Integrated development environment for teaching and research on RISC-V processors (PDC2023-145832-I00)

Integrated development environment for teaching and research on RISC-V processors

CREATOR

D 2.2
User manual, use cases, and materials for microcontrollers and RISC-V boards

Universidad Carlos III de Madrid

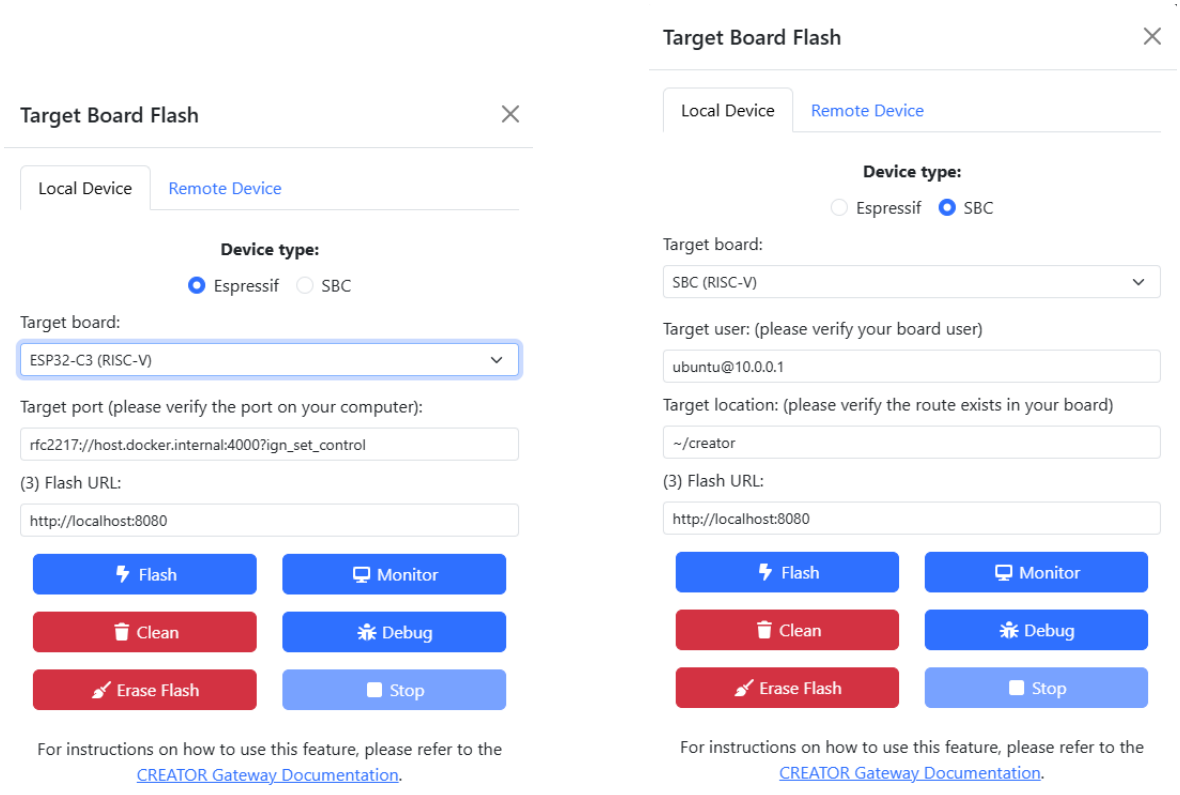
January, 2026

CONTENTS

1. TARGET FLASH MENU USER MANUAL

1.1. Target Flash Menu

CREATOR has real-hardware support for multiple RISC-V processors devices: Espressif and SBC. This is the *Target Board Flash* menu, with all its buttons explained:



(i) Target board ESP32 setup.

(ii) Target board SBC setup.

Figure 1.1.1: Target board setups.

1.1.1. Espressif ESP32

1. **Target Board**: Select between the Espressif ESP32 devices available.
2. **Target Port:** Checkout how does your device recognize UART device connection.
   (a) **Linux:** `/dev/ttyUSB0`.
   (b) **macOS:** Starts with `/dev/cu.usbserial-`.
   (c) **Windows:** `COM3`, but native Windows compatibility is NOT AVAILABLE. Checkout Docker Windows.

3. **Flash URL:** URL direction where the driver will be displayed. By default, is `localhost:8080`, but the user can change if need it.

1.1.2. SBC

1. **Target Board**: Select between the SBC available.
2. **Target User:** User and IP address of the SBC (e.g. `<user>@<ip>`).
3. **Target Location:** Location of the project folder (e.g. `~/creator`).
4. **Flash URL:** URL direction where the driver will be displayed. By default, is `localhost:8080`, but the user can change if need it.

1.1.3. Buttons

1. **Flash:** Builds and flashes CREATOR's program into your development board.
2. **Monitor:** Executes development board's flashed program.

   In ESP32, it can also be stopped with the keyboard shortcuts `Ctrl+]` or `Ctr+T+X`.

3. **Debug**: If all good connected, it will open another tab with GDB UI ready to execute step-to-step programs.
4. **Clean**: Erases computer's copy of the build program.
5. **Erase-flash:** Erases program inside the development program.

2. ESPRESSIF MICROCONTROLLER USER MANUAL

2.1. Espressif ESP32 RISC-V Support

At the moment, CREATOR supports these processors and development boards:

- ESP32-C3:
  - ESP32-C3-DevKitC-02 (includes JTAG).
  - ESP32-C3-DevKitM-1 (no JTAG detected).
- ESP32-C6:
  - ESP32-C6-DevKitC-1 (JTAG + port included).
  - ESP32-C6-DevKitM-1 (JTAG + port included).
- ESP32-H2:
  - ESP32-H2-DevKitM-1 (JTAG + port).

2.2. Native ESP32 Driver Execution

The use of the native version of the driver needs to be in a Linux/macOS environment with the following prerequisites:

1. **ESP-IDF framework**: This driver uses specifically the v.5.3.2 version of the driver, which can be downloaded here: `https://github.com/espressif/esp-idf/releases/tag/v5.3.2`.
2. **Python version from 3.9 to 3.11**: As it's an old version, we recommend creating and using a virtual environment.
   - **Install Python 3.9:**

     ```
     sudo apt install software-properties-common
     sudo add-apt-repository ppa:deadsnakes/ppa
     sudo apt install python3.9
     ```

   - **Create virtual environment:**

     ```
     python3.9 -m venv ~/.espressif/python_env/idf5.3_py3.9_env
     source ~/.espressif/python_env/idf5.3_py3.9_env/bin/activate
     ```

   - **Check and erase if you have newer Python environment versions (by default it will choose the newer one):**

     ```
     rm -rf ~/.espressif/python_env/idf5.3_py3.10_en
     ```

3. **Install debug dependences:** OpenOCD is already installed in ESP-IDF, but to debug it, it is necessary to install gdbgui:

```
pip3 install gdbgui
```

IMPORTANT: When using ESP32-C6 or ESP32-H2 development boards, only another USB-C cable is necessary. However, ESP32-C3 devices need an extra USB-to-DIP device to work, as shown in the Figure 2.2.1:

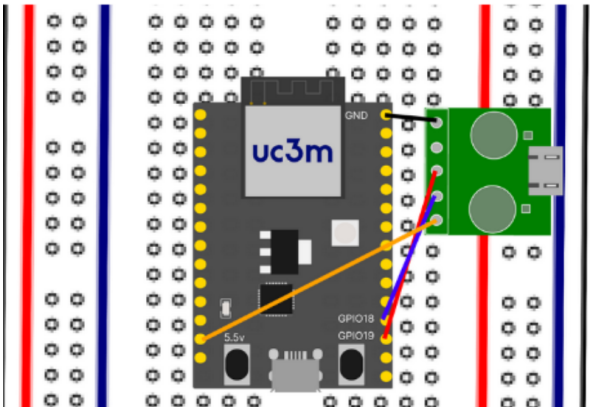


Figure 2.2.1: ESP32-C3 JTAG installation with USB-to-Dip.

Checkout your **board's debug preferences** here: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32c3/get-started/index.html#what-you-need`.

4. **Install driver's dependences:**

```
pip3 install flask flask_cors
```

5. Download the **driver linked into the prerequisites** page.

6. Install ESP-IDF dependencies and export ESP-IDF symbols using Espressif's scripts:

   Install ESP-IDF dependencies inside the virtual environment:

```
$HOME/esp-idf-v5.3.2/install.sh
```

   Load the environment variable for your board with:

```
. $HOME/esp-idf-v5.3.2/export.sh
```

7. **Execute** the gateway [1]:

```
python3 gateway.py
```

[1] `https://github.com/creatorsim/creator-gateway-esp32/blob/main/esp32c3/gateway.py`

2.2.1. Native Debug

To debug locally on your computer, check these conditions:

1. Check out **ports permissions**: To flash and debug these development boards, your user must be in plugdev and dialout (in Ubuntu). In ARCH, there is a uucp user.

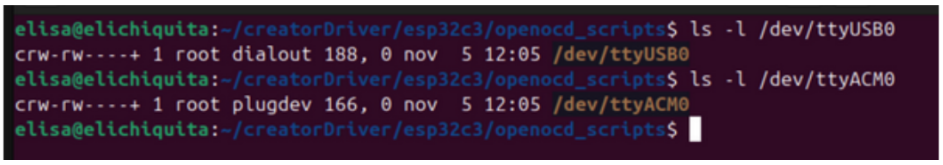


Figure 2.2.2: UART and JTAG devices privileges on Ubuntu.

   Using these commands to add your user to the ones given:

```
sudo usermod -a -G dialout $USER
sudo usermod -a -G plugdev $USER
```

2. If this is not enough, create a new `/etc/udev/rules.d/99-Espressif.rules` file (with superuser permission) containing:

```
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ATTRS{idVendor}=="303a", ATTRS{idProduct}=="1001",
      GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="10c4", ATTRS{idProduct}=="ea60", GROUP="dialout",
      MODE="0660"
```

   And recharge the rules using:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

   To check the ID and vendor, use the `lsusb` command:

```
...
Bus 003 Device 022: ID 303a:1001 Espressif USB JTAG/serial debug unit
...
```

   For more information, check out the original documentation here: `https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/establish-serial-connection.html`

2.3. Linux/macOS Docker

To use this, the UART and JTAG ports must have the correct rules and users assigned.

- **Install** CREATOR's docker image `creatorsim/creator-gateway-esp32` [2]:

```
docker pull creatorsim/creator-gateway-esp32
```

- **Run the image** with the correct ports opened:

```
docker run --init -it --device=/dev/ttyUSB0 --add-host=host.docker.internal:host-gateway -p
    8080:8080 -p 5000:5000 --name creator-gateway-esp32 creatorsim/creator-gateway-esp32 /bin/
    bash
```

  - To simplify this deployment, all these steps can be performed using Docker Compose. To do this, the `compose.yaml` file must be as follows:

```yaml
services:
  creator-gateway-esp32:
    image: creatorsim/creator-gateway-esp32:latest
    ports:
      - "8080:8080" # gateway
      - "5000:5000" # gdbgui
    stdin_open: true
    tty: true
    devices:
      - /dev/ttyUSB0 # device port
    # for debug
    network_mode: bridge
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

  - And deployed with

```
docker compose up -d
docker ps #Check Containers ID
docker attach <container_id>
```

2.3.1. Linux Docker Debug

1. Install OpenOCD inside your Linux/macOS device for Espressif devices [3].
2. Unzip the OpenOCD distro and **copy the binary** on the PATH:

```
sudo cp -r bin/ /usr/local/bin/openocd-esp32
sudo ln -s /usr/local/bin/openocd-esp32/openocd /usr/local/bin/openocd
```

3. **Export routes** (can be also added into `.bashrc` script):

```
export PATH="<openocd_route>/openocd-esp32/bin:$PATH"
export OPENOCD_SCRIPTS="<openocd_route>/openocd-esp32/share/openocd/scripts"
```

[2] https://hub.docker.com/repository/docker/creatorsim/creator-gateway-esp32
[3] https://github.com/espressif/openocd-esp32/releases/tag/v0.12.0-esp32-20241016

4. Start the `openocd_start.sh` script located in the `esp32c3/openocd_scripts` directory with the desired development board.:

```
./openocd_start.sh esp32c3
```

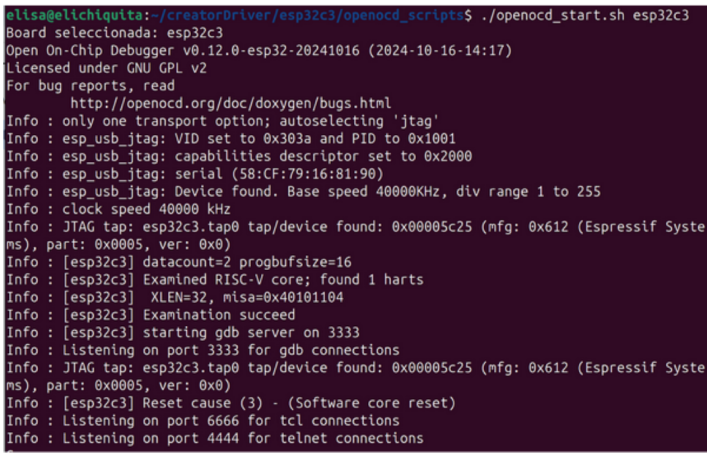


Figure 2.3.1: OpenOCD in Ubuntu output.

5. Press “Debug” button and start debugging.

2.4. Windows Docker

1. Install Docker Desktop.
2. Download esptool.
3. Pull `creator_gateway` image in Docker Desktop:
   (a) Search for `creatorsim/creator-gateway-esp32` in the Docker Desktop browser.
   (b) Click the “Pull” button.
   (c) **Run the image**:
   Using **Windows Powershell:**

```
docker run -it --rm --name creator-gateway-esp32 -p 3333:3333 -p 8080:8080 -p 5000:5000
    creatorsim/creator-gateway-esp32:latest
```

- To simplify this deployment, all these steps can be performed using Docker Compose. To do this, the `compose.yaml` file must be as follows:

```
services:
  creator-gateway-esp32:
    image: creatorsim/creator-gateway-esp32:latest
    ports:
      - "8080:8080" # gateway
      - "5000:5000" # gdbgui
    stdin_open: true
    tty: true
    # for debug
    network_mode: bridge
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

- And deployed with:

```
    docker compose up -d
    docker ps #Check Containers ID
    docker attach <container_id>
```

(d) Connect the computer with `esp_rfc2217_server`:

```
esp_rfc2217_server -v -p 4000 <target_port>
```

For more information: `https://docs.espressif.com/projects/esptool/en/latest/esp32/remote-serial-ports.html`

2.4.1. Windows Docker Debug

1. Install JTAG device onto your Windows device:

   (a) Install **Zadig** [4].

   (b) List all the devices in Options > List All Devices and search for **"USB Jtag/serial debug unit (Interface 2)"**.

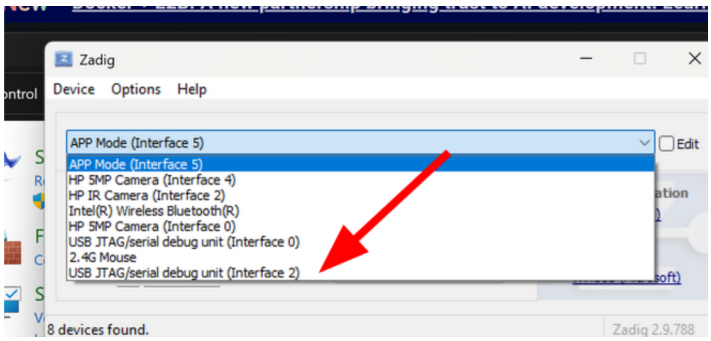


Figure 2.4.1: Zadig list of devices.

[4] `https://zadig.akeo.ie/`

(c) Downgrade the driver.

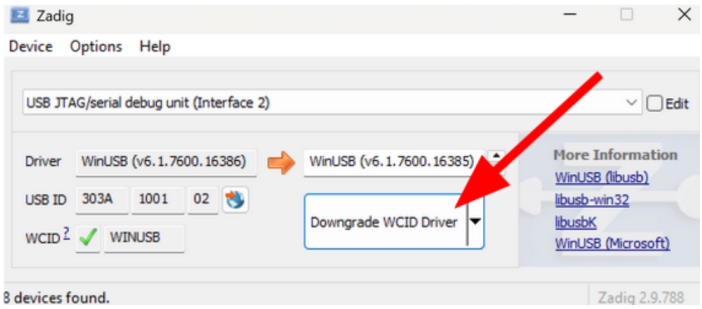


Figure 2.4.2: Zadig downgrade JTAG driver.

2. **Install OpenOCD** in your Windows device [5]:

3. Unzip and move its content to “Program Files”.

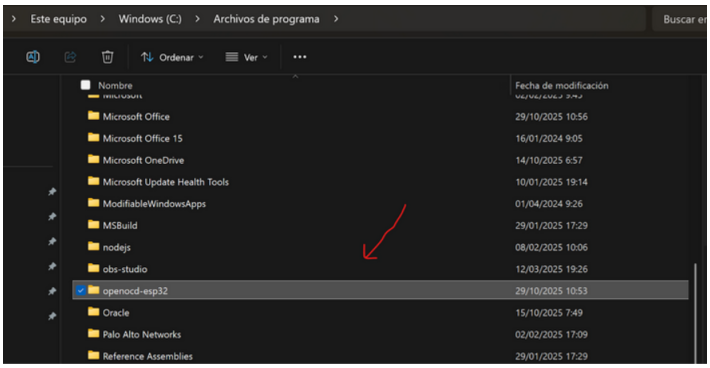


Figure 2.4.3: Program Files OpenOCD position.

4. Control panel > System and Security > System.

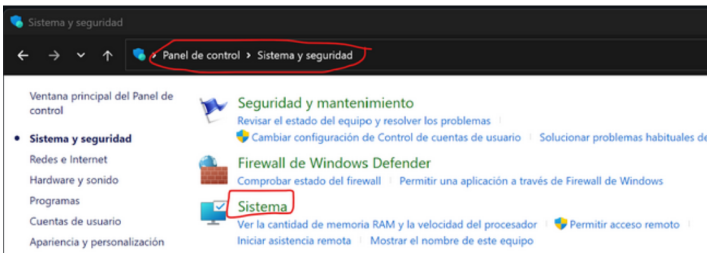


Figure 2.4.4: Control Panel.

[5] https://github.com/espressif/openocd-esp32/releases/tag/v0.12.0-esp32-20241016

5. Go to Information > Advanced system settings.

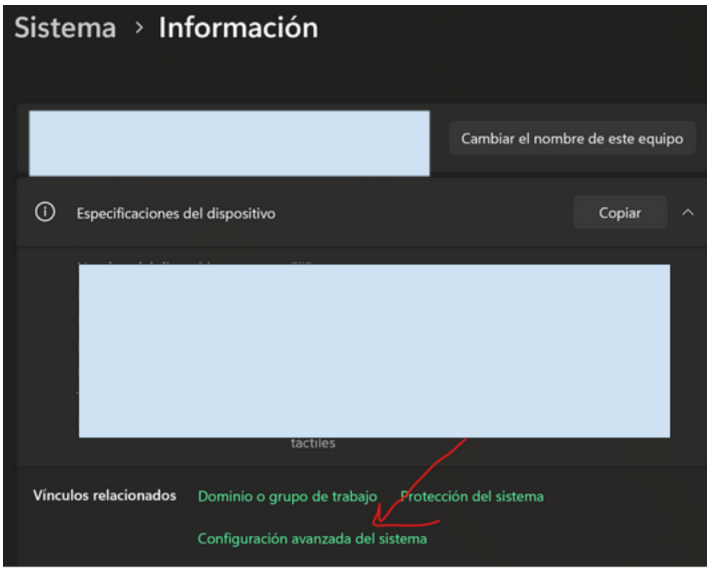


Figure 2.4.5: Advanced configuration.

6. Inside the System Properties window, go to the Advanced tab > Environment Variables > User variables > PATH.

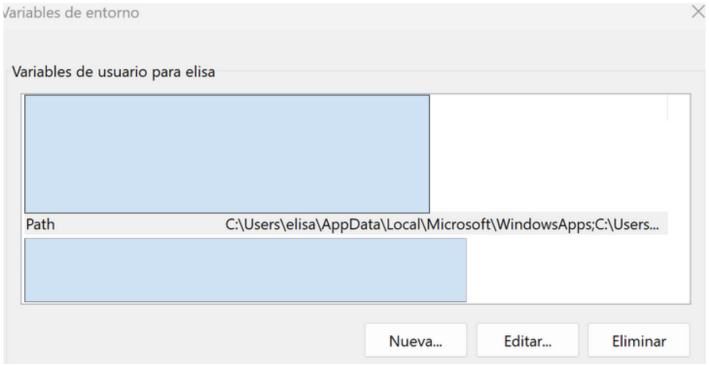


Figure 2.4.6: User environment variables panel.

7. Add OpenOCD path as an environment variable, pressing NEW.

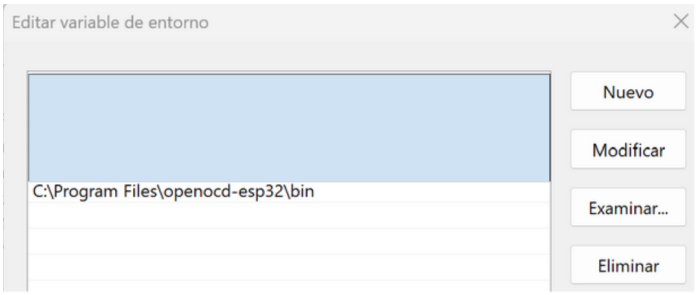


Figure 2.4.7: Environment variables on Windows.

8. Start the `openocd_start.bat` script located in the `esp32c3/openocd_scripts` directory with the desired development board.

```
.\openocd_start.bat esp32c3
```

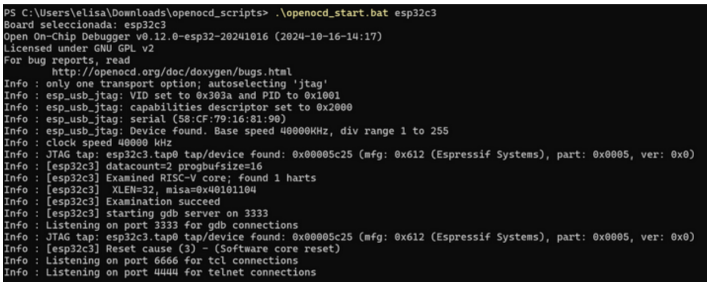


Figure 2.4.8: OpenOCD works on Windows.

3. ESPRESSIF MICROCONTROLLER INTERRUPTS USER MANUAL

This user tutorial is for the RISC-V Espressif Board based on the ESP32-C3 SoC, using a framework other than ESP-IDF to teach students how interrupts work in the microprocessor.

3.1. Hardware Interruptions

1. **Assign the GPIO port selected for the interrupt with its priority level.**

   In this step, we iterate over an auxiliary variable that stores which interrupts are taken.

   In the development environment we use, we do not store interrupts, and we do not allow two interrupts to share the same interrupt ID to prevent failures.

   We verify this by masking the interrupt set against the loop position.

```
.data
allocated_msg:
    .string "Allocated CPU IRQ %d, prio %u\n"

.text
next:
    addi t3, t3, 1 # no++
    j loop_bits

no_free:
    li a0, -1 # no free byte

cpu_alloc_intr:
    la t0, allocated # t0 = &allocated
    lw t1, 0(t0) # t1 = allocated
    # Start loop conditions
    li t2, 1 # t2 = 1, as BIT(1)
    li t3, 1 # t3 = no = 1 (loop start)

loop_bits:
    li t5, 31
    bge t3, t5, no_free # no >= 31, no interruption ID free
    # start loop
    sll t4, t2, t3 # t4 = BIT(no)
    and t5, t1, t4 # allocated & (1 << t3)
    bne t5, zero, next # if used(=/0), go next position
    # mark if not used
    or t1, t1, t4 # allocated |= (1 << t3)
    sw t1, 0(t0)
```

   Once the free ID position has been found, we record it in the interrupt vector, enable interrupts on the board, and assign the chosen interrupt position a memory priority.

The following memory records are modified:

- `INTERRUPT_CORE0_CPU_INT_ENABLE_REG` (to enable interrupts in the system).

```
# (1) Enable CPU interruptions
li t5, INTERRUPT_BASE
addi t5, t5, 0x104 # offset INTERRUPT_COREO_CPU_INT_ENABLE_REG
lw t0, 0(t5)
li t1, 1
sll t1, t1, t3 #BIT(no)
or t0, t0, t1
sw t0, 0(t5)
```

- `INTERRUPT_CORE0_CPU_INT_PRI_n_REG` (to indicate the priority of the interrupt), finishing the assignment.

```
# (2) Assign priority
li t5, INTERRUPT_BASE
addi t5, t5, 0x118 #INTERRUPT_COREO_CPU_INT_PRI_0_REG
addi t1, t3, -1
slli t1, t1, 2 #For GPIO 2
add t5, t5, t1 #INTERRUPT_COREO_CPU_INT_PRI_2_REG
sw s0, 0(t5)
mv a0, t3
j done
```

To sum up, interrupt enable management and pin assignment as interrupts are defined as follows:

```
cpu_alloc_interrupt:
    #Save all save registers, just in case
    addi sp, sp, -12
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw s2, 8(sp)

    #Save argument
    mv s0,a0 #PRIORITY (1-15)

    la t0, allocated # t0 = &allocated
    lw t1, 0(t0) # t1 = allocated
    #Start loop conditions
    li t2, 1 # t2 = 1, BIT(1)
    li t3, 1 # t3 = no = 1 (loop start)
loop_bits:
    li t5, 31
    bge t3, t5, no_free # if no >= 31, every interruption has been allocated.

    sll t4, t2, t3 # t4 = 1 << t3
    and t5, t1, t4 # t5 = allocated & (1 << t3)
    bne t5, zero, next # if used, go to next one

    # if unused, mark it down
    or t1, t1, t4 # allocated |= (1 << t3)
    sw t1, 0(t0)
```

```
27
28      # (1) Enable CPU interruptions REG(C3_INTERRUPT)[0x104 / 4] |= BIT(no);
29      li t5, INTERRUPT_BASE
30      addi t5, t5, 0x104
31      lw t0, 0(t5)
32      li t1, 1
33      sll t1, t1, t3 #BIT(no)
34      or t0, t0, t1
35      sw t0, 0(t5)
36
37      # (2) Assign priority REG(C3_INTERRUPT)[0x118 / 4 + no - 1] = prio; // CPU_INT_PRI_N
38      li t5, INTERRUPT_BASE
39      addi t5, t5, 0x118
40      addi t1, t3, -1
41      slli t1, t1, 2
42      add t5, t5, t1
43      sw s0, 0(t5)
44
45
46      mv a0, t3 # return assigned number
47      j done
48
49  next:
50      addi t3, t3, 1 # no++
51      j loop_bits
52
53  no_free:
54      li a0, -1 # no free byte
55
56  done:
57      # Restore registers
58      lw s0, 0(sp)
59      lw s1, 4(sp)
60      lw s2, 8(sp)
61      addi sp, sp, 12
62      ret
```

2. **Define interrupt routine and interrupt clean-up routine.**

   First, we define the function to be executed whenever our microprocessor detects an interrupt. In this case, we want a message to be printed on the screen when the user presses a button connected to a GPIO pin, as follows:

```
.data
fmt_button:
    .string "Button %s\n"
pressed_msg:
    .string "pressed"
released_msg:
    .string "released"
allocated:
    .word 0 # variable static uint32_t allocated
allocated_msg:
    .string "Allocated CPU IRQ %d, prio %u\n"

.text
    .equ BUTTON_PIN, 9
    .equ INTERRUPT_BASE, 0x600c2000
    .equ GPIO_BASE, 0x60004000
    .global main
    .extern gpio_input
    .extern gpio_read

button_handler:
    addi sp, sp, -4
    sw ra, 0(sp)

    li a0, BUTTON_PIN
    call gpio_read

    #beqz a0, use_released
    bnez a0, finish_handler
    la a1, pressed_msg
    j do_printf

use_released:
    la a1, released_msg

do_printf:
    la a0, fmt_button
    call printf

finish_handler:
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

External functions in C are used, such as the library function `printf` or external functions such as `gpio_read`, which will be discussed later.

On the other hand, the interrupt clean-up function routine is necessary, since one of the values of an interrupt is whether it is pending execution. Therefore, when the interrupt has finished, and we need to return to normal execution, we must write to the GPIO to indicate that all pending interrupts have finished, as shown below.

```
gpio_clear_interrupt:
    # a0 = pin
    li t0, GPIO_BASE
    addi t0, t0, 0x44 #GPIO_STATUS_REG
    lw t1, 0(t0)
    li t2, 1
    sll t2, t2, a0 # BIT(pin)
    not t0, t0 # ~BIT(pin)
    and t1, t1, t0 #t1 & ~(1 << pin) Clear pin
    sw t1, 0(t0)
    ret
```

3. **Store information about how we want our introduction to react in the ISR hosting vector.**

   Expressif boards do not store everything in the interrupt vector, but have another table in memory (a list of structures) to store the ISRs for each interrupt, which in this case we call `g_irq_data`, with as many positions as there are interrupts that can be accommodated and structured as shown:

```
struct irq_data {
    void (*fn)(void *); // ISR.
    void *arg; // ISR arguments
    void (*clr)(void *); //Clean interrupt function
    void *clr_arg; // Clean arguments
};
```

   Each element of the `struct` takes up 4 bytes, so in assembly language we will have to move to the corresponding list (in this case, as many positions as there are interrupts) and then access each element by jumping 4 bytes at a time:

```
# (2)Save pin interrupt
la t0, g_irq_data
la s1, isr_function #ISR
la t2, gpio_clear_interrupt # t2 = gpio_clear_interrupt
#---- Iterate the list
slli t1, s3, 4 # t1 = no * 16
add t0, t0, t1 # t0 = &g_irq_data[no]
#---- Store in the struct
sw s1, 0(t0) # *(t0 + 0) = isr
sw s2, 4(t0) # *(t0 + 4) = arg
sw t2, 8(t0) # *(t0 + 8) = gpio_clear_interrupt
sw s0, 12(t0) # *(t0 + 12) = pin (s0)
```

   Once the interrupt routine is established, we must configure the interrupt characteristics for the assigned GPIO.

   This memory register is `GPIO_PINn_REG`, which is used to configure the pin. Each pin has the following slots, which indicate the characteristics it can support, as shown in Figure 3.1.1.

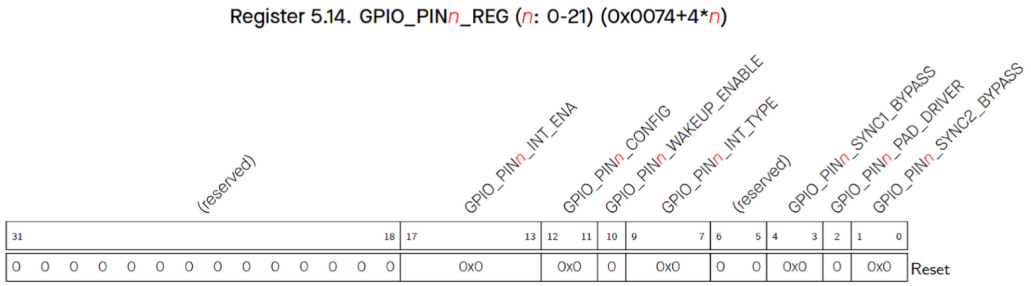


Figure 3.1.1: Diagram of the `GPIO_PINn_REG` memory register in the ESP32-C3 TRM [1].

In this case, when we reach the memory register for the pin we want, we must touch the memory sections `GPIO_PINn_INT_ENA` (to enable this interrupt) and `GPIO_PINn_INT_TYPE` (to set the type of interrupt we want, as discussed above).

We see that `INT_TYPE` has 3 bits and is used to indicate the interrupt type, as shown on page 184 of the manual [1] and in Table 3.1.1.

Interrupt type	**Value**
Disabled	0 (000)
Rising edge	1 (001)
Falling edge	2 (010)
Any edge	3 (011)
Low level	4 (100)
High level	5 (101)

Table 3.1.1: Interrupt types and associated values.

On the other hand, `GPIO_PINn_INT_ENA` is just a bit to indicate that the interrupt is enabled.

Therefore, before writing to the register, we perform the calculations and write the characteristics we want in a single operation.

In this example, we want to indicate that the interrupt we want will accept both edges (since it is to detect whether a button goes up or down), managing it as shown:

```
# (3) Set characteristics for the interrupt;
li t0, GPIO_BASE
addi t0, t0, 0x74
slli t1, s0, 2 #t1 = pin(2) * 4
add t0, t0, t1 #t0 = GPIO_PIN2_REG
lw t2, 0(t0)
li t3, 3 # t3 = 3
slli t3, t3, 7 # t3 = 3 << 7 = 0x180
li t4, 1 # t4 = 1
slli t4, t4, 13 # t4 = 1 << 13 = 0x2000
or t3, t3, t4 # t3 = t3 | t4 = 0x180 | 0x2000 = 0x2180
or t2, t2, t3
sw t2, 0(t0)
```

Lastly, we map the GPIO interrupt routine to the CPU using the `GPIO_INTERRUPT_PRO_MAP_REG` register by adding the interrupt number assigned to it:

```
# (4) Map GPIO IRQ to CPU
li t0, INTERRUPT_BASE
addi t0, t0, 0x40 #GPIO_INTERRUPT_PRO_MAP_REG
sw s3, 0(t0)
```

To summarize, the way in which the behavior of the assigned interrupt is stored can be seen below:

```
gpio_set_irq_handler:
    # a0 = pin, a1 = handler, a2 = arg
    #Move args into saved registers
    mv s0, a0 # pin
    mv s1, a1 # handler
    mv s2, a2 # arg

    # (1) Allocate an interrupt in the CPU with priority 1
    li a0, 1 # priority
    call cpu_alloc_interrupt
    mv s3, a0

    # (2) Save pin interrupt
    la t0, g_irq_data
    slli t1, s3, 4 # t1 = no * 16
    add t0, t0, t1 # t0 = &g_irq_data[no]

    sw s1, 0(t0) # *(t0 + 0) = handler (a1)
    sw s2, 4(t0) # *(t0 + 4) = arg (a2)
    la t2, gpio_clear_interrupt # t2 = gpio_clear_interrupt
    sw t2, 8(t0) # *(t0 + 8) = gpio_clear_interrupt
    sw s0, 12(t0) # *(t0 + 12) = pin (s0)

    # (3) Set characteristics for the interrupt = REG(C3_GPIO)[0x74 / 4 + pin] |= (3U << 7) |
        BIT(13);
    li t0, GPIO_BASE
    addi t0, t0, 0x74
    slli t1, s0, 2 #t1 = pin * 4
    add t0, t0, t1 #t0 = &REG(C3_GPIO)[0x74 / 4 + pin]
    lw t2, 0(t0)
    # li t3, 0x2180 # t3 = (3U << 7) | BIT(13)
    li t3, 3 # t3 = 3
    slli t3, t3, 7 # t3 = 3 << 7 = 0x180
    li t4, 1 # t4 = 1
    slli t4, t4, 13 # t4 = 1 << 13 = 0x2000
    or t3, t3, t4 # t3 = t3 | t4 = 0x180 | 0x2000 = 0x2180
    or t2, t2, t3 #valor REG(C3_GPIO)[0x74 / 4 + pin] |= t3
    sw t2, 0(t0)
    # (4) Map GPIO IRQ to CPU
    li t0, INTERRUPT_BASE
    addi t0, t0, 0x40 #GPIO_INTERRUPT_PRO_MAP_REG
    sw s3, 0(t0)
    ret
```

4. **Assign the pin mode and call the interrupt assigner.**

   In the main function, we will set the button pin to "input". Since, in this case, GPIO input/output management is not relevant to implement, the C functions defined in the system controller of our test environment will be called, as shown below.

```c
void gpio_output(int pin) {
  REG(C3_GPIO)[GPIO_OUT_FUNC + pin] = BIT(9) | 128; // Simple out, TRM 5.5.3
  gpio_output_enable(pin, 1);
}

int gpio_read(int pin) {
  return GPIO->IN & BIT(pin) ? 1 : 0;
}
```

   Therefore, the `main` function is defined in the following listing. An infinite loop is added at the end, so the program does not terminate and continues listening for interrupts until it is stopped manually.

```assembly
main:
    # gpio_input(BUTTON_PIN)
    li a0, BUTTON_PIN
    call gpio_input

    # Set irq handler
    li a0, BUTTON_PIN
    la a1, button_handler
    li a2, BUTTON_PIN
    call gpio_set_irq_handler

loop:
    j loop
```

We verify that this program works, as shown in Figure 3.1.2.

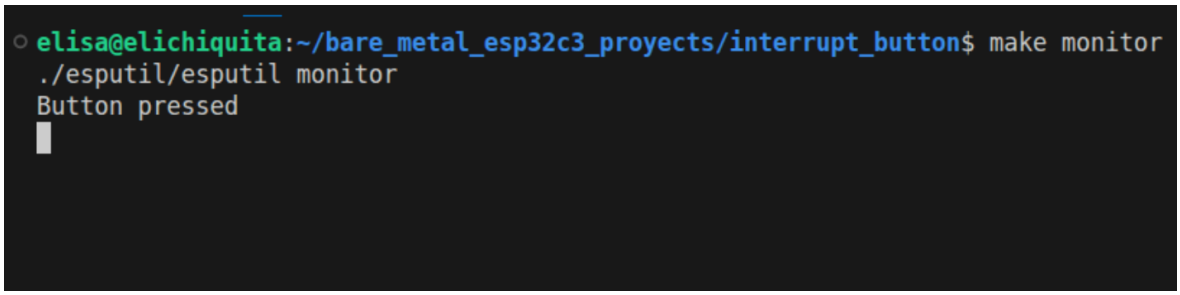


Figure 3.1.2: Successful hardware interrupt.

3.2. Timer Interrupts

A timer is a hardware peripheral that accurately measures time intervals and generates events or signals based on the count of clock cycles. There are three types within the system: SYSTIMER, TIMG (Timer Group), and RTC timers.

In this case, for this use case, we will use **SYSTIMER**: a general-purpose peripheral that provides a 52-bit counter to generate *tick* or heartbeat interrupts, or to serve as a general timer to generate periodic or one-time interrupts.

1. **Define the interrupt routine and clean-up function.**

   For this simple case, this program will count how many milliseconds have passed since we initialized the timer, as the following code shows:

```
timer_handler:
    la t0, systimer_tick
    lw t1, 0(t0)
    addi t1, t1, 1
    sw t1, 0(t0)
    jr ra
```

   In addition, as with all interrupts, we define a clean-up function. Since we are activating all interrupt targets, we must process these three to clear the "pending" status once the desired action completes. This is achieved by manipulating the `SYSTIMER_TARGET0_INT_CLR` memory register, as shown in Figure 3.2.1:

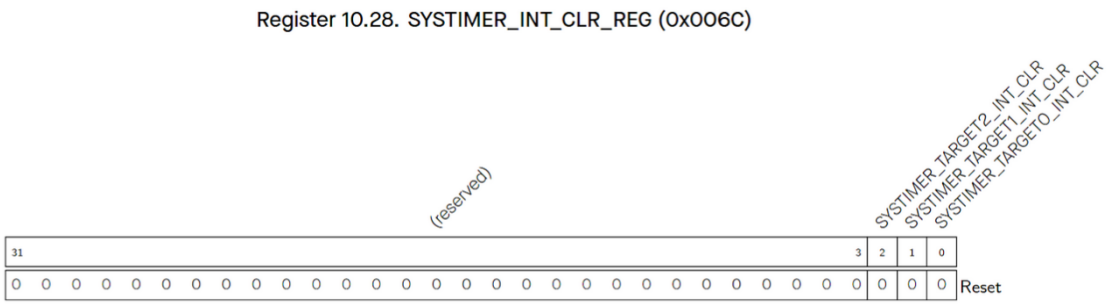


Figure 3.2.1: Diagram of the `SYSTIMER_TARGET0_INT_CLR` memory register according to the ESP32-C3 manual [1].

2. **Timer initialization.**

   First, we need to set the timer's frequency and enable interrupts. The timer frequency is set in the memory register `SYSTIMER_TARGET0_CONF_REG`, which can be seen in Figure 3.2.2:

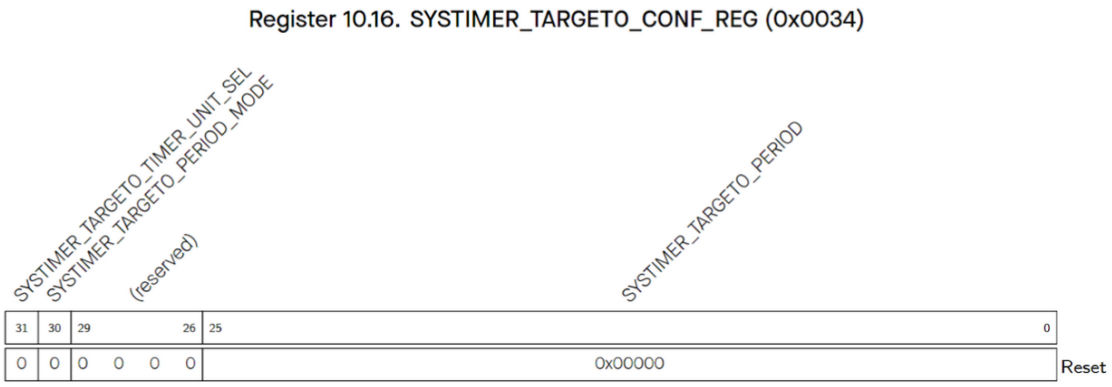


Figure 3.2.2: Diagram of the `SYSTIMER_TARGET0_CONF_REG` memory register according to the ESP32-C3 manual [1].

In this register, what we are going to do is first activate **periodic mode** (which, as we can see in Figure 3.2.2, is bit 30) and indicate in `SYSTIMER_TARGET0_PERIOD` that we want the interrupt to occur every 16,000 *ticks* (knowing that this clock, according to the development board documentation, has a frequency of 16 MHz (16,000,000 cycles per second), indicates that the interrupt will occur every millisecond [1].

This periodic mode activation can be seen below:

```
li t0, SYSTIMER_BASE
addi t0, t0, 0x034 #SYSTIMER_TARGET0_CONF_REG
li t1, 1
sll t1, t1, 30 #Bit(30)
or t1, t1, s0 # t1 = BIT(30) | 16000
sw t1, 0(t0)
```

Then, we reset the counter to zero using *SYSTIMER_COMP0_LOAD_REG*, a memory region structured as shown in Figure 3.2.3.

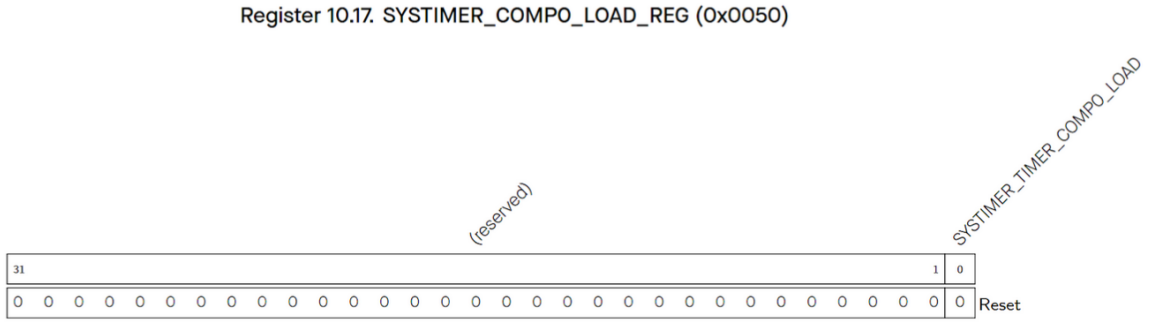


Figure 3.2.3: Esquema del registro de memoria *SYSTIMER_COMP0_LOAD_REG* según el manual de ESP32-C3 [1].

The element we need to modify is located in bit 0, so we activate that part of the memory:

```
li t0, SYSTIMER_BASE
addi t0, t0, 0x050
li t1, 1
sll t1, t1, 0 # t1 = BIT(0)
sw t1, 0(t0)
```

Next, we activate comparator 0. SYSTIMER has up to 3 comparators, as reflected in `SYSTIMER_CONF_REG` and shown in Figure 3.2.4.

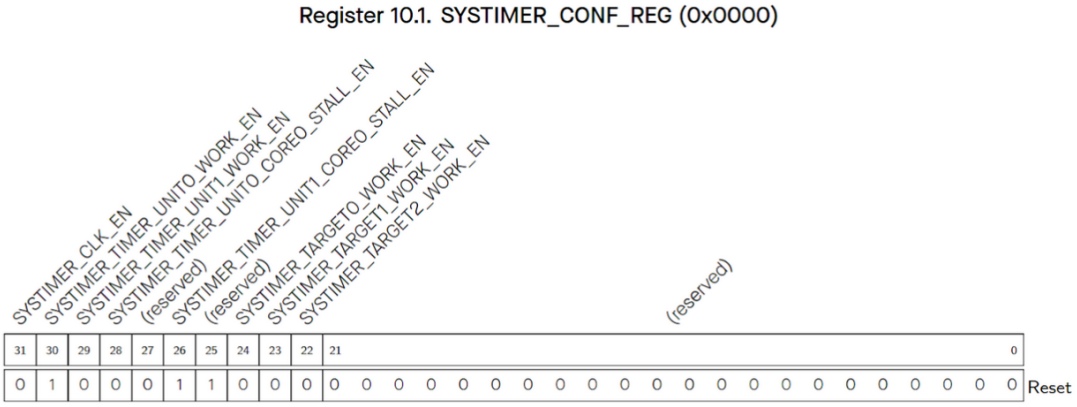


Figure 3.2.4: Diagram of the `SYSTIMER_CONF_REG` memory register according to the ESP32-C3 manual [1].

To activate comparator 0, we see that it is located in bit 24:

```
li t0, SYSTIMER_BASE
addi t0, t0, 0x000 #SYSTIMER_CONF_REG
lw t1, 0(t0)
li t2, 1
slli t3, t2, 24 # t3 = BIT(24)
or t1, t1, t3 # BIT(24) | 0
sw t1, 0(t0)
```

Finally, we enable timer interrupts. In this case, we enable interrupts for all targets using the `SYSTIMER_INT_ENA_REG` memory register, shown in Figure 3.2.5.

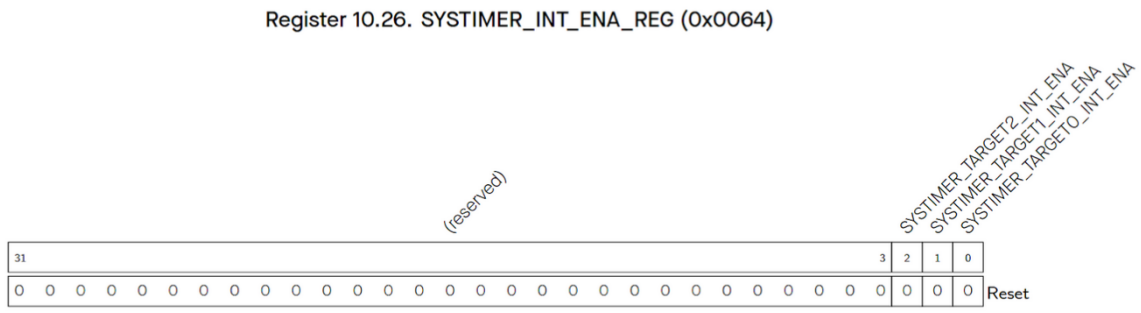


Figure 3.2.5: Diagram of the `SYSTIMER_INT_ENA_REG` memory register according to the ESP32-C3 manual [1].

In this case, we will enable interrupts for the three targets by writing 111 to the register:

```
li t0, SYSTIMER_BASE
addi t0, t0, 0x064 # t0 = SYSTIMER_INT_ENA_REG
lw t1, 0(t0)
li t2, 7 # 7 (111) activate all targets
or t1, t1, t2 # SYSTIMER->INT_ENA |= 7U;
sw t1, 0(t0)
```

Then, once the timer has been set, we register the timer control ISR in the interrupt table, as was done in function `cpu_alloc_interrupt` in hardware interruptions, calling it as shown in the following code:

```
li a0, 1 # priority
call cpu_alloc_interrupt
mv s1, a0 # Save IRQ assigned
# (3) Save ISR in g_irq_data
la t0, g_irq_data
slli t1, s1, 4 # t1 = no * 16 (4 entries x 4 bytes)
add t0, t0, t1
la t2, timer_handler
sw t2, 0(t0)
la t2, systimer_clear_interrupt
sw t2, 8(t0)
```

Finally, as we did with GPIO interrupts, we map the SYSTIMER IRQ to the CPU. In this case, we record the priority in the `INTERRUPT_CORE0_SYSTIMER_TARGET0_INT_MAP_REG` register, as shown in the following code snippet:

```
li t0, INTERRUPT_BASE
addi t0, t0, 0x94
lw t1, 0(t0)
mv t2, s1 #no
sw t2, 0(t0)
```

In summary, the following code snippet initializes and configures a timer interrupt.

```
systimer_init:
    # Arguments: a0 = period
    mv s0, a0

    # (1) Configure timer
    addi sp, sp, -4 # Reserva espacio en la pila
    sw ra, 0(sp) # Guarda ra
    # SYSTIMER->TARGET0_CONF = BIT(30) | 16000; Set period
    li t0, SYSTIMER_BASE
    addi t0, t0, 0x034 #SYSTIMER_TARGET0_CONF_REG
    li t1, 1
    sll t1, t1, 30 #Bit(30)
    or t1, t1, s0 # t1 = BIT(30) | 16000
    sw t1, 0(t0)
    # SYSTIMER->COMP0_LOAD = BIT(0); Reload period
    li t0, SYSTIMER_BASE
    addi t0, t0, 0x050
    li t1, 1
```

```
    sll t1, t1, 0 # t1 = BIT(0)
    sw t1, 0(t0)
    # SYSTIMER->CONF |= BIT(24); // Enable comparator 0
    li t0, SYSTIMER_BASE
    addi t0, t0, 0x000 #SYSTIMER_CONF_REG
    lw t1, 0(t0)
    li t2, 1
    slli t3, t2, 24 # t3 = BIT(24)
    or t1, t1, t3 # BIT(24) | 0
    sw t1, 0(t0)
    # SYSTIMER->INT_ENA |= 7U enable triggers in all targets
    li t0, SYSTIMER_BASE
    addi t0, t0, 0x064 # t0 = SYSTIMER_INT_ENA_REG
    lw t1, 0(t0)
    li t2, 7 # 7 (111) activate all targets
    or t1, t1, t2 # SYSTIMER->INT_ENA |= 7U;
    sw t1, 0(t0)

    # (2) Allocate an interrupt in the CPU with priority 1
    li a0, 1 # prioridad
    call cpu_alloc_interrupt
    mv s1, a0 # Save IRQ assigned

    # (3) Save ISR in g_irq_data
    la t0, g_irq_data
    slli t1, s1, 4 # t1 = no * 16 (4 campos x 4 bytes)
    add t0, t0, t1
    la t2, timer_handler
    sw t2, 0(t0)
    la t2, systimer_clear_interrupt
    sw t2, 8(t0)

    # (4) Map systimer IRQ to CPU
    li t0, INTERRUPT_BASE
    addi t0, t0, 0x94
    lw t1, 0(t0)
    mv t2, s1 #no
    sw t2, 0(t0)

    lw ra, 0(sp) # Restaura ra
    addi sp, sp, 4 # Libera espacio de la pila

    jr ra
```

3. **Interaction with the timer interrupt.**

   With a millisecond counter, we could, for example, print to the screen every time a full second passes. An example of such a function, plus the main function `main`, can be found in the code snippet below:

```
.data
  msg_tick:
   .string "Hi!!! \n"

.text
print_function:
    addi sp, sp,-4
    sw ra,0(sp)
    la a0, msg_tick
    call printf
    li t0, 0 # t0 = 0
    la t1, systimer_tick # t1 = &systimer_tick
    sw t0, 0(t1) # systimer_tick = 0
    lw ra,0(sp)
    addi sp, sp,4
    jr ra

log_task:
    lw t0, systimer_tick
    li t1, 1000 #1 s
    bge t0,t1, print_function
    j log_task

main:
    li a0, 16000
    jal ra, systimer_init
    j log_task

loop:
    j loop
```

This use case can be verified by executing the program, as shown in Figure 3.2.6.

```
elisa@elichiquita:~/bare_metal_esp32c3_proyects/timer_exaple$ make flash
./esputil/esputil flash 0 firmware.bin
Using flash params 0x21f
Written firmware.bin, 1264 bytes @ 0
elisa@elichiquita:~/bare_metal_esp32c3_proyects/timer_exaple$ make monitor
./esputil/esputil monitor
Hi!!!
Hi!!!
Hi!!!
Hi!!!
Hi!!!
```

Figure 3.2.6: Running a program that prints “Hi!” every second.

4. SBC BOARDS USER MANUAL

This user tutorial is for RISC-V SBC boards where a Linux distro which can handle SSH and install gdbgui in their systems. At the moment, the SBC RISC-V board with Ubuntu Support (v.24.04.3) fulfills these requirements.

SBC boards used:

- **OrangePi RV2**: 8 cores RISC-V. Wi-Fi and Bluetooth connection.
- **Nezha D1-H 64 bit RISC-V**: Unicore RISC-V64. Ubuntu does not have Wi-Fi connectivity or a UI.

Recommendations for a correct SBC setup:

- Have the recommended power supply for every SBC.
- Use a class10 microSD from a recognizable brand from Amazon or another reliable retailer.
- A good Ethernet cable.

4.1. SBC Environment Setup

1. Operative System preparation according to the SBC or Canonical Ubuntu's instructions.
2. Create the **default folder** where your CREATOR projects will be saved. By default, this directory will be named "creator".
3. Give the **correct rights** to the directory.

```
sudo chown user:user ~/creator
sudo chmod u+rwx ~/creator
```

4. **Connect the SBC to the Internet** via Ethernet or Wi-Fi if possible.
   - Depending on the SBC's configuration, the SBC IP will change from time to time. Check the IP every first use.

```
ip a
```

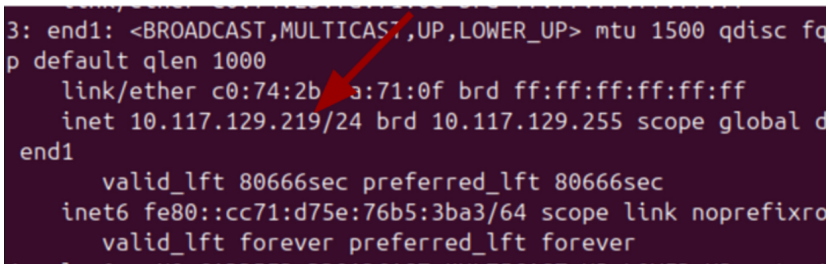


Figure 4.1.1: `ip a` command output.

5. Check **ssh** SBC's status.

```
systemctl status ssh
```

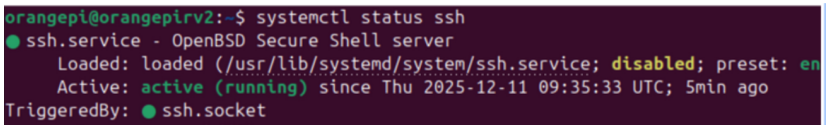


Figure 4.1.2: SSH correct open status.

6. Check username:

   (a) **Default username**: Normally, it is the name on the left side of the "@". However, it can be checked out by typing.

```
whoami
```

   (b) **New SBC user.**

7. Check if **an ssh connection can be established** from the computer connected to CREATOR to the SBC.

```
ssh <usuario>@<ip>
```

   (a) Every time an ssh connection is made, the system will ask for a password. It can be overridden by copying your computer's SSH keys to the SBC.

   (b) Create an SSH key on your computer.

```
ssh-keygen -t rsa -b 4096
```

   (c) Copy the ssh code to SBC.

```
ssh-copy-id <user>@<ip>
```

8. Install gdbgui

   To prevent incompatibility issues, gdbgui will be installed within a virtual environment that the driver will access automatically. It must have the same name as specified in the tutorial.

   - Install **virtual environments** and create one.

```
sudo apt install python3.12-venv
python3 -m venv ~/gdbgui-venv
source /home/ubuntu/gdbgui-venv/bin/activate
```

   - Install **gdbgui** and its dependencies.

```
sudo apt install build-essential python3-dev python3-pip python3-setuptools python3-wheel
    gdb
pip3 install gdbgui
```

   - Small **configuration on gdbgui** (looking forward to the next update to fix this).

```
sed -i "/extra_files=get_extra_files()/a\ allow_unsafe_werkzeug=True," \
~/gdbgui-venv/lib/python3.12/site-packages/gdbgui/server/server.py
```

   - To check if changes were correct.

```
grep -n "allow_unsafe_werkzeug" \
~/gdbgui-venv/lib/python3.12/site-packages/gdbgui/server/server.py
```

4.2. Usage with CREATOR's Simulator

1. Install driver's dependencies.

```
pip3 install flask flask_cors
```

2. Download SBC's driver from prerequisites page [6] and unzip it.

```
unzip creator-gateway-sbc.zip
cd creator-gateway-sbc/driver
```

3. Execute gateway.

```
python3 gateway.py
```

4.3. Use Case: 64 Bit Factorial Example

Using the code example from Figure 4.3.1 inside the RISCV-64 editor on an OrangePi RV2, we can verify the correct computation of factorial values through simulation and real hardware execution using the *Target-Flash SBC* interface.

```
.text
main:
    addi sp, sp, -16
    sd ra, 8(sp)

    li a0, 5
    jal ra, factorial

    li a7, 1
    ecall

    ld ra, 8(sp)
    addi sp, sp, 16
    li a7, 10
    ecall
```

```
factorial:
    addi sp, sp, -32
    sd ra, 24(sp)
    sd fp, 16(sp)
    addi fp, sp, 16

    li t0, 2
    bge a0, t0, b_else
    li a0, 1
    j b_end

b_else:
    sd a0, -8(fp)
    addi a0, a0, -1
    jal ra, factorial
    ld t1, -8(fp)
    mul a0, a0, t1

b_end:
    ld ra, 24(sp)
    ld fp, 16(sp)
    addi sp, sp, 32
    jr ra
```

Figure 4.3.1: Assembly RISC-V 64 program which implements factorial.

[6] https://github.com/creatorsim/creator-gateway-sbc

The driver's results are the same as those from the simulator (120). Figure 4.3.2 demonstrates the correct compilation, transmission, and execution of the program.

```
riscv64-linux-gnu-gcc program.o ecall_macros.o  -o program -static -lc -lm
rm -f program.o ecall_macros.o
make: se sale del directorio '/home/elisa/OrangePiDriver/Driver/main'
program.s                                                     100%  986   289.2KB/s   00:00
script.gdb                                                    100%   40     7.6KB/s   00:00
gdbinit                                                       100%   25     3.8KB/s   00:00
Makefile                                                      100%  395    83.1KB/s   00:00
program                                                       100%  552KB   5.3MB/s   00:00
ecall_macros.s                                                100%   15KB   2.1MB/s   00:00
2025-12-19 09:25:57,726 - INFO - 127.0.0.1 - - [19/Dec/2025 09:25:57] "POST /flash HTTP/1.1" 200 -
2025-12-19 09:25:59,708 - INFO - 127.0.0.1 - - [19/Dec/2025 09:25:59] "OPTIONS /monitor HTTP/1.1" 200 -
2025-12-19 09:25:59,714 - INFO - orangepi@10.117.129.219
120
2025-12-19 09:26:00,694 - INFO - 127.0.0.1 - - [19/Dec/2025 09:26:00] "POST /monitor HTTP/1.1" 200 -
```

Figure 4.3.2: SBC factorial execution.

On the other hand, debugging is also possible using gdbgui, as with the ESP32, as shown in Figure 4.3.3. In this case, the I/O terminal is the one attached to the gdbgui.

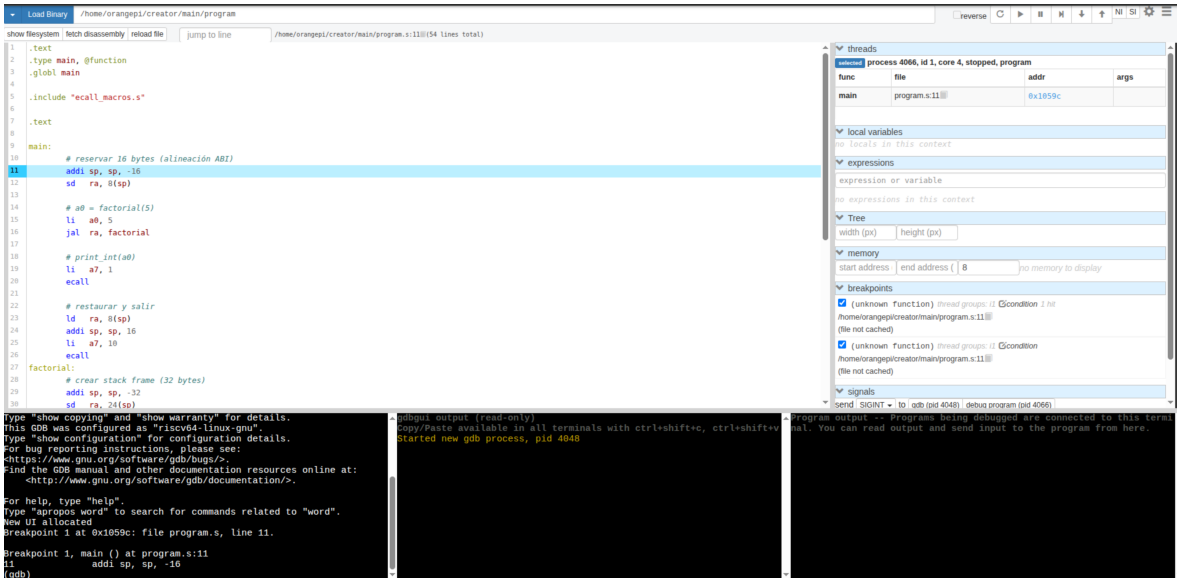


Figure 4.3.3: gdbgui interface.

5. ARDUINO USER MANUAL

5.1. Arduino CREATOR Module

This new library is only for Espressif 32-bit boards added in CREATOR; check out the ESP32 gateway setup information.

5.1.1. How to Start Using CREATino Library Functions

CREATOR lets users add custom libraries via the Library button. This update already provides a shortcut to add an Arduino library in CREATOR via the “Load Arduino Library” option. These functions will be displayed on the right.

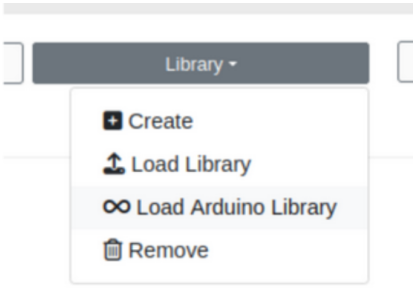


Figure 5.1.1: Arduino Library Shortcut.

Except `serial_printf` function (exclusively for Espressif Arduino devices) all the functions displayed can be founded in Arduino Original’s Documentation and in CREATOR’s Info button “Creatino Help”.

Create your First Program

As in the original Arduino sketches, CREATino programs must have a structure composed of a “setup” and a “loop” function. To address this issue, users can load a template in Example called *Example 1: Template for new examples*, which is ready to use.

```
#Template for Arduino projects
.data

.text
    setup:
        nop
    loop:
        nop
    main:
        addi sp, sp, -16
        sw ra, 12(sp)
        jal ra, initArduino
        jal ra, setup
        lw ra, 12(sp)
        addi sp, sp, 16
        j loop
```

Integrate Arduino Library in Target Flash Menu

Once your program is developed and compiled, you need to activate the "Arduino Support" checkbox below the *Select Target Board* option in the textit Target Flash menu.

Figure 5.1.2: Arduino Support button in Target Flash.

Aspects to Consider Using this Library

1. The supported ESP32 boards do not support float pointers. Arduino functions that return float or double values will fail.
2. Avoid using these GPIO numbers:
   (a) **GPIO 8:** BOOT MODE pin. It might show this error code:

```
Serial port /dev/ttyUSB0
Connecting.....................................

A fatal error occurred: Failed to connect to ESP32-C3: Wrong boot mode detected (0x0)! The
      chip needs to be in download mode.
For troubleshooting steps visit: https://docs.espressif.com/projects/esptool/en/latest/
     troubleshooting.html
```

   (b) **GPIO 18 and 19:** Debug pins.

5.2. Use Cases

5.2.1. Use Case 1: Internal LED Blink

This example is used to verify that GPIO functions work correctly on an ESP32 board.

- **Components:**
  - ESP32-C3-DevKitC-02 board.
- **Steps**:
  1. **Setup**: Establish Internal LED as an output.

     On the ESP32-C3-DevKitC-02, the internal LED pin is pin 30, so it must be configured as an output. We use the **`pinMode`** function for this. Depending on the project necessities, `pinMode` can have these values:

Mode	Value	Usage
INPUT	0x01	Digital input mode
OUTPUT	0x03	Digital output mode
PULLUP	0x04	Enables the internal pull-up resistor on an input pin (if the pin is disconnected, it will read HIGH). Buttons connected to ground
INPUT_PULLUP	0x05	Combines input configuration (0x01) with pull-up (0x04)
PULLDOWN	0x08	Ensures the pin reads LOW when disconnected
INPUT_PULLDOWN	0x09	Ensures the pin reads LOW when disconnected
OPEN_DRAIN (Data buses)	0x10	Output can only pull to ground; reading HIGH requires an external resistor or pull-up
OUTPUT_OPEN_DRAIN (Buses)	0x13	Combines digital output with open-drain mode
ANALOG	0xC0	Configures the pin for analog input (ADC)

Table 5.2.1: GPIO Pin Modes.

     In this case, the setup function will look like this:

```
setup:
    #pinMode(LED_BUILTIN, OUTPUT);
    li a0,30
    li a1, 0x03
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, pinMode
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra
```

  2. **Loop**: Turn the LED on and off.

     The LED has been set up; now it is necessary to toggle it on and off to create a blinking effect. For this, the **`digitalWrite`** function will be used. This function requires the pin number (pin 30) and the write state (0x1 or `HIGH` to turn on the light, 0x0 to turn it off).

On the other hand, we will use the **`delay`** function to introduce a delay between state changes of the LED. This function only needs the delay time in milliseconds. For this example, this quantity of time will be stored in memory.

This is the assembly program:

```
.data
  time:
     .word 1000
.text
loop:
    #digitalWrite(LED_BUILTIN, HIGH);
    li a0,30
    li a1, 0x1
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, digitalWrite
    lw ra, 0(sp)
    addi sp, sp, 4
    #delay(1000);
    la a0, time
    lw a0, 0(a0)
        addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, delay
    lw ra, 0(sp)
    addi sp, sp, 4
    #digitalWrite(LED_BUILTIN, LOW);
    li a0,30
    li a1, 0x0
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, digitalWrite
    lw ra, 0(sp)
    addi sp, sp, 4
    #delay(1000);
    la a0, time
    lw a0, 0(a0)
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, delay
    lw ra, 0(sp)
    addi sp, sp, 4
    j loop
```

Now the LED will start blinking.

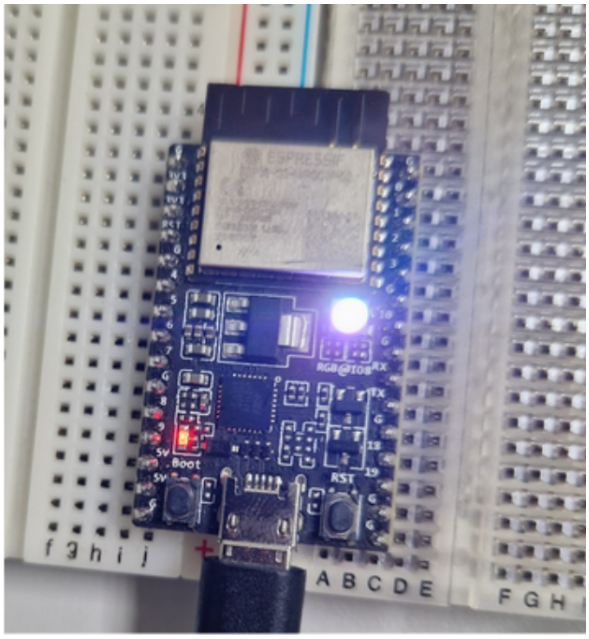


Figure 5.2.1: Use Case 1 Result: Board starts blinking.

5.2.2. Use Case 2: Button + LED

This use case will be overridden using interrupts and a recursive approach.

- **Components:**
  - ESP32-C3-DevKitC-02 board.
  - Button (in GPIO 6).
  - LED (in GPIO 4).

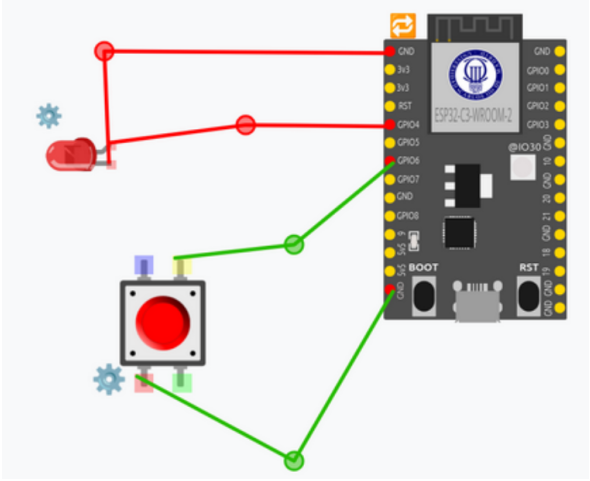


Figure 5.2.2: Use case 2 setup: Button + LED

Recursive Way

- **Steps**:

  1. **Setup**: Establish the internal LED as an output and the button as an input. As shown in Use Case 1, you need to configure the pin mode with **`pinMode`** before using the pins.

     (a) **LED:** GPIO 4 as OUTPUT.

     (b) **Button:** GPIO 6 as INPUT.

     This is the assembly program:

```
.data
   buttonPin: .word 6
   ledpin: .word 4
.text
setup:
    #pinMode(buttonPin,INPUT_PULLUP)
    la a0, buttonPin
    lw a0, 0(a0)
    li a1, 0x05 #INPUT_PULLUP
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, pinMode
    lw ra, 0(sp)
    addi sp, sp, 4

    #pinMode(ledpin,OUTPUT)
    la a0, ledpin
    lw a0, 0(a0)
    li a1, 0x03 #OUTPUT
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, pinMode
    lw ra, 0(sp)
    addi sp, sp, 4

    jr ra
```

2. **Loop**: Read button state. Once set up, we start the infinite loop by reading the button state with **`digitalRead`**, which only requires the button pin (in this case, pin 6).

   In our case, if the button is pressed, the `button_pressed` function is called. Otherwise, the LED will remain off.

   Also, a small `delay` is added to avoid overloading the system.

```
.data
    time: .word 100
.text
loop:
    la a0, buttonPin
    lw a0, 0(a0)
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, digitalRead
    lw ra, 0(sp)
    addi sp, sp, 4

    mv t0,a0

    li t1 ,0 #LOW

    beq t0,t1,button_pressed

    la a0, ledpin
    lw a0, 0(a0)
    li a1, 0x0
    jal ra, digitalWrite

    la a0, time
    lw a0, 0(a0)
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, delay
    lw ra, 0(sp)
    addi sp, sp, 4

    j loop
```

3. **`button_pressed`**: Action when the button is pressed. As shown in Use Case 1, we will turn on the LED when the button is pressed, as shown below:

```
button_pressed:
  la a0, ledpin
  lw a0, 0(a0)
  li a1, 0x1
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, digitalWrite
  lw ra, 0(sp)
  addi sp, sp, 4

  la a0, time
  lw a0, 0(a0)
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, delay
  lw ra, 0(sp)
  addi sp, sp, 4

  jr ra
```

Using Interruptions

Another way to achieve this project is to use GPIO interrupts, which are more immediate but more complex to program.

In this case, we will use the Arduino functions `attachInterrupt` and `digitalPinToInterrupt` to obtain the interrupt number to assign to the interrupt service routine for that pin.

1. **Setup**: Establish the internal LED as an output and the button as an input. As shown in Use Case 1, you need to configure the pin mode with **`pinMode`** before using the pins.

   Then we connect the button pin to an interrupt service routine (ISR), which we call `blink`, that runs automatically when the button is pressed. For this, we will use the `attachInterrupt` function.

   The `attachInterrupt` function takes the following parameters:

   - The interrupt position corresponding to the pin for which we want to detect the interrupt (we use `digitalPinToInterrupt(pin)`).
   - The memory address where the interrupt service routine is located (in this case, we call it `blink`).
   - The interruption mode the ISR will follow.

Mode	Value	Usage
RISING	0x01	Interrupt triggered on the rising edge (when the pin changes from LOW to HIGH)
FALLING	0x02	Interrupt triggered on the falling edge (when the pin changes from HIGH to LOW)
CHANGE	0x03	Interrupt triggered on any change of the pin state (both LOW→HIGH and HIGH→LOW)
ONLOW	0x04	Interrupt triggered while the pin remains LOW
ONHIGH	0x05	Interrupt triggered while the pin remains HIGH
ONLOW_WE	0x06	Same as ONLOW, but with write enable — allows modifications or writing to related registers while the pin is LOW
ONHIGH_WE	0x07	Same as ONHIGH, but with write enable

Table 5.2.2: GPIO Interrupt Modes.

In this case, because we want the interrupt to occur when the button is pressed, we select ON_LOW. This is how the setup will look:

```
.data
    ledPin: .byte 4
    interruptpin: .byte 6
    state: .byte 0 #LOW
    on_low: .byte 0x04
.text
setup:
    #Start pins
    la t1, ledPin
    lb a0, 0(t1)
    li a1, 0x03 #OUTPUT
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, pinMode #pinMode(ledPin, OUTPUT);
    lw ra,0(sp)
    addi sp, sp, 4
    la t1, interruptpin
    lb a0, 0(t1)
    li a1, 0x05 #INPUT_PULLUP
    addi sp, sp, -4
    sw ra,0(sp)
        jal ra, pinMode# pinMode(ledPin, INPUT_PULLUP);
    lw ra,0(sp)
    addi sp, sp, 4

    la t1, interruptpin
    lb a0, 0(t1)
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, digitalPinToInterrupt #digitalPinToInterrupt(interruptpin);
    lw ra,0(sp)
```

```
    addi sp, sp, 4

    la a1, blink

    la t1, on_low
    lb a2, 0(t1)

    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, attachInterrupt #attachInterrupt(digitalPinToInterrupt(interruptPin), blink,
         ON_LOW);
    lw ra,0(sp)
    addi sp, sp, 4
    jr ra
```

2. **`Blink`**: ISR definition. In this case, we want the LED status to change when an interruption is detected.

   Specifically, we will modify a variable stored in memory that indicates the LED's state, as follows:

```
blink:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw t0, 0(sp)

    la t0, state
    lb a0, 0(t0)
    xori a0, a0, 1 # 0->1, 1->0
    sb a0, 0(t0)
    lw t0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    jr ra
```

3. **Loop**: Change LED state.

   LED will be turned off until the button is pressed, as shown in the following code:

```
loop:
    la t1, ledPin
    lb a0, 0(t1)
    li a1,0
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, digitalWrite# digitalWrite(ledPin, state)
    lw ra,0(sp)
    addi sp, sp, 4

    li a0, 100
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, delay # delay(100)
    lw ra,0(sp)
    addi sp, sp, 4
    j loop
```

```
18
19 setup:
20     #Start pins
21     la t1, ledPin
22     lb a0, 0(t1)
23     li a1, 0x03 #OUTPUT
24     addi sp, sp, -4
25     sw ra,0(sp)
26     jal ra, pinMode #pinMode(ledPin, OUTPUT);
27     lw ra,0(sp)
28     addi sp, sp, 4
29     la t1, interruptpin
30     lb a0, 0(t1)
31     li a1, 0x05 #INPUT_PULLUP
32     addi sp, sp, -4
33     sw ra,0(sp)
34     jal ra, pinMode# pinMode(ledPin, INPUT_PULLUP);
35     lw ra,0(sp)
36     addi sp, sp, 4
37     la t1, interruptpin
38     lb a0, 0(t1)
39     addi sp, sp, -4
40     sw ra,0(sp)
41     jal ra, digitalPinToInterrupt #digitalPinToInterrupt(interruptpin);
42     lw ra,0(sp)
43     addi sp, sp, 4
44
45     la a1, blink
46
47     la t1, change
48     lb a2, 0(t1)
49
50     addi sp, sp, -4
51     sw ra,0(sp)
52     jal ra, attachInterrupt #attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE)
           ;
53     lw ra,0(sp)
54     addi sp, sp, 4
55     jr ra
```

This is the result: In both cases, the LED turns on when the button is pressed.

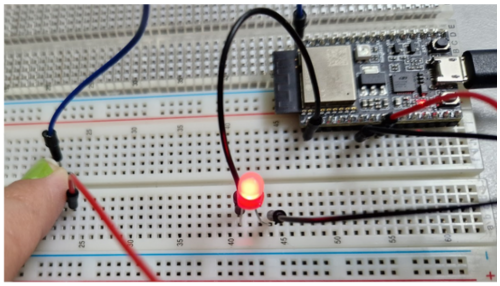


Figure 5.2.3: Use Case 2 result.

5.2.3. Use Case 3: Text Input/Output Using Serial Output

For this use case, we will use a few basic functions from the Arduino Serial library. For more information, see the official documentation (not all functions are available in this library, as they are not well-suited to the environment we are working in): `https://docs.arduino.cc/language-reference/en/functions/communication/serial/`.

By definition, serial input functions are very fast (they do not wait for you to press Enter) and do not display a callback of what has been written. The purpose of this library is to familiarize you with these functions; therefore, **it is the user's responsibility to monitor each step** of what is being done.

Just as we previously used the `ecall` instruction to print messages in assembly CREATOR programs, here we will use the `serial_printf` function, which is part of Espressif's Arduino component.

Unlike higher-level functions like `Serial.print` and `Serial.println`, `serial_printf` requires explicit format specifiers to indicate the data types being printed; otherwise, the output is interpreted as a string.

On the other hand, we will use the `serial_readBytes` function to see how text input works. There is the `serial_read` variant (which only takes one character), `serial_readBytesUntil` (which stops storing characters when it finds the specified character), and those that parse, such as `serial_parseInt` (which only takes numbers).

- **Components:**
  - ESP32-C3-DevKitC-02 board.
- **Steps**:
  1. **Data**: Save messages to print in memory. For this use case, we are going to declare in the data section:
     - The initial message to be displayed.
     - The buffer allocated to store the input text.
     - The message that includes a placeholder for outputting the entered text.
     - An auxiliary callback used to display the entered number.

     This would be reflected in the code as follows:

```
.data
  space: .zero 100 #Buffer to place the string
  initial: .string "Introduce number of letters:\n"
  aux: .string "%d\nType your message\n"
  print: .string "Your message: %s\n"
```

  2. **Serial**: Start terminal output and input. On most boards, such as Arduino and Espressif, the terminal must be initialized at a specific frequency. In this case, we will use the `serial_begin` function with a baud rate of 115200.

     If a value other than the one given is entered, strange characters may be printed, or nothing may be printed at all.

It is declared in the code as follows:

```
setup:
    li a0,115200
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_begin
    lw ra, 0(sp)
    addi sp, sp,4
    jr ra
```

3. **Loop**: Check if the terminal is correctly open. A good practice in Arduino is to use `serial_available` to verify that the terminal has opened correctly.

   If its value is greater than 0, it is operational and ready to use.

   > For simplicity's sake, you will often find that this check is not performed... but using `serial_available` is a very good error control.

   So, we start our loop as follows:

```
loop:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_available
    lw ra, 0(sp)
    addi sp, sp, 4
    beqz a0,aux_print
    j loop
```

   - `aux_print` is an auxiliary function to indicate to the user that they should enter the number only once. This is purely aesthetic, but it can improve user clarity.

```
aux_print:
    #serialPrintf
    la a0,initial
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_printf
    lw ra, 0(sp)
    addi sp, sp, 4
    j read_num
```

   - `read_num`: read a number from terminal.

     In this case, we use the `serial_parseInt` function to get a number per terminal. This function is used instead of `serial_read` because `serial_read`, although it returns a number, interprets the input as ASCII.

     If we enter the number "4" via the terminal, the values change:

     * In serial_read: 52
     * In serial_parseInt: 4

Since `serial_read` is very fast, we will not move on to the next step if there is no number greater than 0 collected in a0, leaving the following code snippet:

```
read_num:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_parseInt
    lw ra, 0(sp)
    addi sp, sp,4
    mv t0, a0
    bnez t0, print_int
    j read_num
```

A small auxiliary function called `print_int` was created to view the callback of the value specified as length, as follows.

```
print_int:
    la a0, aux
    mv a1,t0
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_printf
    lw ra, 0(sp)
    addi sp, sp, 4
    j read_function
```

- `read_function`: Read the text of the requested length. Once we have the string length, we read the text to add with `serial_readBytes`, which blocks until the requested number of characters has been read.
  We would end up with an implementation like this:

```
read_function:
    la a0, space
    mv a1, t0 # number of letters it will have
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_readBytes
    lw ra, 0(sp)
    addi sp, sp,4
    bne t0,a0,read_function
    la a0, print
    la a1, space
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_printf
    lw ra, 0(sp)
    addi sp, sp, 4
    jr ra
```

And the result is:

```
Started program...
---Introduce number of letters:
4
Type your message
Your message: hola
Finished program: 283430235 cycles
```

Figure 5.2.4: Use Case 3 result.

5.2.4. Use Case 4: Daytime Running Lights with `analogRead`

This time, we are going to make a small LED that lights up when it detects low light, using an LDR.

> The sensor's sensitivity may vary depending on the user's location and device. We recommend keeping a flashlight or a light source handy and using the debugger or print statements.

- **Components:**
  - ESP32-C3-DevKitC-02 board.
  - LDR or photoresistance (in GPIO 2).
  - LED (in GPIO 4).
  - 10kOhm resistance.

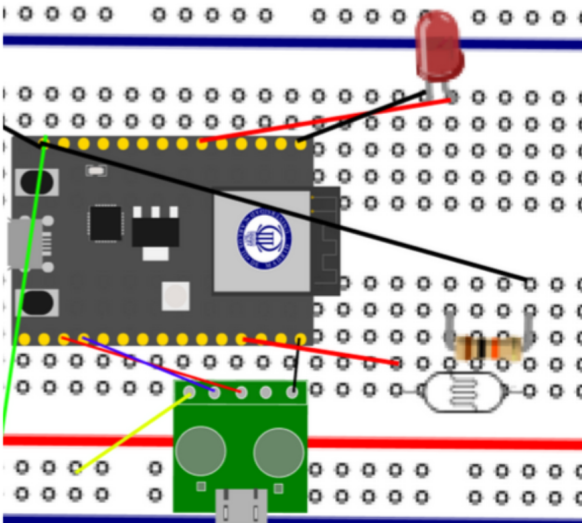


Figure 5.2.5: Setup case 4.

> To find resistors with the correct value, look at the color code on their bands. You can either look at the color guide, use this calculator, or simply look for a resistor that matches the one in the image.

MECATRONIUM CHIPS	CÓDIGO DE COLORES DE RESISTENCIAS				
COLOR	VALOR 1	VALOR 2	VALOR 3	MULTIPLICADOR	TOLERANCIA
NEGRO	0	0	0	1Ω	± 1%
MARRÓN	1	1	1	10Ω	± 2%
ROJO	2	2	2	100Ω	
NARANJA	3	3	3	1KΩ	
AMARILLO	4	4	4	10KΩ	
VERDE	5	5	5	100KΩ	± 0.5%
AZUL	6	6	6	1MΩ	± 0.25%
VIOLETA	7	7	7	10MΩ	± 0.10%
GRIS	8	8	8		± 0.05%
BLANCO	9	9	9		
ORO				0.1Ω	± 5%

Figure 5.2.6: Resistance color guide.

- **Steps**:

1. **Setup**: Initialize LDR, pin, and serial output. To use the sensor, we must initialize it with `pinMode` to INPUT (0x01), as shown in Figure 2.1. The setup code would look like this:

```
.data
    lightSensorPin: .word 2
    ledPin: .word 4
    time: .word 100
    aux_msg: .string "LDR value: %d\n"

.text
setup:
#Serial
    li a0,115200
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_begin
    lw ra, 0(sp)
    addi sp, sp,4
# PinMode LED
    la t1, ledPin
    lw a0, 0(t1)
    li a1, 0x03 #OUTPUT
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, pinMode #pinMode(ledPin, OUTPUT);
    lw ra,0(sp)
    addi sp, sp, 4
# Light sensor pin
    la t1, lightSensorPin
    lw a0, 0(t1)
    li a1, 0x01
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, pinMode# pinMode(lightSensorPin, INPUT);
    lw ra,0(sp)
    addi sp, sp, 4

    jr ra
```

2. **Loop**: Analog sensor reading. Once everything is up and running, we read the light level in the room using `analogRead`.

   In this tutorial, after a few tests, we found that the values reached were **between 1100 and 1400**. The darker it is, the higher the resistance value. **Therefore, a threshold of 1200 is set.**

   - If the sensor reaches a value higher than 1200, the LED will turn on (`lightUp`).
   - If, on the other hand, it does not reach that value, the LED remains off (`turnDown`).

   In both cases, we add a small `delay` to slow the readings.

   We finish with this code:

```
            # delay
            la a0, time
            lw a0, 0(a0)
            addi sp, sp, -4
            sw ra, 0(sp)
            jal ra, delay
            lw ra, 0(sp)
            addi sp, sp, 4

            j loop

        loop:
        #read LDR
            addi sp, sp, -4
            sw ra,0(sp)
            jal ra, analogRead #analogRead(lightSensorPin);
            lw ra,0(sp)
            addi sp, sp, 4
        # Print value
            mv t1, a0
            mv a1,a0
            la a0,aux_msg
            addi sp, sp, -4
            sw ra, 0(sp)
            jal ra, serial_printf
            lw ra, 0(sp)
            addi sp, sp, 4
        # turn up led
            li t0, 1200 #Minumun value
            bgt t1, t0, lightUp
            blt t1, t0, turnDown

            j loop
```

And this is the result:

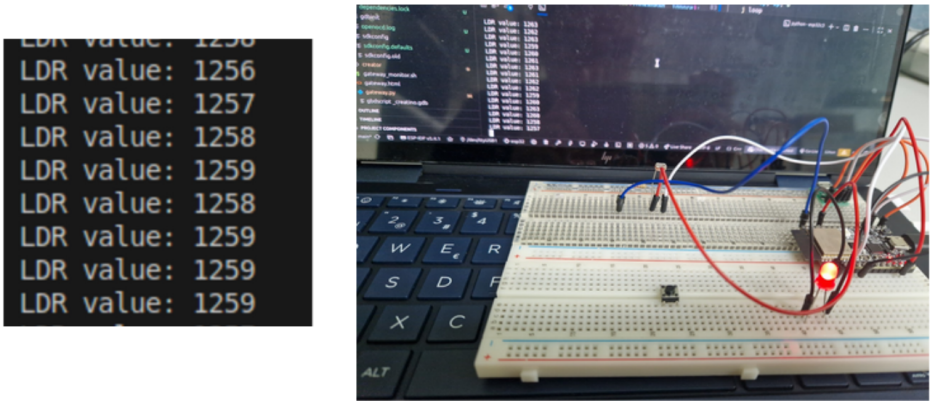


Figure 5.2.7: Arduino Use Case 4 result.

5.2.5. Use Case 5: [Advanced] Piano Using `tone()`

This use case is slightly more complex than the others, but very creative. It is recommended to have a breadboard with plenty of space.

Each musical note corresponds to a frequency that drives air into the buzzer membrane. Consult this page to learn how to calculate the other possible notes according to the desired tempo: `https://www.tibotinspirationlab.com/reto/musica-con-arduino/`

In this case, we will proceed with interruptions. This example may incur slightly more latency because it involves continuous memory accesses and because the `tone` function launches tasks.

Polling could be used.

- **Components:**
  - ESP32-C3-DevKitC-02 board.
  - 4 buttons.
  - A passive buzzer.

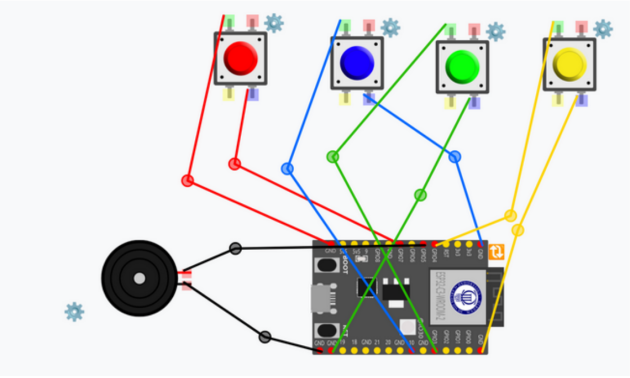


Figure 5.2.8: Setup to Use Case 5.

- **Steps**:

1. **Data**: Declaration of pins and values of musical notes. We indicate in the data which pins we will use (avoiding "dangerous" pins such as GPIO 8 and GPIO 1) and the frequency values of the musical notes.

   Care must be taken with the placement of these values, as we will use these memory positions as arrays.

   In this example, we have positioned them as follows:

```
.data
    # GPIO
    buzzerPin: .word 5
    button_C4: .word 7
    button_D4: .word 10
    button_E4: .word 3
    button_G4: .word 4

    # Notes
    note_C4: .word 262
    note_D4: .word 294
    note_E4: .word 330
    note_G4: .word 392
    #aux
    button_count: .word 4
    time_delay: .word 10
    anyPressed: .word -1
    .align 4
    button_handlers:
      .word handleButton0
      .word handleButton1
      .word handleButton2
      .word handleButton3
```

   Another way to position them is like this:

```
.data
    # GPIO
    buzzerPin: .word 5
    buttons: .word 7,10,3,4
    # Notes
    notes: .word 262,294,330,392
    #aux
    button_count: .word 4
    anyPressed: .byte -1
 .align 4
    button_handlers:
      .word handleButton0
      .word handleButton1
      .word handleButton2
      .word handleButton3
```

2. **Setup**: Configure all pins. In this case, we will complicate things slightly, as there are many pins to configure. First, we will configure pin 5 of the buzzer as an OUTPUT mode (since it emits sound externally).

```
setup:
#buzzer
    la a0, buzzerPin
    lw a0, 0(a0)
    li a1, 0x03 #OUTPUT
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, pinMode
    lw ra, 0(sp)
    addi sp, sp, 4
```

Then, to configure the remaining buttons, we created a loop that iterates over all selected buttons. First, in `setup`, after initializing the buzzer, we initialize the variables used in the loop.

```
#buttons
la s0, button_C4 #Button list
li s1, 0 # Position in the list
la t3, button_count
lw s2, 0(t3) #List length
```

Next, we move on to the `loop_buttons` function, where we will:

(a) Check if we have already gone through the entire list.

```
loop_buttons:
    bge s1, s2, end_loop_buttons # loop until all the buttons are positioned
```

(b) If this is not the case, we scroll through the data to find the PIN number we want.

```
# Take position of the button
slli s3, s1, 2
add t1, s0, s3 #shift
lw s4, 0(t1) #Button value
mv a0,s4
```

(c) Once the position has been obtained, we configure the button with `INPUT_PULLUP` mode, as they are buttons.

```
li a1, 0x05 #INPUT_PULLUP
addi sp, sp, -4
sw ra, 0(sp)
jal ra, pinMode
lw ra, 0(sp)
```

(d) As we indicated in the statement, we are going to use interrupts, so we have to assign them now, knowing which button we are on (that is, we take advantage of the fact that the order of `button_handlers` is the same as the order followed by the button array).
For more information on assigning interrupts, see use case 2. We will indicate what the ISRs are like in the next point.

```
#----Attach Interrupts
#Transform digitalPin into an interrupt
mv a0, s4
addi sp, sp, -4
sw ra,0(sp)
jal ra, digitalPinToInterrupt #digitalPinToInterrupt(interruptpin);
lw ra,0(sp)
addi sp, sp, 4
# Search the correct pointer
la t0, button_handlers
add t1, t0, s3
lw a1, 0(t1)
# Let the interrupt jump when the button is pressed (on_low)
li a2, 0x03
addi sp, sp, -4
sw ra,0(sp)
jal ra, attachInterrupt #attachInterrupt(digitalPinToInterrupt(buttonPin[i]), handler[
    i], ON_LOW);
lw ra,0(sp)
addi sp, sp, 4
```

(e) We add up a position and re-enter the loop.

```
addi s1, s1, 1 # next button
   j loop_buttons
```

(f) If we have finished going through the list, we return to the main by executing a ret (or a jr ra).

```
end_loop_buttons:
   ret
```

3. **Loop**:

   This step can be implemented in two ways: polling (i.e., repeatedly checking the status of all buttons) or interrupts (pressing the button triggers an interrupt).

   The easiest way to do this exercise is to assign an interrupt to each button and, when pressed, have each one have its own ISR.

   Therefore, before starting the loop, we will create each required ISR and assign it to its corresponding button. Refer to use case 2, which uses interrupts, for a better understanding.

   In this case, we use a "flag" change since the `tone()` function underneath launches "tasks" or "processes" underneath, which is not safe to do in an ISR.

```
#ISR
handleButton0:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 0 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton1:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 1 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton2:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 2 # Assign button
    sw t2, 0(t0)
    jr ra

handleButton3:
    la t0, anyPressed
    lw t1, 0(t0)
    li t2, 3 # Assign button
    sw t2, 0(t0)
    jr ra
```

Then, inside the loop, we check the status of the `anyPressed` flag and add a small `delay` to prevent the watchdog from tripping. This loop checks whether the flag is not -1, and if so, plays the tone for that position.

```
loop:
    la t0, anyPressed
    lw s0,0(t0)
    li t1, -1
    bne s0, t1, startTune
    la t0, time_delay
    lw a0, 0(t0)
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, delay #delay(200);
    lw ra,0(sp)
    addi sp, sp, 4
    j loop
```

4. `startTune`: start playing the tone.

   In this case, once we have the pressed button's position, we search the memory array for the corresponding note value, as we did when searching for the button.

   To prevent the tone from lasting indefinitely, we have set a timeout of 200 ms. Once the tone is played, it returns to the loop.

```
startTune:
    # Clean value
    la t0, anyPressed
    li t1, -1
    sw t1, 0(t0)
    #Search value
    la t0, note_C4
    slli s1, s0, 2
    add t1,s1,t0
    lw s2, 0(t1) #Note value
    #Play Tone
    la t0, buzzerPin
    lw a0, 0(t0)
    mv a1,s2
    li a2, 200
    addi sp, sp, -4
    sw ra,0(sp)
    jal ra, tone #tone(buzzerPin, notes[i], duration);
    lw ra,0(sp)
    addi sp, sp, 4
    j loop
```

And there we have our piano:

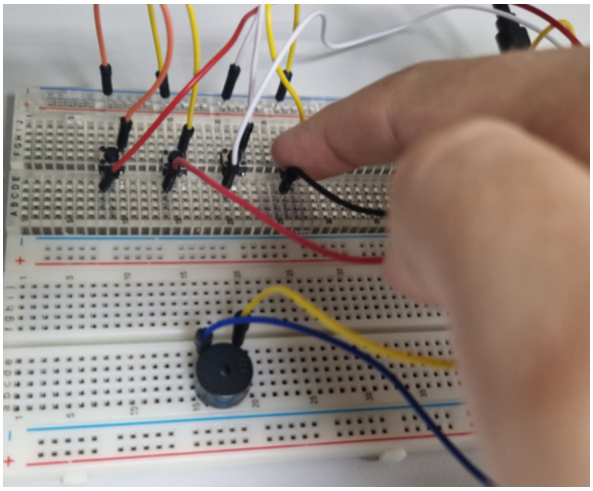


Figure 5.2.9: Arduino Case 5 Result.

6. REMOTE LABORATORY SERVICE USER MANUAL

This section introduces the remote laboratory service that enables users to execute assembly programs on real hardware via multiple CREATOR Gateways.

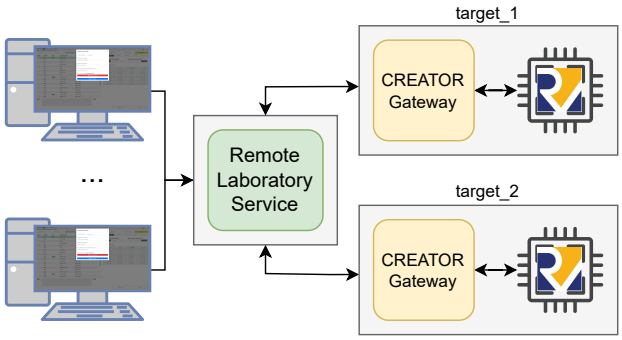


Figure 6.0.1: Deployment of the remote laboratory service.

Specifically, the steps required to deploy the remote laboratory service using ESP32 C3 microcontrollers, which are currently the only ones supported, will be presented in Section 6.1. Next, in Section 6.2, a use case will be presented to explain how to use this remote laboratory service from the CREATOR web simulator.

6.1. Remote Laboratory Service Deployment

To facilitate the deployment of the remote laboratory service, a Docker container has been provided [7]. The steps to deploy the remote laboratory service using this Docker container and Docker Compose are detailed below.

First, create a file structure such as this:

```
creator-remote-lab/
|- compose.yaml
|- results/
|- config/
| |- deployment.json
```

Second, on the remote laboratory service container configuration:

- `results/` must be mounted to `/app/results/` and `config/` to `/app/config/`.
- The server will listen on port `5000`, so we should expose that port.
- To send the results via email, you must set your Google Mail address in the `EMAIL` environment variable and your app password in the `PASSW` environment variable.

[7] `https://hub.docker.com/repository/docker/creatorsim/creator-remote-lab`

Third, a gateway must be deployed for each microcontroller you intend to use in the remote laboratory service.

To simplify this deployment, all these steps can be performed using Docker Compose. To do this, the `compose.yaml` file must be as follows:

```yaml
services:
  creator-remote-lab:
    image: creatorsim/creator-remote-lab:latest
    volumes:
      - ./config:/app/config
      - ./results:/app/results
    ports:
      - "5000:5000"
    environment:
      - EMAIL=test@inf.uc3m.es # google mail address
      - PASSW=abcdefghijklmnop # google app password

  target-0:
    image: creatorsim/creator-gateway-esp32:latest
    devices:
      - /dev/ttyUSB0
    stdin_open: true
    tty: true

  target-1:
    image: creatorsim/creator-gateway-esp32:latest
    devices:
      - /dev/ttyUSB1
    stdin_open: true
    tty: true
```

The configuration of the target boards is placed in `config/deployment.json`. You must provide the type of board (`target_board`), the USB port of the board (`target_port`), and the URL of the gateway server (`target_url`). As we're doing everything with Docker Compose, we can just use their container names and prevent exposing the ports to `localhost`. An example of a `deployment.json` file using Docker Compose would be as follows:

```json
{
  "target-0": {
    "target_board": "esp32c3",
    "target_port": "/dev/ttyUSB0",
    "target_url": "http://target-0:8080"
  },
  "target-1": {
    "target_board": "esp32c3",
    "target_port": "/dev/ttyUSB1",
    "target_url": "http://target-1:8080"
  }
}
```

Then, we can deploy everything with:

```
docker compose up
```

Finally, the results of all executions performed in the remote laboratory service will be stored in `results/` directory.

6.2. Remote Laboratory Service Use Case

When developing a program in CREATOR, the common execution workflow begins with using the text editor integrated into CREATOR to implement the assembly program shown in Figure 6.2.1. This program is designed to calculate the factorial of 10 using a recursive subroutine and to measure the number of clock cycles required for the calculation. Additionally, it displays the number of clock cycles used on-screen once the factorial has been computed.

```
.text

# Subroutine that calculates the factorial
factorial:
   # create stack frame
   addi sp, sp, -12
   sw ra, 8(sp)
   sw fp, 4(sp)
   addi fp, sp, 4

   # if (a0 < 2):
   li x5, 2
   bge a0, t0, b_else

   # a0 = 1
   # return a0
   li a0, 1
   beq x0, x0, b_efs

   # else:
 b_else:

   # a0=a0*factorial(a0-1)
   # return a0
   sw a0, -4(fp)
   addi a0, a0, -1

   jal x1, factorial

   lw t1, -4(fp)
   mul a0, a0, t1

 b_efs:
   # restore the stack
   lw ra, 8(sp)
   lw fp, 4(sp)
   addi sp, sp, 12

   # return a0
   jr ra


 main:
    # create stack frame
    addi sp, sp, -20
    sw ra, 16(sp)
    sw s0, 12(sp)
    sw s1, 8(sp)
    sw s2, 4(sp)
    sw s3, 0(sp)

    # s=0
    li s1, 0
    li s0, 3
    # while (s1 < 3) {
w1: bge s1, s0, w1_end

    # s2=get_cycles();
    # factorial(10);
    # s3=get_cycles()
    rdcycle s2
    li a0, 10
    jal x1, factorial
    rdcycle s3

    # print(s3-s2)
    sub a0, s3, s2
    li a7, 1
    ecall

    # s1++
    addi s1, s1, 1
    beq zero, zero, w1
    # }

w1_end:
    # restore the stack
    lw s3, 0(sp)
    lw s2, 4(sp)
    lw s1, 8(sp)
    lw s0, 12(sp)
    lw ra, 16(sp)
    addi sp, sp, 20
    jr ra
```

Figure 6.2.1: RISC-V program that calculates the factorial of 10.

It is important to highlight that this program gets the CPU cycles in three consecutive executions of the factorial. This feature allows users to observe how the microcontroller's cache memory reduces the number of cycles required to execute the same code. This cache effect is often not detectable in existing assembly simulators.

After implementation, we must compile the program to ensure there are no errors in the code. If the compilation process is error-free, we can proceed to the next step in the workflow: running the program in CREATOR to verify its expected performance and to avoid execution errors before running it on real hardware.

Once we have verified that execution is as expected using CREATOR, the program can be sent to the remote laboratory service for execution on a real device. The Remote Device Target Flash menu can be accessed from Tools → Flash → Remote Device in the simulator view.

Next, the form fields shown in Figure 6.2.2 will be filled in to do this.

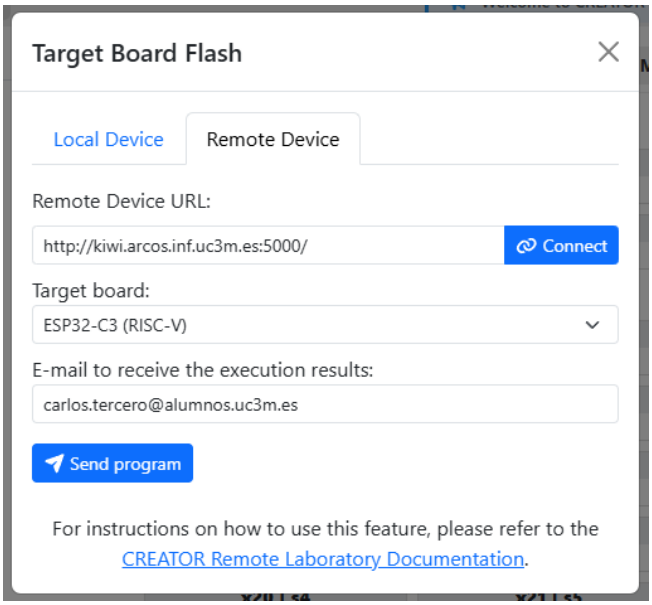


Figure 6.2.2: Form to send the assembly program to the remote laboratory service.

1. Connect to the Remote Device URL where the server is deployed (e.g. `http://kiwi.arcos.inf.uc3m.es:5000`).
2. Select one of the available board types that the server supports.
3. Enter your E-mail address to receive the execution results.
4. Execute your program by using the *Send program* button.

When the assembly program is sent to the remote lab service, it is stored in the request queue and waits for a real device to become available, as shown in Figure 6.2.3. Also, you can cancel the execution by using the *Cancel last program* button.

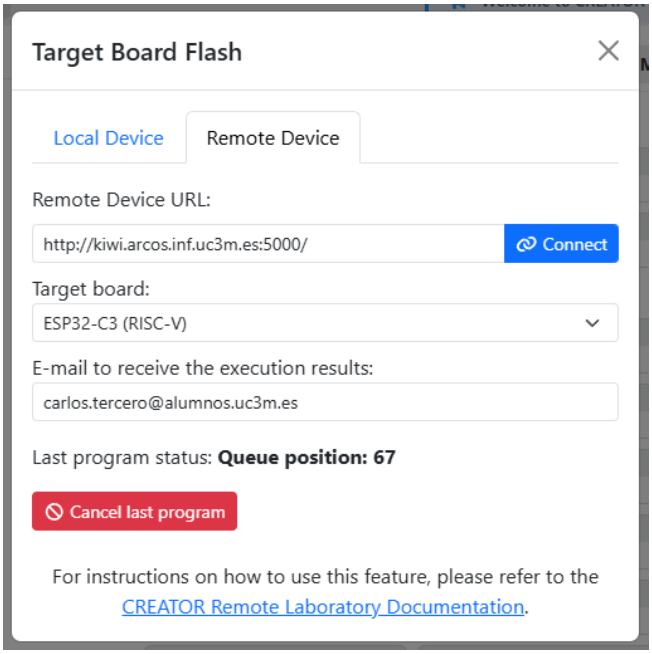


Figure 6.2.3: Form for sending programs to the remote laboratory service during the execution of a program on the device.

When a real device is available, the program will be executed on the microcontroller, and, finally, the results will be sent back by E-mail to the user, as shown in Figure 6.2.4.

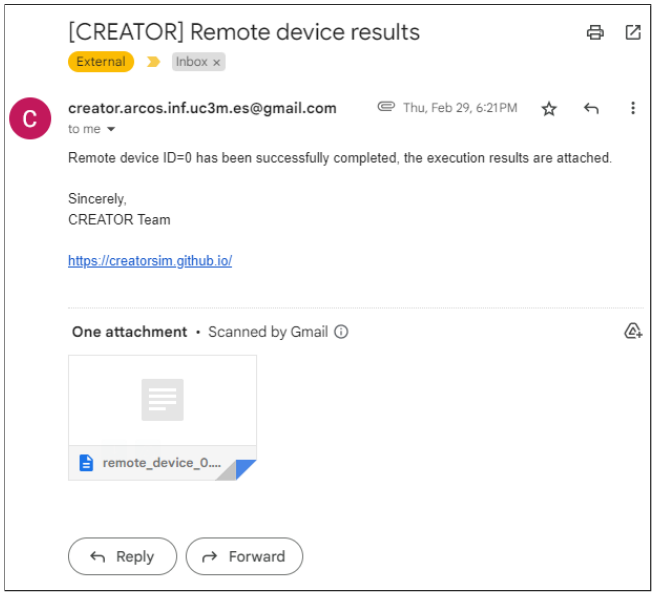


Figure 6.2.4: E-mail sent by the remote laboratory service with the results of the requested on-device execution.

In Figure 6.2.5, you can see the output of running the *flash* action on the microcontroller to load the assembly program to be executed into memory. Next, you can see the output of executing the *monitor* action on the microcontroller to see the output of the program execution. This is the program output that the user receives by E-mail.

```
...
Configuring flash size...
Flash will be erased from 0x00000000 to 0x00005fff...
Flash will be erased from 0x00010000 to 0x00051fff...
Flash will be erased from 0x00008000 to 0x00008fff...
Compressed 20832 bytes to 12760...
Writing at 0x00000000... (100 %)
Wrote 20832 bytes (12760 compressed) at 0x00000000 in
     0.7 seconds (effective 243.5 kbit/s)...
Hash of data verified.
Compressed 268928 bytes to 107296...
Writing at 0x00010000... (14 %)
Writing at 0x0002e3b0... (57 %)
Writing at 0x0004d967... (100 %)
Wrote 268928 bytes (107296 compressed) at 0x00010000 in
      3.9 seconds (effective 544.7 kbit/s)...
Hash of data verified.
```

```
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.1
     seconds (effective 413.3 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done

Started program...
-------------------
>1382
>220
>220
Finished program: 53638 cycles
-------------------
```

Figure 6.2.5: Results of program execution in the remote laboratory service sent by E-mail.

If we analyze the result of the execution of the assembly program presented in Figure 6.2.5. It can be noted that the first time the factorial of 10 is computed, 1,382 clock cycles are required, whereas only 220 cycles are needed for subsequent calculations. This is because the code to be executed is in cache memory, and reading it from main memory is unnecessary, reducing the number of cycles the `factorial` subroutine must execute.

BIBLIOGRAPHY

[1] Espressif, *Espressif ESP32-C3 Datasheet*. Accessed: Dec. 26, 2025. [Online]. Available: `https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf`.