Maseeh College of Engineering and Computer Science

PORTLAND STATE UNIVERSITY

Portland State University

Team 8

Final Report: RISC-V Betaflight Port

(Formerly RISC-V Cleanflight Autopilot)

Date: June 10, 2020

Brass, Bliss , Maldonado, Ruben

Nikolov, Nikolay, Schulte, Eric

brass@pdx.edu , mruben@pdx.edu

nnikolov@pdx.edu , eschulte@pdx.edu

Document Revisions:

Version	Date	Comment
0.5	05/20/2020	Initial draft
1.0	06/07/2020	Initial Release
1.1	06/08/2020	Updating main content
1.2	06/09/2020	Extending content
2.0	06/10/2020	Final review & submission

Table of Contents:

Abstract **5**

Introduction **5**
- Original project goals 6
- Updated project goals 6
- Short Betaflight summary 6
- Short RISC-V Summary 7
- Final Deliverables 7

Overview of Porting Process **8**
- Initial Design 8
- Adjustment and Implementation 8
  - Min Port 9
  - Adding Kendryte Examples 9
  - Port Betaflight initialization process using Kendryte API 10

Technical Details **11**
- Overview 11
- File editing 11
  - Makefile 11
- Creating the target 12
  - Target folder 12
    - Target.c 12
    - Target.h 12
    - Target.mk 12
    - RISCV_K210.mk (target MCU) 13
    - Targets.mk 13
    - Source.mk 14
    - target/common_pre.h 14
    - riscv_flash_k210.ld 14
    - startup_riscv_k210.s 15
    - RISCV_K210 API from Kendryte 15
    - BSP (Board Support Package) 15
    - Drivers 15

Testing **16**
- Integration testing 17
- Unit testing 17
- Functional Testing 18

Challenges **18**

TEAM Lessons **23**
- Communicating effectively 23

Understanding team dynamics 23
Defining goals 23
Time Management 24

Project Management 24
Daily Standup Meetings 24
Weekly Sprint Planning (Monday) 25
Weekly Team Retrospective (Friday) 25
Frequent Updates 25
Pair Programming 26
Trello Cards 26
Scrum Master 26
Weekly Reports 26
Sprint Planning Workflow 27
Definition of Done 27
Velocity Capacity 28
Frequent 1-on-1 with Capstone Advisor 28
Summary of Project Management 28

Conclusion 30

References: 31

Appendix 32
Sources: 32
Primary responsibilities per member 33
How to build and use 34

Abstract

This document is about the process and difficulties we, capstone team 8, went through for porting Cleanflight to a RISC-V board. It explains our initial struggles and why we had to switch to Betaflight and a new RISC-V board. It also lays out the steps we took to achieve the porting of Betaflight's low-level configuration process. It also covers the basics of various tools and equipment that were used.

Note: *Throughout this document MAiX BiT and Kendryte K210 are used interchangeably. MAiX BiT is the development board and Kendryte K210 is the RISC-V SoC on the board.*

Introduction

At first, we would like to thank our Capstone sponsor Galois for giving us the opportunity to attempt the project. In addition, we would like to thank our Capstone advisor Roy Kravitz for the continuous support for the past two months.

Furthermore, in the following sections we will discuss in detail the timeline of the project, by examining how it started and what was delivered. The report begins with briefly discussing the original goals of the project and how they changed. Additionally, we will attempt to explain what were the main challenges along the way, and key learnings and takeaways. We will pursue a high level overview of the processes that we followed in order to port the initialization process using the K210 MCU using the standard libraries provided by Canaan (the manufacturer of the mcu). We will further talk about the project management style changes we made during the second half of the project and what that entailed to us.

Last but not least, we will touch upon Testing and verification. At the end we have a brief list with References that we found to be very helpful not only during the development of the project, as well as outside the project.

Betaflight is a fundamental example that captures many aspects of writing firmware using C. The lessons on compiling, linking, writing good and well-documented code along with portability are prime examples of importance and what we learned during this project.

Original project goals

The original goal was to port the open-source Cleanflight racing drone control software to a SiFive HiFive1 Rev-B RISC-V architecture board. After completing the port, we were to assemble a drone and fly it using the RISC-V ported software on the RISC-V board. We were to also create a Github merge request to merge our ported code with the upstream master branch.

Updated project goals

The original goals were revised 2.5 months in when we switched from Cleanflight to Betaflight, and from SiFive's HiFive1 Rev-B to Sipeed's Maixbit. As we learned from the Betaflight devs, Cleanflight is now outdated and obsolete. We switched dev boards because of a lack of RAM space on the HiFive1 Rev-B. More details on our hardware and software switch can be found in the Challenges section.

After porting Betaflight's Make process, we began digging into the C source code. It quickly became evident that Betaflight was a much larger, much more complex code base than originally anticipated. This, coupled with the progress lost by switching hardware and software platforms, forced us to reconsider the final deliverables. We ultimately settled on delivering a functional RISC-V port of Betaflight's configuration process.

Short Betaflight summary

Betaflight is an open-source racing drone operating system (OS) written in C. Betaflight is a fork project of Cleanflight. Betaflight is among the top-3 drone flight control softwares in use for competitive racing and enthusiasts at the date of this document. It offers a plethora of controls and options to tweak a racing drone to maximize performance. It's also fairly user-friendly with its Chrome app web-based interface that allows users to simply plug in a supported board and immediately get things like telemetry data and individual motor control in real-time for testing.

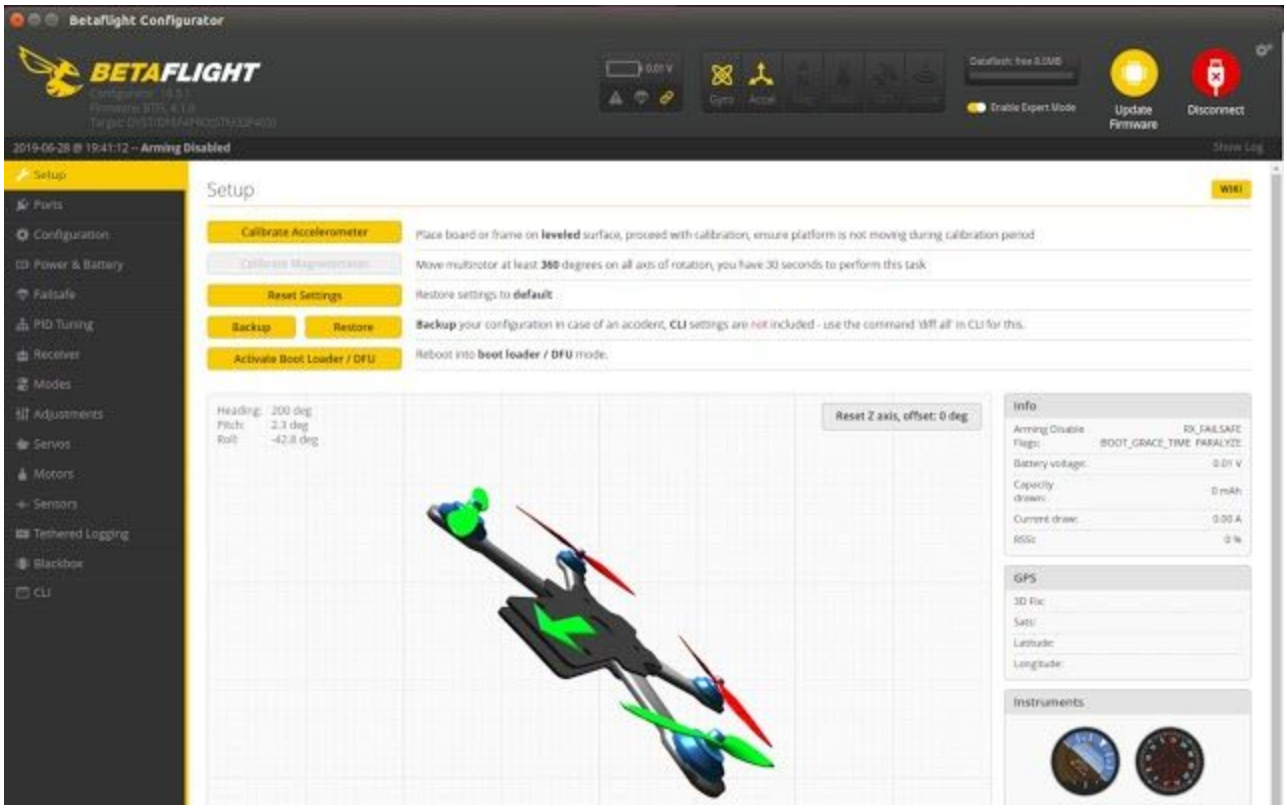

Figure 1. Betaflight configurator main screen after connecting a flight controller.

Short RISC-V Summary

RISC-V is the open-source 5th iteration of the Reduced Instruction Set Computer assembly code. Though not as diametrically opposed to the CISC (Complex Instruction Set Computer) machines of yore as both styles have at times made an exception to borrow from one another, it still remained fundamentally a more simplified way of generating machine code, with the idea being that more, but simpler instructions can execute more efficiently than longer, complex instructions. The 5th iteration promises even more improvements in both options and efficiency.

Final Deliverables

- Port of Betaflight's Make process to RISC-V
- Port of Betaflight's initialization process to RISC-V
- Test plan and report
- Final project report
- Porting guide
- Betaflight RISC-V wiki

Overview of Porting Process

Initial Design

We, the team, met early and with each other and then the sponsor back in December 2019 to try and gauge the scope and desired outcome of the project. We initially planned to do a “min port” as we call it which got the Make code side of Cleanflight ported and ready for use on the new RISC-V board.

About a month and a half into this process we decided to purchase and build the drone to have it ready for live testing once we got to the C code drivers that came after the min. port. After significant setbacks and finding out from the maintainers of Cleanflight/Betaflight that our board lacked the hardware capabilities (insufficient RAM) to run the program, we decided on a new target based off the suggestion of the maintainer Jflyper (as they are known on Betaflight community Slack) to be the SiPeed MAiX BiT using the Kendryte K210 RISC-V microprocessor.

We also revised our project schedule as it became clear the significance of the Make code portion of the project was vastly unrealized. This was almost a restart of the project.

Adjustment and Implementation

We took what was transferable from the Cleanflight Make hierarchy and applied it to the Betaflight Make hierarchy to cut down on the bring-up of what we still called the min. port. We decided to switch to Trello and began meeting 5 days a week utilizing SCRUM. With more depth of knowledge on the Make code side and with the aid of Jflyper we broke down the flashing process and stages required to port a very basic version of Betaflight onto a new architecture.

Min Port

The ‘minimum port’ (shortened to min. port) was the team’s first step in porting to the RISC-V hardware. The term ‘min. port’ refers to the idea of creating a simple binary using the project structure without calling any Betaflight functions. This simple binary would act as a test; a way to see if we could get something to build for the new RISC-V architecture using Betaflight’s make process. The basic steps included in the min port are listed below. For more information on any of these files refer to the Modified Files section.

1. Modify Makefile
2. Create target folder for Maixbit:
    - Target.c
    - Target.h
    - Target.mk
3. Create RISCV_K210.mk (MCU makefile for K210)
4. Alter Targets.mk
5. Add Kendryte API to Betaflight directory
    - Board Support Package (BSP)
    - Drivers
    - Linker script (riscv_flash_k210.ld)
    - Startup file (startup_riscv_k210.s)

Adding Kendryte Examples

Once we verified that the make process worked for the RISC-V hardware (via the min. port), we needed a way to test the functionality of the executable. To do this we inserted a basic blink LED example from Kendryte’s SDK into the empty main.c of Betaflight. After rebuilding the executable with the Kendryte example included, we were able to demonstrate the functionality of a RISC-V executable created by Betaflight’s make process. This was an important step; it was the first time our team saw something that was produced by Betaflight functioning on the RISC-V hardware.

Port Betaflight initialization process using Kendryte API

After proving that our executable was working on the RISC-V hardware, our team began porting our first chunk of Betaflight code - the initialization process. The initialization process involves reading and writing configuration data to and from flash storage. The Kendryte K210 processor utilizes an external flash which communicates via the SPI protocol.

The first step in porting the initialization process was enabling SPI on the K210. This was accomplished by invoking a Kendryte API function from Betaflight's init.c. With SPI initialized, the next step was to initialize the external flash device (flash chip is on the Maixbit dev board). Similar to SPI, this was done by invoking an API function from init.c.

With SPI and flash up and running, we began porting Betaflight's PG config storage process. PG stands for 'Parameter Groups', these are basically groups of features which get either enabled or disabled depending on the type of flight controller. The PGs determine the ultimate functionality of the board, features like black box telemetry, GPS, Barometers all get enabled or disabled via the PGs. The PG config storage process is a somewhat complex procedure that involves copying these PGs from flash memory to RAM at board boot time. Without getting into too much detail here, we ported this process to the K210 by inserting API functions at some of the lowest levels of Betaflight.

The above firmware allows Betaflight to read and write to the K210's external flash and RAM, enabling the PG configs to be transferred.

Technical Details

Overview

Betaflight is written in C with a massive web of Make calls and flags. You need to understand the intricacies of creating a project for an embedded system and what compilers and tools you need to compile a program for a different target than the system it is written on.

File editing

Makefile

Betaflight's top-level Makefile is used to build executable files for the dozens of supported flight controller models. Betaflight employs a tree-like hierarchical structure during the make process. During the make process, execution is routed through several makefile fragments - other .mk files used to accomplish specific tasks during the make process. These files are called from this top-level makefile. The top-level makefile is responsible for the overall execution flow of the make process - everything starts and ends here.

The Makefile is quite large, and our group spent a lot of time at the beginning of the project figuring out how it works. The Makefile includes GNU tool names and locations, compiler, linker, and assembler flags, and the recipes and rules used to create executables for the dozens of supported boards. It's important to note: not everything in the Makefile is crucial for building an executable - there's a lot of extra functionality present. So it's not critical to understand everything within the Makefile.

The Makefile served as our team's entry point into understanding the Betaflight code base. It is quite useful to thoroughly understand it, as it can lead to insights about the overall structure of the code base.

Our team didn't add or change much within the Makefile (since most of its functionality is split out across several Makefiles), but we did include the RISC-V GNU tool suite necessary for building RISC-V executables. Up until now, Betaflight has served only ARM STM chips, so only the ARM GNU toolchain was needed. To get the Makefile to build something for our board, we needed to include our own toolchain. We did not modify any executable rules or recipes. Since it acts as the central hub of the make process (everybody uses it), we tried to keep our footprint as small as possible in the top-level makefile.

Creating the target

Target folder

Each flight control board supported by Betaflight has its own target folder (located within src/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:

1. **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.

2. **Target.h**

This file is one of the most critical during the Make process, along with target_mcu.mk and Makefile. The defines in this file 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 flight controller 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 src/main/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 is the file you must edit to create your custom Betaflight 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.

3. **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.

RISCV_K210.mk (target MCU)

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 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.

Our team created a new MCU makefile for our Maixbit target and K210 microprocessor. This file is similar to the other MCU makefiles in make/mcu folder, but it points to RISC-V resources instead of ARM. We've added a few custom variables which point to peripheral header files, and also added a variable to create a .map during the make process. We've intentionally kept the Maixbit makefile as similar to the others as possible to conform with Betaflight's make process.

Targets.mk

Targets.mk is a small makefile fragment 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.

TARGET_MCU is used at the bottom of the file to assign microprocessor-specific compiler flags. For example, if the target board has an STM32F4, one set of compiler flags will be applied; if the board has an STM32F7, a different set of compiler flags gets applied.

Source.mk

Source.mk is a makefile fragment found within the top-level make folder. It's purpose is to collect and organize all the C source files needed for Betaflight to compile for a given target. The broad strategy of source.mk is to collect large groups of source files into makefile variables. These variables are then accessed by the top-level makefile for compilation. The source files are generally organized into variables by the type of optimization needed.

To optimize program execution, Betaflight will optimize certain source files for speed, and other files for size. For example, the variables SPEED_OPTIMISED_SRC and SIZE_OPTIMISED_SRC contain source files that require speed and size optimization, respectively. The COMMON_SRC variable contains all source files that require no optimization. Source files are differentiated by optimization type so that the compiler knows what to do with each file. It's important to note too, source.mk receives several source file variables from other locations. For example, the variable MCU_COMMON_SRC gets populated in the MCU.mk file, and TARGET_SRC gets populated in the MCU makefile. External source file variables get combined into one greater variable, SRC, towards the bottom of source.mk.

target/common_pre.h

This is the first file where developers can specify which features are activated for a flight controller. Common_pre.h specifies the features enabled depending on the memory flash size of the flight controller, and several other conditions. Features are enabled by defining certain flags (macros). These flags are then checked throughout the Betaflight code base to control the flow of program execution.

As per the instruction of Betaflight devs, this file should not be modified. But it's useful to see what features you can activate or deactivate in your custom build.
However, there is a file where developers may enable more flags or disable flags enabled in common_pre.h. This can be done within the target's unique target.h file, located in src/main/target/<Target_Flight_Controller>/target.h (see the Target.h section above).

riscv_flash_k210.ld

This is the linker script for the K210 target, located in src/link. This file describes the memory map. The RAM, ROM, stack addresses and their lengths are described in this file. The different sections like .text, .bss, .data, pg_registry are also described in this file. The external flash storage can also be described.

startup_riscv_k210.s

This file contains assembly language instructions for booting the K210 processor (located in src/main/startup/startup_riscv_k210.s). Nothing inside this file needs to be changed, but it does need to be included in the make process. The startup file (for each board) gets included in the make process via the MCU makefile (in our case, RISCV_K210.mk).

RISCV_K210 API from Kendryte

The Kendryte API (application programming interface) comes with several folders. The two most important folders are 'bsp' (board support package) and 'drivers'. Both of these folders must be included in the make process (via the MCU makefile, RISCV_K210.mk), but it shouldn't be necessary to alter the contents of either folder. We placed Kendryte's APU within the Betaflight directory here: lib/main/RISCV_K210.

BSP (Board Support Package)

The bsp folder contains K210 specific drivers and other subroutines which allow outside operating systems (Betaflight) to function in the K210 hardware environment.

Drivers

This folder contains the software needed to use the K210's various peripherals (GPIO, UART, PWM, timers, etc.). The files within this folder contain chunks of Kendryte's API that must be inserted into Betaflight, at the low-level, to make Betaflight run on the new RISC-V hardware.

Initialization and Flash

Our first approach to integrate K210's API into Betaflight code was to start with the init.c source file located at ./src/main/fc/. This file initializes all the features (like telemetry, GPS, blackbox, etc), parameter groups, motors, etc. The section of code that we ported was

code that handled the parameter groups. We made edits to source files that managed reading and writing to persistent memory like SD card or external flash. Betaflight labels this persistent memory as ‘EEPROM’. The remaining source files are config.c, config_eeprom.c, config_streamer.c, config_reset.h, and are located at ./src/main/config/. The source file that we used for interfacing with the onboard flash chip is flash_riscv_k210.c located at ./src/main/drivers/. We also created a parameter group for our k210 flash, source file riscv_k210.c located at ./src/main/pg/.

Testing

The overall goal of the test plan is to ensure Betaflight's initial boot sequence works on a RISC-V K210 target. If successful, the sequence should execute just as it would on any of Betaflight's existing ARM targets. This time though, it will be utilizing a new API specific to Kendryte's K210 RISC-V architecture. But the end result (an initialized target board) should be the same.

Integration testing

To test both branches of the initialization process, our team inserted conditional print statements. These print statements act as integration tests by checking the returns of key functions along both branches. Each print statement will display a message indicating whether each function passed or failed. If program execution fails at any point along either branch, it's easy to pinpoint exactly where it failed. If the boot sequence is successful through both branches, all of our expected test prints will execute successfully. Our team used a text-based serial port communications program called Minicom to interface with the board during unit and integration testing. For more information about Minicom see the How to Build and Use section at the end of the appendix.

Unit testing

Our team also included unit tests to check the behavior of individual functions. These tests involved examining function outputs given certain input (or in most cases just a function call, since many Betaflight functions don't take inputs).

To assist in conducting unit tests, our team used a framework called Unity. Unity is an open-source unit test framework available as a Github repo. Unity offers a number of assertion functions that can be used to test a function's behavior. Assertions are statements of what is expected to be true about the source code being tested. The most useful Unity assertion for us was TEST_ASSERT_EQUAL, which compares function returns against an expected value. Betaflight source code functions are plugged into Unity assertions, run on the hardware, and then compared against expected values.

The Unity framework uses its own makefile to create a test executable that runs on hardware. The test results appear in the terminal, and can be observed using a text-based serial port communications program (Minicom).

We created a folder within our repo called unit_testing_k210 to house the Unity source code, and our test scripts. We also developed a shell script to automate the unit testing, called test_runner.sh. This script checks for the Unity source code within the repo (if it's not there, it clones it), creates the test executable, flashes the executable to the board, and runs the executable through Minicom.

Functional Testing

Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (unlike white-box testing).

Functional testing is conducted to evaluate the compliance of a system or component with specified functional requirements. Functional testing usually describes what the system does.

Challenges

Switch to Betaflight

Throughout the course of the project, our team encountered several technical challenges. The first challenge was having to switch software from Cleanflight to Betaflight in early March. The original project proposal from our sponsor, Galois Inc., requested that we port Cleanflight to RISC-V. Cleanflight is essentially the precursor to Betaflight; it is one big branch off of Cleanflight.

After getting in touch with several Betaflight developers, it quickly became clear that Cleanflight is an outdated software. All development on Cleanflight has ceased, and there haven't been any updates to the project repo in about 2 years.

So here we faced a dilemma: Do we continue working on a port of an obsolete software, which most likely won't be of any use to anyone outside our capstone project; or do we switch to the newest version of the software, with more community support and relevance (but also lose weeks of technical progress)?

After discussing the two options with our project sponsors and Betaflight devs, we decided to switch to porting Betaflight. The biggest reasons behind this decision were 1) increased community support - there are currently dozens of devs working on Betaflight, whom could field our questions, and 2) Our work on the project would remain relevant after the end of spring term - devs can pick up our branch and run with it, unlike if we had stuck with Cleanflight.

During our discussions, the Betaflight devs expressed their prior interest in RISC-V hardware, and the fact that they themselves were thinking of eventually porting Betaflight over. So, even though it was a very hard decision, and we lost weeks of technical progress, we believe we made the correct call for the interests of all parties involved.

Switch to Kendryte K210

Shortly after switching to Betaflight, our team learned about another roadblock - our hardware. In the original project description our project sponsors asked us to do the port using the SiFive's HiFive1 Rev. B. The HiFive1 Rev. B is similar to a RISC-V equivalent of an Arduino - a versatile yet low-power dev board. About halfway through March a

Betaflight developer told us the HiFive1 Rev. B likely would not be up to the task of running Betaflight. Specifically, the bottleneck was the RAM. Current implementations of Betaflight can require up to 80kB of RAM, and even stripped-down versions running minimal features require at least 50kB. Our HiFive1 board offered only 16kB of RAM. Clearly, it would not be up to the task.

From here we had a couple options. We could (again) switch software platforms, to one with a much smaller RAM requirement. We could try creating the smallest, most stripped-down version of Betaflight to try and squeeze into 16kB. Or we could switch hardware, and scale up to a more powerful dev board.

The decision really came down to the question of which alternative would cost us the least time and effort - switching hardware or software? With new hardware we'd have to get up to speed on a new SDK and IDE (since we were using SiFive's proprietary IDE Freedom Studio); but with new software we'd be dealing with a completely new source code base (Cleanflight and Betaflight share the same structure).

We ultimately decided switching hardware would cause the least amount of setback. Based on the recommendation of several Betaflight devs we switched to Sipeed's Maixbit dev board. The Maixbit uses Kendryte's K210 RISC-V processor, which has 6MB of RAM. Again, this wasn't an easy decision to make. But we feel confident we made the right decision, and selected the right hardware for the project. The K210 is large and powerful enough to accommodate years of future development on the RISC-V port of Betaflight.

STM nature of Betaflight

Our team faced a few challenges from Betaflight itself while porting to RISC-V. The first challenge was Betaflight's ARM dependencies. From the beginning Betaflight was designed around ARM STM hardware. The first iterations of Betaflight ran on STMF1 and F3 chips, now they're up to F7s and H7s. And while the devs have made it easy to port to new STM chips, Betaflight has never been ported outside the STM family.

This is most apparent at the lowest level of Betaflight, where Betaflight and peripheral drivers meet. A lot of this low-level code uses data types and structs that get defined within STM peripheral driver files. So in a sense, low-level Betaflight is "infused" with STM-specific code. This makes it difficult to use existing Betaflight functions for

low-level tasks like reading or writing to flash. So when porting Betaflight to a new architecture, a fair bit of rewriting needs to take place, which is time-consuming.

Betaflight's layered structure

Another challenge our group faced was Betaflight's complex layering (the term 'spaghetti code' comes to mind). Betaflight isn't the best-structured code base; this is probably a consequence of years of constant changes and upgrades from dozens of devs. Throughout the initialization procedure, which we focused primarily on, it's common to see functions calling 3-4 layers of subroutines. To compound the challenge, comments are nearly non-existent throughout the source code. And there is hardly any wiki/markdown documentation for what goes on under the hood. New devs are forced to closely examine the code to understand what it's doing. This is a problem our team tried to address by creating a project wiki, meant to be a reference for anyone who attempts to continue the project.

Team's mutual lack of porting experience

All of the team members were pretty green to the process of porting software when the project began. One of the most important early steps was defining the philosophy of the port. We eventually decided upon injecting parts of the Si-Five API (what would eventually be Kendryte's API) into the lowest levels of Betaflight. This is basically how Betaflight is operating with the existing ARM STM boards. This process was hard to visualize at first, before we started working on the C source code, but became much more clear once we began understanding how Betaflight utilizes API functions.

Additionally, most team members lacked several basic requisite skills needed to develop Betaflight. The most apparent example of this was the Make language, and overall make process - only one team member had prior make experience. So we all had to get caught up with the make language in parallel with the actual porting work. In fact, we were all continually learning new skills, requisite to the porting work, along the way.

This included learning and using the syntax of linker scripts, and finding and using programs needed to accomplish tasks like flashing, and displaying terminal output (Kflash and Minicom, respectively). It's never a bad thing to learn new skills and languages, but having to do so during the course of the project definitely had an impact on the development schedule.

Soft challenges

Our group also faced a number of ‘soft’ challenges - non-technical challenges which nonetheless affected the project outcome.

The most obvious of these was COVID-19. Being that our project was software-oriented, we expected the pandemic to not have much of an effect - but this was far from the case. The pandemic has basically caused a lot of ‘little’ inconveniences; things like not being able to meet face-to-face, staying home 24/7 (in some cases with children or other roommates), and the challenges of remote learning for other courses. All of these added up over the course of the last two months, and have had more of an impact on the final outcome than any of us probably expected.

Two teammates have also been holding down full-time engineering jobs throughout the course of the project. They both did a tremendous job of contributing to the project while balancing their 40-hour schedules - nice job Niko and Ruben! And those of us not currently working have also been busy, trying to find employment after graduation. These are not meant to be excuses, but it’s important to take into account the effect these “soft” challenges had on the project.

TEAM Lessons

Communicating effectively

Communication plays a critical role for any project. Throughout the project the team experimented with different methods of communication (Slack, Email, Hangouts). The communication styles among the team members varied.

An important lesson was understanding how to communicate deliverables, performance, and sensitive personal issues, while maintaining the integrity of the team and making everyone feel included. In addition, it was critical to learn how to deliver feedback and to recognize topics that are sensitive to other members of the team.

Overall, effective communication begins and ends with listening and trying to understand.

Understanding team dynamics

A critical lesson throughout the project was the team dynamics. All four team members had different lives and schedules - some were working full time, others were looking for jobs, and some had families. These different schedules and outside demands meant that every team member approached the project differently - some worked on it for a few hours everyday, while others had to squeeze project work into a weekend, or during late nights.

Here is where the Agile Method and daily standup meetings (detailed in the next section) really helped team organization. No matter when team members worked on the project, the daily meetings kept everyone informed of each member's progress and current tasks.

Defining goals

Initially we defined project goals and their timelines using the waterfall method and a Gantt chart. While this provided a nice overview of the entire project schedule, the goals were too broad, and were not defined well enough. As part of our team's switch to Agile we began holding weekly Sprint Planning meetings on Mondays where we outlined the next week's goals in Trello. These meetings helped us define more specific goals that were much more manageable. For a project like porting Betaflight, where the next steps

usually aren't entirely obvious, the Agile methodology is much more effective than creating broad 6-month long Gannt charts.

Time Management

Six months isn't much time to port a project like Betaflight to a new architecture, so time management was always critical to our project. During the project's first term, winter quarter, we were not as conscious about time management as we should have been.

The team spun its wheels a lot, and had trouble making progress. This changed during spring term, when we started doing bi-weekly (or so) peer programming sessions. During these meetings, one person would drive and demonstrate to the group what they'd been working on the past few days. A lot of brainstorming also happened during these meetings, allowing team members to springboard off other's work. After starting peer programming, development progress on the port rapidly began increasing.

Project Management

The team initially followed a waterfall style of Project Management. We created a Gantt Chart and we estimated how much time each task would require to be completed. We had 2 hour meetings twice a week.

This particular project management style did not work well for our team and particular software project. It was not flexible enough to accommodate the learning curve of doing something we had never done before. What is more, it led to unrealistic expectations and it was making it hard to adapt to the challenges and changes.

For the second part of the project we switched to Agile. The major changes are detailed below.

Daily Standup Meetings

The first change we did was to increase the frequency of our meetings. We had daily 15-30 minute meeting sessions. Those were our Stand Up meetings, in the definition of Scrum. During these meetings the team members explained what their tasks were, blocks and major short-term goals in the project. It was a good time to exchange information, ask questions and share know-how.

Weekly Sprint Planning (Monday)

In the spirit of Scrum we had weekly sprints. Generally, sprints are two weeks, however we found it to be more effective having a weekly Sprint Planning. During the Sprint Planning the team created user stories or pulled user stories from the Backlog. In accordance with the project progress during that week and new-found information, the team took a direction and established critical goals for what needed to be achieved during the upcoming week.

Weekly Team Retrospective (Friday)

The team held Team Retrospectives at the end of each week. During the Team Retrospectives each team member had to describe what went well during the week, what needed to be improved and a general assessment of the work performed. During the Retrospectives the team would decide major tasks that need to be scheduled during the Sprint Planning.

It is important to emphasize the aggressive schedule the team had in place. The main goal was keeping a strong feedback loop between the project and the team. Hence, a two week sprint would not have been as desirable.

Frequent Updates

Communication became an integral part of the project management. The team held meetings for code reviews, exchanged information in Slack, and maintained a strong link that enhanced the feedback loop between the project and the team.

The team also had performance surveys that were intended to help the team voice concerns over performance, or topics that were hard to discuss in a team setting. This allowed our team members to voice opinion and feedback to the team. There were also weekly progress updates delivered to our sponsor and advisor. The weekly updates included a high-level summary of what each team member had accomplished throughout the week. The main sections were `TODO` , `DONE`, `DOING`.

Pair Programming

The frequent exchange of know-how, the continuous feedback loop and the improved communication were solidified with pair programmings. During that time the team was able to make significant breakthroughs. Instead of each team member working isolated and on a separate piece of code, one person drove the coding part and the rest of the team was helping debugging and thinking out loud. It is often where the team gathered better understanding of the project, established team morale, helped people get on the same page and developed the dynamic needed to continue improving.

Trello Cards

The team used Trello as the software of choice for the project management. We tried experimenting with other project management tools, however most were too complicated and had a steep learning curve. We used Trello only once a week during our spring planning. We had four sections, Backlog, Todo, In-progress, and Done. The Backlog has the user stories/cards (major tasks) that we had defined. We continued adding to our Backlog and during the Sprint Planning we picked the user stories/cards that were relevant to the most immediate goals.

Scrum Master

The team had a scrum master. Ruben had the role of the Scrum master. The Scrum Master is responsible for promoting and supporting Scrum. Scrum Masters do this by helping everyone understand Scrum theory, practices, rules, and values.

The Scrum Master was a servant-leader for the Scrum Team. The Scrum Master helped those outside the Scrum Team understand which of their interactions with the Scrum Team are helpful and which aren't.

The Scrum Master helped everyone change these interactions to maximize the value created by the Scrum Team. ["What Is a Scrum Master?" *Scrum.org*]

Weekly Reports

The team shared weekly reports with the capstone sponsor and with the capstone advisor. This was part of the continuous feedback loop; keeping the information flowing

within the team and keeping the customer and other stakeholders updated on the project progress.

Sprint Planning Workflow

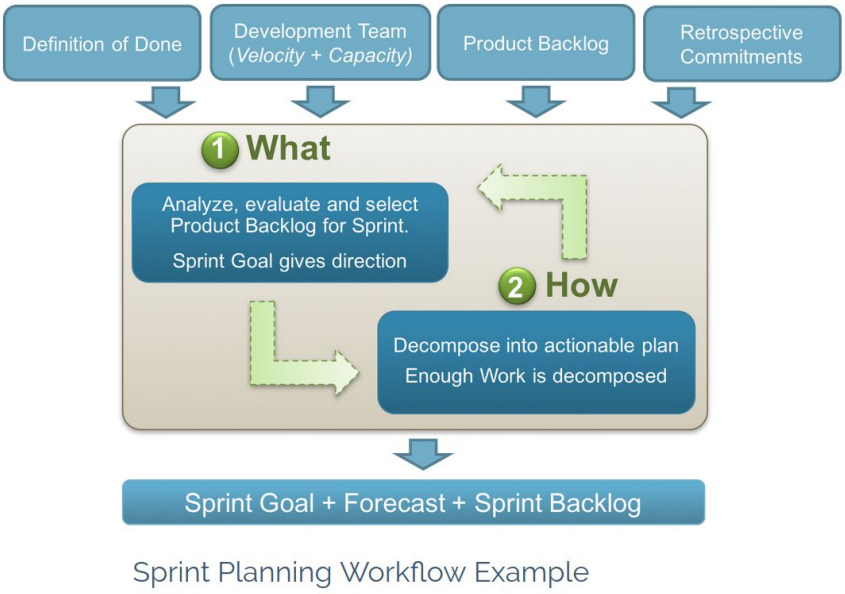

Figure 1: Sprint Planning Workflow

Above is an example of how the team utilized scrum.

Definition of Done

During our sprint planning we created user stories that had a specific definition of done (DoD). Below is an informative part that describes the DoD.

- The Definition of Done provides a checklist which usefully guides pre-implementation activities: discussion, estimation, and design.
- The Definition of Done limits the cost of rework once a feature has been accepted as “done”. Having an explicit contract limits the risk of misunderstanding and conflict between the development team and the customer or product owner.

Pitfalls:

- Obsessing over the list of criteria can be counter-productive; the list needs to define the minimum work generally required to get a product increment to the “done” state.

- Individual features or user stories may have specific “done” criteria in addition to the ones that apply to work in general.

- If the definition of done is merely a shared understanding, rather than spelled out and displayed on a wall, it may lose much of its effectiveness; a good part of its value lies in being an explicit contract known to all members of the team.

The above was borrowed from “Definition of Done.” Agile Alliance, 24 Sept. 2019.

Velocity Capacity

During the sprint planning we evaluated the time each team member can contribute for the project.

In general, the velocity in our project was typically an average of all previous sprints. Velocity was used to plan how many product backlog items the team should bring into the next sprint.

In addition, capacity is how much availability the team has for the sprint. This varied between team members due to exams, homeworks, etc. The team considered capacity in determining how many product backlog items to plan for a sprint.

Frequent 1-on-1 with Capstone Advisor

During the project the team frequently met with the capstone advisor to discuss important topics related to technical issues encountered during the project, along with personal issues and team dynamics.

Summary of Project Management

Switching from Waterfall to Agile had a large impact on the team’s performance. It helped team members to exchange information and collaborate more effectively. Furthermore, it helped the team avoid getting stuck for too long on parts of the project that could have been broken on smaller parts. It improved morale and created a more collaborative atmosphere in the team. In some occasions, during moments of disagreement the fact that the members had to continue communicating enforced them to iron our issues, which during the Waterfall would not have been addressed.

Last but not least, it promoted transparency among the team and gave an opportunity to everyone to voice their ideas and opinions.

Conclusion

The project served as a great learning experience for all members. On the technical side, we got exposed to myriads of new concepts. We had to revisit topics such as linking, compiling, C syntax, Make, ISA, device drivers and serial ports that we had previously learned in ECE 371-373. In addition, we got experience on how to navigate inside a larger set of code, and use Git more effectively.

On the soft skills side, we learned how to communicate better, adapt to personal attributes and deal with challenges related to personal life and emergencies.

In the project management area, we used and learned different modes of operation. The first one, Waterfall, was more static and with strictly defined roles, responsibilities and timeline. The meetings were long, and relatively infrequent. The team rarely paired or performed any tasks together. It was hard to track what each person did and it was hard to enforce any kind of peer performance evaluation. What is more, even if the team was heading the wrong way, we had to stick to the pre-charted plan and continue chugging. Oftentimes this rigidity and static planning can be helpful for projects that have been previously completed successfully, and there is an established process. Although, for us this method was not effective.

On the other hand, during the second-half of the project we adopted Agile, a more plastic and flexible approach to how we established our goals and deliverables. It was not tied to a long term process. Rather, we continually asked ourselves what is the next best thing we can do in the following week. This is how long our goals spanned. One week. Along with adopting a more flexible perspective we increased the frequency of our meetings.

Yet, an approach like this might be scattered for projects that need rigidity. However, we predict that in the future most projects will have to adapt to an ever-changing and fast-paced environment, and being held rigidly to a set course of action will become increasingly inefficient. As Charles Darwin observed, it is not the fittest that survive, but the most adaptable. In actuality this is what he was describing. Only species capable of adapting to their environment can survive. Physical fitness is just an attribute related to adaptability.

Therefore, as we are looking back on the project, what is most rewarding is the fact that we did not give up; rather we adapted to our environment and we were able to continue. And this is the success of the project. Not the porting. The porting is merely a wish casted upon us.

References:

- *Barr, Michael, and Anthony Massa. "Programming Embedded Systems, 2nd Edition." O'Reilly Online Learning, O'Reilly Media, Inc., www.oreilly.com/library/view/programming-embedded-systems/0596009836/ch04.html.*
- *"What Is a Scrum Master?" Scrum.org, www.scrum.org/resources/what-is-a-scrum-master.*
- *"Definition of Done." Agile Alliance, 24 Sept. 2019, www.agilealliance.org/glossary/definition-of-done*
- *Barr, Michael, and Anthony Massa. Programming Embedded Systems with C and GNU Development Tools. MTM, 2013.*
- *Grenning, James W. Test-Driven Development for Embedded C. The Pragmatic Bookshelf, 2014.*
- *Harwani, B. M. C Programming Cookbook: over 40 Recipes Exploring Data Structures, Pointers, Interprocess Communication, and Database in C. Packt Publishing, 2019*
- *Patterson, David A., and John L. Hennessy. Computer Organization and Design: the Hardware/Software Interface. W. Ross MacDonald School Resource Services Library, 2019.*
- *Loukides, Michael Kosta., and Andrew Oram. Programming with GNU Software. O'Reilly & Associates, 1997.*
- *Andersen, Paul K., et al. Essential C: an Introduction for Scientists and Engineers. Saunders College, 1995.*

Other sources:

- Project Wiki : https://github.com/GaloisInc/betaflight/wiki
- Capstone How-To Port: https://github.com/GaloisInc/betaflight/blob/capstone-port/docs/Capstone%20How-To%20Port

Appendix

Primary responsibilities per member

- **Bliss**
  - Wiki contributor
  - Testing and Verification (hardware)
  - Weekly Reports
  - Built, tested and flew the drone
  - Wrote Kflash and Minicom documentation
  - Team moderator
- **Niko**
  - Contributor:
    - Makefile(s), Linker file, SDK
    - Flash, SPI, startup file
    - Test Suite with Unity
    - Wiki
  - Code Review, Testing
  - Project Management
  - JFlyper best friend forever
- **Eric**
  - Helped develop min. port process
  - Project management and Planning
  - Ported initial config storage steps
  - Unit Testing and Test Planning
    - Integrating Unity (helped with Makefile and created runner script)
  - Wiki documentation
- **Ruben**
  - Created initial min port process
  - Code changes to Betaflight
  - Researched I2C - SiFive API
  - Configuration settings
  - Integrated Flash and SPI
  - Script to automate compiling and flashing of firmware

How to build and use

- **Make** https://github.com/GaloisInc/betaflight/wiki/Makefle
- **Flashing to the board**
  https://github.com/GaloisInc/betaflight/wiki/Building-a-Demo-Binary-and-Flashing-K210-Using-the-Kflash-Utility
- **Minicom**
  https://github.com/GaloisInc/betaflight/wiki/Debugging-Serial-Output-with-Minicom
- **Runner**
  To automate the make, flashing to board, and running minicom, we created a bash script. You can run runner.sh located in ./k210_tools/. You can run the script from ./k210_tools/ but if you run it from anywhere else then just update the script with the paths.

  The script will pull the required tools like RISC-V toolchain, kflash, minicom, unity if they are missing on your machine. After compiling the binary and flashing the binary onto the board, the script will open a separate terminal and run the binary using minicom. The script uses ttyUSB0 as the default USB port but you can update if needed. If no binary compiled, the script will indicate and stop.

To run script:

```
$ sudo ./runner.sh
```

You can also run the script and only compile the binary by including the argument 'no'

```
$ sudo ./runner.sh no
```

- How to install the Toolchain:
  https://github.com/GaloisInc/betaflight/wiki/How-to-setup-Kendryte-RISC-V-GNU-Compiler-Toolchain-(Ubuntu)