1. ARM GCC tool chain

<https://developer.arm.com/-/media/Files/downloads/gnu-rm/10.3-2021.10/gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2?rev=78196d3461ba4c9089a67b5f33edf82a&hash=5631ACEF1F8F237389F14B41566964EC>

sudo apt install gcc-arm-none-eabi

sudo apt install gdb-multiarch

arm-none-eabi-gcc --version

gdb-multiarch --version

Install build dependencies:

sudo apt install -y git build-essential libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev

Clone the QEMU STM32 fork:

git clone https://github.com/beckus/qemu\_stm32.git

cd qemu\_stm32

Configure and build QEMU:

[Fix: sudo apt install install qemu-system-arm

**3. Debug with GDB**

1. **Start GDB**:  
   Open another terminal and start gdb-multiarch:

gdb-multiarch stm.elf

1. **Connect to QEMU**:  
   Inside the GDB prompt:

gdb

target remote localhost:1234

1. **Load and Run**:
   * Load the ELF symbols:

gdb

load

* + Start execution:

gdb

continue

1. **Debugging Commands**:
   * Set breakpoints:

gdb

break main

* + Step through the code:

gdb

step

next

* + Inspect variables:

gdb

print RCC\_AHB1ENR

print GPIOA\_ODR

Let’s go step-by-step from scratch. Here’s how to set up the stm32\_flash.ld linker script, compile the firmware (stm.elf), and run it on QEMU STM32.

**1. Folder Setup**

Your folder structure should look like this:

bash

Copy code

tools/

qemu\_engine/

qemu\_stm32/

build\_firmware.sh # Script for building firmware

run\_qemu.sh # Script for running QEMU

stm32\_flash.ld # Linker script for STM32

stm.elf # Compiled firmware (output)

apps/

watch\_record/

watch\_record.cpp # Your main C++ firmware source code

test/

log.txt # QEMU serial output log

**2. Create the Linker Script (stm32\_flash.ld)**

Save the following content as tools/qemu\_engine/qemu\_stm32/stm32\_flash.ld:

/\* Linker script for STM32F103 microcontroller \*/

MEMORY

{

FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 128K /\* Flash memory \*/

RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 20K /\* SRAM \*/

}

SECTIONS

{

/\* Vector table and code \*/

.text : {

KEEP(\*(.isr\_vector)) /\* Interrupt vector table \*/

\*(.text\*) /\* Program code \*/

\*(.rodata\*) /\* Read-only data \*/

. = ALIGN(4);

} > FLASH

/\* Initialized variables \*/

.data : {

\_sidata = LOADADDR(.data);

\*(.data\*)

. = ALIGN(4);

} > RAM AT > FLASH

/\* Uninitialized variables \*/

.bss : {

\*(.bss\*)

\*(COMMON)

. = ALIGN(4);

} > RAM

/\* Stack configuration \*/

\_estack = ORIGIN(RAM) + LENGTH(RAM); /\* Top of stack \*/

\_\_stack = \_estack;

}

**3. Create the Build Script (build\_firmware.sh)**

Save the following as tools/qemu\_engine/qemu\_stm32/build\_firmware.sh:

#!/bin/bash

set -e # Exit on error

# Navigate to qemu\_stm32 directory

cd "$(dirname "$0")"

# Path to your firmware source code

FIRMWARE\_SRC=../../../apps/watch\_record/watch\_record.cpp

# Output firmware ELF file

FIRMWARE\_OUT=../stm.elf

# Build the firmware

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -Wall -g \

-T stm32\_flash.ld \

-o $FIRMWARE\_OUT \

$FIRMWARE\_SRC

echo "Firmware built successfully: $FIRMWARE\_OUT"

**4. Create the Run Script (run\_qemu.sh)**

Save the following as tools/qemu\_engine/qemu\_stm32/run\_qemu.sh:

#!/bin/bash

set -e # Exit on error

# Navigate to qemu\_stm32 directory

cd "$(dirname "$0")"

# Path to the compiled firmware

FIRMWARE\_ELF=../stm.elf

# Log file path

LOG\_FILE=../../../test/log.txt

# Run QEMU

qemu-system-arm -M stm32-p103 -kernel $FIRMWARE\_ELF -serial file:$LOG\_FILE -S -gdb tcp::1234

echo "QEMU started. Logs are being saved to $LOG\_FILE"

**5. Compile the Firmware**

Run the build\_firmware.sh script to compile the firmware:

cd tools/qemu\_engine/qemu\_stm32

./build\_firmware.sh

After running this command:

* **stm.elf** will be created in tools/qemu\_engine/.

**6. Run QEMU**

Start the QEMU simulation with the compiled firmware:

./run\_qemu.sh

* QEMU will start, and logs will be written to test/log.txt.
* QEMU will listen for GDB connections on port 1234.

**7. Debugging with GDB (Optional)**

Open another terminal and connect to QEMU using GDB:

gdb-multiarch ../stm.elf

target remote localhost:1234

You can now debug your firmware, set breakpoints, and step through the code.

**8. Testing**

To verify the QEMU output, check test/log.txt:

Example Python test script (test/verify\_log.py):

# Path to the log file

log\_file = "log.txt"

with open(log\_file, 'r') as log:

content = log.read()

# Check for expected output

if "Expected Output" in content:

print("Test Passed: Output matches expected.")

else:

print("Test Failed: Output mismatch!")

Run the test script:

cd test

python3 verify\_log.py

**Summary**

1. Create stm32\_flash.ld in qemu\_stm32/.
2. Create build\_firmware.sh and run\_qemu.sh in qemu\_stm32/.
3. Place your firmware source code (watch\_record.cpp) in apps/watch\_record/.
4. Build firmware using build\_firmware.sh.
5. Run QEMU using run\_qemu.sh.
6. Optionally debug with GDB or test with Python.

**2. Update build\_firmware.sh**

Ensure that your linker script (stm32\_flash.ld) is correctly referenced in the build\_firmware.sh file. For example:

arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -Wall -g \

-T stm32\_flash.ld \

-o ../stm.elf \

../../../apps/watch\_record/watch\_record.cpp

**3. Clean and Rebuild**

Sometimes, leftover object files can cause issues. Clean up previous builds:

rm -f ../stm.elf

Rebuild using the script:

./build\_firmware.sh

**4. Debug Linker Script Issues**

If the problem persists, manually check if \_bss\_start and \_bss\_end are defined:

arm-none-eabi-nm ../stm.elf | grep bss

If the symbols are missing, double-check the linker script (stm32\_flash.ld) and ensure all memory sections are defined properly.

Stop QEMU:

pkill qemu-system-arm

**2.Start QEMU with Your ELF**

1. **Run QEMU with the Compiled ELF**:  
   Use the following command to launch QEMU:

qemu-system-arm -M stm32-p103 -kernel stm.elf -S -gdb tcp::1234

* + -M stm32-p103: Simulates the STM32-P103 board.
  + -kernel stm.elf: Loads your compiled firmware.
  + -S: Starts in paused mode.
  + -gdb tcp::1234: Opens a GDB server on port 1234.

Here is a complete **step-by-step guide** to build your firmware, run QEMU, debug with GDB, generate logs, and test your application.

**1. Build the Firmware**

Use the following commands to build your firmware using the linker script:

bash

Copy code

cd tools/qemu\_engine/qemu\_stm32

# Ensure the build script is executable

chmod +x ./build\_firmware.sh

# Build the firmware

./build\_firmware.sh

If the firmware builds successfully, it should generate the stm.elf file in the specified directory.

**2. Run QEMU**

To run QEMU with your STM32 emulation:

bash

Copy code

chmod +x ./run\_qemu.sh

# Start QEMU

./run\_qemu.sh

* **Inside run\_qemu.sh**, ensure it points to your ELF file and provides appropriate settings, e.g.:

bash

Copy code

#!/bin/bash

qemu-system-arm \

-M stm32-p103 \

-nographic \

-kernel ../stm.elf

**3. Debug with GDB**

Start debugging with GDB:

1. **Start QEMU with GDB Stub:** Add -S -gdb tcp::1234 options to run\_qemu.sh to enable debugging:

bash

Copy code

qemu-system-arm \

-M stm32-p103 \

-nographic \

-kernel ../stm.elf \

-S -gdb tcp::1234

Then, run QEMU:

bash

Copy code

./run\_qemu.sh

1. **Connect to GDB:**

Open another terminal and connect GDB to QEMU:

bash

Copy code

arm-none-eabi-gdb ../stm.elf

Inside GDB, connect to the QEMU GDB server:

gdb

Copy code

target remote :1234

1. **Debug Commands in GDB:**
   * Set a breakpoint:

gdb

Copy code

break main

* + Continue execution:

gdb

Copy code

continue

* + Step through code:

gdb

Copy code

step

* + Print variables:

gdb

Copy code

print <variable\_name>

* + Quit GDB:

gdb

Copy code

quit

**4. Kill QEMU**

After running or debugging, stop QEMU by:

* Pressing **Ctrl+C** in the QEMU terminal.
* Or, killing the process:

bash

Copy code

pkill qemu-system-arm

**5. Generate log.txt**

Modify your run\_qemu.sh script to capture logs:

bash

Copy code

#!/bin/bash

qemu-system-arm \

-M stm32-p103 \

-nographic \

-kernel ../stm.elf \

> log.txt 2>&1

Run QEMU again, and all outputs will be saved to log.txt.

**6. Automate Testing**

Write a test script (test\_firmware.sh) to automate building, running, and testing:

bash

Copy code

#!/bin/bash

# Build Firmware

./build\_firmware.sh || { echo "Build failed!"; exit 1; }

# Run QEMU

./run\_qemu.sh &

QEMU\_PID=$!

# Wait for QEMU to start

sleep 2

# Test output (example: check log.txt for success messages)

if grep -q "Expected Output" log.txt; then

echo "Test passed!"

else

echo "Test failed!"

fi

# Kill QEMU

kill $QEMU\_PID

Make it executable and run:

bash

Copy code

chmod +x test\_firmware.sh

./test\_firmware.sh

Once QEMU is running, you can interact with your firmware to test its behavior. Here's what you need to do while QEMU is running:

**1. Start QEMU**

Run the following command to start QEMU with your firmware:

bash

Copy code

./run\_qemu.sh

**2. Observe Firmware Output**

* If your firmware outputs data to a UART or console, it will appear in the terminal where QEMU is running (assuming -nographic is used in run\_qemu.sh).
* For example, if your firmware toggles an LED or writes logs, you should see related messages or behavior in the QEMU console.

**3. Debug Firmware on QEMU**

You can debug your firmware running on QEMU by attaching GDB.

**Step-by-Step Debugging**

1. **Modify run\_qemu.sh for GDB Stub:** Update the script to include GDB debugging options:

bash

Copy code

qemu-system-arm \

-M stm32-p103 \

-nographic \

-kernel ../stm.elf \

-S -gdb tcp::1234

1. **Run QEMU:**

bash

Copy code

./run\_qemu.sh

1. **Connect GDB:** Open another terminal and connect to the GDB server using arm-none-eabi-gdb:

bash

Copy code

arm-none-eabi-gdb ../stm.elf

1. **Attach to QEMU:** Inside GDB:

gdb

Copy code

target remote :1234

1. **Debugging Commands:**
   * Set breakpoints:

gdb

Copy code

break main

* + Run firmware until breakpoint:

gdb

Copy code

continue

* + Inspect registers:

gdb

Copy code

info registers

* + Print variable values:

gdb

Copy code

print <variable\_name>

* + Step through code:

gdb

Copy code

step

* + Quit GDB:

gdb

Copy code

quit

**4. Interact with Peripherals**

QEMU emulates STM32 peripherals like GPIO, UART, and timers. If your firmware controls peripherals:

* **GPIO:** Check for GPIO changes in your code. For example, if your firmware toggles an LED, QEMU might log messages or change a GPIO state.
* **UART:** If your firmware uses UART for communication, output should be visible in the QEMU terminal. You can also send data to UART using GDB or scripts.

**5. Check Logs**

If your run\_qemu.sh is set to redirect output to log.txt:

bash

Copy code

qemu-system-arm \

-M stm32-p103 \

-nographic \

-kernel ../stm.elf \

> log.txt 2>&1

You can inspect log.txt after running QEMU:

bash

Copy code

cat log.txt

**6. Run Automated Tests**

Write a test script (test\_firmware.sh) to run QEMU, check its behavior, and validate output. Example:

bash

Copy code

#!/bin/bash

# Run QEMU

./run\_qemu.sh &

QEMU\_PID=$!

# Wait for QEMU to initialize

sleep 2

# Check log output

if grep -q "Expected Output" log.txt; then

echo "Test passed!"

else

echo "Test failed!"

fi

# Kill QEMU

kill $QEMU\_PID

Run the test script:

bash

Copy code

chmod +x test\_firmware.sh

./test\_firmware.sh

**7. Kill QEMU**

After you finish debugging or testing, stop QEMU:

* Press Ctrl+C in the QEMU terminal.
* Or, kill the process from another terminal:

bash

Copy code

pkill qemu-system-arm

FOR ARDUINO SIMLUTION

https://github.com/arduino/ArduinoCore-avr.git

sudo apt install gcc-avr

sudo apt install gdb-avr

sudo apt install simulavr

sudo apt install avr-libc gcc-avr

avr-g++ -g -Os -mmcu=atmega328p -DF\_CPU=16000000UL -I/home/rps/wipro/Embeded\_Testing/Day\_4/Assignments/Ass\_1/ArduinoCore-avr/cores/arduino -I/home/rps/wipro/Embeded\_Testing/Day\_4/Assignments/Ass\_1/ArduinoCore-avr/variants/standard -c /home/rps/wipro/Embeded\_Testing/Day\_4/Assignments/Ass\_1/ArduinoCore-avr/cores/arduino/blink.cpp -o blink.elf

avr-gdb blink.elf

**2. Build AVR Firmware**

Compile the firmware using gcc-avr:

bash

Copy code

avr-gcc -mmcu=atmega328p -DF\_CPU=16000000UL -o blink.elf blink.c

avr-objcopy -O ihex -R .eeprom blink.elf blink.hex

**3. Simulate with SimulAVR**

Run the firmware using simulavr:

bash

Copy code

simulavr --device atmega328p --clock 16000000 --hex-file blink.hex

**4. Debug with GDB**

1. Start SimulAVR with GDB:

bash

Copy code

simulavr --gdbserver --device atmega328p --clock 16000000 --hex-file blink.hex

1. Connect with GDB:

bash

Copy code

avr-gdb blink.elf

target remote localhost:1212