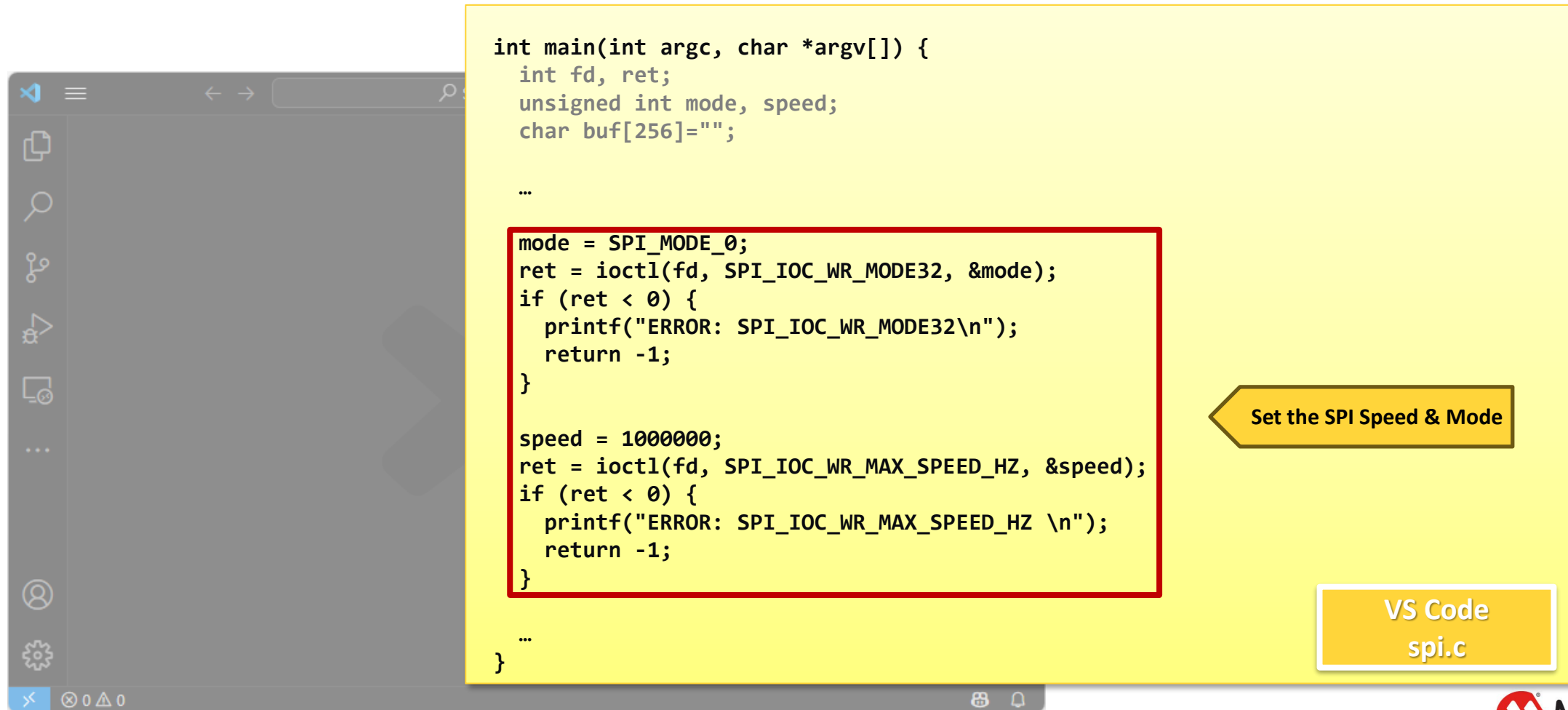
Get the File Descriptor of the device

VS Code spi.c

Lab7 – Use Tools and C Language to Control SPI

Step 2.6

- **Create a C language application** for Embedded Linux OS.



```
int main(int argc, char *argv[]) {
    int fd, ret;
    unsigned int mode, speed;
    char buf[256]="";

    ...

    mode = SPI_MODE_0;
    ret = ioctl(fd, SPI_IOC_WR_MODE32, &mode);
    if (ret < 0) {
        printf("ERROR: SPI_IOC_WR_MODE32\n");
        return -1;
    }

    speed = 1000000;
    ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret < 0) {
        printf("ERROR: SPI_IOC_WR_MAX_SPEED_HZ \n");
        return -1;
    }

    ...
}
```

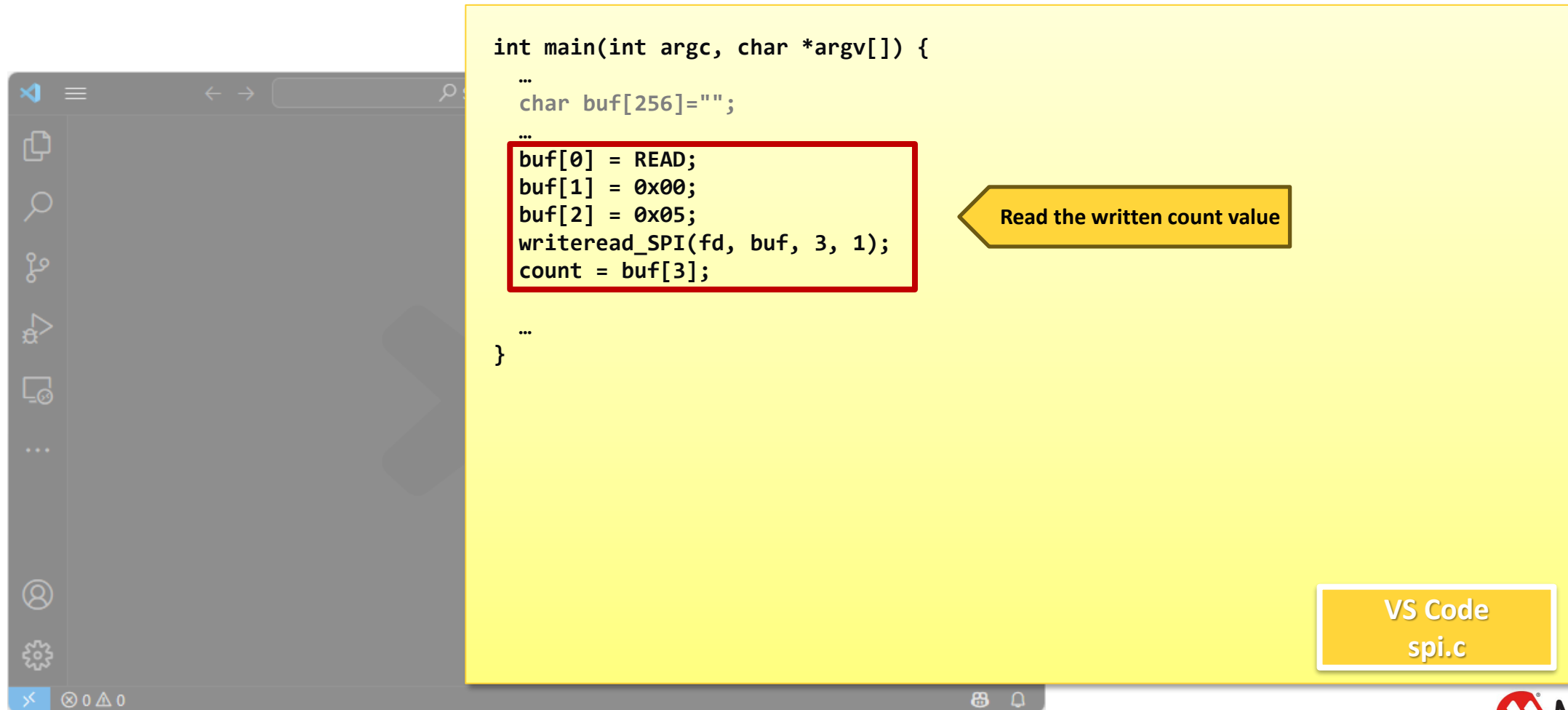
Set the SPI Speed & Mode

VS Code
spi.c

Lab7 – Use Tools and C Language to Control SPI

Step 2.7

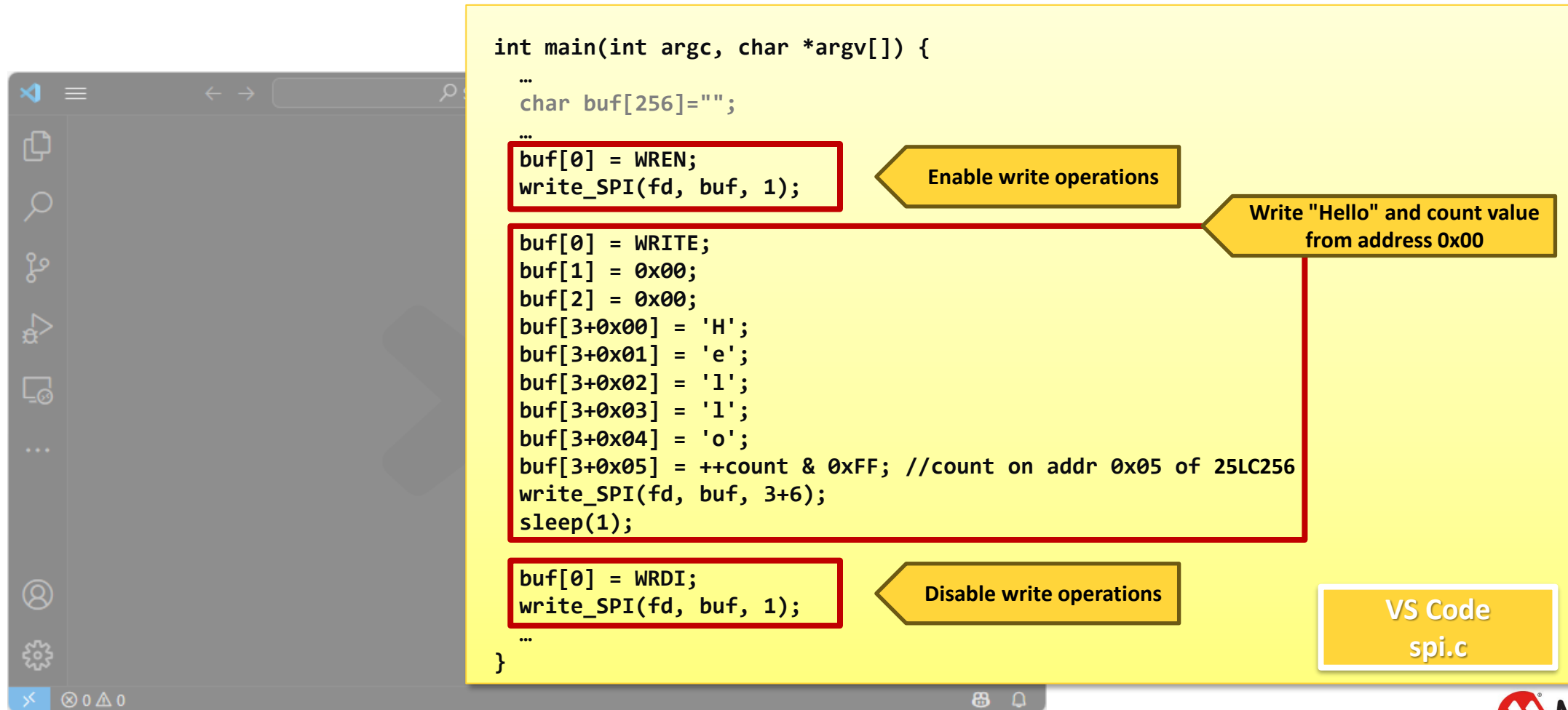
- **Create a C language application** for Embedded Linux OS.



Lab7 – Use Tools and C Language to Control SPI

Step 2.8

- **Create a C language application** for Embedded Linux OS.



The image shows a VS Code editor window with a C program for SPI control. The code is as follows:

```
int main(int argc, char *argv[]) {  
    ...  
    char buf[256]="";  
    ...  
    buf[0] = WREN;  
    write_SPI(fd, buf, 1);  
    ...  
    buf[0] = WRITE;  
    buf[1] = 0x00;  
    buf[2] = 0x00;  
    buf[3+0x00] = 'H';  
    buf[3+0x01] = 'e';  
    buf[3+0x02] = 'l';  
    buf[3+0x03] = 'l';  
    buf[3+0x04] = 'o';  
    buf[3+0x05] = ++count & 0xFF; //count on addr 0x05 of 25LC256  
    write_SPI(fd, buf, 3+6);  
    sleep(1);  
    ...  
    buf[0] = WRDI;  
    write_SPI(fd, buf, 1);  
    ...  
}
```

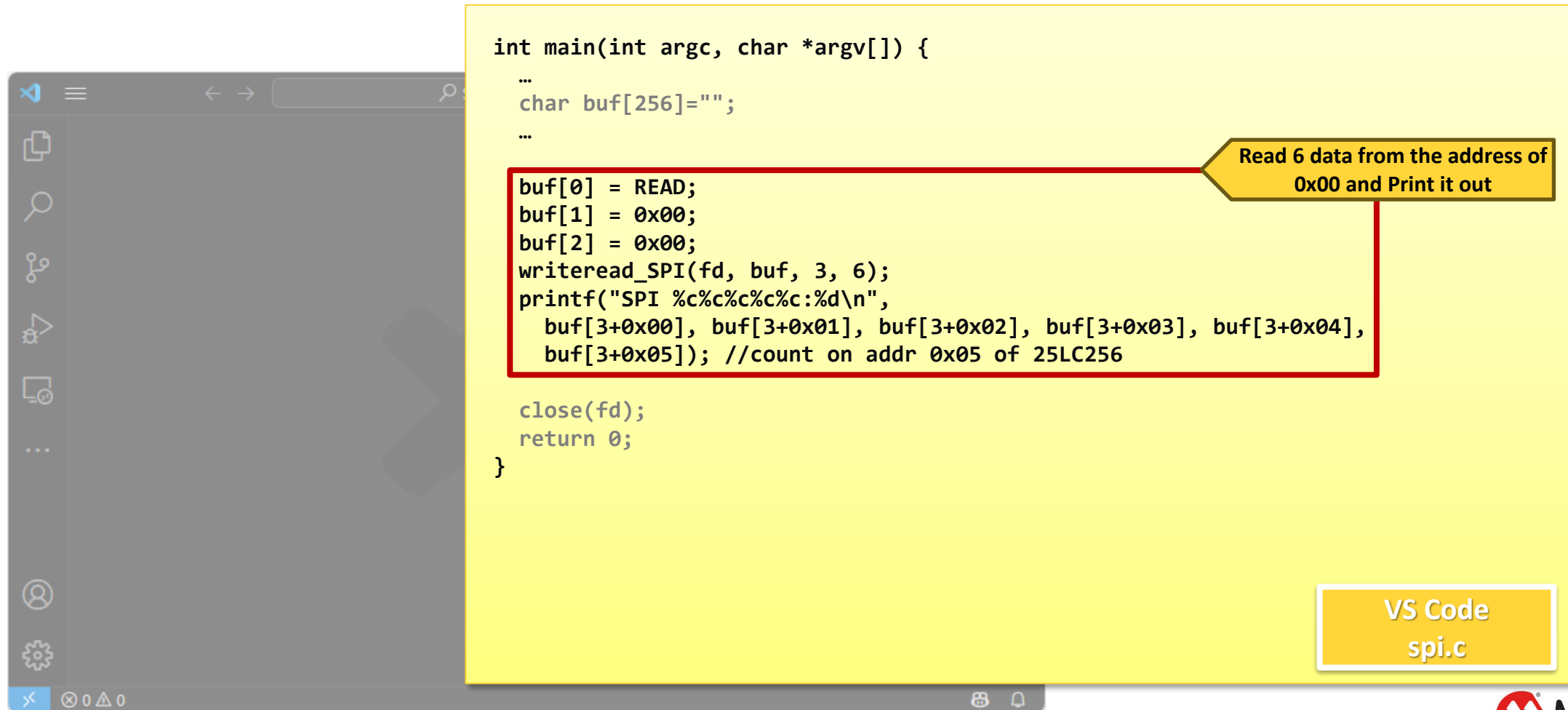
Annotations in the image:

- A red box highlights the code block: `buf[0] = WREN; write_SPI(fd, buf, 1);`. An arrow points from this box to a yellow callout box labeled "Enable write operations".
- A red box highlights the code block: `buf[0] = WRITE; ... buf[3+0x05] = ++count & 0xFF; //count on addr 0x05 of 25LC256; write_SPI(fd, buf, 3+6); sleep(1);`. An arrow points from this box to a yellow callout box labeled "Write 'Hello' and count value from address 0x00".
- A red box highlights the code block: `buf[0] = WRDI; write_SPI(fd, buf, 1);`. An arrow points from this box to a yellow callout box labeled "Disable write operations".
- A yellow box in the bottom right corner is labeled "VS Code spi.c".

Lab7 – Use Tools and C Language to Control SPI

Step 2.9

- **Create a C language application** for Embedded Linux OS.



```
int main(int argc, char *argv[]) {  
    ...  
    char buf[256]="";  
    ...  
    buf[0] = READ;  
    buf[1] = 0x00;  
    buf[2] = 0x00;  
    writeread_SPI(fd, buf, 3, 6);  
    printf("SPI %c%c%c%c%c:%d\n",  
        buf[3+0x00], buf[3+0x01], buf[3+0x02], buf[3+0x03], buf[3+0x04],  
        buf[3+0x05]); //count on addr 0x05 of 25LC256  
  
    close(fd);  
    return 0;  
}
```

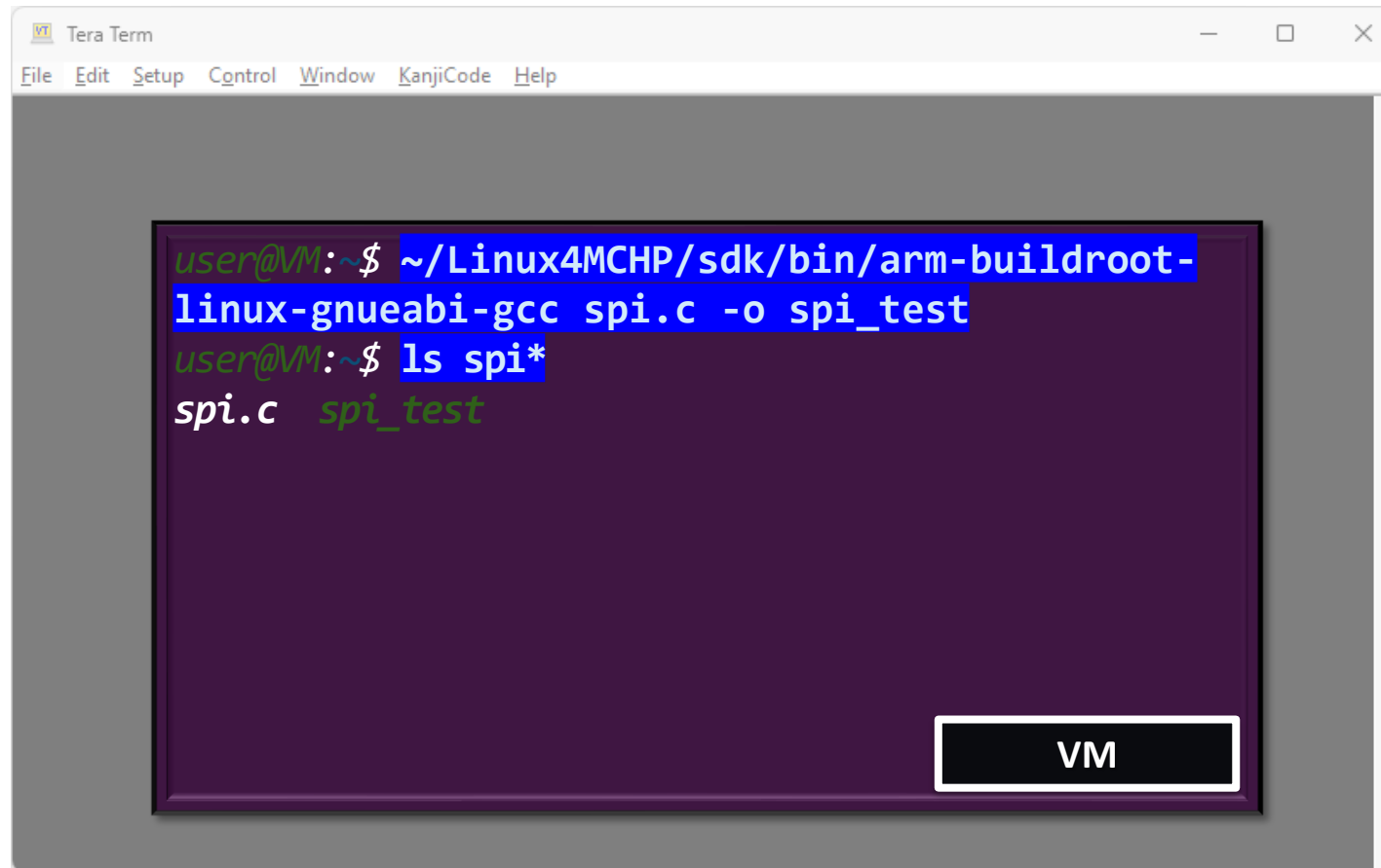
Read 6 data from the address of 0x00 and Print it out

VS Code
spi.c

Lab7 – Use Tools and C Language to Control SPI

Step 3

- **Compile the C language application** for Embedded Linux OS.



The screenshot shows a Tera Term terminal window with a menu bar (File, Edit, Setup, Control, Window, KanjiCode, Help) and a dark purple terminal area. The terminal displays the following commands and output:

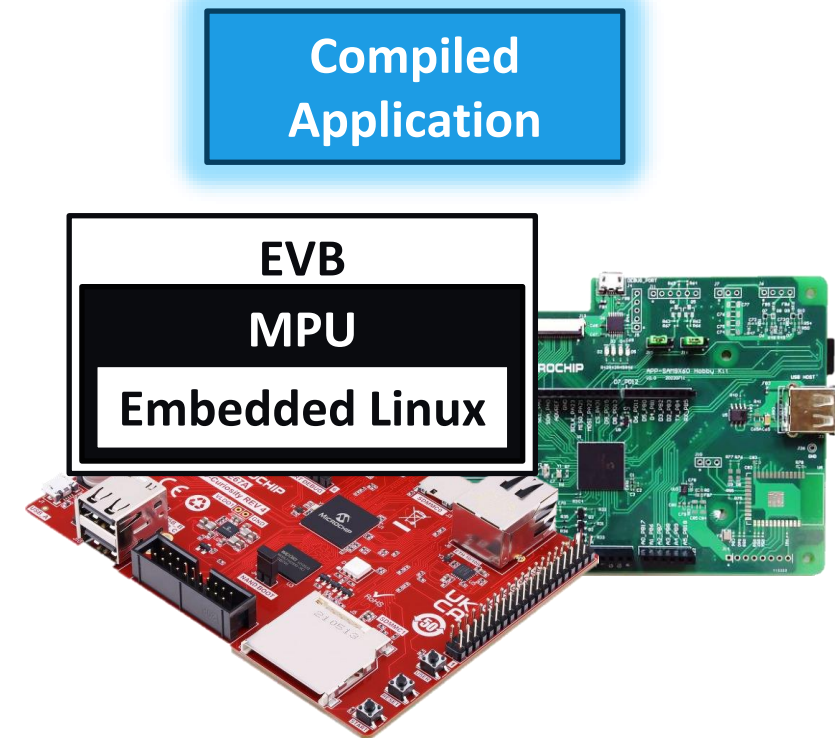
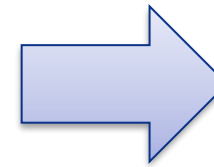
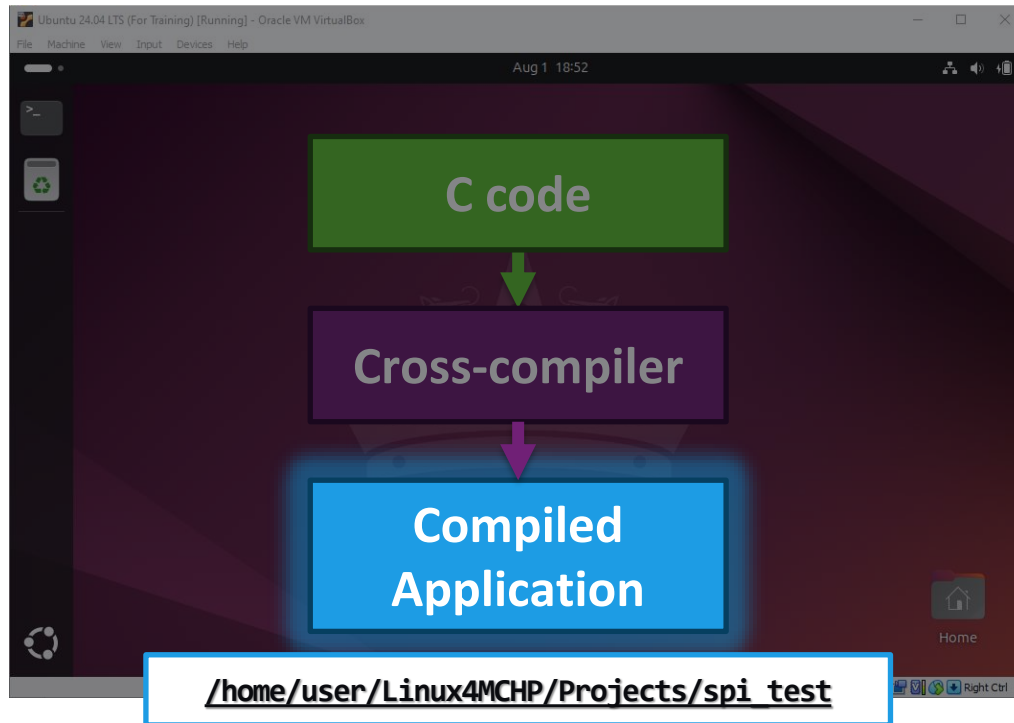
```
user@VM:~$ ~/Linux4MCHP/sdk/bin/arm-buildroot-  
linux-gnueabi-gcc spi.c -o spi_test  
user@VM:~$ ls spi*  
spi.c  spi_test
```

A white rectangular box with the text "VM" is located in the bottom right corner of the terminal window.

Lab7 – Use Tools and C Language to Control SPI

Step 4

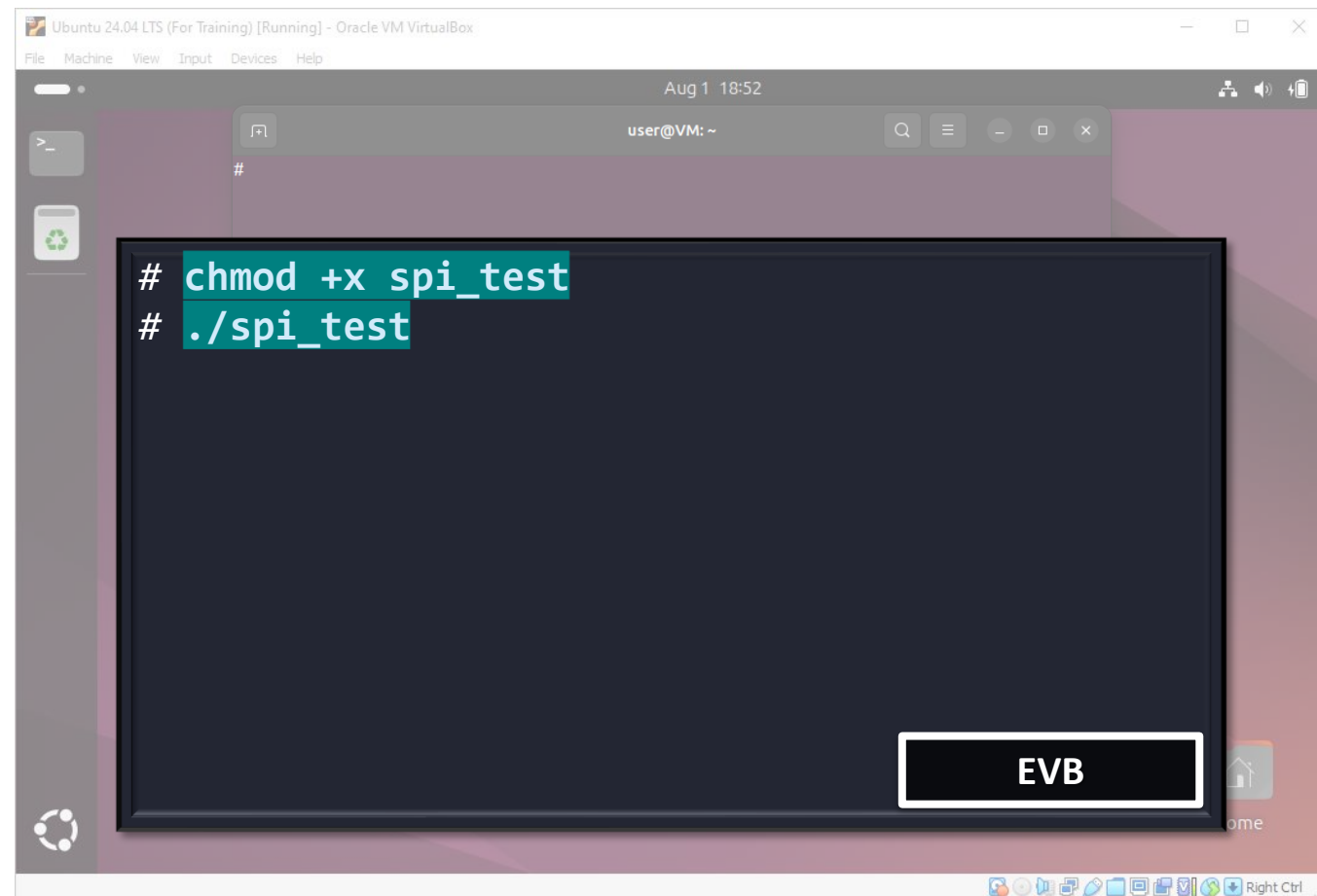
- Send the application to Embedded Linux OS.
(Refer to the appendix to send files to Embedded Linux System)



Lab7 – Use Tools and C Language to Control SPI

Step 5

- **Execute the application** for Embedded Linux OS.



Lab7 – Use Tools and C Language to Control SPI

Result

- The application will first read the EEPROM (25LC256) once and obtain the count value from SPI Bus (/dev/spidev-0.0), then add the count value and write the "Hello" text to the EEPROM, and then read the EEPROM to obtain the written data and print it out.

```
user@VM: ~  
# chmod +x spi_test  
# ./spi_test  
SPI Read Data : Hello:26  
# ./spi_test  
SPI Read Data : Hello:27  
# ./spi_test  
SPI Read Data : Hello:28  
# ./spi_test  
SPI Read Data : Hello:29  
# ./spi_test  
SPI Read Data : Hello:30  
#
```

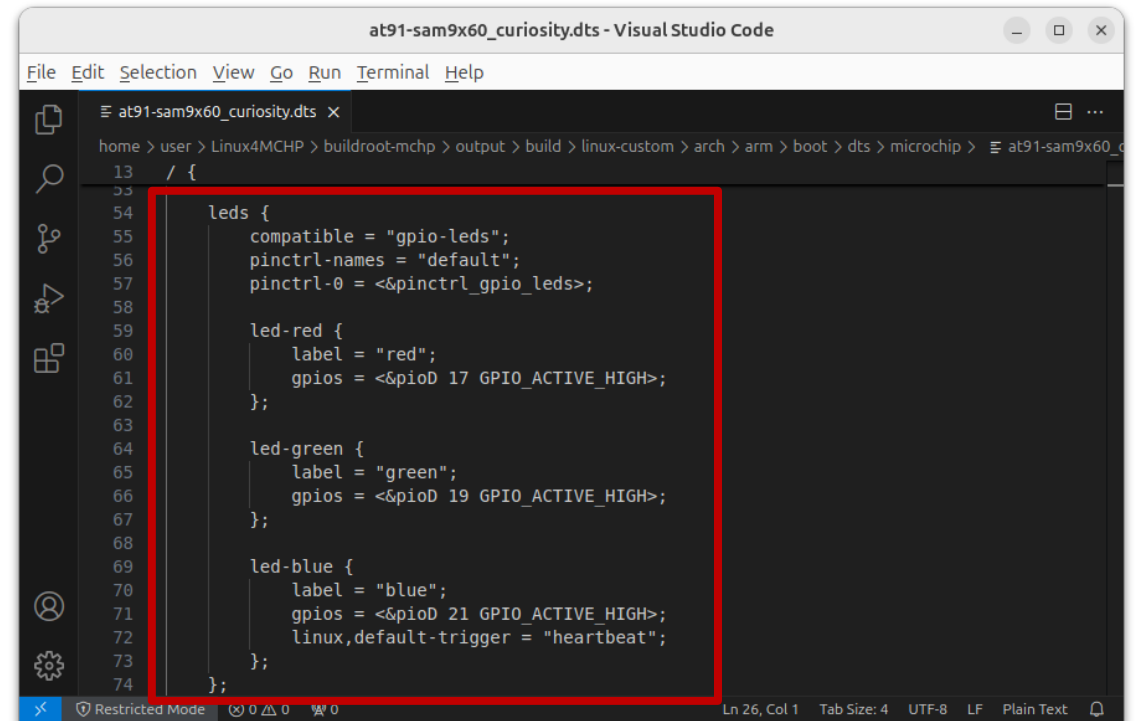
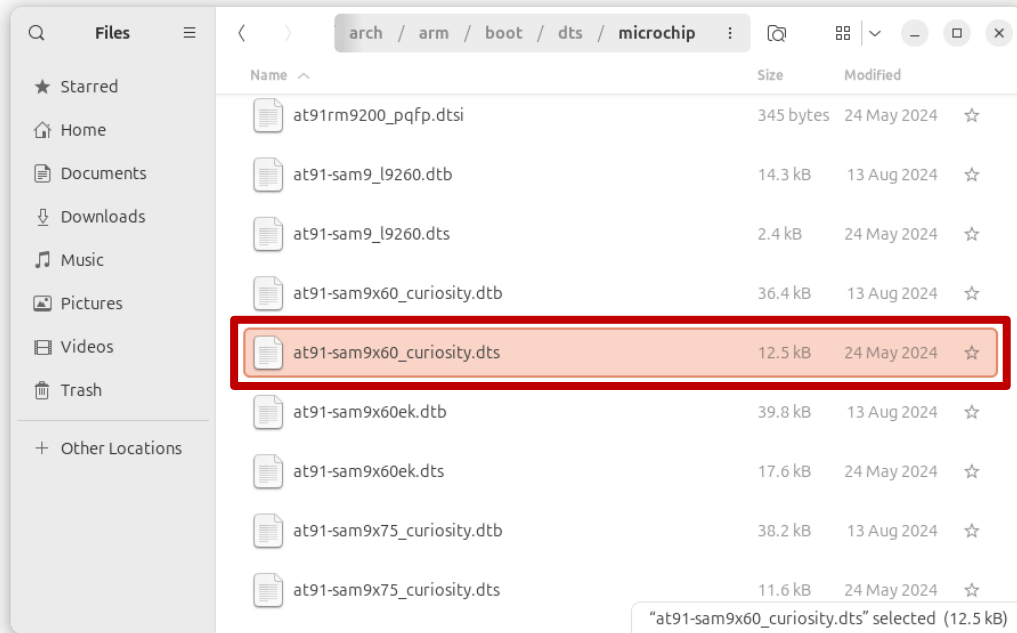
mikroBUS PIN	External
SCK	SPI EEPROM (25LC256), SCK
MISO	SPI EEPROM (25LC256), MISO
MOSI	SPI EEPROM (25LC256), MOSI
CS	SPI EEPROM (25LC256), CS
GND	GND

Appendix

Device Tree and Device Driver

Device-Tree-Source for LEDs

- **Check** the Device-Tree-Source for the LEDs .
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for LEDs

- Check the required device drivers.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```

2

3

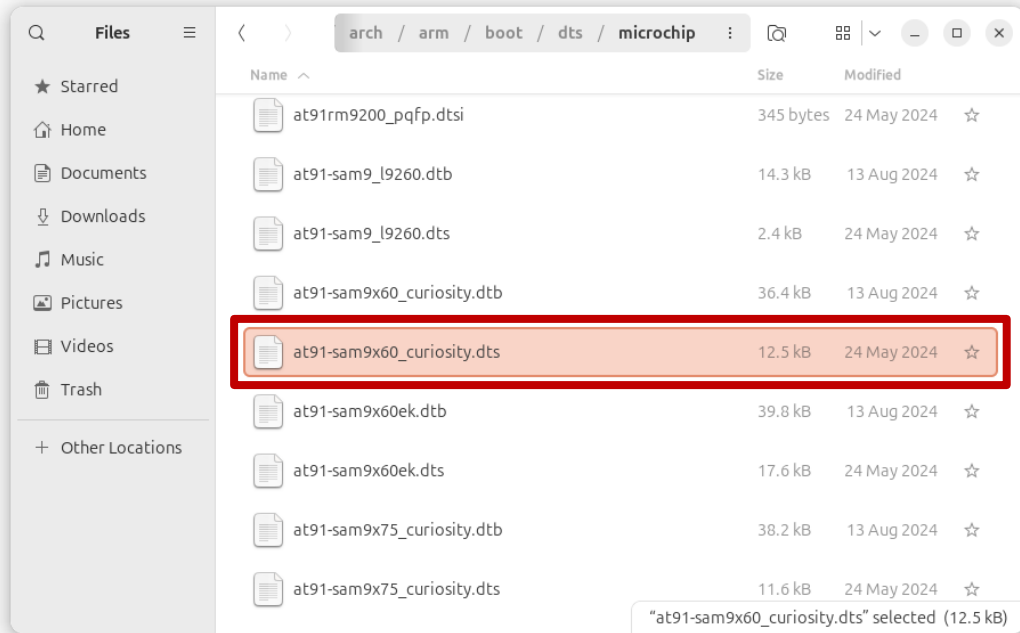
4

5

VM

Device-Tree-Source for ADC

- **Check** the Device-Tree-Source for the ADC.
(Use GUI or Command to open the file with Visual Studio Code)



```
at91-sam9x60_curiosity.dts - Visual Studio Code
File Edit Selection View Go Run Terminal Help
at91-sam9x60_curiosity.dts x
home > user > Linux4MCHP > buildroot-mchp > output > build > linux-custom > arch > arm > boot > dts > microchip > at91-sam9x60_curio
101
102
103 &adc {
104     vddana-supply = <&vdd1_3v3>;
105     vref-supply = <&vdd1_3v3>;
106     pinctrl-names = "default";
107     pinctrl-0 = <&pinctrl_adc_default &pinctrl_adtrg_default>;
108     status = "okay";
109 }
```

```
at91-sam9x60_curiosity.dts - Visual Studio Code
File Edit Selection View Go Run Terminal Help
at91-sam9x60_curiosity.dts x
home > user > Linux4MCHP > buildroot-mchp > output > build > linux-custom > arch > arm > boot > dts > microchip > at91-sam9x60_curio
256
257
258 &pinctrl {
259     adc {
260         pinctrl_adc_default: adc-default {
261             atmel,pins = <AT91_PIOB 14 AT91_PERIPH_A AT91_PINCTRL_NONE>;
262         };
263         pinctrl_adtrg_default: adtrg-default {
264             atmel,pins = <AT91_PIOB 18 AT91_PERIPH_B AT91_PINCTRL_PULL_UP>;
265         };
266     };
267 }
```

```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for ADC

- **Check** the required device drivers for the ADC.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```

2

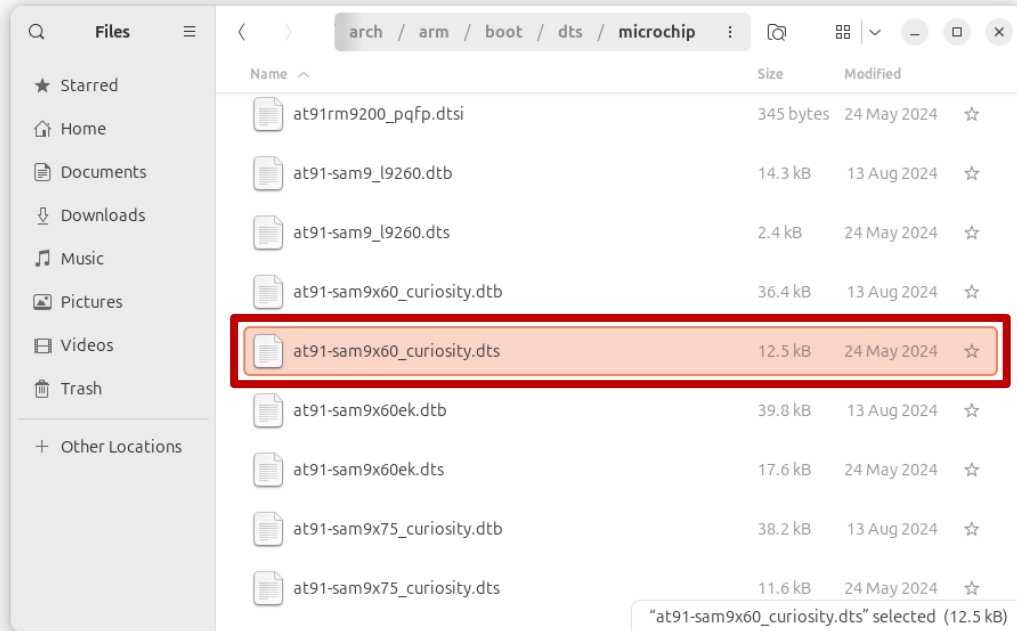
3

4

VM

Device-Tree-Source for PWM

- **Check** the Device-Tree-Source for the PWM.
(Use GUI or Command to open the file with Visual Studio Code)



```
at91-sam9x60_curiosity.dts - Visual Studio Code
File Edit Selection View Go Run Terminal Help
at91-sam9x60_curiosity.dts x
home > user > Linux4MCHP > buildroot-mchp > output > build > linux-custom > arch > arm > boot > dts > microchip > at91-sam9x60_curio
439 &pwm0 {
440     pinctrl-names = "default";
441     pinctrl-0 = <&pinctrl_pwm0_0 &pinctrl_pwm0_1 &pinctrl_pwm0_2>;
442     status = "okay";
443 };
444
```

```
at91-sam9x60_curiosity.dts - Visual Studio Code
File Edit Selection View Go Run Terminal Help
at91-sam9x60_curiosity.dts x
home > user > Linux4MCHP > buildroot-mchp > output > build > linux-custom > arch > arm > boot > dts > microchip > at91-sam9x60_curio
257 &pinctrl {
381     pwm0 {
382         pinctrl_pwm0_0: pwm0-0 {
383             atmel,pins = <AT91_PIOB 12 AT91_PERIPH_B AT91_PINCTRL_NONE>;
384         };
385     };
386     pinctrl_pwm0_1: pwm0-1 {
387         atmel,pins = <AT91_PIOB 13 AT91_PERIPH_B AT91_PINCTRL_NONE>;
388     };
389     pinctrl_pwm0_2: pwm0-2 {
390         atmel,pins = <AT91_PIOD 16 AT91_PERIPH_B AT91_PINCTRL_NONE>;
391     };
392 };
393
394
395
```

```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for PWM

- **Check** the required device drivers for the PWM.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```

2

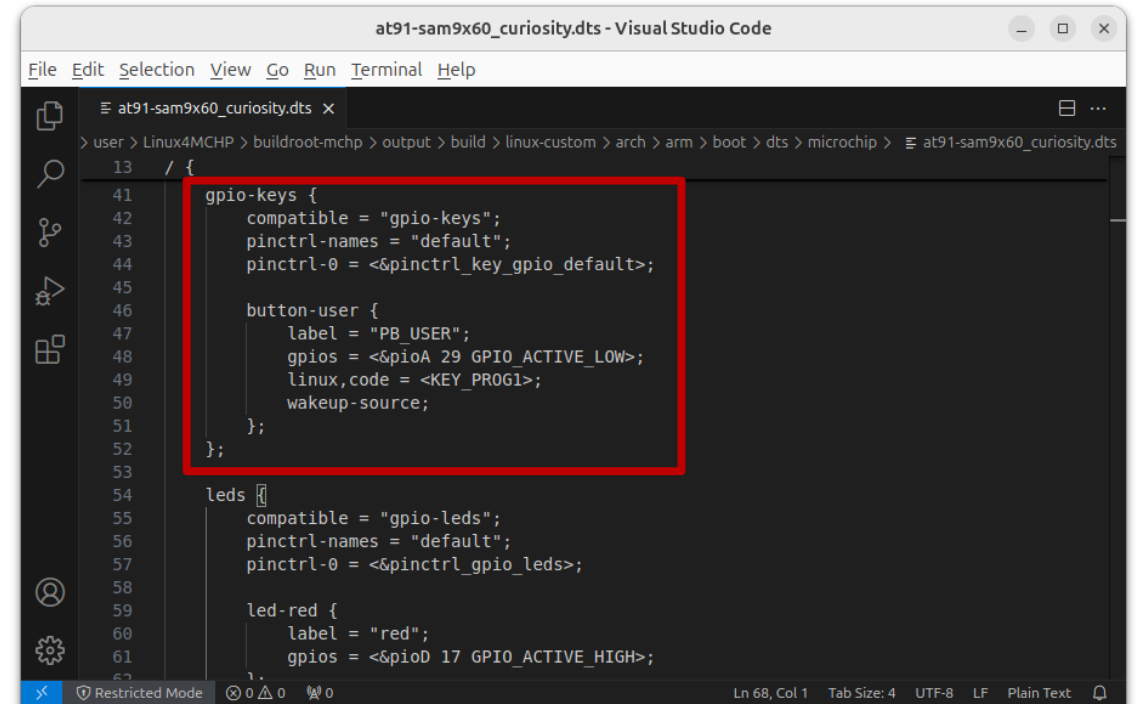
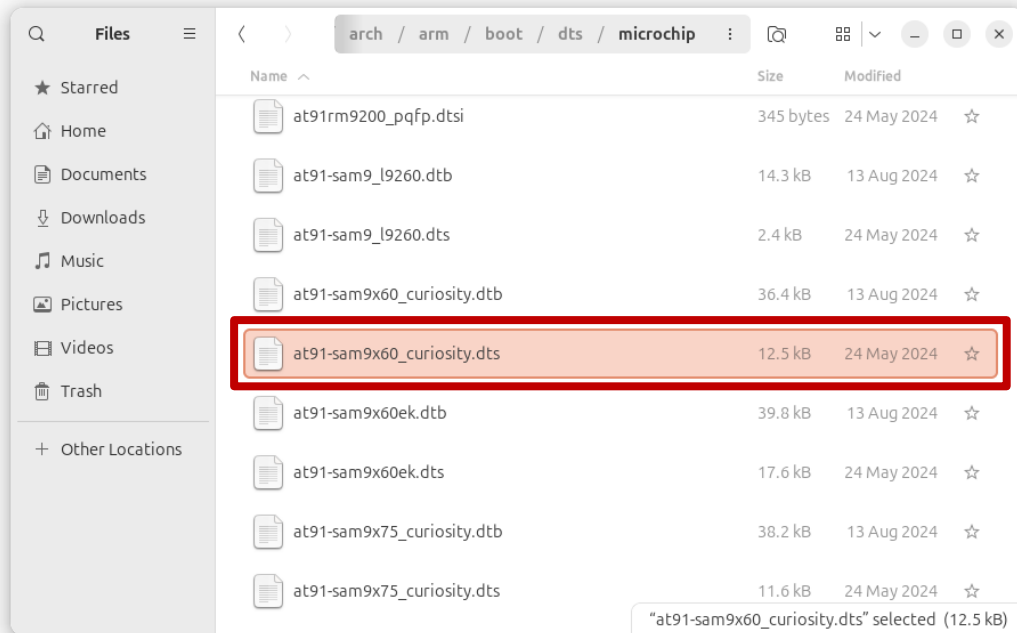
3

4

VM

Device-Tree-Source for Button

- **Check** the Device-Tree-Source for the GPIO-Keys.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for Button

- **Check** the required device drivers for the GPIO-Keys.

The image illustrates the process of configuring device drivers for GPIO keys in a Linux VM. It consists of several overlapping screenshots:

- Terminal Window (Step 1):** Shows the user navigating to the buildroot directory and running the configuration tool.

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```
- Kernel Configuration Menu (Step 2):** Shows the main menu where 'Device Drivers' is selected.
- Input Device Support (Step 3):** Shows the 'Input device support' submenu where 'Keyboards' is selected.
- GPIO Buttons (Step 4):** Shows the 'GPIO Buttons' submenu where 'GPIO driven matrix keypad support' is selected.

Each step is annotated with a yellow circle containing a number (1, 2, 3, 4). A black box labeled 'VM' is positioned at the bottom center of the terminal window.

Tools for Button

- **Check** the required tools for the GPIO-Keys.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make menuconfig
```

2

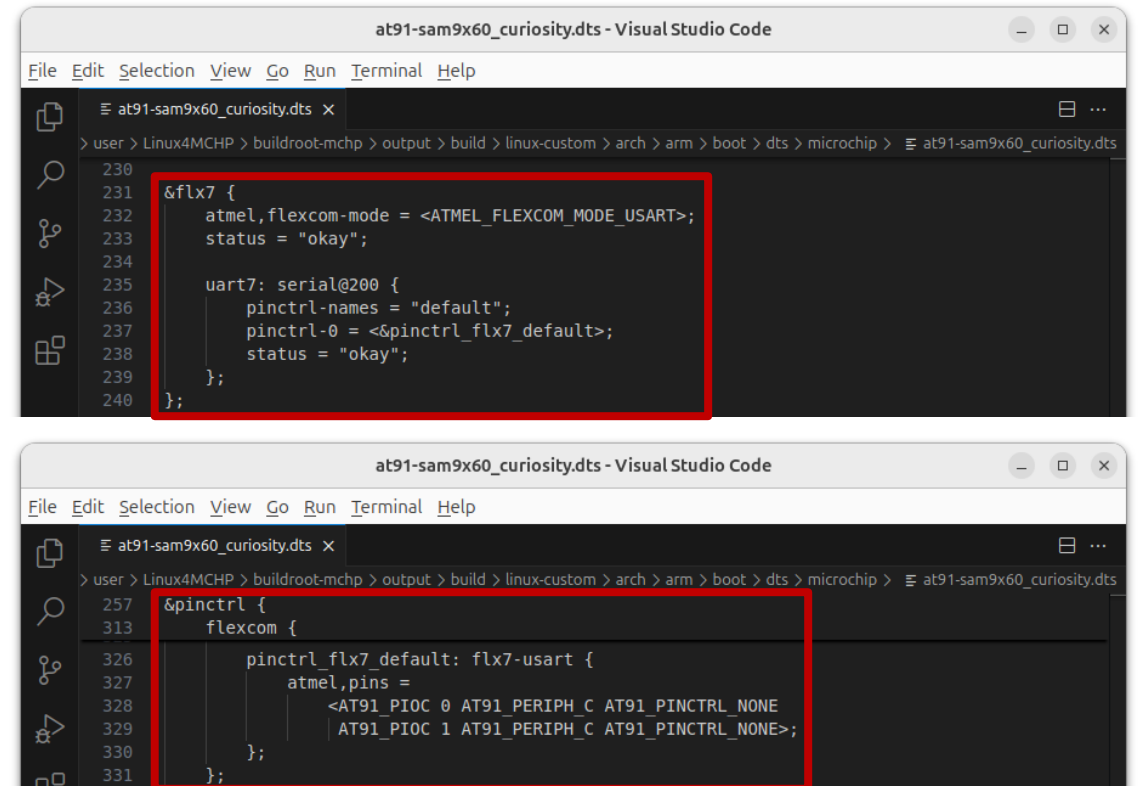
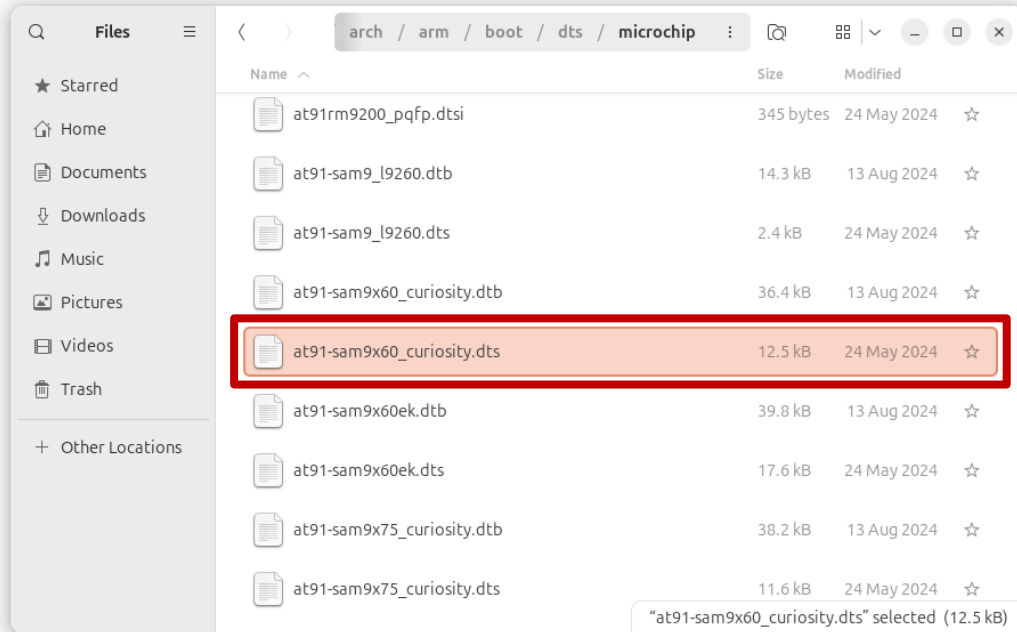
3

4

VM

Device-Tree-Source for FLEXCOM - UART

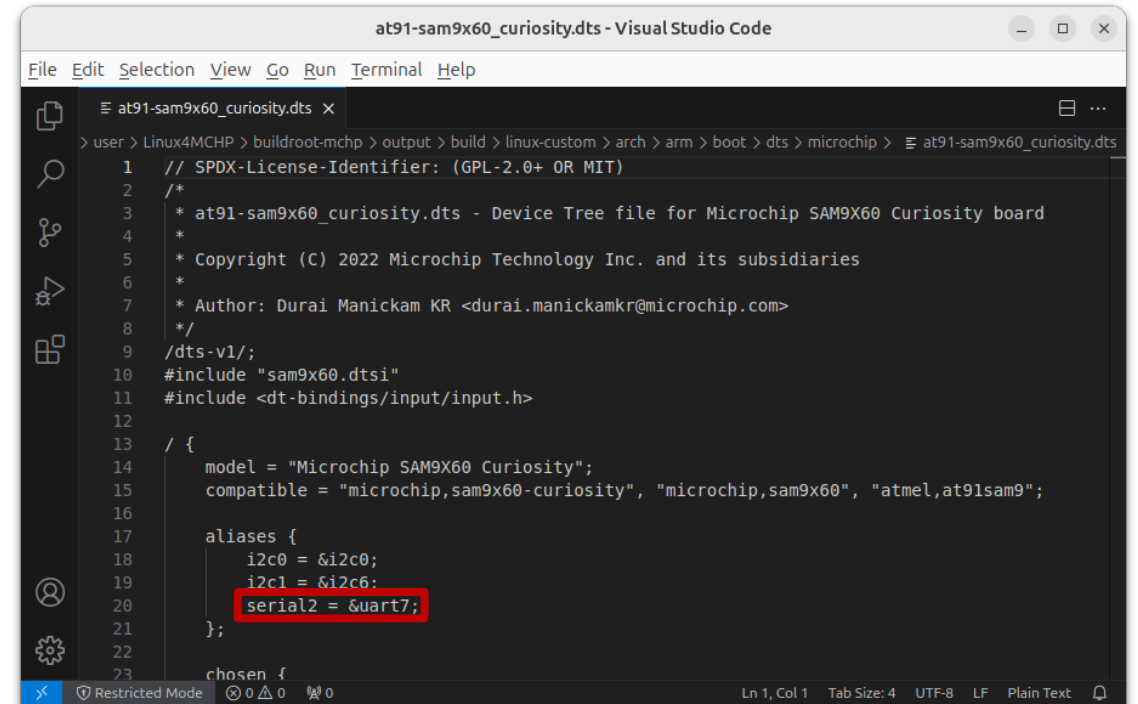
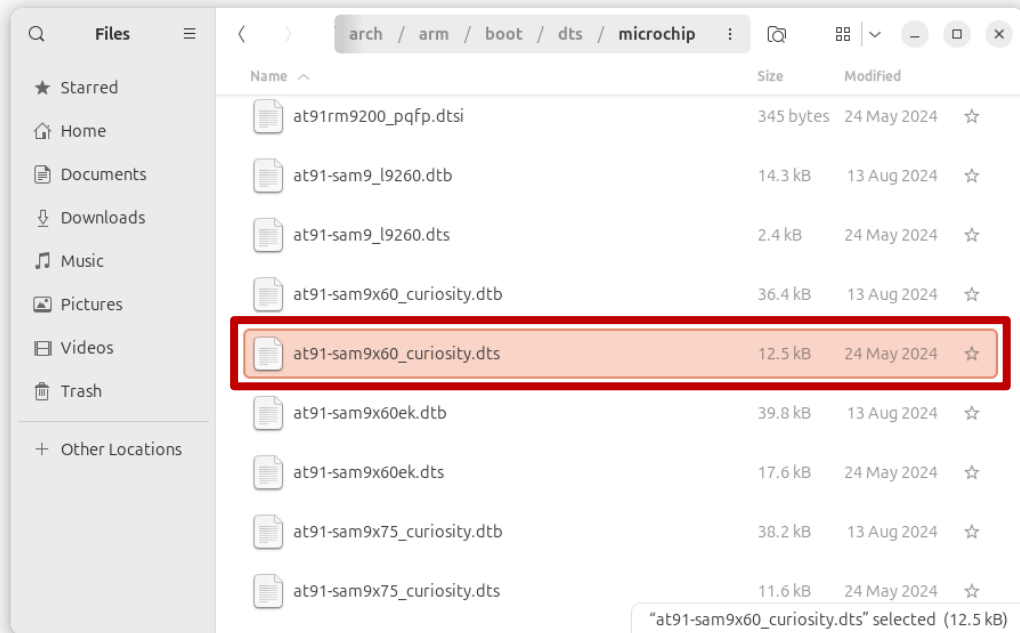
- **Check** the Device-Tree-Source for the UART.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Tree-Source for FLEXCOM - UART

- **Check** the Device-Tree-Source for aliases of the UART (serial2, ttyS2).
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for FLEXCOM - UART

- **Check** the required device drivers for the UART.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```

2

3

4

5

VM

Tools for FLEXCOM - UART

- **Check** the required tools (microcom) for the UART.

1

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make busybox-menuconfig
```

2

BusyBox 1.36.1 Configuration

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < >

- Init Utilities --->
- Login/Password Management Utilities --->
- Linux Ext2 FS Progs --->
- Linux Module Utilities --->
- Miscellaneous Utilities --->**
- Networking Utilities --->
- Print Utilities --->
- Mail Utilities --->
- Process Utilities --->

<Select> < Exit > < Help >

3

BusyBox 1.36.1 Configuration

Miscellaneous Utilities

Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [] excluded <M> module < >

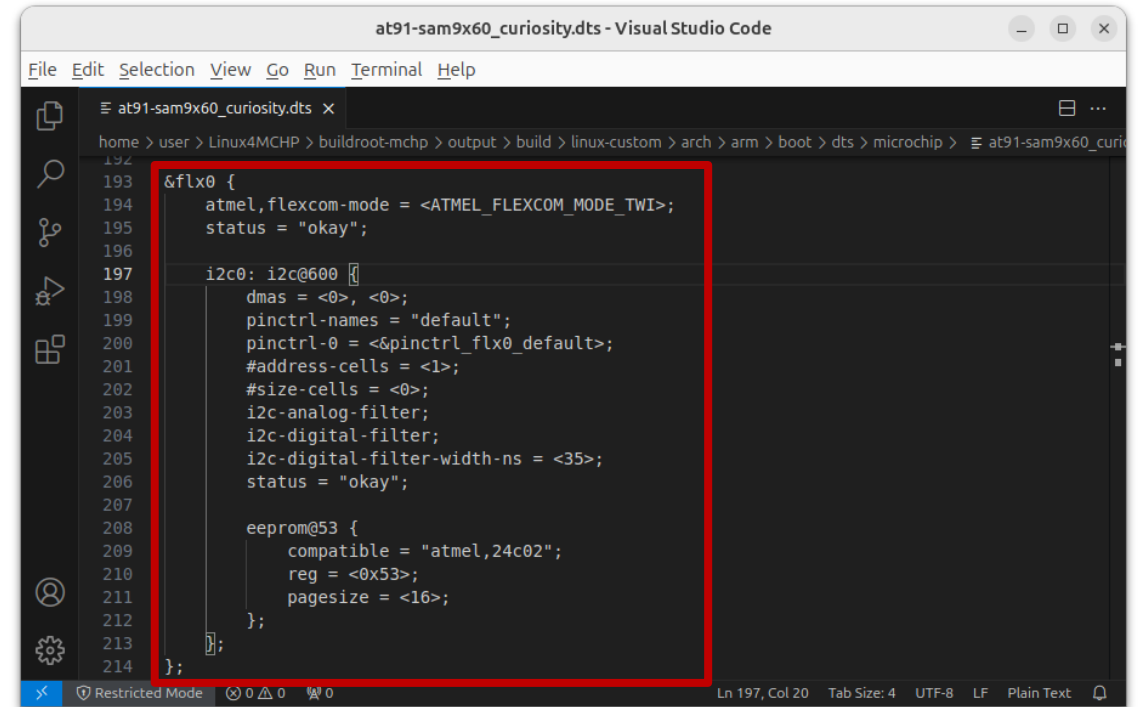
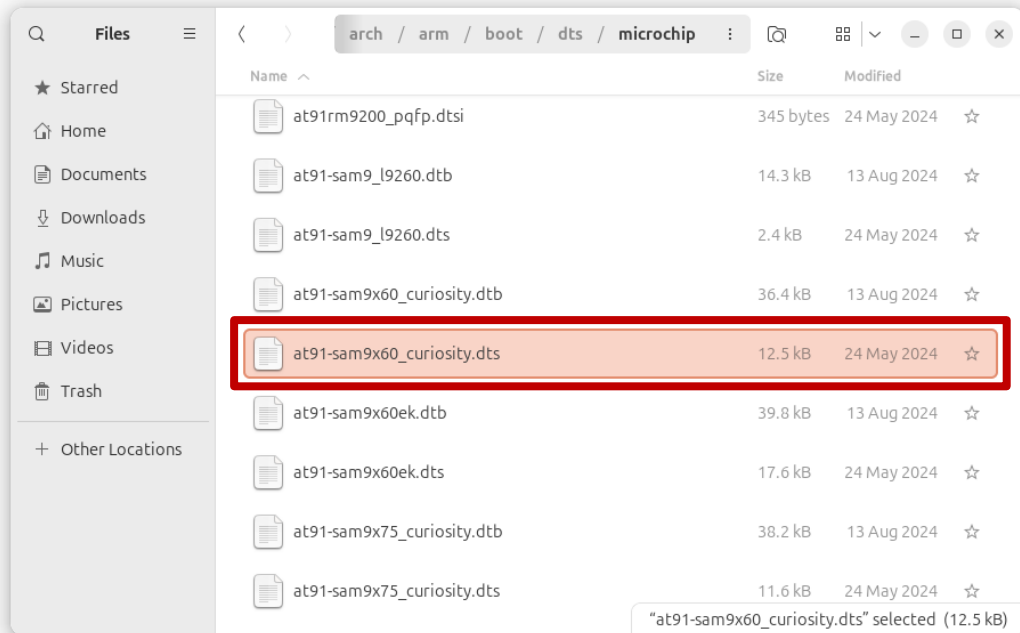
- [*] lsscsi (2.5 kb)
- [*] makedevs (9.2 kb)
- Choose makedevs behaviour (table) --->
- [*] microcom (5.7 kb)**
- [] min (0.5 kb)
- [*] mt (2.5 kb)
- [] nandwrite (4.8 kb)
- [] nanddump (5.2 kb)
- [*] partprobe (3.5 kb)

<Select> < Exit > < Help >

VM

Device-Tree-Source for FLEXCOM - I²C

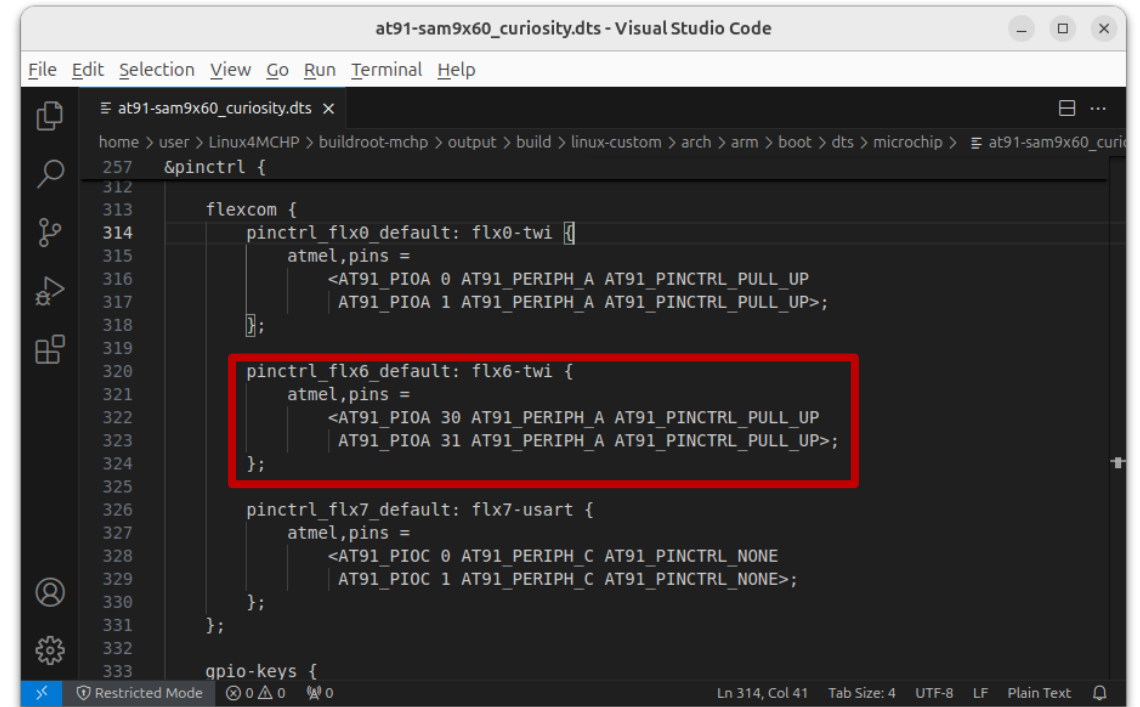
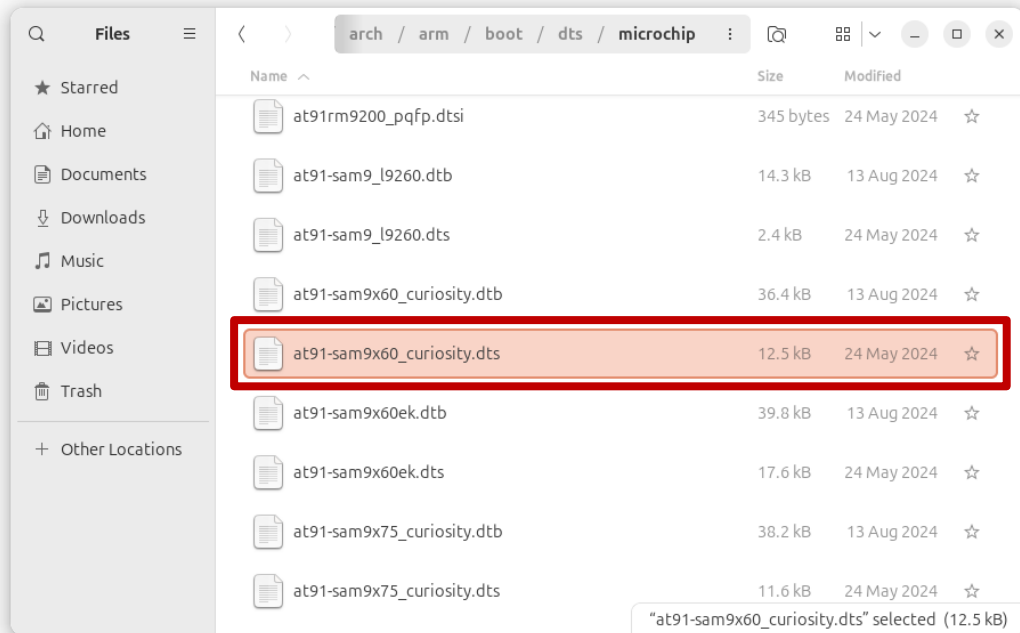
- **Check** the Device-Tree-Source for the I²C.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Tree-Source for FLEXCOM - I²C

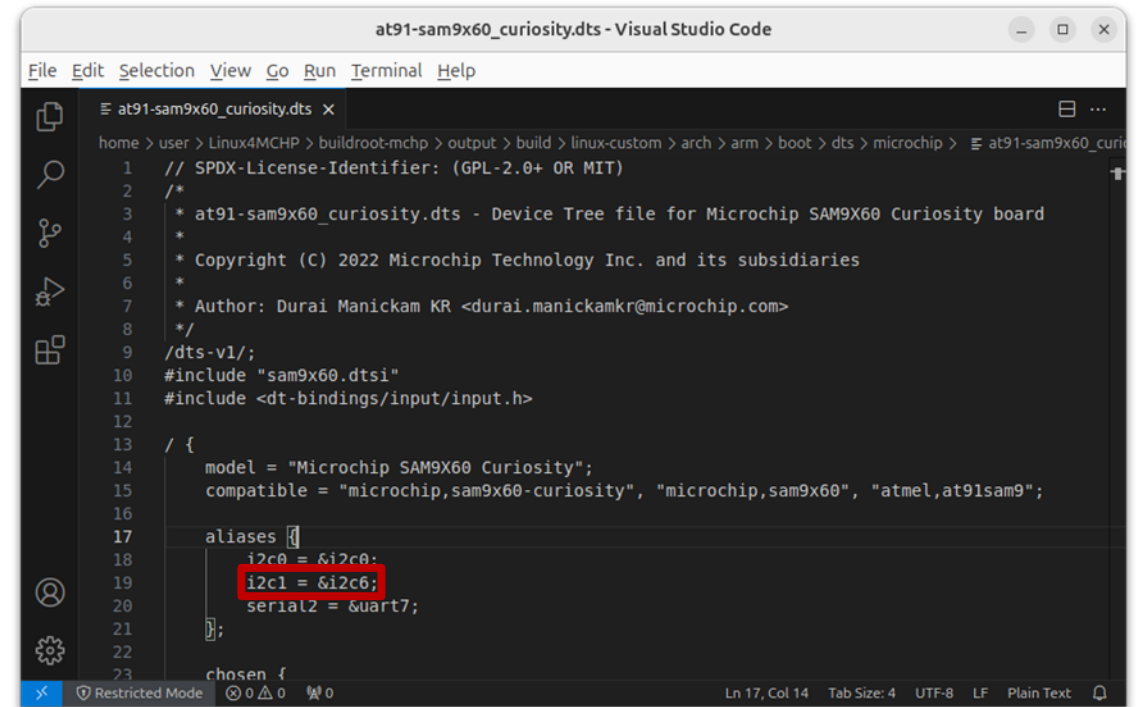
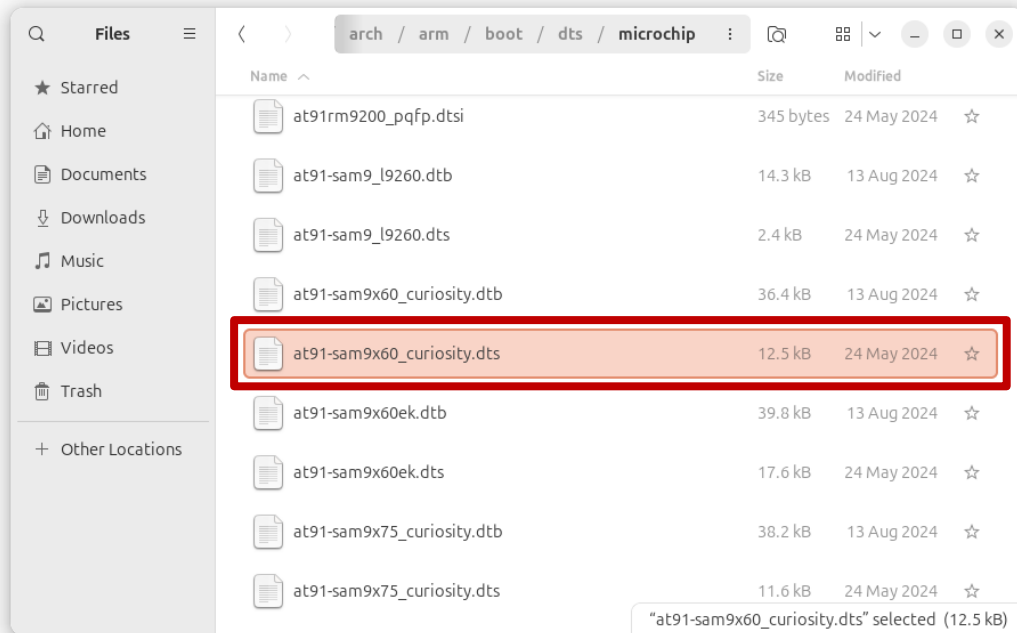
- **Check** the Device-Tree-Source for the I²C.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Tree-Source for FLEXCOM - I²C

- **Check** the Device-Tree-Source for the I²C.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
```

Device-Driver for FLEXCOM - I²C

- **Check** the required device drivers for the I²C.

The image illustrates the process of checking and configuring I²C device drivers in a Linux VM. It consists of a terminal window and three screenshots of the kernel configuration menu.

Terminal Window (Step 1): The terminal shows the user navigating to the build directory and running the configuration command:

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make linux-menuconfig
```

Kernel Configuration Screenshots (Steps 2-4):

- Step 2:** The 'Device Drivers' menu is highlighted.
- Step 3:** The 'I2C support' menu is highlighted.
- Step 4:** The 'Atmel AT91 I2C Two-Wire interface (TWI)' option is highlighted.

The terminal window is labeled 'VM' in a black box at the bottom right.

Tools for FLEXCOM - I²C

- **Check** the required tools (i2c-tools) for the I²C.

The image illustrates the process of configuring i2c-tools in a VM environment. It consists of several overlapping screenshots:

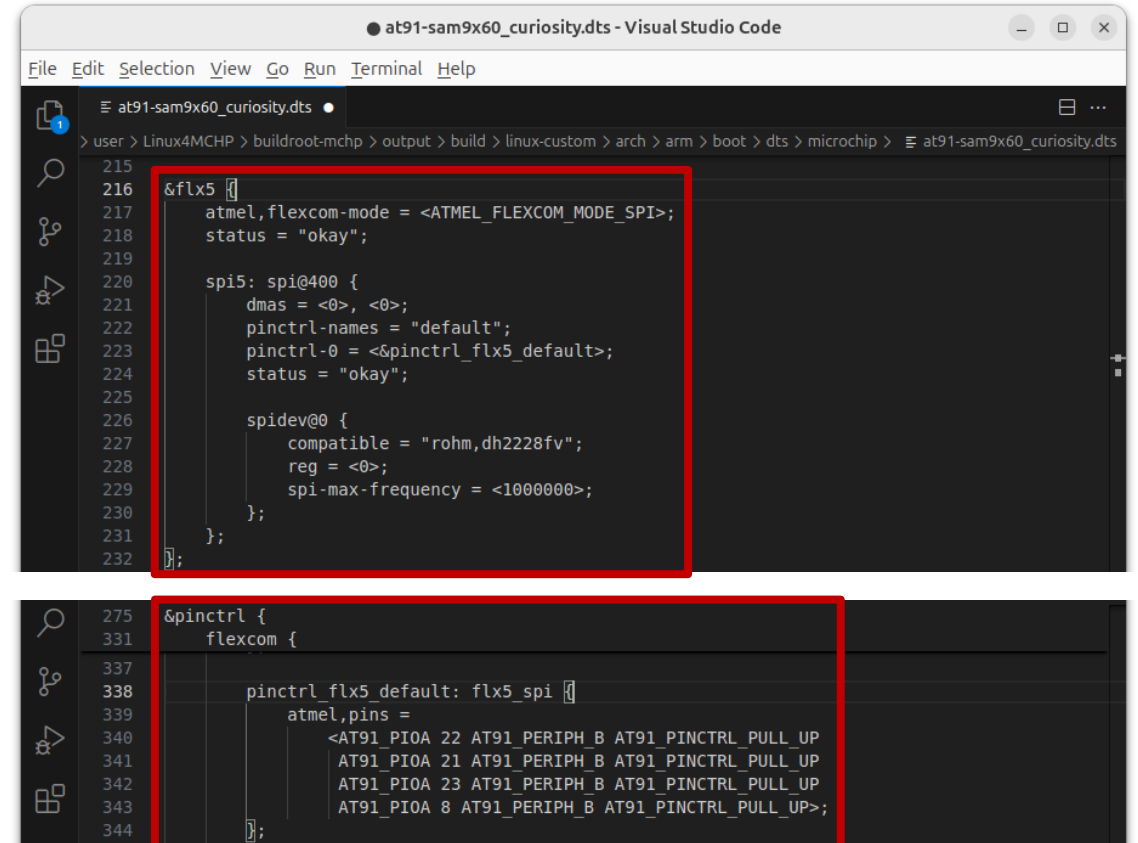
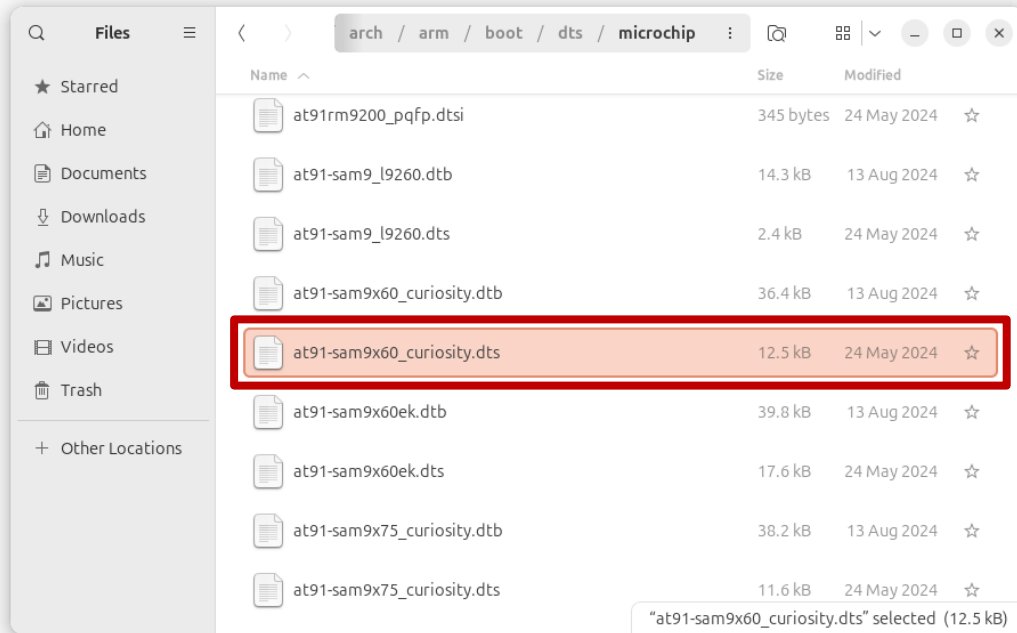
- Terminal Window (Left):** Shows the user navigating to the buildroot directory and running the configuration command. A red circle with the number '1' is next to the first command.

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make menuconfig
```
- Buildroot Configuration Menu (Top Right):** Shows the 'Target packages' menu. A red circle with the number '2' is next to the 'Target packages' option, which is highlighted with a red box.
- Hardware Handling Menu (Right):** Shows the 'Hardware handling' menu. A red circle with the number '4' is next to the 'i2c-tools' option, which is highlighted with a red box.
- Multifunction Device Drivers Menu (Bottom Right):** Shows the 'Multifunction device driver' menu. A red circle with the number '3' is next to the 'AT91 USART Driver' option, which is highlighted with a red box.

A black box with the text 'VM' is located at the bottom right of the terminal window.

Device-Tree-Source for FLEXCOM - SPI

- **Check** the Device-Tree-Source for the SPI.
(Use GUI or Command to open the file with Visual Studio Code)



```
user@VM:~$ code ~/Linux4MCHP/buildroot-mchp/output/build/linux-custom/arch/arm/boot/dts/microchip/at91-sam9x60_curiosity.dts
(with added spi node)
```

Device-Driver for FLEXCOM - SPI

- **Check** the required device drivers for the SPI.

The image illustrates the process of checking and enabling required device drivers for SPI in a Linux kernel configuration. It consists of a terminal window and three screenshots of the kernel configuration menu.

Terminal Window:

```
user@VM: ~  
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp  
user@VM:~$ make linux-menuconfig
```

Kernel Configuration Screenshots:

- Step 1:** The terminal window shows the command `make linux-menuconfig` being executed.
- Step 2:** The first screenshot shows the `Linux/arm 6.6.23-linux4microchip-2024.04 Kernel Configuration` menu. The `Device Drivers` option is highlighted with a red box and a yellow circle labeled '2'.
- Step 3:** The second screenshot shows the `Device Drivers` menu. The `[*] SPI support` option is highlighted with a red box and a yellow circle labeled '3'.
- Step 4:** The third screenshot shows the `SPI support` menu. The `<*> Atmel SPI Controller` option is highlighted with a red box and a yellow circle labeled '4'.

VM

Tools for FLEXCOM - SPI

- **Check** the required tools (spi-tools) for the SPI.

The image illustrates the process of configuring buildroot for SPI tools, showing a terminal window and three screenshots of the buildroot configuration menu.

Terminal Window (Step 1):

```
user@VM:~$ cd ~/Linux4MCHP/buildroot-mchp
user@VM:~$ make menuconfig
```

Buildroot Configuration Menu (Step 2):

The menu shows the following options:

- Target options --->
- Toolchain --->
- Build options --->
- System configuration --->
- Target packages --->**
- Package management --->
- Bootloaders --->
- Host utilities --->

Target packages menu (Step 3):

The menu shows the following options:

- Filesystem and flash utilities --->
- Fonts, cursors, icons, sounds and themes --->
- Games --->
- Graphics applications (graphic/text) --->
- Hardware handling --->**
- Interpreter languages and scripting --->
- Libraries --->
- Mail --->
- Miscellaneous --->
- Networking applications --->

Hardware handling menu (Step 4):

The menu shows the following options:

- [] sigrok-cli
- [] sispnctl
- [] smartmontools
- [*] spi-tools**
- [] stl
- [] statserial
- [] stm32flash
- [] sunxi-cedarx

*** sunxi-mali-utgard needs an EABIhf glibc toolchain ***

Navigation keys: <select> <Exit> <Help> <Save> <Load>

VM

Microchip Trademarks

<https://www.microchip.com/en-us/about/legal-information/microchip-trademarks>

Trademarks	Description/Appropriate Nouns	Notes
Adaptec® "a" logo		Registered in other countries (not US)
Adaptec®	products, solutions, cables, software, drivers, adapters, controllers, series, family	
Adjacent Key Suppression™	technology	
AgileSwitch®	digital programmable gate driver, gate driver, drivers, family, device	US Only
AKS™	technology	
Analogue-for-the-Digital Age™	AIPD family tagline	
Any Capacitor™	stable	
AnyIn™	feature, structure, capability	
AnyOut™	feature, capability	
Augmented Switching™	technology	
AVR®	devices, microcontrollers, MCUs, CPU, architecture, family	
AVR Logo		
AVR Freaks®	forum	
BesTime®	technology	
BitCloud®	SDK	
BlueSky™	firewall, subscription service, technology	
BodyCom™	technology, system, development kit	
ClockStudio™	application, software, program, utility	
ClockWorks®	Configurator, products, series, family, devices	US Only
CodeGuard™	security	
CryptoAuthentication™	development library, library, devices, ICs, family	
CryptoAutomotive™	technology, security ICs, development kits	
CryptoCompanion™	chip	
CryptoController™	solution, Trusted Platform Module (TPM)	
CryptoMemory®	cryptographic security ICs, ICs	

NOTE: *Always check with the Legal Department for the most current trademarks.

Trademarks	Description/Appropriate Nouns	Notes
PIC ³² Logo		
PICDEM™	demonstration board	
PICDEM.net™	Internet/Ethernet demonstration board	
PICKit™	starter kit, programmer, debugger, in-circuit debugger, development tool, tool, on-board	
picoPower®	technology	
PICSTART®	development programmer	
PICtail™	daughter board, board	
PolarFire®	FPGA, family, kits, devices, SoC, System-on-Chip	
Power MOS IV™		
Power MOS 7™		
Powermite 3®	package	US Only
PowerSmart™	digital control library designer, development suite	
Precision Edge®	family, product family	US Only
ProASIC®	FPGA, architecture, board, kit, family	US Only
ProASIC Plus®	FPGA, devices, family, architecture, switch	US Only
ProASIC Plus Logo		US Only
Prochip Designer®	software suite	
PureSilicon™	oscillator, technology	
QMatrix™	sensor, devices	
QTouch®	sensor, devices, library, tools, development platform, project builder, Composer, Project Builder, board, solution, technology, Configurator	
Quiet-Wire®	enhanced EMI technology, family, technology	
Ripple Blocker™	technology, attenuators, product line, family of linear regulators	
REAL ICE™	In-circuit emulator	
RTAX™	FPGAs	
RTG4™	FPGAs	
SAM-BA®	software, application, monitor	

NOTE: *Always check with the Legal Department for the most current trademarks.

Trademarks	Description/Appropriate Nouns	Notes
CryptoRF®	transponder, devices	
dsPIC®	Digital Signal Controllers (DSCs)	
dsPIC Logo		
dsPICDEM™	demonstration board, development board	
dsPICDEM.net™	board	
Dynamic Averaged Matching™ (DAM™)	architecture	
ECAN™	technology, solution	
Espresso T1S™	device	
EtherGREEN™	platform, technology, family	
EtherSynch®	platform, technology, family	US Only
EyeOpen™	cable compensation	
Flashtec™	controller, products, family	US Only
flexPVR®	technology	
Frequency on Demand®		Registered in other countries (not US)
GestIC®	technology, sensing, controllers	
GridTime™	device, time server, GNSS time server	
HELDO®	family, regulators	
Hyper Speed Control®	family, architecture	US Only
HyperLight Load®	mode	US Only
ICSP™ or In-Circuit Serial Programming™	programming capability	
IdealBridge™	rectifier, dual MOSFET-bridge rectifier	
IGAT™	board, demonstration board	
IGLOO®	FPGAs, family, devices, series	
INICNet™	technology, sniffer	
Intelligent Paralleling™	technology	
IntelliMOS™	family, power stage family	US Only
Inter-Chip Connectivity™	technology	
JitterBlocker™	family, attenuators	
JukeBlox®	technology, platform	

NOTE: *Always check with the Legal Department for the most current trademarks.

Trademarks	Description/Appropriate Nouns	Notes
SAM-ICE™	emulator	
SenGenuity®	sensors, products	
Serial Quad I/O™ or SQI™	family, flash device, product	
Silicon Storage Technology®		Japan only
SimpleMAP™	protocol	
SimpliPHY™	family, products, devices	
SmartBuffer™	devices, ICs	
SmartFusion®	FPGA, evaluation kit, SoC, System-on-Chip,	US Only
SmartHLS™	IDE, compiler software, compiler, software	
SMART-I.S.™	technology	
SpyNIC®	device	
SST®		
SST Logo		
storClad™	encryption technology, technology	
SuperFlash®	technology, kit, macro, device, memory	
SuperSwitcher™	family, regulator, buck regulator	
SuperSwitcher II™	family	
Switchtec™	technology, family, switch(es), product line	China only
Symmmcom®		
Symmetricon®		
SynchroPHY™	family, products, devices	
SyncServer®	server, instrument, clock, oscillator, device	
SyncWorld®	timing, Ecosystem Program	US only
Tachyon®	products, family, controllers, platform, technology	
The Embedded Control Solutions Company®	MarCom Use Only	
TimeCesium®	products, primary frequency reference	US only
TimeHub®	platform, system	US only
TimePictra®	software suite, platform, Synchronization Management System	US only

NOTE: *Always check with the Legal Department for the most current trademarks.

Trademarks	Description/Appropriate Nouns	Notes
JukeBlox with design	Registered word mark JukeBlox with a design	Image is not registered, this is a graphic treatment with the word mark
KEELOQ®	security ICs, protocol, technology	
Kleer®	technology	
Kleer with design	Registered word mark Kleer with a design	Image is not registered, this is a graphic treatment with the word mark
Knob-on-Display™/KoD™	technology, solution, HMI, user interface, touch controller, family, devices, Widget Configurator, Knob Designer, Position Wizard	
LANCheck®	online design review, online review, review	
Libero®	SoC Design Suite, tools, software, design files, IDE, designer	US registration
LinkMD®	cable diagnostics, technology, family, diagnostics engine	
MarginLink™	signal integrity testing	
maxCrypto™	controller-based encryption, encryption	
maXStylus™	solution	
maXTouch®	touchscreen controllers, technology, studio, family	
maxView™	storage manager, tools, drivers	
MediaLB®	bus, controller, device	
megaAVR®	devices, MCUs	
memBrain™	neuromorphic memory product, product, solution, technology	
Microchip Logo		
Microchip Name and Logo		
Microsemi Logo		
Microsem®		

NOTE: *Always check with the Legal Department for the most current trademarks.

Trademarks	Description/Appropriate Nouns	Notes
TimeProvider®	series, server, clock, products, grandmaster, ePRTC, enhanced PRTC	US only
TimeSource®	Enhanced PRTC	
tinyAVR®	microcontroller, MCU	
Total Endurance™	software model	
Trusted Time™	technology	
TSHARC™	touch screen controller, controller	
Turing™	hardware security module, programmer	
UNIQ®	memory chips, bus, hardware port, communication protocol, firmware	
USBCheck™	design review	
VariSense™	technology	
VectorBlox™	accelerator, software development kit, SDK, intellectual property, IP, solution, platform, tools, tool chain, tool flow	
Vector®		
VeriPHY™	cable diagnostics algorithm, cable diagnostics software suite	
ViewSpan™	technology	
WiperLock™	technology	
XMEGA®	microcontroller, MCU, devices, Custom Logic (XCL), family, series	
XpressConnect™	retimer	
ZENA™	software, analyzer or wireless adapter	
ZL®		

NOTE: *Always check with the Legal Department for the most current trademarks.

Microchip Technology Inc. Servicemarks*

Trademarks	Description/Appropriate Nouns	Notes
SQTP™	Serial Quick Turn Programming capability	

NOTE: *Always check with the Legal Department for the most current servicemarks.

MICROCHIP RESERVES THE RIGHT TO CHANGE THESE GUIDELINES SOLELY AT ITS DISCRETION.

Trademarks	Description/Appropriate Nouns	Notes
MindI™	analog simulator, analog simulator tool	
MIWI™	wireless networking protocol, protocol, network, stack, software, applications, mesh, coordinator	
MOST®	technology, NetServices, evaluation kit, board, system, device, product, network, specification, platform, cooperation, design	
MOST Logo		
motorBench®	Development Suite	US Only
MPASM™	assembler	
MPF™	products, devices	
MPLAB®	Integrated Development Environment, In-Circuit Debugger, In-Circuit Emulator, IDE, ICD, REAL ICE™ in-circuit emulator, starter kit, compiler, XC compiler, C compiler, Harmony, Integrated Programming Environment, IPE, Xpress, Code Configurator, development ecosystem, PICkit™, server, library, Snap, Connect Configurator, Network Creator, Analog Designer, Discover, MindI™, Analysis Tool Suite, Discover, Universal Device Programmer, Machine Learning Development Suite, PTG, Programmer-to-Go, Programmer-to-Go Mobil App, SiC Power Simulator	
MPLAB Certified Logo		
MPLIB™	object librarian or librarian	
MPLINK™	object linker or linker	
mSIC™	MOSFETs, diodes, products, gate drivers, modules, bare die, technology, solutions	
mTouch®	solution, sensing solution, evaluation kit, technology, development kit, library	US Only
MultiTRAK™	technology	
NetDetach™	technology	
Omniscient Code Generation™ (OCG)	compilation technology	
OptoLyzr®	tool, network analysis tool, Studio	
PIC®	microcontroller, MCU, device(s), assembler	

NOTE: *Always check with the Legal Department for the most current trademarks.

