Bare-Metal Embedded C Project with CMake + External Libraries

Example: Integrating libopencm3 into STM32 Firmware Builds

This repository demonstrates how to structure a **bare-metal embedded firmware project** using **CMake**, where external libraries (such as *libopencm3*) are built and linked into your firmware automatically.

Even though this specific example uses **libopencm3** on an **STM32F4**, the techniques here are **generic** and apply to any library or architecture (ARM Cortex-M, AVR, RISC-V, etc).

⁂ Project Goals

- Write clean firmware in C using a modern build system.
- Automatically build an external dependency (like libopencm3).
- Link the library into firmware cleanly.
- Avoid vendor IDEs (CubeIDE, Keil, IAR, etc.).
- Produce `firmware.elf` and `firmware.bin`.
- Flash the board using `st-flash` (or any tool you choose).

⁂ Directory Structure

```
project/
├── app/
│   ├── src/firmware.c
│   ├── inc/common-defines.h
│   └── linkerscript.ld
├── libopencm3/          <- External library (git submodule or copy)
├── CMakeLists.txt        <- Main build script
└── README.md
```

You can replace `libopencm3` with any other embedded library (CMSIS, FreeRTOS, TinyUSB, LWIP, etc).

⁂ How CMake Fits Into Embedded Development

Unlike desktop software, embedded firmware needs:

- A cross-compiler (e.g., `arm-none-eabi-gcc`)
- A custom memory layout (linker script)
- No OS, no dynamic loader
- No exceptions, no unwind tables
- No `crtbegin.o` / `crtend.o`
- No system libraries unless explicitly added

CMake **does not know any of this by default**, so *you must teach it*.

This repository shows how to do it cleanly.

⁂ Key Concepts to Understand

1. Bare-metal toolchain setup

You must explicitly tell CMake which compiler to use:

```
set(CMAKE_C_COMPILER arm-none-eabi-gcc)
set(CMAKE_ASM_COMPILER arm-none-eabi-gcc)
```

Without this, CMake will mistakenly use your PC compiler (`/usr/bin/cc`).

---

2. Compiler flags for Cortex-M

Bare-metal firmware requires:

- Architecture flags
- Optimization
- Removing exceptions / C++ features
- Float ABI
- Family define (`-DSTM32F4`)

Example:

```
-mcpu=cortex-m4 -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16
-ffunction-sections -fdata-sections
-fno-exceptions -fno-unwind-tables
```

These flags ensure:

- The code actually runs on your MCU.
- No unwanted runtime objects are linked.
- The compiled objects match the library ABI.

---

3. Linker flags

Since this is bare-metal:

- No system startup files
- No default system libs
- Custom linker script
- Garbage-collection of unused sections

```
-T app/linkerscript.ld
-nostdlib -nostartfiles -nodefaultlibs
-Wl,--gc-sections
```

This guarantees your firmware maps correctly into flash and RAM.

---

4. ExternalProject_Add — compiling external libraries

Most embedded libraries do **not** use CMake.

So CMake must *call their build system* (usually Make):

```
ExternalProject_Add(libopencm3_project
    SOURCE_DIR ${CMAKE_SOURCE_DIR}/libopencm3
    BINARY_DIR ${CMAKE_SOURCE_DIR}/libopencm3
    BUILD_COMMAND make TARGETS=stm32/f4
)
```

This makes your project reproducible:

- CMake drives the build (no manual “cd libopencm3 && make”)
- libopencm3 always builds before your firmware
- Works with CI (GitHub Actions, GitLab CI, etc.)

---

5. Linking the generated library

Once built, the library is linked manually:

```
target_link_libraries(firmware.elf
    ${OPENCM3_LIB}
    c
    nosys
)
```

The only C libraries normally needed for bare-metal:

- `libc` (newlib-nano if you want small size)
- `libnosys` (stubs for syscalls)

---

6. Generating a firmware.bin

CMake itself only produces `.elf`.
You convert it using objcopy:

```
add_custom_command(TARGET firmware.elf POST_BUILD
    COMMAND ${CMAKE_OBJCOPY} -O binary firmware.elf firmware.bin
)
```

⚒ Building the Project

```
mkdir build
cd build
cmake ..
make -j
```

You will see:

```
libopencm3_project → builds first
firmware.elf → linked with correct flags
firmware.bin → auto-generated
```

⚒ Flashing the Firmware (STM32 example)

Using **st-flash**:

```
st-flash write firmware.bin 0x08000000
```

Check connection:

```
st-info --probe
```

This workflow works for any debugger/loader:

- OpenOCD
- pyocd
- ST-Link
- JLink
- BlackMagic Probe

⚒ Applying This Template to Future Projects

You can reuse this structure by changing only:

1. MCU architecture and float ABI

Examples:

- Cortex-M0: `-mcpu=cortex-m0 -mthumb -mfloat-abi=soft`
- Cortex-M7: `-mcpu=cortex-m7 -mfpu=fpv5-d16 -mfloat-abi=hard`

2. Memory layout (linkerscript.ld)

Different MCU → different flash/RAM map.

3. Library build command

Replace libopencm3 with:

- FreeRTOS
- CMSIS
- TinyUSB
- FatFS
- LWIP

Just change:

```
BUILD_COMMAND make
```

to whatever that library needs.

4. Target sources

```
add_executable(firmware.elf
    app/src/main.c
    app/src/drivers/uart.c
    ...
)
```

5. Flash tool

Change ST-Link → OpenOCD or JLink as needed.

⚙️ Why This Structure is Modern & Scalable

- No vendor lock-in (CubeIDE / IAR / Keil)
- Reproducible builds (good for CI/CD)
- Automatic dependency building
- Clean separation of:
    - app code
    - external libraries
    - build logic
- Language-agnostic (C, ASM)

This is how professional embedded teams structure their firmware builds.

🧠 Summary

This example teaches how to:

✔ Set up a bare-metal ARM toolchain
✔ Build external libraries inside CMake
✔ Link firmware cleanly without vendor IDEs
✔ Produce `.elf` and `.bin` artifacts
✔ Flash with ST-Link
✔ Scale this structure to any MCU

Use this template as your starter for all future embedded projects.

🚀 Need more?

I can also generate:

- A complete **template repo** (startup, linker script, example firmware)
- VSCode Cortex-Debug configurations
- GDB helper scripts
- Newlib-nano integration
- FreeRTOS + CMake template

Just ask.