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 1.2
New development environment design report

Universidad Carlos III de Madrid

January, 2026

CONTENTS

1. SUPPORT FOR INTEGRATION AND DEBUGGING OF DIFFERENT RISC-V BOARDS AND MICROCONTROLLERS

This chapter will address the real-hardware support expansion with RISC-V-based microcontrollers, such as the development boards from Espressif (ESP32) or the recent SBCs with RISC-V processors released with Ubuntu support (OrangePi, Nezda D1-H).

These new functions aim to make it easier for users to test their projects on real-hardware without the learning curve of Espressif's framework (ESP-IDF) or the need to manage SBC systems directly.

These features were introduced in the Jornadas SARTECO Conference on June 26 in Sevilla, Spain [1].

1.1. New Features and Interface Improvements in CREATOR's *Target Flash Menu*

This new features can be visualized in *Target Board Menu* (illustrate in Figures 1.1.1 and 1.1.2), where the user can choose:

- The kind of **device** they want to use (ESP32 development boards or Ubuntu-based RISC-V SBCs).
- In case of Espressif devices, the **development boards model** to use (ESP32-C3, ESP32-C6, or ESP32-H2).
- **Configuration parameters** related to the driver's and device configuration (target port, flash URL...).
- **Buttons** offering the **new functionalities** on the board, connected to the driver displayed.

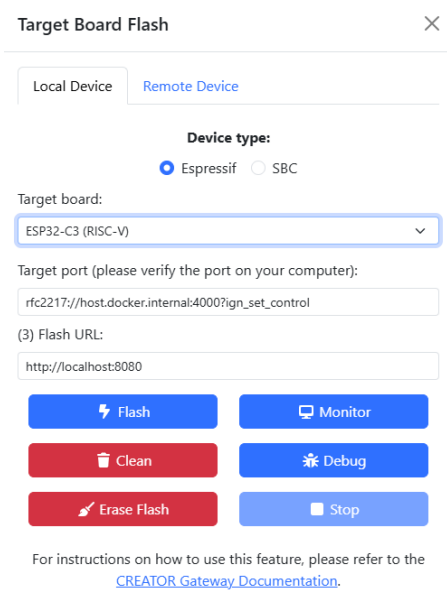


Figure 1.1.1: Espressif Interface.

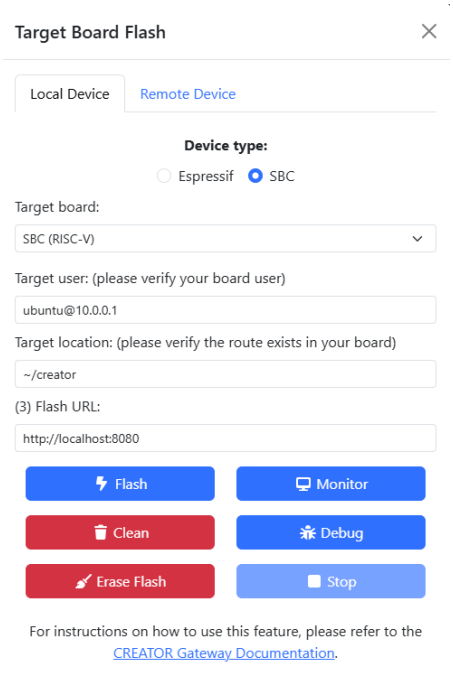


Figure 1.1.2: SBC Interface.

1.2. Extension of Real-Hardware Management Driver on ESP32 Development Boards

This extension is suitable for development boards based on the ESP32-C3, ESP32-C6, and ESP32-H2 SoCs. Also, this new driver, with new features, can run natively on Linux/macOS or via Docker.

1.2.1. Integration of New Functionalities Using ESP-IDF

The new functionalities are fully supported by Espressif framework ESP-IDF, using v.5.3.2 version. Integration of these new features will be explained in detail below:

- **Clean**: Code's build inside the environment chosen is erased.
- **Erase Flash**: Code flashed inside the development board is erased.
- **Flash**: Code written in CREATOR's editor is built and flashed into the development board using `idf.py` functions by ESP-IDF, as shown in Figure 1.2.1.

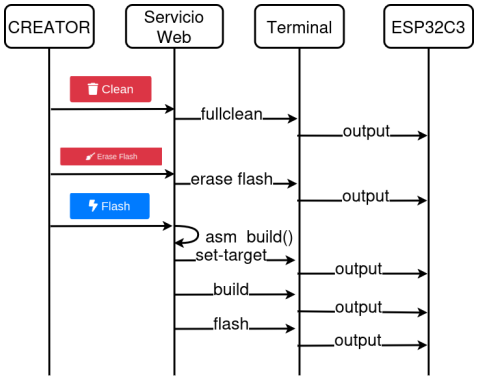


Figure 1.2.1: Clean and flash functions.

- **Monitor**: Code flashed correctly into the development board is executed on the driver's terminal.
- **Debug**: An external tab with GDB UI (gdbgui) is displayed to debug the development board flashed. This operation requires OpenOCD and a JTAG connector between the board and the computer, as shown in Figure 1.2.2.

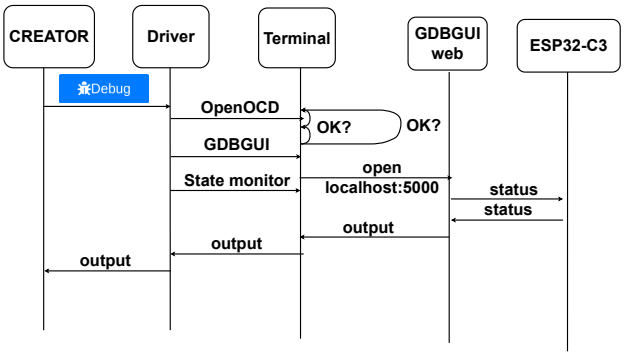


Figure 1.2.2: Debug function order.

1.2.2. Modification of ESP-IDF Panic Handler to Support CREATOR's `ecall` Functions

On the other hand, ESP32 devices detect `ecall` instructions because they are RISC-V-based microprocessors. However, Espressif's framework detects them as Unhandled Exceptions, as the TMR manual of ESP32-C3 claims [2]. Using wrapping compilation techniques, `ecall` handlers are added to match CREATOR's `ecall` instructions response. `ecall` functions regarding float or double numbers are excluded, as this development board does not have FPU support at the moment.

Call Code	Input Registers	Output Registers	Description
1	a0	-	Prints an int number
4	a0	-	Prints a string
5	-	a1	Writes a number asked on terminal inside a register
8	a0, a1	a0	Reads and writes a string inside a memory position (a0) with a length given (a1)
9	a0	a0	SBK
10	-	-	Ends program execution
11	a0	-	Prints a char
12	-	a0	Reads a char on terminal and writes it down on a0

Table 1.2.1: CREATOR's ESP32 system call interface summary.

An example of the correct implementation of the new system calls can be seen in Listing 1.1, and its correct execution on microcontrollers in Figure 1.2.3.

Listing 1.1: RISC-V program using syscall to read and write strings

```
.data
    string1: .string "Insert the string length (no more than 100 characters) "
    string2: .string "Insert the string "
    space: .zero 100

.text
main:
    # print "Insert string length..."
    la a0, string1
    li a7, 4
    ecall

    # read int
    li a7, 5
    ecall

    add t0, a0, zero

    # print "Insert string..."
    la a0, string2
    li a7, 4
    ecall


```

```
    # read string
    la a0, space
    add a1, t0, zero
    li a7, 8
    ecall

    # print string
    la a0, space
    li a7, 4
    ecall

    # return
    jr ra
```

```
I (283) coexist: coexist rom version 5b8dcfa
I (284) main_task: Started on CPU0
I (284) main_task: Calling app_main()
Started program...
------------------
Insert the string length (no more than 100 characters)
5
Insert the string
hello
hello
Finished program: 1046399811 cycles
------------------
```

Figure 1.2.3: RISC-V I/O program with `ecall` inside ESP32 development board.

This program can be properly debugged in GDBGUI as shown in Figure 1.2.4:

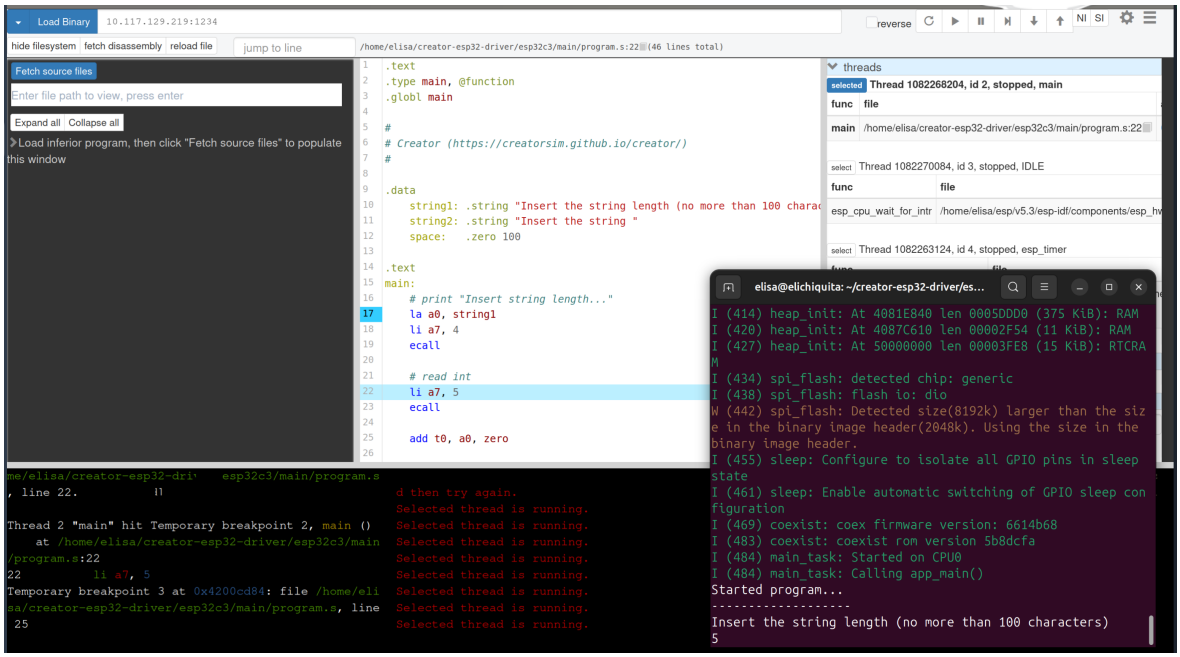


Figure 1.2.4: GDBGUI interface for RISC-V program inside an ESP32.

This new custom panic handler in the ESP32 driver has been tested in the Computer Structure optional laboratory and has performed well.

1.3. New Driver for Managing Real-Hardware on SBC Boards with Linux and SSH

Complementing the ESP32 support, another driver has been developed to manage Ubuntu-based RISC-V SBCs remotely using SSH, as shown in Figure 1.1.2. This driver adapts Espressif-like services and CREATOR's `ecall` handling to the Linux environment, offering, as shown in Figure 1.3.1:

- **Erase-flash and Clean**: Can clean the implementation inside the SBC and the build inside the computer.
- **Flash**: The driver will cross-compile CREATOR's program and send it to the SBC via SSH (using the `scp` command).
- **Monitor**: Executes programs as a normal executable in an Ubuntu environment.
- **Debug**: This option will open the GDBGUI interface, but in a more stable way, so it does not depend on OpenOCD to work.

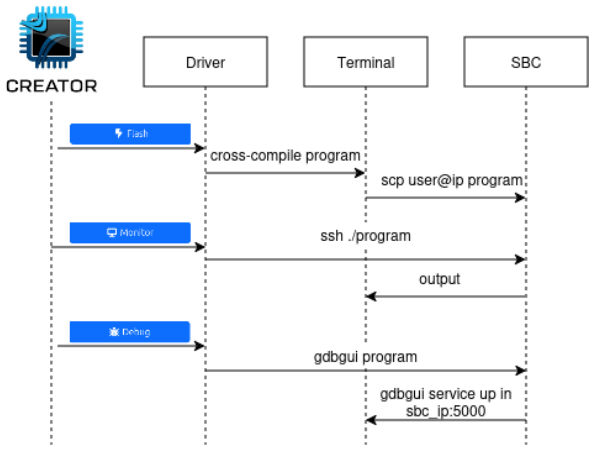


Figure 1.3.1: Flow diagram showing SBC functions in CREATOR.

The programs will be cross-compiled with a modified `ecall` command different from the RISC-V original syscall convention[1]. In this case, reading and writing floats and doubles `ecall` functions are available, as can be seen in Table 1.3.1.

[1]Original RISC-V syscall can be seen in here: `https://jborza.com/post/2021-05-11-riscv-linux-syscalls/`

Call Code	Input Registers	Output Registers	Description
1	a0	-	Prints an int number
2	fa0	-	Prints a float number
3	fa0	-	Prints a double number
4	a0	-	Prints a string
5	-	a0	Writes a number introduced on terminal inside a register
6	-	fa0	Writes a float number introduced on the terminal inside a register
7	-	fa0	Writes a double number introduced on the terminal inside a register
8	a0, a1	a0	Reads and writes a string inside a memory position (a0) with a length given (a1)
9	a0	a0	SBK
10	-	-	Ends program execution
11	a0	-	Prints a char
12	-	a0	Reads a char on terminal and writes it down on a0

Table 1.3.1: CREATOR's SBC system call interface summary.

Initial testing has been performed on OrangePi and Nezda D1-H boards, demonstrating the feasibility and scalability of this approach for real-hardware experimentation in educational settings.

An example of execution is a 64-bit factorial implementation shown in Figure 1.3.2, which produces the results in Figure 1.3.3 and is correctly debugged in Figure 1.3.4.

```
.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 1.3.2: Assembly RISC-V 64 program which implements factorial.

```
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 1.3.3: SBC factorial assembly program flash and execution.

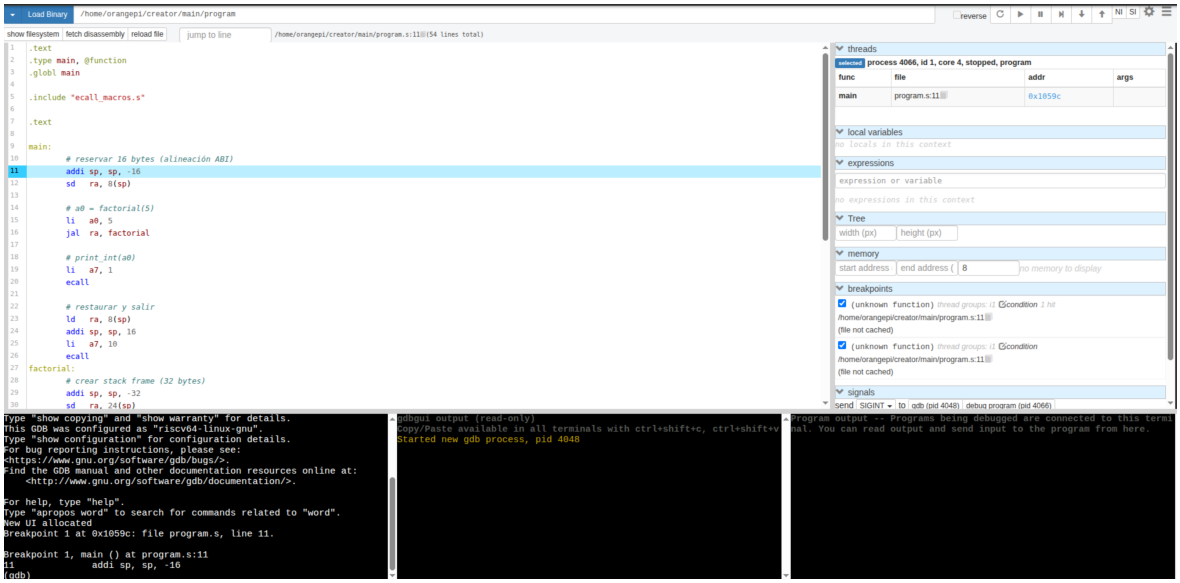


Figure 1.3.4: Factorial assembly program inside SBC debug in GDBGUI.

2. SUPPORT FOR ARDUINO-BASED RISC-V MICROCONTROLLERS

This chapter will address the integration of Arduino functions into CREATOR's engine, including high-level functions for the simulator, a real-time graphics ESP32-C3 simulator for small projects, and the ability to generate flashable code for Espressif RISC-V development boards. The CREATino integration was presented on June 26 at the Jornadas SARTECO Conference 2025 in Sevilla, Spain [1]

The following implementation is not in the official CREATOR repository at the moment and is expected to be included in a future version.

2.1. CREATino Library Inside CREATOR

To start, CREATOR's library function can accept Arduino functions within CREATOR's engine. This new library is added as a shortcut under the "Library" button and has functionality in the simulator, as shown in Figure 2.1.1.

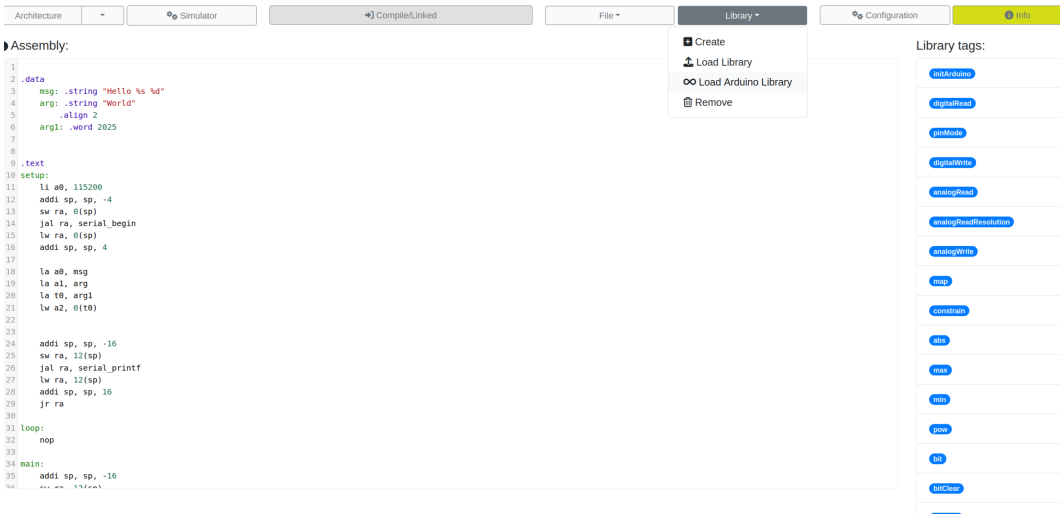


Figure 2.1.1: CREATino library button and display.

This library can be divided into three parts:

- **GPIO functions**: Effects will be shown in the Graphic Environment. To the moment of publishing this deliverable, only a few functions will be added (`digitalWrite()` and `digitalRead()`). An example can be found in Figure 2.2.2 and its output in Figure 2.2.3.

- **Non-GPIO functions**: High level operations will be reflected in the a0 registers. This includes **Math**, **Bits and Bytes**, **Characters**, **Time**, and **Random Numbers** functions from the original documentation [2]. An example can be found in Figure 2.1.2 and its output in Figure 2.1.3.

```
.data
    msg: .string "%d"
    newline: .string "\n"

.text
main:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, initArduino
    lw ra, 0(sp)
    addi sp, sp, 4

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

setup:
  # MIN TEST
  li a0, 40
  li a1, 80

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, min # expected: 40
  lw ra, 0(sp)
  addi sp, sp, 4

  mv a1, a0
  la a0, msg

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4

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

```
  # MAX TEST
  li a0, 40
  li a1, 80

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, max # expected: 80
  lw ra, 0(sp)
  addi sp, sp, 4

  mv a1, a0
  la a0, msg

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4

  # newline
  la a0, newline
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4
  # CONSTRAIN
    li a0, 1000
  li a1, 0
  li a2, 255

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, constrain # Expected
        result: 255
  lw ra, 0(sp)
  addi sp, sp, 4

  mv a1, a0
  la a0, msg

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4
```

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

  li a0, 1000
  li a1, 0 # fromLow
  li a2, 1023 # fromUpper
  li a3, 0 # toLow
  li a4, 255 # toUpper

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, map
  lw ra, 0(sp)
  addi sp, sp, 4

  mv a1, a0
  la a0, msg

  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4

  # newline
  la a0, newline
  addi sp, sp, -4
  sw ra, 0(sp)
  jal ra, serial_printf
  lw ra, 0(sp)
  addi sp, sp, 4

  jr ra
```

Figure 2.1.2: RISC-V with Arduino functions: Non-GPIO functions example.

```
40
80
255
249
```

Figure 2.1.3: RISC-V with Arduino functions: Non-GPIO function example output.

[2]Original Arduino Documentation can be found at `https://docs.arduino.cc/language-reference/#functions`

- **Serial functions**: I/O operations will be executed using the terminal and keyboard elements inside the simulator. An example can be found in Figure 2.1.4 and its output in Figure 2.1.5.

```
.data
    space: .zero 100
    print: .string "%s"

.text
main:
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, initArduino
    lw ra, 0(sp)
    addi sp, sp, 4

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

    j loop
```

```
loop:
    addi sp, sp, -4
    sw ra, 0(sp)

    la a0, space
    li a1, 5
    jal ra, serial_readBytes

    la a0, print
    la a1, space
    jal ra, serial_printf

    lw ra, 0(sp)
    addi sp, sp, 4

    j loop
```

Figure 2.1.4: RISC-V with Arduino functions: Serial function example.

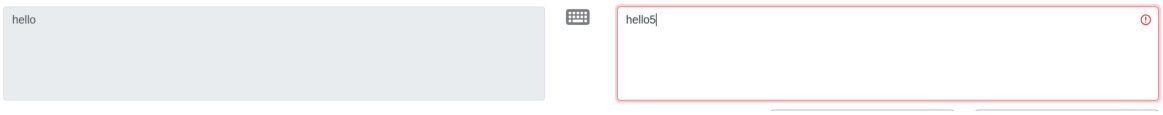


Figure 2.1.5: Serial Function Simulator output.

2.2. CREATino Library Inside ESP32 Sevices

Not only will it be reflected in CREATOR's simulator, but it will also have a real effect on ESP32 boards via the Arduino-ESP32 component, which displays C++ Arduino functions within the Espressif ESP-IDF framework. This is due to a library within the library that can call Arduino high-level functions from assembler's CREATOR programs, as shown in Figure 2.2.1.

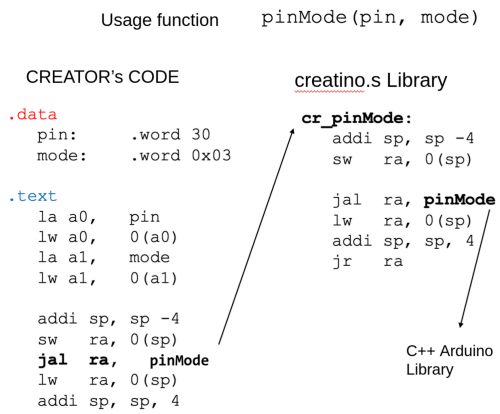


Figure 2.2.1: CREATino functions inside real hardware.

An example of using this new library is the compilation and execution of the code shown in Figure 2.2.2, which uses GPIO and Time functions.

```
.data
    delay:.word 1000
    buttonPin:.word 4
    ledpin: .word 5
    buttonState: .word 0
.text
setup:
    li a0, 115200
    addi sp, sp, -4
    sw ra, 0(sp)
    jal ra, serial_begin
    lw ra, 0(sp)
    addi sp, sp, 4

    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

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

    jr ra
```

```
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, delay
    lw a0, 0(a0)
    addi sp, sp, -16
    sw ra, 12(sp)
    jal ra, delay
    lw ra, 12(sp)
    addi sp, sp, 16

    j loop
```

```
button_pressed:
    la a0, ledpin
    lw a0, 0(a0)
    li a1, 0x1
    jal ra, digitalWrite

    la a0, delay
    lw a0, 0(a0)
    addi sp, sp, -16
    sw ra, 12(sp)
    jal ra, delay
    lw ra, 12(sp)
    addi sp, sp, 16

    jal ra, loop
main:
    addi sp, sp, -16
    sw ra, 12(sp)
    jal ra, initArduino
    jal ra, setup
    lw ra, 12(sp)
    addi sp, sp, 16
    li t4, 0
    beqz t4, loop
    jr ra
    ret
```

Figure 2.2.2: RISC-V with Arduino functions about a button + LED.

Then, the library is verified when we flash it on the board, with the expected results:

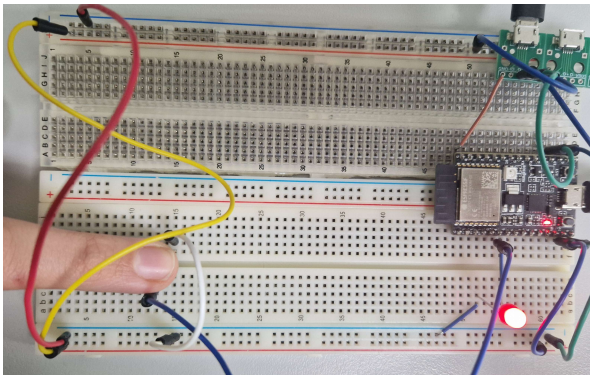


Figure 2.2.3: Button + LED program in esp32c3.

2.3. CREATino Graphic Environment Prototype

Inside CREATOR's stats, a "Creatino Maker" mode will be displayed. It will display a graphic simulation environment similar to Wokwi [3] or other famous Arduino graphic simulators, but using CREATOR's function executor [3].

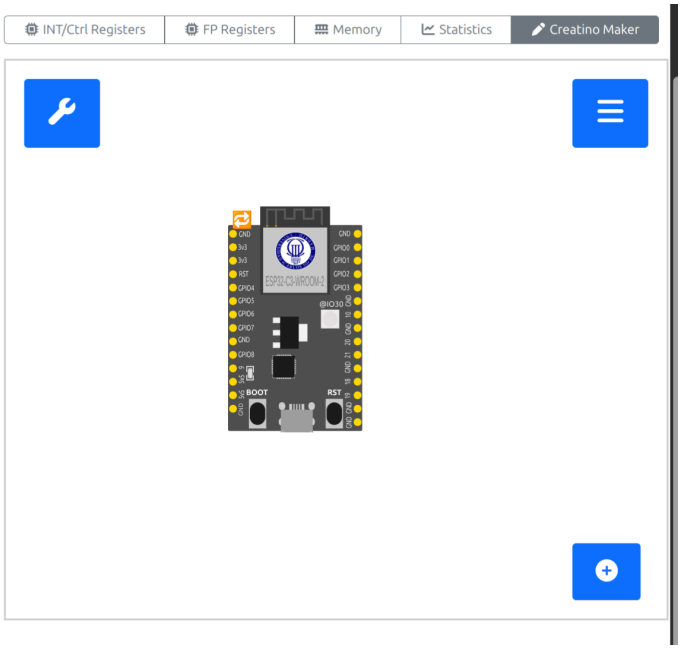


Figure 2.3.1: CREATINO's Maker graphic interface.

This graphic environment connects to CREATOR's instruction executor via CREATOR's API (CAPI), as shown in Figure 2.3.2.

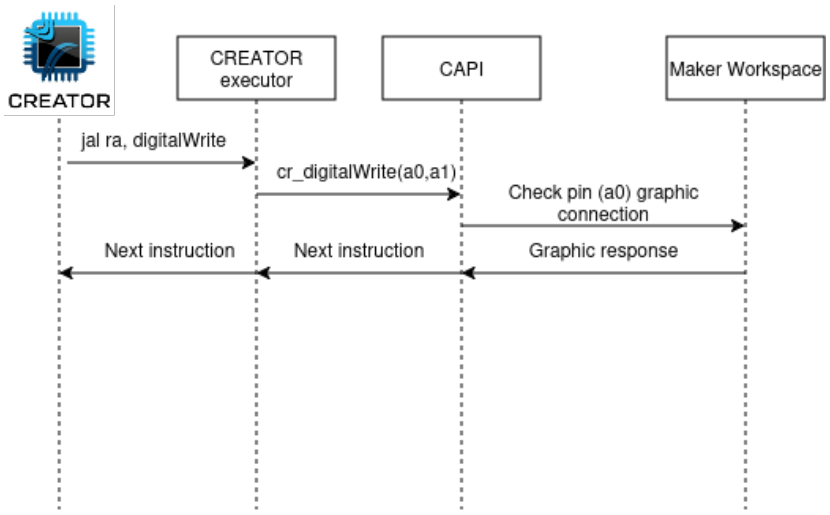


Figure 2.3.2: CREATino Maker workflow with CREATOR.

[3]Component prototype to add in CREATOR can be found here `https://github.com/EUtrilla2002/CreatinoMaker`

As we can see in Figure 2.3.1, inside CREATINOS's Maker graphic environment, it is allowed to:

- **Add new components**: In this first version, LEDs, buttons, and Buzzers can be added.

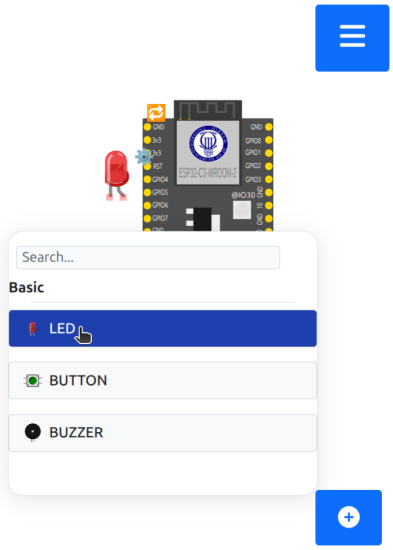
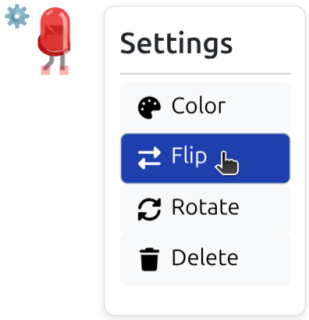
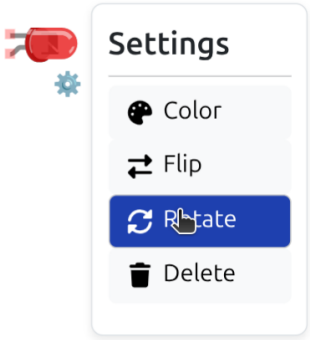


Figure 2.3.3: Components menu.

All components can be rotated, moved, deleted, and connected.



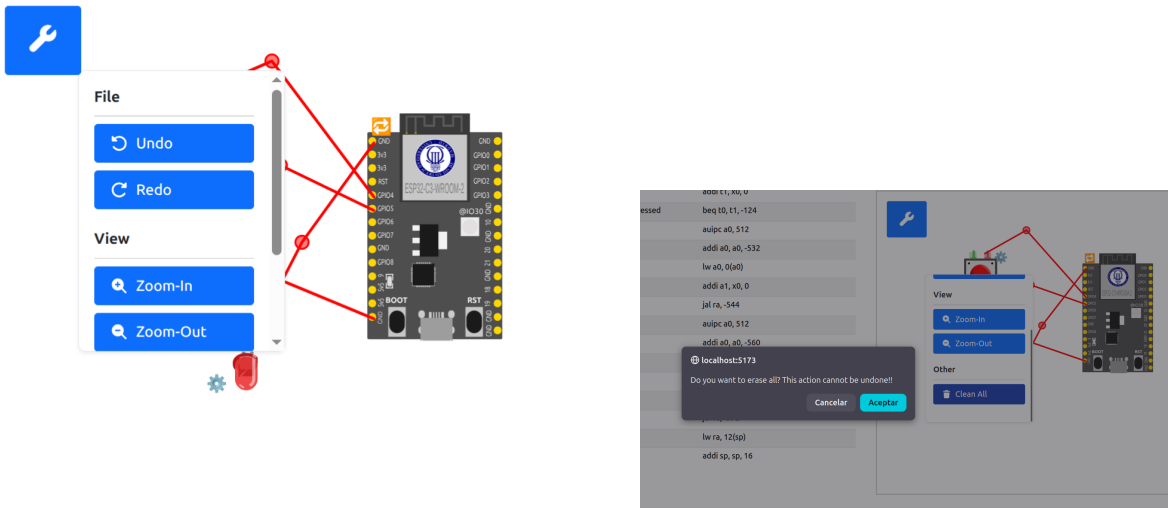
(i) Flip component.



(ii) Rotate component.

Figure 2.3.4: Component manipulation menus.

- **Modify Graphic Environment view**: Using the button on the left, the user can zoom in or out of the scene or clean all elements.



(i) Workspace options menu. (ii) Erase all confirmation pop-up.

Figure 2.3.5: Workspace management options.

Also, connections can change their color and their edge position.

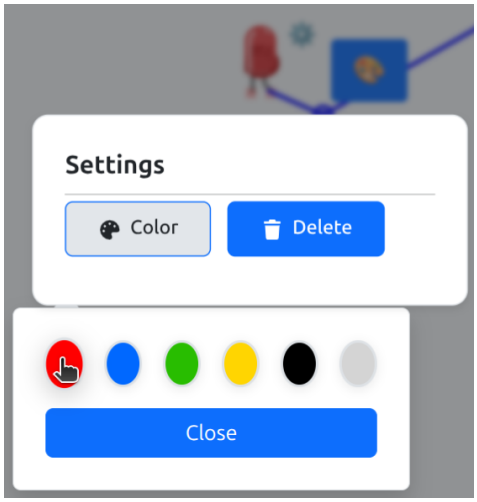
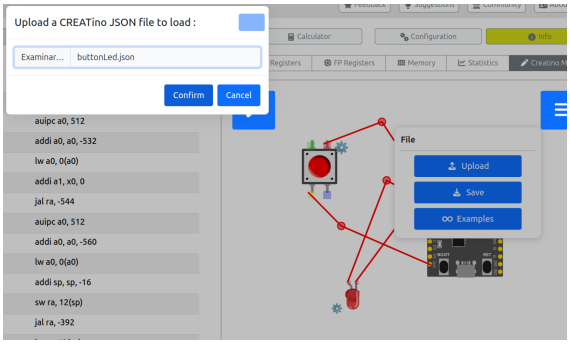
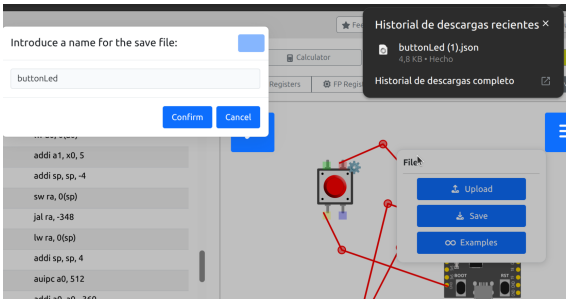


Figure 2.3.6: Connections color change menu.

- **Configure files**: Users can save their CREATino maker scenarios and load them whenever needed. The downloaded files indicate the exact locations of the components displayed and the code executed.



(i) Upload CREATino popup + result.



(ii) Download state.

Figure 2.3.7: Configure flash options in CREATino.

An example of a graphical interface setup is shown in Figure 2.3.8 for a correct button-and-LED program.

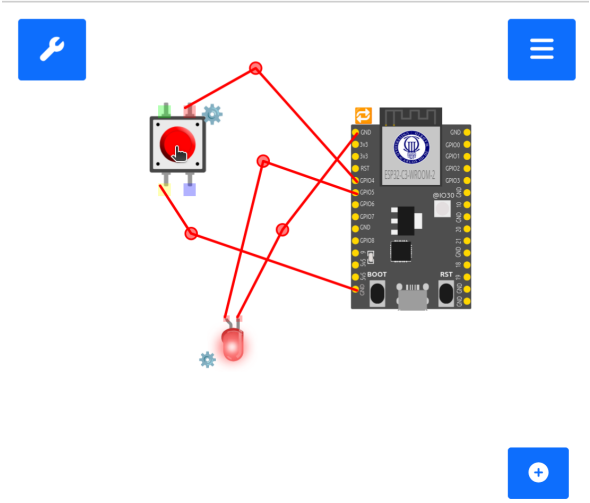


Figure 2.3.8: CREATino Maker button + led setup.

3. REMOTE LABORATORY SERVICE

CREATOR's new remote laboratory service is designed to allow users to execute their assembly programs on remote hardware devices. The approach is important for two reasons: it eliminates the need to purchase hardware and configure the environment on their computer. Furthermore, this laboratory service allows several users to work at the same time on the same hardware devices set because it has a queuing system that schedules the execution requests of the users and sends the results of these executions by E-mail, avoiding that the users have to wait for their program to be executed to close the CREATOR simulator in their web browser.

This new service has been developed using Python 3 and the Flask framework, which facilitates the implementation of web services. In addition, to facilitate the deployment of the remote laboratory service on any device, a Docker-based container [4] has been created with the necessary environment configured and ready to work.

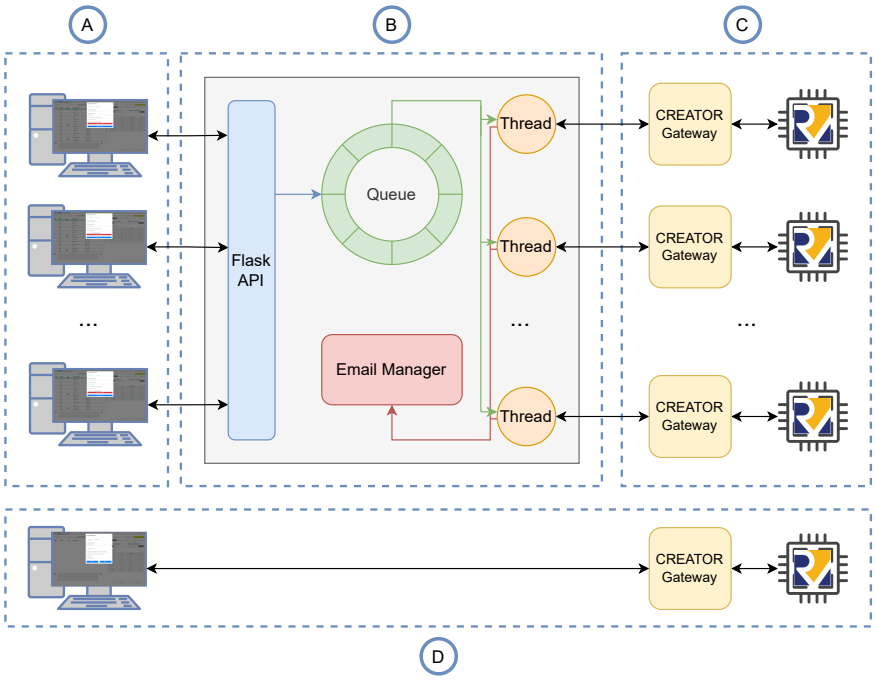


Figure 3.0.1: CREATOR remote laboratory service modular design.

The remote laboratory service has been specially designed to be modular and to allow greater flexibility in its deployment. In Figure 3.0.1 it can be seen that this service is composed of three modules: module A (Section 3.1) corresponds to the CREATOR simulator user interface, module B (Section 3.2) corresponds with the new laboratory service and module C (Section 3.3) corresponds with the hardware device itself and the gateway developed to be able to interact with the microcontroller's drivers. Finally, it should be noted that a local hardware device can also be used to run the assembly programs developed in CREATOR, as can be seen in module D of Figure 3.0.1.

[4] `https://hub.docker.com/repositories/creatorsim`

3.1. User Interface

First, the CREATOR simulator's user interface has been slightly modified to add a new dialog box for connecting to and sending assembly programs to the remote laboratory service. As can be seen in Figure 3.1.1, this dialog box allows users to indicate the necessary parameters to be able to run their assembly programs on a remote device:

- Remote laboratory URL: Specifies the URL where the remote laboratory service is being executed.
- Target board: Indicates the microcontroller model to be used to run the assembly program. This field will only show the device models available in the remote laboratory specified in the previous field.
- E-mail: This field indicates the user's E-mail address to receive the results of the assembly program execution when it is finished.

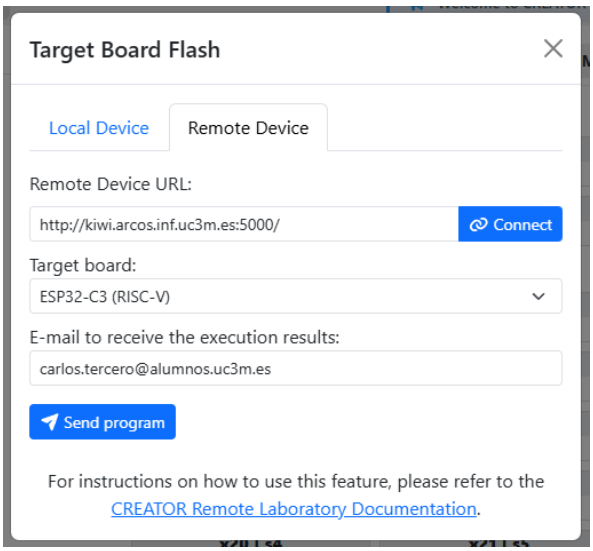


Figure 3.1.1: Form to send the assembly program to the remote laboratory service.

Once all fields are completed, the program can be sent to the remote laboratory service for execution on a device.

When the assembly program has been submitted, this dialog box will show the queue position of the last submitted program and the possibility of canceling the last submitted program, as shown in Figure 3.1.2. In addition, it is possible to submit a new program to the remote laboratory even if the previous program has not yet been completed.

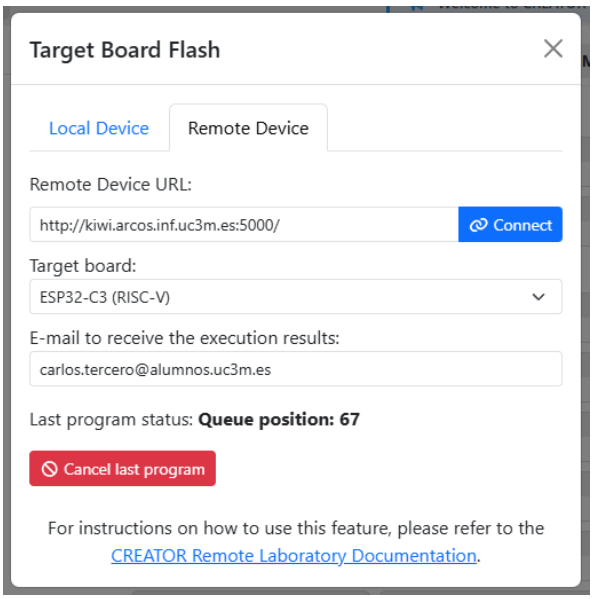


Figure 3.1.2: Form for sending programs to the remote laboratory service during the execution of a program on the device.

3.2. Remote Laboratory Service

The remote laboratory service is a new component that acts as an intermediary, as shown in Figure 3.0.1, between the CREATOR user interface and the gateway, which is responsible for communicating with the hardware device drivers.

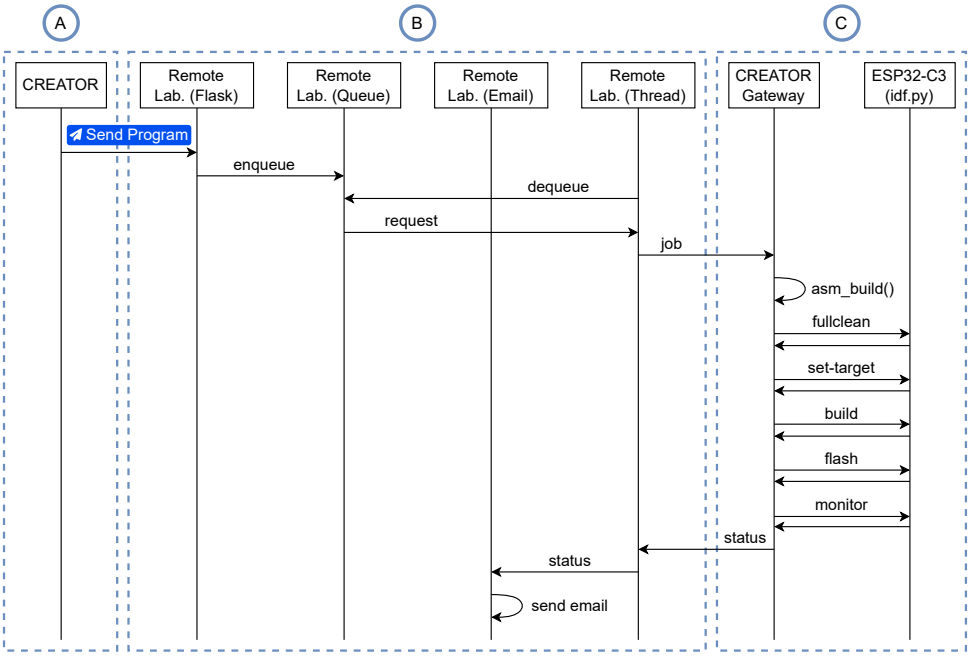


Figure 3.2.1: Exchange of operations among the different modules of the remote laboratory service.

Specifically, this new component is in charge of receiving the requests made by the users when they want to run their assembly program on a remote device, sending the assembly program to the first available hardware device by communicating with the gateway of CREATOR, and finally, sending by E-mail to the user the result of the execution. For this purpose, this laboratory service is composed of different components that communicate with each other and the rest of the modules, as seen in Figure 3.2.1.

The steps that this new service follows to execute the user request could be described as follows:

1. The user uses the CREATOR user interface to request the remote execution by sending all related information (assembly program, hardware device model, and E-mail) to the remote laboratory.
2. Once the web service receives the user's execution request, it is stored in the execution request queue. This request queue stores all execution requests and their corresponding data in the order of arrival, so that the assembly program can be executed when a hardware device becomes available.
3. There is a thread for each hardware device deployed in the remote laboratory, enabling one program per device to be executed in parallel. Specifically, these threads are in charge of extracting from the request queue the first available request and sending the assembly program and the execution information to the gateway module of the real device associated with the thread.
4. Once the gateway executes the user's assembly program on the hardware device, the results of these executions are returned to the thread that sent the request.
5. The thread will send the results of the execution to the E-mail manager, who will be responsible for sending these results in a file by E-mail to the user, as shown in Figure 3.2.2. It should be noted that the Python 3 module smtplib has been used to implement this E-mail manager, which allows sending E-mails using the SMTP protocol (Simple Mail Transfer Protocol).

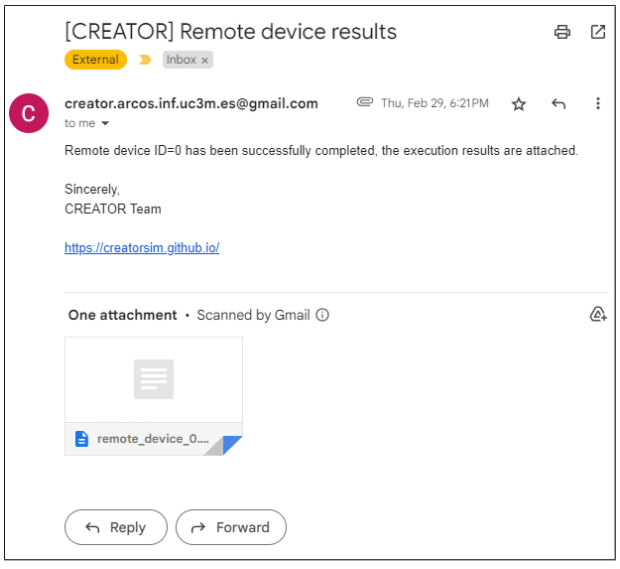


Figure 3.2.2: E-mail sent by the remote laboratory service with the results of the requested on-device execution.

Therefore, the laboratory service design allows users to submit multiple assembly programs for execution on real devices without waiting for the previous program to complete before submitting another. In addition, sending the results via E-mail allows users to avoid waiting for the program to finish executing on the device before closing the open session in the simulator.

3.3. CREATOR Gateway and Hardware Device

Adjacent to the hardware device is the CREATOR gateway. The gateway will receive the assembly program to be executed through a request from the remote laboratory service. Then, it will preprocess the received assembly program to emulate the system calls implemented in CREATOR on the microcontroller.

Finally, once the preprocessing has been done, the gateway will perform the actions *flash* (loading the program in the microcontroller) and *monitor* (the program execution visualization) using the microcontroller's drivers, and the result of the execution of both actions is returned to the remote laboratory service.

4. SIMULATOR ARCHITECTURE AND USER INTERFACE IMPROVEMENTS

4.1. The new simulator architecture

The architecture of CREATOR was divided into three distinct modules: a *core* module, containing all the base functionality of the simulator, and two "consumer" modules, the *Command-Line Interface* (CLI) and the *Web Application*. Both consumer modules are separate applications that act as the interface between the user and the main system's functionality (the core module)

The Figure 4.1.1 shows the new general architecture of CREATOR.

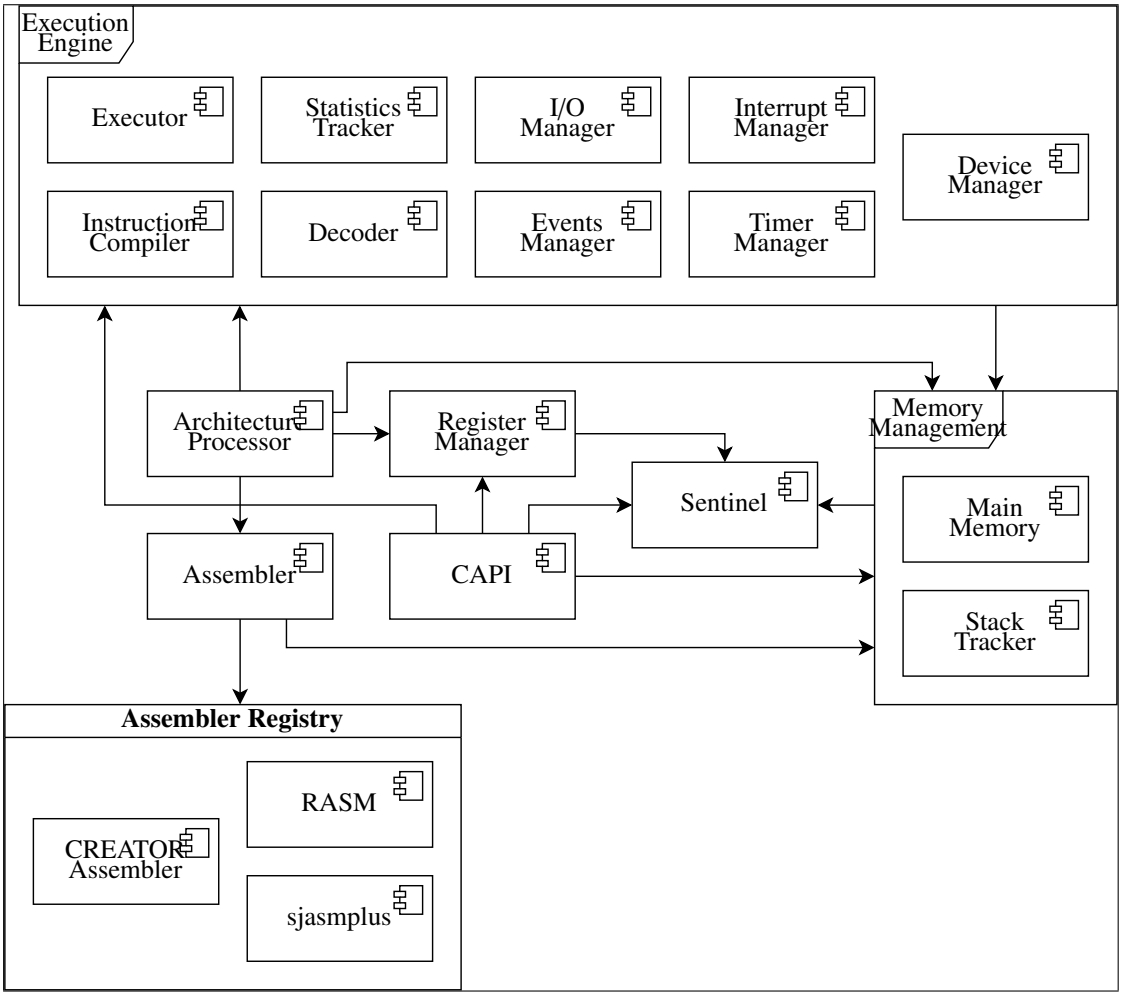


Figure 4.1.1: Core architecture.

Its main components are the following:

- *Architecture Processor*: In charge of parsing the architecture configuration file and initializing up the rest

of components according to it.

- *Register Manager*: Manages the registers and register operations.
- *Memory Management*: Manages memory access and operations.
- *Execution Engine*: Executes the loaded instructions and the rest of operations in the execution cycle, including interrupts, timers and devices.
- *Assembler*: Assembles and links the source assembly code into a set of executable instructions.
- *Sentinel*: Monitors the execution of the program, providing feedback on whether parameter passing conventions are followed.
- *CAPI*: An API that defines functions to interact with the rest of the components of the system. Includes the plugin system.

4.2. Key updates on the web-based user interface

The *Web Application* component implements de new Web-based user interface. This new User Interface has three main working areas:

- *Editor*. This area let users to edit the assembly code, and compile this assembly code.
- *Simulator*. This area allows to execute step-by-step, run until breakpoint or run until the end of the assembly code.
- *Architecture*. In this area the ISA can be displayed or edited.

In the *architecture* working area, the ISA is now defined in YAML, and Web Applications have a new Monaco-based editor that lets us edit the full ISA. Former syntax was based on JavaScript, and the ISA was edited instruction-by-instruction, with a separate form for each instruction. Also, ISA can be displayed, were instructions representation includes the instruction format, operands, etc.

The *simulator* working area now shows all register files (integer, floating-point, and control registers) in a compact layout, and the terminal (keyboard and screen) is available in an associated detail panel. The executed code now shows the memory address (Address), the assembly instruction (User Instruction), and the associated machine instruction (Loaded Instructions), which may differ for pseudo-instructions. There are two bars to highlight the current instruction and the next instruction to be executed.

The execution cycle of the simulator had to be modified, as shown in 4.1, to incorporate the interrupt, and device handling subroutines.

Listing 4.1: New CREATOR instruction execution cycle

```
1 Decode _instruction_
2 Increment 'PC'
3 Execute _instruction_
4 Handle interrupts()
5 Handle devices()
6 Fetch _instruction_
```

The main change from the first version of CREATOR's execution cycle is that interrupt detection and instruction fetch are performed at the end of the cycle, rather than at the beginning. With the current approach, the simulator can infer the state after the current instruction's execution and display it in the User Interface in the current cycle. This is relevant for step-by-step execution, as the User Interface needs to show the next instruction before the user decides to execute it. In the case of the first instruction, some extra logic is performed by the assembler in order to fetch the first instruction.

4.3. The interrupts management

A new part of the simulator's architecture is the management of interrupts and devices.

Simulating interrupts in a "generic" way is somewhat difficult. Each ISA can define its own way of handling interrupts, But all approaches tend to converge to the same core idea: something unexpected happens: the processor either ignores it or stops execution, handles it, and then resumes execution.

The approach in CREATOR is to divide the interrupt model into a set of common actions whose behavior is then defined by each specific ISA. The selected interrupt handler will implement the behavior of each action. Some ISAs (such as RISC-V) also support globally enabling and disabling interrupts, that is, all interrupt types.

The set of actions in the model is the following:

- Generating a specific type of interrupt.
- Evaluating what types of interrupts are currently pending, and returns the highest priority one.
- Clearing pending interrupts, both globally and by type.
- Handling a pending interrupt type.
- Enabling and disabling interrupts, either one specific type of interrupt or globally.
- Evaluating if interrupts are enabled, both globally and by type.

Each ISA defines its own set of interrupt types, but in a generic model, it is possible to differentiate the following:

- *Maskable*: Generic interrupts that can be disabled.
- *NonMaskable*: Generic interrupts that can't be disabled.
- *Timer*: Interrupts generated by a timer.
- *External*: Interrupts generated by an external device.
- *Software*: Interrupts generated by software.
- *EnvironmentCall*: A subset of *software* interrupts, in which the switch into a higher privilege mode

of execution (e.g., an OS) is required. Note that not all interrupt types need to be defined in all ISAs; the approach is to allow the user to select a subset of these types.

The handling subroutine, specified in Algorithm 1, evaluates the pending interrupts and executes its handler if interrupts are globally enabled and that type of interrupt is enabled. This algorithm allows only one interrupt to be handled per call, and it's up to the implementation of the evaluating action to assign the correct priority to each interrupt type and return the highest-priority interrupt. Instructions, timers, and devices can generate interrupts by using the specified action.

Algorithm 1 Interrupts handle subroutine

```
procedure HANDLEINTERRUPTS
    if not interruptsGlobalEnabled() then
        return
    end if
    if not interruptPending() then
        return
    end if
    pending_interrupt ← getInterrupt()
    if not interruptEnabled(pending_interrupt) then
        return
    end if
    interruptHandle(pending_interrupt)
end procedure
```

For the *EnvironmentCall* CREATOR has two functional modes: one where system calls are implemented with interrupts, and another different mode where system calls are implemented in the 'ecall/syscall' ISA instruction definition. A configuration option lets CREATOR know which one will be used. In the CREATOR API (CAPI), there are several I/O functions that let ISA designers implement 'ecall/syscall' definition directly without interrupts. This gives more flexibility.

As mentioned before, there are two handlers implemented (see Figure 4.3.1): the default CREATOR interrupt handler, and an architecture handler. The former handler treats all interrupt types except `EnvironmentCall` as unexpected errors, and generates an exception that is managed by the User Interface. For `EnvironmentCall` interrupts, it executes the architecture-defined system call handler. The architecture handler obtains the definition of its actions from the architecture definition, and just executes them.

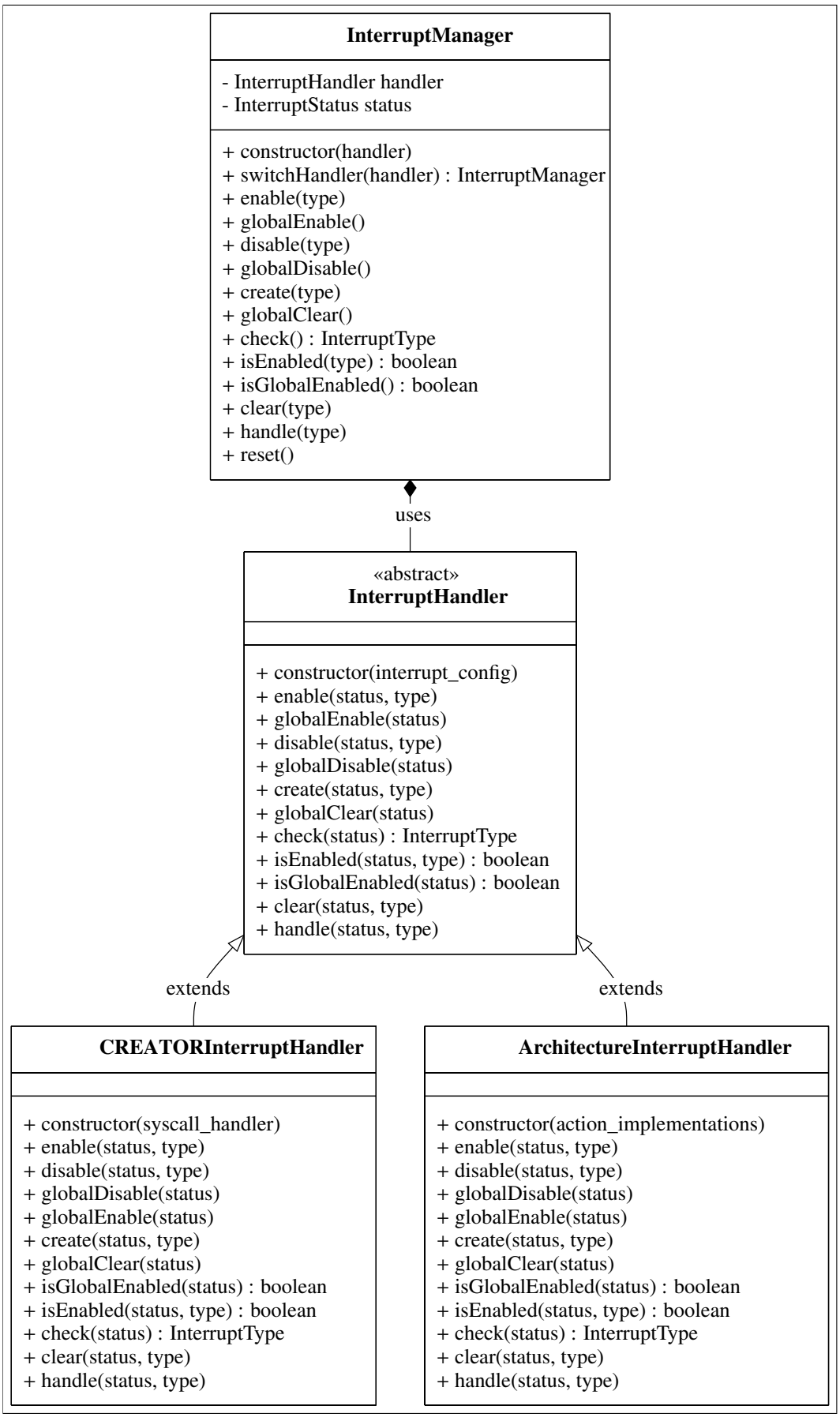


Figure 4.3.1: Interrupts handler classes.

4.3.1. The CREATOR API for interrupts

The CREATOR API (CAPI) is an interface that interacts with the different simulator core components. It is used in the architecture definition, especially in the instructions and interrupt/timer actions.

In order to allow interactions with interrupts and devices, the following methods have been added:

- *CAPI.INTERRUPTS.create(type: InterruptType)*: Creates a new interrupt of the specified type.
- *CAPI.INTERRUPTS.enable(type: InterruptType)*: Enables the specified type of interrupts.
- *CAPI.INTERRUPTS.disable(type: InterruptType)*: Disables the specified type of interrupts.
- *CAPI.INTERRUPTS.isEnabled(type: InterruptType)*: Checks if interrupts of the specified type are enabled.
- *CAPI.INTERRUPTS.clear(type: InterruptType)*: Clears a pending interrupt of the specified type.

- *CAPI.INTERRUPTS.globalEnable()*: Globally enables interrupts.
- *CAPI.INTERRUPTS.globalDisable()*: Globally disables interrupts.
- *CAPI.INTERRUPTS.isGlobalEnabled()*: Checks if interrupts are globally enabled.
- *CAPI.INTERRUPTS.globalClear()*: Clears all pending interrupts.
- *CAPI.INTERRUPTS.setUserMode()*: Sets the user privilege mode
- *CAPI.INTERRUPTS.setKernelMode()*: Sets the kernel privilege mode.

- *CAPI.INTERRUPTS.setCREATORHandler()*: Sets the interrupt handler to the one defined in the architecture as CREATOR handler.
- *CAPI.INTERRUPTS.setCustomHandler()*: Sets the interrupt handler to the one defined in the architecture as custom handler.

- *CAPI.INTERRUPTS.setHighlight()*: Start hightlighting the interrupted instruction.
- *CAPI.INTERRUPTS.clearHighlight()*: Stop hightlighting the interrupted instruction.

4.3.2. Architecture section for interrupts

The architecture definition file has the following properties related to interrupts management:

- interrupts.create: [string]: JavaScript arrow (lambda) function to be executed in order to set an interrupt given an interrupt type.

Listing 4.2: Example of create for RISCV32

```
create: |
  switch (type) {
    case InterruptType.Software:
        registers.mcause = 2n ** 3n;
        registers.mip |= 2n ** 3n; // MSIP
        break;
    case InterruptType.External:
        registers.mcause = 2n ** 31n + 11n;
        registers.mip |= 2n ** 11n; // MEIP
        break;
    case InterruptType.EnvironmentCall:
        registers.mcause = 2n ** 8n;
        registers.mip |= 2n ** 3n; // MSIP
        break;
    case InterruptType.Timer:
        registers.mcause = 2n ** 31n + 7n;
        registers.mip |= 2n ** 7n; // MTIP
        break;
  }
```

- interrupts.check [string]: JavaScript code to be executed in order to check whether an interrupt happened. It must return a InterruptType (if an interrupt happened) or null (if it didn't).

Listing 4.3: Example of check for RISCV32

```
check: |
  if (registers.mip & (2n ** 11n)) return InterruptType.External;
  if (registers.mip & (2n ** 3n)) return InterruptType.Software;
  if (registers.mip & (2n ** 7n)) return InterruptType.Timer;
  return null;
```

- interrupts.is_enabled [string, "null"]: JavaScript code to be executed in order to check whether interrupts are enabled.

Listing 4.4: Example of is_enable for RISCV32

```
is_enabled: |
  switch (type) {
    case InterruptType.Software:
    case InterruptType.EnvironmentCall:
        return !! (registers.mie & 2n ** 3n); // MSIE
    case InterruptType.External:
        return !! (registers.mie & 2n ** 11n); // MEIE
    case InterruptType.Timer:
        return !! (registers.mie & 2n ** 7n); // MTIE
  }
  return false;
```

- interrupts.enable [string, "null"]: JavaScript code to be executed in order to enable the specified interrupt 'type'. Defaults to 'global_enable'

Listing 4.5: Example of enable for RISCV32

```
enable: |
```

```
  switch (type) {
    case InterruptType.Software:
    case InterruptType.EnvironmentCall:
        registers.mie |= 2n ** 3n; // MSIE
        break;
    case InterruptType.External:
        registers.mie |= 2n ** 11n; // MEIE
        break;
    case InterruptType.Timer:
        registers.mie |= 2n ** 7n; // MTIE
        break;
  }
```

- interrupts.disable [string, "null"]: JavaScript code to be executed in order to disable the specified interrupt ‘type‘. Defaults to ’global_disable’

Listing 4.6: Example of disable for RISCV32

```
disable: |
  switch (type) {
    case InterruptType.Software:
    case InterruptType.EnvironmentCall:
        registers.mie &= ~(2n ** 3n); // MSIE
        break;
    case InterruptType.External:
        registers.mie &= ~(2n ** 11n); // MEIE
        break;
    case InterruptType.Timer:
        registers.mie &= ~(2n ** 7n); // MTIE
        break;
  }
```

- interrupts.clear [string, "null"]: JavaScript code to be executed in order to clear an interrupt of the specified ‘type‘. Defaults to ’global_clear’

Listing 4.7: Example of clear for RISCV32

```
clear: |
  switch (type) {
    case InterruptType.Software:
    case InterruptType.EnvironmentCall:
        registers.mip &= ~(2n ** 3n); // MSIP
        break;
    case InterruptType.External:
        registers.mip &= ~(2n ** 11n); // MEIP
        break;
    case InterruptType.Timer:
        registers.mip &= ~(2n ** 7n); // MTIP
        break;
  }
  registers.mcause = 0n;
```

- interrupts.handlers [string]: Interrupt handler configuration.
  - interrupts.handlers.creator_syscall [string, "null"]: JavaScript handler for syscall interrupt.

Listing 4.8: Example of syscall handler for RISCV32

```
creator_syscall: |
  let [fa0Value, fa0Type] = CAPI.ARCH.toJSNumberD(registers.fa0);
  switch (registers.a7) {
    case 1n:
        CAPI.SYSCALL.print(registers.a0, 'int32');
        break;
    ................
    case 10n:
        CAPI.SYSCALL.exit();
        break;
    case 11n:
        CAPI.SYSCALL.print(registers.a0, 'char');
        break;
    case 12n:
        CAPI.SYSCALL.read('a0', 'char');
        break;
  }
  CAPI.INTERRUPTS.clearHighlight();
```

– interrupts.handlers.custom [string, "null"]: JavaScript handler for the custom interrupt handler.

Listing 4.9: Example of custom handler for RISCV32

```
custom: |
  CAPI.INTERRUPTS.globalDisable();
  CAPI.INTERRUPTS.setKernelMode();
  registers.mepc = CAPI.REG.read("pc"); // get "real" PC (next instruction)

  if (registers.mtvec & 1n) { // vectored mode
      registers.pc = (registers.mtvec >> 2n) + 4*(registers.mcause & (2**32 - 1)) ;
  } else { // direct mode
      registers.pc = registers.mtvec >> 2n ;
  }
```

4.4. The device management

It is possible to model a "generic" device as a set of registers and a handler function. Most devices have at least a control register for the CPU to signal an action to perform, and optionally a status register for the device to broadcast its status.

Instead of a single data register, to enable more flexibility in device definition, a data range is provided: a set of contiguous memory that can be used for communication and to represent several different registers. The handler is responsible for interacting with the different registers and the simulator, including *Direct Memory Access*. A device can also be enabled or disabled in the architecture definition.

Figure 4.4.1 shows the representation for this "generic" device model. Devices all derive from the same abstract class, which implements common methods such as ‘isDeviceAddr()‘ (which validates whether the given address belongs to the device), a reset method (which clears the associated memory), and a clear method (which resets the control register). Each device then implements its own handler and has its own associated memory. This approach makes the system extensible, making it easier to add new devices.

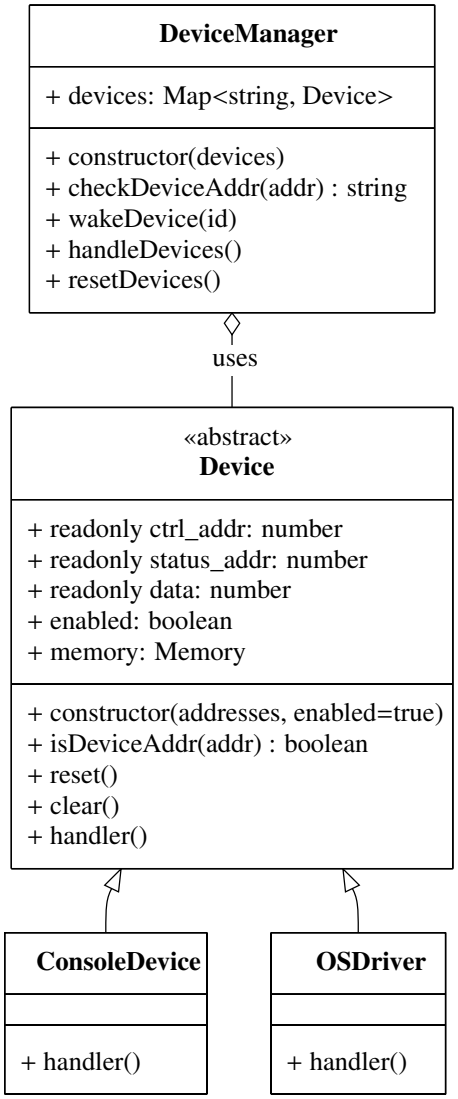


Figure 4.4.1: Device model.

To give the users flexibility over ISA definition, the addresses of the different devices' registers are configured in the architecture definition. This also allows the user to map registers to addresses both inside and outside main memory. For address decoding, any memory access first checks whether the address points to a device register and performs the operation in the device's memory if it does.

A device is defined by the following:

- An identifier (id), which uniquely identifies the device.
- A control register address (ctrl_addr), which is typically used by the processor to signal an action to perform on the device.
- A status register address (status_addr), which is typically used by the device to signal its status to the processor.

- A data range (data), which defines a section of memory (start and end) that is shared between the processor and the device, typically for the exchange of data.
- A handler function (handler()), which is called once per cycle, and defines the behavior of the device.
- An enabled flag (enabled), which controls whether the device is enabled or not. If it's not, the callback is not called.

To create a new device, create a new class extending Device and implementing its handler method. Then, instantiate that object and add it to devices with its corresponding ID.

4.4.1. Device handling

After executing each instruction, the executor calls *handleDevices()*, which executes the handlers for all enabled devices.

Typically, a device reads its control register and checks its value. If it's 0, it exits. If it's not, it works with the other data and clears the control register when it ends by calling *reset()*.

The device handling subroutine, as shown in 2, executes the handler of all enabled devices once per cycle.

Algorithm 2 Device handling subroutine

```
procedure HandleDevices
    for each device in devices do
        if not device.enabled() then
            continue
        end if
        device.handler()
    end for
end procedure
```

4.4.2. Implemented devices

ConsoleDevice

A device for interacting with CREATOR's console.

Depending on the value stored in the control register (ctrl_addr), it executes one of the following:

- 1 - print int: Reads a word from the data.start address and writes it as an integer value in the console.
- 2 - print float: Reads a word from the data.start address and writes it as a float value in the console.
- 3 - print double: Reads a word from the data.start address and writes it as a float value in the console.
- 4 - print string: Reads the main memory address of a string from the data.start address and writes it in the console.
- 5 - read int: Reads an integer from the console and stores it as a word in data.start.

- 6 - read float: Reads a float from the console and stores it as a word in data.start.
- 7 - read double: Reads a double from the console and stores it as a word in data.start.
- 8 - read string: Reads a string from the console of the length specified in data.start + 4 and stores it in the main memory address specified in data.start.
- 11 - print char: Reads a byte from the data.start address and writes it as a char in the console.
- 12 - read char: Reads a char from the console and stores it as a byte in data.start.

Memory addresses:

- ctrl_addr: 0xf0000000
- status_addr: 0xf0000004
- data:
  - start: 0xf0000008
  - end: 0xf000000f

OSDriver

A device to simulate system calls to a "Operating System" services for memory and execution.

Depending on the value stored in the control register (ctrl_addr), it executes one of the following:

- 9 - sbrk: Allocates a segment of main memory of the size specified in data.start and stores its address in data.start.
- 10 - exit: Terminates the current program's execution.

Memory addresses:

- ctrl_addr: 0xf0000010
- status_addr: 0xf0000014
- data:
  - start: 0xf0000018
  - end: 0xf000001f

5. SAIL-BASED MODEL DESIGN

This chapter will address the design and implementation of the simulator based on Sail. Also, this chapter is part of the article presented at the SARTECO 2025 Conference [4].

5.1. Design of the Simulator

Both variants of the architecture based on Sail have been designed as follows:

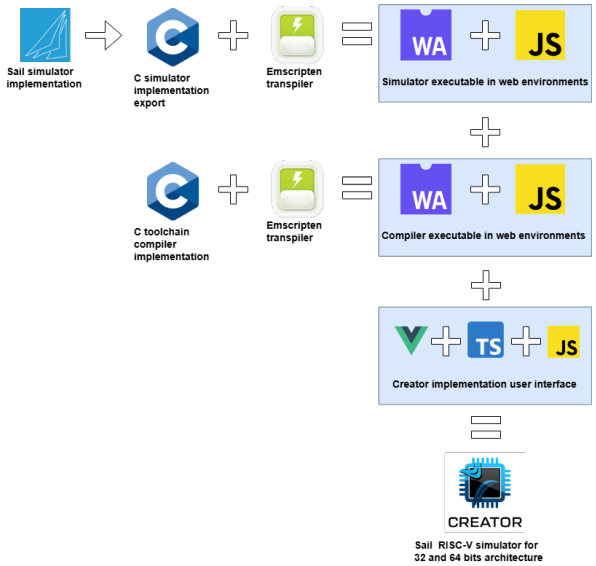


Figure 5.1.1: Design of the simulator (Compiler + Executor).

As shown in Figure 5.1.1, the simulator consists of two main parts: the compiler and the executor. The first part is responsible for compiling the assembly code entered and generating the corresponding binary to be executed in the simulator architecture. The simulator was implemented in the Sail specification language and exported to C language. The main reason is that Sail is not a compiled language, so it must be compiled to C to generate an executable program. After the export, Emscripten was used to transpile the code into a binary format for execution in web environments. In the case of the compiler, the same methodology is followed: the C implementation is transpiled, and the corresponding binary is generated for execution in web environments. Finally, both binaries are integrated into the CREATOR interface thanks to WebAssembly's dependency on JavaScript, allowing the execution environment to be created and easily integrated into web environments.

The compilation process consists of three phases:

1. **Assembly**, verify that the code has been implemented correctly and there are no syntax or implementation errors in order to generate an object file (*.o*).
2. **Link**, proceed to the link phase with a linker script (*.ld*) that contains a map of the regions and memory addresses where the binary is expected to run in the RISC-V simulator.

3. **Disassembly**, once the executable binary has been generated, proceed with a disassembly phase to translate the memory addresses assigned to the encoded data and instructions. This last step can facilitate the simulator's debugging and provide a better understanding of its behavior.

This compiler replaces the actual compiler available in CREATOR. It is a compilation tool that generates executable binaries on RISC-V architectures. The main reason is that significantly increasing the size of the instruction set to be executed can negatively affect JavaScript performance. The tool was developed by RISC-V, and in the project, it has been used to generate executable binaries for RISC-V architectures. This tool is a toolchain[5] that generates programs from assembly or C code and has been adapted to the simulator's needs. Specifically, the RISC-V code assembly, linking, and disassembly tool has been used. It has also been integrated into the simulator using WebAssembly to run the native application in CREATOR.

The second part is responsible for running binaries compiled with the compiler toolchain. The executor is primarily the simulator based on Sail introduced in the previous sections. This executor allows the user to run programs coded in assembly language and debug them during execution by displaying the current status of execution in the different modules of the architecture.

The simulator was implemented in Sail and transpiled with Emscripten to generate a WebAssembly component for integration and execution in web environments. The compiler toolchain was written in C and transpiled with Emscripten to allow compiling assembly-coded programs in web environments.

5.2. Integration of Components in the User Interface

The new simulator tool (compiler and executor) was integrated into CREATOR via JavaScript and WebAssembly, and adapted to the current implementation's needs. This integration has expanded the simulator's functionalities. These functionalities are as follows:

- Full support for the RISC-V instruction set: The user can generate a program using any instruction defined in the RISC-V standard and execute it in the simulator.
- Architecture module update: Vector file register, control and status file register, and cache memory module have been implemented in the web environment to display the program's behavior during simulation.
- Execution of a custom kernel coded in assembly language: The simulator includes a default kernel on RISC-V architecture, but the simulator allows the user to code their own kernel to their personal objectives and explore more in the RISC-V architecture.

This has been implemented using JavaScript and WebAssembly for the simulation tool, and TypeScript, Bootstrap + Vue for the implementation of the different modules now available in the simulator. Now, the new functionalities will be shown in the web environment.

[5]**Available at** `https://github.com/riscv-collab/riscv-gnu-toolchain`

Actually, the RISC-V architecture in CREATOR supports only the 32-bit variant and has a reduced instruction set (*IMFD*). This simulator is very useful for small projects and in teaching environments, but considering the trend that many processors are designed to support a 64-bit architecture, and with the new integrations presented in previous sections, the aim is to take a step forward to offer a complete solution to professional developers and researchers in this type of architecture.

The new integrations in CREATOR have led to changes in the user interface and its use, as well as in the behavior of the features already implemented in the web simulator.

Both the 32-bit and the new 64-bit simulator for the RISC-V architecture are now available, as shown in Figure 5.2.1.

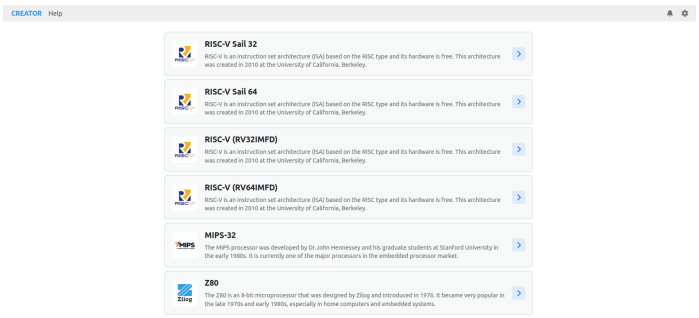


Figure 5.2.1: CREATOR home page.

In addition, each architecture includes the full instruction set defined by the RISC-V specification. This allows the user to generate binaries that execute any instruction in the specification.

Within the configuration menu, you can select whether you want to work with the kernel integrated into the simulator itself or with a kernel implemented by yourself, as shown in Figure 5.2.2.

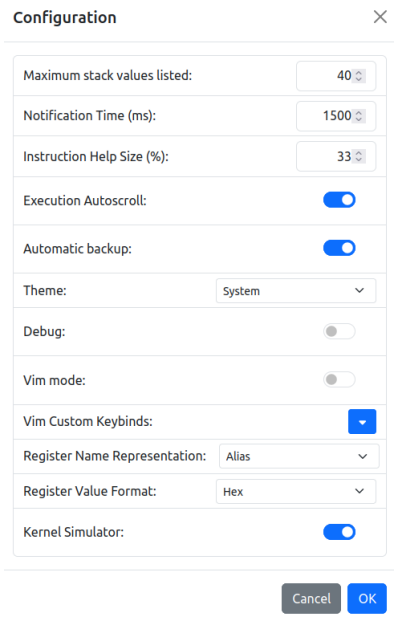


Figure 5.2.2: CREATOR configuration menu.

On the simulation view, the interface has been updated to display all the elements of the architecture that can be interacted with in the simulator.

A vector file of registers has been added to the existing file module registers, as can be seen in Figure 5.2.3. These show the changes in the architecture when vector extension instructions are executed.

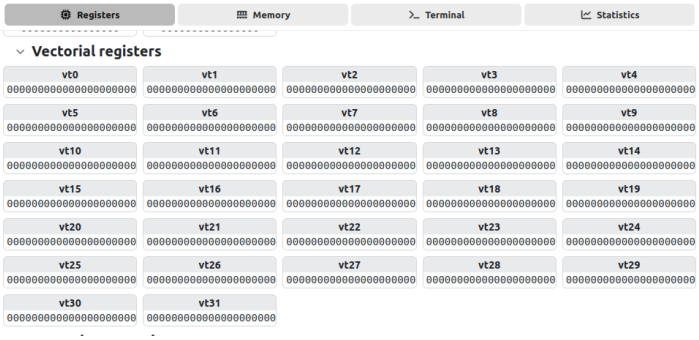


Figure 5.2.3: Vector Register file.

Additionally, the Control State Register file has been added, as can be seen in Figure 5.2.4. The aim is to illustrate and understand the architecture's configuration and justify its behavior during instruction execution.

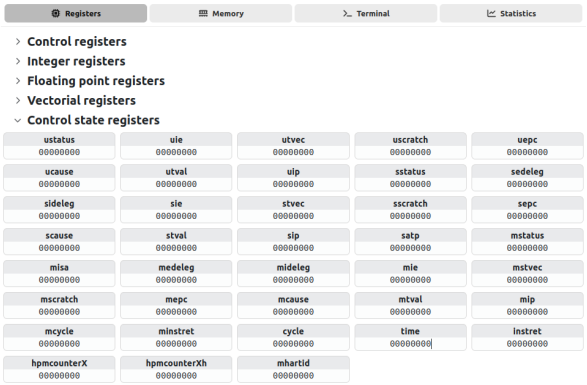


Figure 5.2.4: CSR Register file.

This register file is composed of different sections, differentiated by execution permissions (user, supervisor, and machine), to understand instruction execution behavior and, in the event of an exception, the reason it occurred. This can be modified through the execution of instructions and with appropriate execution permissions to avoid unwanted behavior during execution and ensure correct architecture management. In addition, it can be very useful for debugging to justify behavior during project development.

Finally, the integrated web editor has been updated and can now generate libraries from compiled assembly code, import them, or import assembly files from previous projects. Also displays the option to use the editor in Vim mode, as shown in Figure 5.2.5.

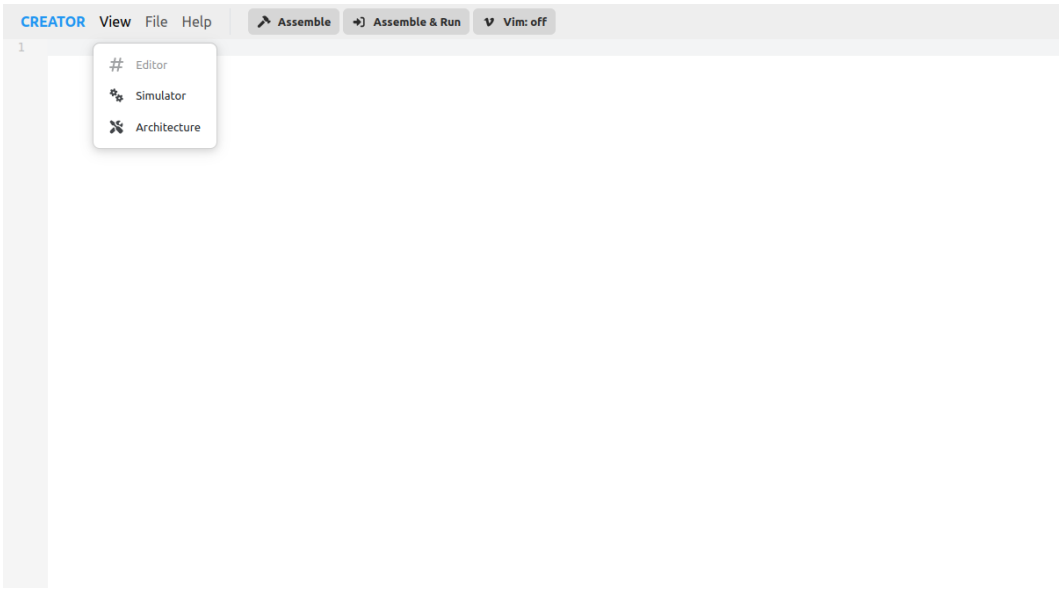


Figure 5.2.5: CREATOR editor view.

Once the program is compiled, the developer can run it directly in the simulator or download it as a library of subroutines if they want to use it in another project.

BIBLIOGRAPHY

[1] E. Utrilla Arroyo, D. Camarmas Alonso, F. Garcia Carballeira, and A. Calderon Mateos, "Extensión del simulador creator para integrar funcionalidades de arduino: Caso de estudio con el microcontrolador esp32," in *Avances en Arquitectura y Tecnología de Computadores. Actas de las Jornadas SARTECO*, (Sevilla, Spain), Zenodo, Jun. 2025, pp. 637–643. DOI: 10.5281/zenodo.15773284. [Online]. Available: https://doi.org/10.5281/zenodo.15773284.

[2] Espressif, "Esp32-c3 technical reference manual version 1.3," Espressif, Tech. Rep., 2023.

[3] WOKWI, *WOKWI*. [Online]. Available: https://wokwi.com/.

[4] J. C. Cano Resa, F. Garcia Carballeira, D. Camarmas Alonso, and A. Calderon Mateos, "Simulador web para risc-v basado en la especificación sail," in *Avances en Arquitectura y Tecnología de Computadores. Actas de las Jornadas SARTECO*, (Sevilla, Spain), Zenodo, Jun. 2025, pp. 367–376. DOI: 10.5281/zenodo.15773218. [Online]. Available: https://doi.org/10.5281/zenodo.15773218.