Port Process

Date: June 10 2020

Brass, Bliss
Maldonado, Ruben
Nikolov, Nikolay
Schulte, Eric

If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

- *Linux 1.3.53 CodingStyle documentation*

Revision:

Version	Date	Comment
0.5	05/20/2020	Initial draft
1.0	06/07/2020	Initial Release
1.1	06/09/2020	Updating main content

Table of Contents:

Introduction:

The purpose of this document is to explain how we ported Betaflight's configuration initialization process to RISC-V architecture, specifically the Maixbit dev board with Kendryte K210 SoC (system-on-chip). It also includes future steps necessary to continue the port (steps we would have eventually gotten to with more time).

The purpose of this document is not to explain each and every line of code we added. However it will cover some code. It assumes that the reader has prior knowledge of C and has some familiarity with the topic.The intent behind it is to help the reader continue working where we left without the need to start over. Although, it also provides a strategy of how to port to a new board.

We hope that the end result provides an outline of a process that can be easily followed and replicated, and provides a useful insight when porting Betaflight to other microprocessors, or even other architectures.

Important note for our reader:
This guide is meant for Linux. If you are using Windows, then you should install Linux and forget about Windows. It will save time from waiting 5 minutes daily to boot and it will never crash (unless you start tweaking things). Ubuntu is a good choice for this project, there is a plethora of tutorials and great support. Mint is excellent too. It is a fork of Ubuntu.

Step 1: Gathering critical information

The most critical part related to this project is by far gathering the required resources. This will prevent future failures and misfortune. A general list is outlined below. It is applicable for a variety of projects and is an excellent place to start.

Identify:

1. **Hardware requirements**
    - How large is your hardware's Memory, CPU, and FLASH?
    - How many IO pins are available?
2. **Software requirements and support**
    - Support from community - proprietary or open source
    - Identify main points of contact for support
        - Libraries, API , Toolchain(s)
    - Is the hardware supported from the vendor or do you have to build everything from scratch?
    - Track down datasheets, technical manuals, and repositories
    - Identify dependencies and critical components before begin porting
        - Does the selected hardware have the capabilities to run the software?
        - If not, then change hardware, software or both parameters
3. **Outline the high-level flow of the process you are porting**
    - Where does it start, what gets compiled and linked, how does it get built?
        - What files have the highest dependencies?
        - What are these dependencies?
4. **Build couple of examples from the architecture you are porting**

In the next section we will present and touch more on Betaflight-specific material. However, it is easy to abstract the problem-solving aspect to other similar projects.

Step 2: The Min. Port - Building a binary from the minimum required sources

The next step is to create a simple binary using the project's Makefiles. The goal here is to successfully build. In Betaflight that entails modifying the Makefiles, creating a target directory and adding the most minimal set of functionalities that can prove a successful build.

For the sake of simplicity we will call that '*min. port,*' since it involves building an executable from the least amount of required source files with no or limited Betaflight sources (empty main.c).

The min. port executable should include architecture-specific files (drivers and board support package) as source files, but should not include any Betaflight source code (yet). References to architecture-specific compiler tools will need to be referenced into the Makefile, so that the software can be built for the new target architecture (vanilla Betaflight only includes ARM tools).

At the end we are looking to establish the infrastructure needed to build a minimal executable for the new target architecture, bypassing many components that are going to be introduced later.

Useful Notes:

- Throughout the process compile and build frequently to test. If something works, add, commit and push.
- Read compiler and linker errors carefully, often they can provide useful insight. Errors are there to help you, don't get scared.
- Is a file missing? Where do you have the file? How are you making it clear to the compiler/linker where to look?
- Go slow, keep notes/log on what has been changed.
- The steps are not entirely sequential. It is rather like a pipeline, you begin modifying one file, then you start modifying another one and you observe dependencies, and you are trying continually to fix them.

Getting Started with the ‘Min. Port’

We can begin by priming some files and folders.These are the basic changes that need to be done in the root folder Makefile and leaf Makefiles. It is expected some familiarity with GNU Makefiles and C. GNU has excellent tutorials on both that we highly recommend.

1. Add the toolchain

We can start by referencing the toolchain provided from Kendryte in the Makefile. It is advisable to install the toolchain prior to modifying the Makefile and testing it by building and flashing a few demo examples from Kendryte. The links are provided in the wiki.

After testing the demo examples and flashing to the Maixbit board we can go ahead and begin modifying and setting up the foundations.

It is important to note that the permanent addition of the toolchain and the toolchain reference in the ecosystem of BF requires a more sophisticated and elegant approach than the one presented. Potentially, another Makefile or use of CMake.

The change below can go right after the reference for the Arm toolchain in the Makefile. What is more, if you are not familiar with Make, keep in mind that Makefiles are scripts, everything happens sequentially. Hence, the sequence of calls does matter for most things.
Here is an example of a code snippet that introduces the toolchain in Make.

```
ifneq ($(TARGET),$(filter $(TARGET),$(MAIXBIT)))
RISCV64_SDK_PREFIX = /opt/kendryte-toolchain/bin/riscv64-unknown-elf-

CROSS_CC     := $(RISCV64_SDK_PREFIX)gcc #$(CCACHE) $(ARM_SDK_PREFIX)gcc
CROSS_CXX   := $(RISCV64_SDK_PREFIX)g++  #$(CCACHE) $(ARM_SDK_PREFIX)g++
CROSS_GDB   := $(RISCV64_SDK_PREFIX)-gdb  #$(ARM_SDK_PREFIX)gdb
OBJCOPY      := $(RISCV64_SDK_PREFIX)objcopy  #$(ARM_SDK_PREFIX)objcopy
OBJDUMP      := $(RISCV64_SDK_PREFIX)objdump  #$(ARM_SDK_PREFIX)objdump
SIZE    := $(RISCV64_SDK_PREFIX)size  #$(ARM_SDK_PREFIX)size
DFUSE-PACK  := src/utils/dfuse-pack.py
endif
```

2.a Add the board in the VALID_TARGETS macro

The second thing we need to do is to create a directory in `src/main/target/<TARGET_NAME>`. This is our target directory.

In general, two things are always needed, 1) a method (not a class method) to gather the source code and build, and 2) to orchestrate that method around a Target (not the store). Target is an abstraction. It can be anything. In our case the target is the board, and we are aiming to compile everything that this board needs to make it work.

2.b In the src/main/target/ create 3 dummy files.

The 3 files are target.c, target.h, target.mk. In target.mk we need only one line on the top of target.mk <TARGET_MCU>_TARGETS += $(TARGET). In our example we named it RISCV_K210_TARGETS += $(TARGET). This will add the board's mcu in the list of targets. We will continue filling in these files in a later step.

The board has to be in the list of valid targets, otherwise it is impossible to do anything useful. The valid targets reside in make/targets_list.mk.

Note, the line below in targets_list.mk is looking for that particular directory.

```
Line 1: ALT_TARGET_PATHS = $(filter-out %/target,$(basename $(wildcard
$(ROOT)/src/main/target/*/*.mk) // this is why we created the directory
above
```

3. Add the board in make/targets.mk

Once we have created the target directory we can continue with adding the insurance of our target in the rest of the Makefiles. In make/targets.mk we will add the board in the existing list with boards and then identify it with a TARGET_MCU.

```
ifeq ($(filter $(TARGET),$(F1_TARGETS) $(F3_TARGETS) $(F4_TARGETS)
$(F7_TARGETS) $(H7_TARGETS) $(SITL_TARGETS) $(RISCV_K210_TARGETS)),)
$(error Target '$(TARGET)' has not specified a valid STM group, must be one
of F1, F3, F405, F411, F446, F7X2RE, F7X5XE, F7X5XG, F7X5XI, F7X6XG or
H7X3XI. Have you prepared a valid target.mk?) endif

##Add an ifeq statement like below to add the board as TARGET_MCU

else ifeq ($(TARGET),$(filter $(TARGET), $(RISCV_K210_TARGETS)))
```

```
TARGET_MCU := RISCV_K210
RISCV_K210 = yes
```

4. Create an empty MCU Makefile - <TARGET_MCU>.mk in the make/mcu

In this step we will create a placeholder for the Makefile that will provide most of the relevant sources, flags and settings for our target mcu. It is preferable to be an empty file in the beginning, rather than copying an existing file from another board, and later we will add only what is needed. The name for the file is the name of the MCU that is on the board.

In our example, it was Kendryte 210 or K210, e.g RISCV_K210.mk (we included the name of the architecture as well). Later, the file will contain specific compiler and linker flags as well source files.

5. Create an empty directory with the TARGET_MCU name in lib/main

In our case we had RISCV_K210 as TARGET_MCU. Later, we will revisit that folder and add the SDK files.

6. Modify source.mk that defines common source files

We can add a piece as seen below in order to selectively add sources as needed rather than commenting the entire file. That way we can toggle what files are compiled and linked, starting with only main.c as a common source and the board-specific drivers included in the lib/main/<TARGET_MCU> directory, and sourced from TARGET_MCU.mk (or potentially in target.mk as well).

```
# -------------------------------------------- +
#### RISCV                                                          |
#### -----------------------------------------+
ifneq ($(TARGET),$(filter $(TARGET),$(MAIXBIT)))
COMMON_SRC = \
              build/build_config.c \
              build/version.c \
              capstone_print.c \
              fc/init.c \
              pg/pg.c\
              sensors/initialisation.c \
              sensors/gyro.c \
              drivers/flash.c \
              config/config_eeprom.c \
              config/config_streamer.c\
              config/config.c \
              common/crc.c \
              config/feature.c \
              fc/board_info.c \
              pg/board.c\
              drivers/bus_spi.c \
              drivers/bus_spi_pinconfig.c \
              drivers/bus_spi_config.c \
              pg/flash.c\
              rx/rx.c \
              drivers/io.c \
              pg/flash_riscv_k210.c

ifneq ($(TEST),yes)
MCU_COMMON_SRC := main.c
endif

endif

SRC := $(MCU_COMMON_SRC) $(STARTUP_SRC) $(TARGET_SRC) $(COMMON_SRC)
#--------------------------------------------------+
```

Understanding the process

The porting starts with the top-level makefile in the root directory. It is responsible for building and linking the entire project through helper makefiles. The makefile is responsible for invoking the most architecture-specific component: the toolchain. Using the architecture-specific toolchain, the makefile will produce the binary executable.

Make accesses the Makefile and its helper makefiles recursively. It starts from the top-level makefile in root and accesses helper makefiles using include statements inserted throughout the makefile. This order matters - because you can overwrite configurations along the way throughout the helper makefiles. The order of how you specify arguments in the Makefile matters.

Start small. Comment out almost everything. Keep the barebones that the Makefile needs to build a simple 'Hello world.' Comment most besides where the Target gets invoked and start from there. Once we can successfully invoke *Make* and specify a target , we can try to create a simple binary. The successful miniport/build is a stepping stone in order to be able to build the project and continue forward.

Useful Note*: Don't be afraid of errors. The errors point us to the correct dependencies that we need to fix. Don't try to have everything right from the beginning. The process is fairly recursive. It is good to keep that in mind. It is not linear, nor sequential.*

It is good to think of it as a network of nodes depending on each other. The Makefile is the central node and then you are moving from branch to branch and you are establishing the proper network between the nodes. Hence, in every attempt we *Make* and build we will uncover what is missing.

That part of the process is really just grinding and digging. There is nothing elegant. There are no shortcuts either. Potentially some tools that uncover dependencies or the debug mode for Make can help.

You have to traverse the nodes and build these connections. Either supplementing and creating directories, adding files, adding libraries, and modifying existing sources. This part greatly differs from project to project. But the concept is similar. First build with the least amount of source files.

```
The process goes like this:
Did you build successfully?
yes - good, move to the next thing below in the file and build again
No - check the errors and  fix the broken parts
Add, modify, and change and build again
```

Useful Note: DO NOT attempt to blindly copy defines, includes or flags from different files related to a different architecture. You should copy/add only the ones which you know you need and gradually add new pieces after testing. It might cost you more time to find out through trial and error, than actually being smart and doing your research.

Modifying and creating makefile dependencies

Once we have created some sort of a framework we will go back and add more features to our Makefiles. Remember, the first thing we need to accomplish is to build.

We will begin with the MCU makefile target_mcu.mk:

1. Populate the MCU makefile

Each type of microprocessor supported by Betaflight has its own MCU makefile. These files are located within ./make/MCU. The general purpose of these MCU makefiles is to provide the top-level makefile with MCU-specific source files needed to boot and run programs on each of the supported MCUs. Each MCU model needs its own makefile because each model uses different source files. MCU source files include vendor-supplied peripheral driver software, assembly startup files, and linker scripts.

Additionally, the MCU makefiles are where device-specific compiler and linker flags are set. Each MCU makefile follows a similar structure to source.mk; source files, flags, and other data are saved as variables that get accessed by the top-level Makefile during the make process.

The main features that we need to add are:

1) **Important directories and their paths:**

```
vpath %.ld ../../src/link/riscv_flash_k210.ld
TARGET_FLASH_SIZE    :=
LINKER_DIR          := $(ROOT)/src/link/
STDPERIPH_DIR        := $(ROOT)/lib/main/
SRC_DIR             := $(ROOT)/src/main
LD_SCRIPT           = $(LINKER_DIR)
STARTUP_SRC          = $(ROOT)/src/main/startup/
```

2) **Important flags**

```
TARGET_FLAGS         = -D$(TARGET_MCU)
OPBL = yes
DEVICE_FLAGS                +=-D$<some_taget_flag>
```

3) **Source file for Make**

```
TARGET_SRC                := \
                          $(wildcard $(STDPERIPH_DIR)/drivers/*.c) \
                          $(wildcard $(STDPERIPH_DIR)/bsp/*.c) \
                          target/$(TARGET)/target.c \
```

4) **Directories for header files "Includes"**

```
DRIVER_INCLUDES             = $(STDPERIPH_DIR)/drivers/
UTILS_INCLUDES              = $(STDPERIPH_DIR)/utils/include
SYSTEM                      = $(ROOT)/sysroot
```

```
INCLUDE_DIRS         := \
                          $(INCLUDE_DIRS) \
                          $(STDPERIPH_DIR) \
                          $(DRIVER_INCLUDES) \
                          $(ROOT)/src/main/target/$(TARGET) \
                          $(ROOT)/src/main/build \
                          $(ROOT)/src/main \
                          $(ROOT)/src
```

5) **Architecture Specific Flags**

```
ARCH_FLAGS          = -march=rv64imafc -mabi=lp64f -mcmodel=medany
```

6) **Assembly flags**

```
ASFLAGS                   = $(ARCH_FLAGS) \
                          -x assembler-with-cpp -D __riscv64 \
                          $(addprefix -I,$(INCLUDE_DIRS))
```

7) **Linker FLAGS**

Note: Be careful, you might need different flags. This is only an example

```
LD_FLAGS                  = \
                          -nostartfiles \
                          -T $(LD_SCRIPT) \
                          -Wl,-static \
                          -Wl,--start-group \
                          -Wl,--whole-archive \
                          -Wl,--no-whole-archive \
                          -Wl,--end-group \
                          -Wl,-EL \
                          -Wl,--no-relax \
                          -g \
                          -O3 -O2 \
                          -lm \
                          -lstdc++ \
                          -lc -lgloss -lgcc\
                          -specs=nano.specs -specs=nosys.specs\
                          -lnosys \
                          -Wl,-gc-sections,-Map,$(TARGET_MAP) \
                          -Wl,--cref \
                          -Wl,--print-memory-usage \
                          -lnosys \
                          $(LTO_FLAGS)
```

8) **GCC FLAGS (Compiler flags)**

Usually many of the flags for the linker will be present here. In this example we have a stripped version of the flags used in the project.

```
CFLAGS        = \
                         $(ARCH_FLAGS) \
                         -std=gnu11 \
                               -Os\
                         -ggdb \
                         -Wall \
                         $(addprefix -I,$(INCLUDE_DIRS)) \
                         $(TARGET_FLAGS) \
                         $(DEVICE_FLAGS) \
                         -D$(TARGET) \
                         -D'__FORKNAME__="$(FORKNAME)"' \
                         -D'__TARGET__="$(TARGET)"' \
                         -D'__REVISION__="$(REVISION)"' \
                         -DUSE_DEVICE_STDPERIPH_SRC \
                         -g
```

9) **Debug, this part of flags is almost optional and related to the GCC flags**

```
ifneq ($(DEBUG),GDB)
OPTIMISE_DEFAULT  := -Os
OPTIMISE_SPEED          :=
OPTIMISE_SIZE           :=
LTO_FLAGS               := $(OPTIMISATION_BASE) $(OPTIMISE_DEFAULT)
endif
```

2. Add new target to targets.mk

Targets.mk is a Makefile found within the top-level make folder. It’s purpose is to organize all target boards by their microprocessor, and ensure the microprocessor is supported by Betaflight. Based on the type of its microprocessor, target boards are separated into groups: F1_TARGETS, F3_TARGETS, F4_TARGETS, F7_TARGETS, H7_TARGETS. The variable TARGET_MCU gets set depending on which group the target board belongs to. To accommodate a target board with a new architecture, a new group must be created within targets.mk.

3. Create target folder for target board (target folder = TARGET)

Each flight control board supported by Betaflight has its own target folder (located: rc/main/target). We created the ‘MAIXBIT’ folder for our RISC-V board. Within each board’s folder there are (at least) the same three files:

a. Target.c

This file contains the definitions of timer hardware and pin numbers. In this project, this file was not critical to be done at first - and it’s something to be completed down the road. We left this file empty within the MAIXBIT folder, with only some timer header files included.

b. Target.h

This file is one of the most critical, along with target_mcu.mk and Makefile. The defines in target.h determine the overall functionality and what is expected to be built. Each flight controller has its own file to specify which features are enabled/disabled only for it. Sometimes features may need to be disabled for space limitations, or limited computing capacity, or a bug, etc.

Each board’s target.h is located in target/[FLIGHT_CONTROLLER_NAME]/target.h. It gets loaded after target/common_pre.h. So any changes in this file will overwrite the default settings common among all targets found within common_pre.h (see the target/common_pre.h section below). This file is the place where you must touch to create your custom firmware.

In terms of development work, target.h is a good place for dummy data structs, variables, and instantiations that are not used or get replaced later by the custom drivers/api, but which need to be initially defined for building an executable.

c. Target.mk

Target.mk is a small makefile fragment found within src/main/target/<target board>. It's main purpose is to set several variables that are used in the mainline make process. Target.mk is most useful when you have many flavors from the same type of architecture, where the nuances matter. In our case there are no conflicts from other boards. Currently, the MAIXBIT's target.mk only has one line: RISCV_K210_TARGETS += $(TARGET). This line assigns the MAIXBIT target board to the RISCV_K210_TARGETS target group found in targets.mk.

4. Integrate SDK into Betaflight

The min. port executable must include architecture-specific files (drivers and board support package) as source files. These files will be included in the target's specific SDK.

These architecture-specific files should be integrated into Betaflight's file system prior to building. It's recommended to organize these files similar to their Betaflight counterparts (i.e. store the drivers next to the ARM drivers, linker file next to the ARM linker files, etc.). Modify any naming conflicts, such as platform.h in the Betaflight code and platform.h in the SDK.

Additionally, the K210 linker script and startup file must also be integrated into Betaflight's directory. We put the K210 linker script with the rest of the ARM scripts, in src/link. Similarly, we placed the K210's startup file with the other startup files in src/main/startup.

5. Build a simple example that will demonstrate the makefile works for the new target (Blink LED example)

The final step of the min. port is to build a "hello world" type executable. The idea is to create an executable that can demonstrate whether Betaflight's make process is functioning correctly. (I.e. does the product of the makefile actually work?)

For our RISC-V target, our team used an example program from Kendryte's SDK that blinked an LED on then off. We placed this simple demo program within Betaflight's main.c. Once we confirmed the LED blinked as it should, we knew that the make process was functioning correctly. Once we can confirm that the make process is working correctly for the new target, it's time to begin integrating pieces of Betaflight into the executable.

Step 3: Re - Integrating the most critical components back into the system

Before adding any APIs we first have to re-integrate some critical parts such as the configs, build version, etc. back in and inject small additions like #ifn defs.

During the min. port we commented mostly everything out in order to ensure that we can successfully build. In this step we will begin the process of re-integrating parts of Betaflight. This process will take some time and will continue further as we go. If we were to uncomment everything at once then we will get so many errors and warnings that it would be very hard to identify the most critical parts that need our attention.

Initially we will uncomment or add parts that are not doing anything useful but at least can get compiled successfully. Once we have files such as the configurations able to build we can start poking around and introduce our API code.

Integrating platform.h

Platform.h is located in src/main. The file is very important because it serves as a link between the platform-specific header files for each CPU and the rest of the system. The main modifications during the initial re-integration is adding our board in the preprocessor directives. It is critical having them there in order to ensure they are called during compile time instead of run time. What is more, at the end of the file we are defining and associating the board with the target_mcu.

Example:

```
#elif defined(MAIXBIT)

#include "riscv_k210_entry.h"
#include "riscv_k210_fpioa.h"
#include "riscv_k210_atomic.h"
#include "riscv_k210_bsp.h"
#include "riscv_k210_dump.h"
#include "riscv_k210_encoding.h"
#include "riscv_k210_interrupt.h"
#include "riscv_k210_platform.h"
#include "riscv_k210_printf.h"
#include "riscv_k210_sleep.h"
#include "riscv_k210_syscalls.h"
#include "riscv_k210_util.h"
```

```
#define U_ID_0 (*(uint32_t*)0x04E4796B)
#define U_ID_1 (*(uint32_t*)0x00000000)
#define U_ID_2 (*(uint32_t*)0x00000000)

#ifndef RISCV_K210
#define RISCV_K210
#endif
```

Uncommenting init and config files

Inside the main loop within main.c (located in src/main) the first function to get called is init. This call routes execution to init.c (located in src/main/init.c) where the board initialization process begins. For the sake of just getting an executable to compile, we started out with init.c completely commented. Gradually, we uncommented parts relevant to the initialization of our K210-based board.

Within init.c, the first section we uncommented was the section beginning with #ifdef CONFIG_IN_EXTERNAL_FLASH, located on line 408. Init.c is divided into sections based on the type of flash on the flight control board. Since our RISC-V K210 utilizes external flash, the CONFIG_IN_EXTERNAL_FLASH section is what's key to our init process. The code within this section takes the board through all its initialization steps.

To do this it calls several config files, where the init work actually takes place. These files are located in src/main/config. The most important for our port were: config.c, config_eeprom.c, and config_streamer.c. Most of the init process bounces around inside these three config files. A good resource for understanding the exact flow of the init process is the PG Config Flow document within our project wiki. Here, the process is explained in much greater detail, with each step listed.

Config.c

Config.c houses the first layer of configuration functions. The most important functions within this file are readEEPROM, writeUnmodifiedConfigToEEPROM, writeEEPROM, and resetEEPROM. These functions perform the important tasks of checking the validity of the PG config file within flash, and writing that into RAM. writeUnmodifiedConfigToEEPROM creates a new PG config file if one doesn't exist in flash or if it is corrupted.

Config_eeprom.c

Config_eeprom.c holds the second layer of configuration functions - functions that get called by the first layer functions within config.c. The most important of these second-level functions are the ‘streamer’ functions such as config_streamer_init, config_streamer_start, and config_streamer_write. The ‘streamer’ functions are what call the API functions which interact with the flash hardware.

Config_streamer.c

Lastly, config_streamer.c holds the function definitions of the streamer functions that get called within Config_eeprom.c. It’s within these definitions that we placed Kendryte API functions dealing with reading and writing to/from flash such as: flash_write_data, flashPageProgramFinish, flashEraseSector, and several others. Our team housed these Kendryte API functions within a define statement, #if defined(RISCV_K210), so that they can only be accessed by a K210 RISC-V target. Config_streamer.c is the deepest layer of Betaflight’s init process, and the functions here are what’s actually interacting with the RISC-V hardware.

To keep the length of this document reasonable, this has been a very high-level overview of Betaflight’s init process. It’s a difficult task to concisely sum up the process within one document (this probably explains the lack of Betaflight code documentation). We’ve tried to briefly explain what needs to be included to make the process function on a new type of hardware, and how we brought it all up. For any new dev teams continuing the RISC-V port, we highly suggest carefully tracing the init source code within the capstone-port branch while referring to the PG Config Flow document as a guide.

Step 4: Integrating APIs

After successful completion of the min. port Betaflight should successfully be able to create an executable for the new target. At this point, the only Betaflight source code included in the executable will be main.c (with the simple demo program inserted). Here is the point in the port where we start gradually introducing Betaflight source code into the executable, piece by piece. Starting at the empty main.c we began by tracing the execution of the board's initialization sequence through the source code.

PG config storage

The first thing Betaflight does after booting up the target board is to look for the PG configuration file in Flash. Each target board has its own PG configuration file, which stores the parameter groups (PG) for that specific board. PGs are a Betaflight construct that define certain properties of the flight controller. These include things like how many SPI instances the device uses, which SPI instances are being used, if it has blackbox, telemetry, external flash, etc. Upon boot-up Betaflight will copy this file from Flash into RAM. For more detail on the execution flow of the PG config storage process, see the PG Config Flow document in the Betaflight RISC-V wiki.

To access the Flash hardware, Betaflight will need to utilize certain parts of the manufacturer's API (Kendryte API, in our case). For more information about Kendryte's API visit https://github.com/kendryte/kendryte-standalone-sdk/tree/develop/lib. The API acts as a layer between Betaflight and the board's hardware, enabling Betaflight source code to talk to the hardware. This is where integration of the API into Betaflight begins.

SPI

For our board, the external flash uses SPI protocol and gets initialized using Kendryte's API called from within Betaflight. SPI can also be used for other sensors like IMU.

Flash

The Maixbit dev board has flash storage (16MB) which is used to store the PG configuration setting file. Similar to SPI, flash also gets initialized using Kendryte's API which gets initialized after SPI.

Define source and header files inside target/<dev_board>/target.h. Start by just compiling with empty main.c. This means that your source and header files are being found and no dependency issues have occurred. Then you can start with the first item required which is the configuration file which stores parameter groups (PG). For the K210, you will need to use the SPI and flash driver. Start tracing the initialization sequence execution through the source code and begin adding sections of the init sequence to the compiled binary. Start small and comment out all processes not needed for the PG config file (these can be reintroduced later).

Conclusion: Next Steps - recommendations

The next steps for the project depend on many factors.

The code introduced for the port should be cleaned up and abstracted in order not to drastically interfere with the already existing structure of Betaflight in case of a pull request. Hence, #ifdefs and #defines in some areas will have to go along with custom functions to K210's API.

What is more, a good implementation, and granted the large amount of RAM, would be creating a lookup table to translate some of the existing data structs from STM to RISCV. While it is more straightforward to use the API/Libraries provided from Kendryte, it is also a good way to introduce bugs and more complexity to the existing complexity of Betaflight.

Therefore, it is more valuable in the long run to create a stable link between the existing infrastructure of Betaflight and the K210. This will make adding more drivers easier, more straightforward and more compliant. Most of the code provided from kendryte is not suitable for production, either due to licensing or other potential risks.

However, if the long term portability into Betaflight is not a major concern, the next thing to be implemented is adding more drivers, while cleaning up the configurations and uncommenting more sections of the code. The UART is used in many peripherals. The successful addition/configuration of it can enable the board to communicate with the Betaflight configurator app. Hence, if the main goal of the project is proof of concept, this would be the next step that we recommend.

Appendix:

Helpful GNU toolchain documentation

A handy place to look for many things is here:

https://gcc.gnu.org/wiki/Building_Cross_Toolchains_with_gcc?action=AttachFile&do=get&target=billgatliff-toolchains.pdf

Betaflight RISC-V PG Config Flow document

https://github.com/GaloisInc/betaflight/wiki/PG-Config-Flow

Kendryte Standalone SDK

https://github.com/kendryte/kendryte-standalone-sdk/tree/develop/lib