Coral

Build apps with FreeRTOS

All development for the Coral Dev Board Micro is done in C/C++ and relies heavily on FreeRTOS: a real-time operating system kernel that provides a variety of system features that are traditionally not available on microcontrollers, such as multitasking.

This page describes the basic workflow to create an app for the Dev Board Micro with FreeRTOS, including as how to set up a new project, build it, and flash it to the board.

If you instead want information about the FreeRTOS platform features, refer to the FreeRTOS documentation. Or for details about using board-specific features, such as the camera, microphone, GPIO pins, and TensorFlow Lite, refer to the coralmicro API reference.

Project overview

The basic project setup requires just two files: A C++ source file with an `app_main()` function and a `CMakeLists.txt` file that defines your build requirements.

For example, here's a "Hello World" source file (it prints to the serial console):

```
#include <cstdio>

#include "third_party/freertos_kernel/include/FreeRTOS.h"
#include "third_party/freertos_kernel/include/task.h"

extern "C" [[noreturn]] void app_main(void *param) {
  (void)param;
  printf("Hello world!\r\n");
  vTaskSuspend(nullptr);
}
```

> Note: If you're familiar with FreeRTOS, you probably expected to see a `main()` function here. However, that function is defined internally to initialize hardware you might need and then it calls your `app_main()` within a new FreeRTOS task.

And here's the `CMakeLists.txt` file to go with it:

```
add_executable_m7(hello_world
    hello_world.cc
)

target_link_libraries(hello_world
    libs_base-m7_freertos
)
```

This CMake configuration declares the executable name and source file with `add_executable_m7()` (a wrapper for `add_executable()`) and specifies the library dependencies with `target_link_libraries()`. You can learn more about how to write this file in the CMake documentation.

That's basically all you need. That is, assuming CMake can find the `libs_base-m7_freertos` library that you require. To ensure that it can, your project needs to include the `coralmicro` source code. There are two common ways you can set up your project to do this, as described in the next section.

Project setup

To start a new project that includes `coralmicro`, you first need to decide where you want to keep your code. You basically have two options:

- Out-of-tree project: Your code exists independent from the `coralmicro` source tree, and includes `coralmicro` as a submodule. (It's also possible to link to the `coralmicro` source from another external location, but using a submodule is a bit easier so that's what we'll covered here.)

  This is a good choice if you don't plan to modify `coralmicro`. It will be easier to keep the `coralmicro` submodule up-to-date with any mainline changes.

- In-tree project: Your code exists inside the `coralmicro` source tree, just like the included `examples` and `apps`. Basically, you will fork the `coralmicro` repo and put your project(s) inside it.

  This is a good choice if you plan to modify some of the `coralmicro` libraries.

The following sections provide a step-by-step guide to get started with either option.

Create an out-of-tree project

To create an out-of-tree project, you need to add `coralmicro` as a submodule. To get you started, we've created an example project you can clone or fork:

1. Clone our out-of-tree example project (it prints "Hello World"):

   > Tip: If you will manage your project in GitHub, fork this out-of-tree-sample and clone that fork instead.

   ```
   git clone https://github.com/google-coral/coralmicro-out-of-tree-sample
   ```

2. Initialize the `coralmicro` submodule and all its submodules:

   ```
   cd out-of-tree-sample

   git submodule add https://github.com/google-coral/coralmicro coralmicro

   git submodule update --init --recursive
   ```

3. Install the required development tools (such as CMake) with this script:

   ```
   bash coralmicro/setup.sh
   ```

That's it. You now have an out-of-tree project and you can start coding. Or continue to the next section and try flashing it to your board.

Build and flash

You can build and flash your out-of-tree project like this:

1. Generate the project Makefile (run this from the `out-of-tree-sample` root):

   ```
   # -B specifies the path for your build output path and
   # -S specifies the path to the CMakeLists.txt file.
   cmake -B out -S .
   ```

2. Build the app:

   ```
   make -C out -j4
   ```

   To maximize your CPU usage, replace `-j4` with either `-j$(nproc)` on Linux or `-j$(sysctl -n hw.ncpu)` on Mac.

3. Flash the app to your board:

   ```
   python3 coralmicro/scripts/flashtool.py --build_dir out --elf_path out/coralmicro-app
   ```

   `coralmicro-app` is the executable name that's specified in `CMakeLists.txt`, so that's the ELF file name you must specify with `--elf_path`.

   > Note: In addition to specifying the path to your ELF file with `--elf_path`, you must specify the build output directory with `--build_dir` because flashtool needs to get the elf_loader (bootloader) program from there. Whereas, when flashing in-tree examples and apps, `--build_dir` can be ommitted because flashtool uses the default in-tree "build" directory. Similarly, in-tree examples/apps don't need to specify `elf_path` because those files reside in the same build directory, so you can instead specify just the project name with `--example` (or `-e`) and `--app` (or `-a`).

When flashing is done, the board reboots and loads the app. You should see the green LED turn on. To see the "Hello World" message, connect to the serial console.

When you modify the source code or CMake configuration, just rebuild and reflash the app:

```
make -C out -j4

python3 coralmicro/scripts/flashtool.py --build_dir out --elf_path out/coralmicro-app
```

Create an in-tree project

To create an in-tree project, you'll start with the `coralmicro` repository and add your project inside it:

1. Clone the `coralmicro` repo (you probably already did this during setup):

   > Tip: If you will manage your project in GitHub, fork the coralmicro repo and clone that fork instead.

   ```
   git clone https://github.com/google-coral/coralmicro/
   ```

2. Create a new directory in `coralmicro/apps/` and populate it with some files by copying code from an existing example project. For example:

   ```
   cd coralmicro

   # Copy the "hello world" code
   cp -r examples/hello_world apps/my_project

   # Rename the executable
   mv apps/my_project/hello_world.cc apps/my_project/main.cc
   ```

3. Update the project and file names in `apps/my_project/CMakeLists.txt`, respective to their new names:

   ```
   add_executable_m7(my_project
       main.cc
   )

   target_link_libraries(my_project
       libs_base-m7_freertos
   )
   ```

4. Add the project as a new directory in `coralmicro/apps/CMakeLists.txt`:

   ```
   add_subdirectory(my_project)
   ```

5.

Install the required development tools (such as CMake) with the script that's appropriate for your system (run this from the `coralmicro` root):

- On Linux:

```
bash setup_linux.sh
```

- On Mac:

```
bash setup_mac.sh
```

That's it. You now have an in-tree project and you can start coding. Or continue to the next section and try flashing it to your board.

Build and flash

You can build and flash your in-tree project like this:

1. Build the app along with everything else in the tree (run this from the `coralmicro` root):

```
bash build.sh
```

2. Flash the app to your board:

```
python3 scripts/flashtool.py --app my_project
```

> Note: By default, flashtool looks for all binaries in "build" directory, so you need only specify the project name with `--app`. If you specify a different build path with `cmake -B`, then you must specify that path with the `--build_dir` argument, because that path is also where flashtool needs to find the elf_loader (bootloader) program.

When flashing is done, the board reboots and loads the app. You should see the green LED turn on. To see the "Hello World" message, connect to the serial console.

When you modify the source code or CMake configuration, just rebuild and reflash the app:

```
make -C build/apps/my_project -j4

python3 scripts/flashtool.py --app my_project
```

To maximize your CPU usage, replace `-j4` with either `-j$(nproc)` on Linux or `-j$(sysctl -n hw.ncpu)` on Mac.

FreeRTOS tasks

By default, your main app is automatically executed as a FreeRTOS task, so it shares execution time on the MCU with other coralmicro tasks that automatically run as needed. For example, when you use the camera API, the camera task manages the camera hardware and sends images back to your app.

The default priority level assigned to your app is defined by the constant `kAppTaskPriority` (from `libs/base/tasks.h`), so you should always use this level use when starting a new task with the FreeRTOS function `xTaskCreate()`. For example:

```
#include "libs/base/led.h"
#include "libs/base/tasks.h"
#include "third_party/freertos_kernel/include/FreeRTOS.h"
#include "third_party/freertos_kernel/include/task.h"

[[noreturn]] void blink_task(void* param) {
  auto led_type = static_cast<coralmicro::Led*>(param);
  bool on = true;
  while (true) {
    on = !on;
    coralmicro::LedSet(*led_type, on);
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}

extern "C" void app_main(void* param) {
  (void)param;
  auto user_led = coralmicro::Led::kUser;
  xTaskCreate(&blink_task, "blink_user_led_task", configMINIMAL_STACK_SIZE,
              &user_led, coralmicro::kAppTaskPriority, nullptr);
  vTaskSuspend(nullptr);
}
```

coralmicro libraries

Although FreeRTOS provides the foundational OS features for the Dev Board Micro, most of the behavioral features in your app will come from the `coralmicro` libraries, which you can explore in the coralmicro API reference.

Fortunately, a lot of `coralmicro` libraries are included by default with the `libs_base-m7_freertos` library, such as libraries to use the board GPIOs, camera, audio, TensorFlow, filesystem, and much more. You need to include this library anyway because it also provides the `main()` function that FreeRTOS requires (which calls through to your program's `app_main()` function).

However, if your build fails with an "undefined reference," then you're missing the link for a library you're trying to use. Take note of which code is undefined and then go to the source file for the corresponding header, which you should be able to find linked in each section of the coralmicro API reference. In the same directory as the header file, look in the local `CMakeLists.txt` and you'll see the library name specified with `add_library_m7()` and/or `add_library_m4()` (these are thin wrappers for `add_library()`), followed by the `.cc` files that are included with that library name.

For example, if your code uses `libs/curl/curl.h`, you'll need the library name that's defined in `libs/curl/CMakeLists.txt`:

```
add_library_m7(libs_curl STATIC
    ...
)
```

So simply update your `CMakeLists.txt` file by adding `libs_curl` in the `target_link_libraries()` command:

```
target_link_libraries(my_project
    libs_base-m7_freertos
    libs_curl
)
```

Beware that some libraries offer both an M7 and an M4 version (such as `libs_base-m7_freertos` and `libs_base-m4_freertos`), so it's important that you specify the library name that corresponds to the MCU core (either M7 or M4) where your executable will run. For information about how to use the M4 core, see how to create a multi-core app.