

Capstone Project: Betaflight Port

TEST Plan and Results

Date: June 10 2020

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

Sponsor

Galois, Inc.

Faculty advisor

Roy Kravitz

"Intelligence is the ability to avoid doing work, yet getting the work done."

~ Linus Torvalds

Revision History:

Date	Version	Comment
05/13/2020	1.0	Initial Test planning
05/20/2020	1.1	Added Unit Tests and Integration Tests
05/28/2020	1.2	Added functionality testing
05/29/2020	1.3	Added more unit tests and refactored existing definitions and explanations

Table of Contents

Introduction:	3
Betaflight Boot Sequence	3
Flow 1: Valid config file exists:	3
Flow 2: Invalid config file exists (or is missing):	4
Integration testing	4
Additional integration testing documentation	5
Unit testing	5
Functional Testing	6
TEST Results	6
Unit test results	6
Integration test results	8
Test Flow 1: Valid PG config file in EEPROM	8
Test Flow 2: PG config file is NOT valid (has been corrupted or doesn't exist)	12
Functional Testing - Black Box	17
Tests:	17
References:	20
Appendix:	21
Power supply reference and pins	21
Screenshots/Test Results:	22
Design Flowcharts:	25

Introduction:

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.

To explain how the test plan works, we must first explain Betaflight's initial boot sequence.

Betaflight Boot Sequence

The boot sequence of a flight control board running Betaflight begins by calling the `init()` function in the main loop within `main.c`. From there execution jumps to `init.c`, where most of the configuration process occurs. The first step in `init.c` is resetting the parameter group (PG) configs (user settings, and enabled features) for the particular board. Betaflight then initializes the SPI instance and Flash hardware on the board. After the Flash initializes, Betaflight will attempt to load the PG config file from Flash into a reserved section of RAM. Betaflight will then check if the file is valid, or if it has been corrupted somehow. Here is where the boot sequence diverges down one of two branches:

Flow 1: Valid config file exists:

If the PG config file is valid (i.e. not corrupted), Betaflight will check that the EEPROM version number within the header of the PG config file is correct. If that check passes, Betaflight deems that the PG config file is correct, and the system configuration process completes, returning execution to `main.c`.

Flow 2: Invalid config file exists (or is missing):

If the PG config file is invalid, Betaflight will create a new PG config file. Betaflight will then write this new file to Flash, and from there will write it back to the reserved section in RAM. Betaflight will check if the header section in file has value (0xBE), if not then it creates a new PG config file. This will happen in the case that Betaflight runs for the first time after flashing binary. Betaflight will then check that the EEPROM version number is correct, just like it would if the PG config file had been valid. Once that check passes, the system configuration process is complete and execution returns to main.c.

To emulate both paths during unit testing, we included code in main.c. The code will read a last state flag from flash at address 0x80200000 to determine which was the last state if either valid config exists (branch 1) or invalid config file exists or missing (branch 2). If the last state was branch 1, then we will set the last state flag and erase the flash sector where the config exists. This will cause Betaflight to write the config file at the next boot up of the device. If the last state was branch 2, then we will just set the last state flag. This will cause Betaflight not to write the config file at the next boot up of the device

Integration testing

To test both branches, 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 (see the screenshot section in the appendix).

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.

Additional integration testing documentation

In the appendix we have included screenshots of the actual terminal test prints referred to in the integration test results tables. Two flowcharts are also provided; these map out the test prints along both branches. The flowcharts are meant to be cross-referenced with the screenshots and test results tables to give a better idea of the program's overall flow, and to describe where each test print comes from within the Betaflight source code.

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, 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[1] 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).[2] Functional testing is conducted to evaluate the compliance of a system or component with specified functional requirements.[3] Functional testing usually describes what the system does.

(https://en.wikipedia.org/wiki/Functional_testing)

TEST Results

Unit test results

Test	Function Name	Input	Expected Result	Pass/Fail
1	loadEEPROMFrom ExternalFlash	None	1	PASS
2	readEEPROM	None	1	PASS
3	loadEEPROM	None	1	PASS
4	isEEPROMStructure Valid	None	1	PASS
5	isEEPROMVersionV alid	None	1	PASS
6	writeSettingsToEEP ROM	None	1	PASS
7	config_streamer_wr ite	None	0	PASS
8	config_streamer_flu sh	None	0	PASS
9	config_streamer_fin ish	None	0	PASS
10	flash_read_id	&manuf_id, &device_id	0	PASS
11	flash_is_busy	None	0	PASS

12	flash_sector_erase	addr = 0x00150000	0	PASS
13	flash_32k_block_erase	addr = 0x00150000	0	PASS
14	flash_chip_erase	None	0	PASS
15	flash_write_data	1. addr = 0x00150000 2. DATA= (1024+1024) 3. length =1024	0	PASS
16	flash_read_data	1. addr = 0x00150000 2. length =1024 3. data_buf[length]	0	PASS
17	flash_init	(3,0)	0	PASS

Integration test results

Test Flow 1: Valid PG config file in EEPROM

	Function Name	Call Location	Definition	Pass/Fail
Test: 1	pgResetAll	init in init.c	src/main/pg/pg.c	PASS
Expected function Behavior	Resets all parameter group (PG) configs.			
Expected Function Output	The function should print "Reset All Configs - successful" to terminal after it gets called from init in init.c.			
Overall	Success - The function executes correctly, returns back to the main branch in init.c.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 2	Spi_init, called from init in init.c	init in init.c	lib/main/RISC_V_K210/drivers	PASS
Expected function Behavior	Kendryte API function called by init that initializes a K210 SPI device. Parameters are SPI device number, work mode, frame format, data length (bits), device endianness).			
Expected Function Output	The function should print "SPI_3 Initialized - successful" to the terminal after it gets called from init in init.c. If executed properly, function will return NULL and no errors will be reported.			
Overall	Success - The function executes correctly - returns no errors.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 3	Flash_init, called from init in init.c	init in init.c	src/main/drivers/flash_riscv_k210.c	PASS
Expected function Behavior	Kendryte API function that initializes K210 external flash. Parameters are (SPI index, SPI chip select pin). SPI index must not be greater than 4 (only 4 SPI instances on K210).			
Expected Function Output	The function should return 0 when the input is 3, 0. It will also print "SPI_3 Initialized - successful" to the terminal if the external flash initialization was successful.			
Overall	Success - The function returns 0, indicating external flash has been initialized properly.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 4	initEEPROM, called from init in init.c	init in init.c	src/main/config/config_eeprom.c	PASS
Expected function Behavior	Determines where to get PG configs (external flash for K210). Then calls subroutine loadEEPROMFromExternalFlash() to load PG configs into EEPROM.			
Expected Function Output	loadEEPROMFromExternalFlash() should return 1 if flash read into EEPROM was successful. Returns 0 if unsuccessful. Prints "Loaded EEPROM from Flash - successful" to the terminal if successful.			
Overall	Success - loadEEPROMFromExternalFlash() returns 1, success message gets printed to the terminal.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 5	loadEEPROMFromExternalFlash(), called from init in init.c	init in init.c	src/main/config_eeeprom.c	PASS
Expected function Behavior	Loads parameter group configs from an external flash to EEPROM. Like many Betaflight functions, it doesn't accept any parameters, it just gets called.			
Expected Function Output	The function should return 1 when parameter group configs have been loaded into EEPROM.			
Overall	Success - The function returns 1.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 6A	isEEPROMStructureValid, called from init in init.c	init in init.c	src/main/config/config.c	PASS
Expected function Behavior	Checks if the PG config file within EEPROM (loaded in the previous step) is valid by conducting a CRC check.			
Expected Function Output	Returns 1 if the PG config file is valid. Also prints "EEPROM Structure Valid - successful" to the terminal if successful.			
Overall	Success - Function returns 1, prints success message.			

Here is the point where program execution will diverge. If the PG config file located in EEPROM is valid (i.e. not corrupted), program execution will continue to test 7. If the file is not valid (has been corrupted somehow) execution will branch, and BF will begin to create a new PG config file within EEPROM. Test Flow 2, after Test Flow 1, demonstrates how we tested this branch.

	Function Name	Call Location	Definition	Pass/Fail
Test: 7A	isEEPROMVersionValid	init in init.c	src/main/fc/init.c	PASS
Expected function Behavior	Checks if EEPROM config version number in PG config file matches what has been previously defined in config_eeprom.h			
Expected Function Output	Function will return true and print "EEPROM Version Valid - successful" to the terminal if successful.			
Overall	Success - Function returns true, prints success message.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 8A	readEEPROM, called from init in init.c	init in init.c	src/main/config/config.c	PASS
Expected function Behavior	Initializes features and activates configs within EEPROM.			
Expected Function Output	Function should return true and print "System Configuration Ready - successful".			
Overall	Success - Function returns true, prints success message.			

At this point the system configuration process is complete for Test Flow 1. SPI and Flash have been initialized, a valid PG config file was found in Flash and written to EEPROM, and the features and configs have been initialized. Program execution will return to main.c and print the message "Board Initialized: OK"

Test Flow 2: PG config file is NOT valid (has been corrupted or doesn't exist)

If the PG config file from Flash has been corrupted (determined in test 6B), program execution will jump here and begin to create a new PG config file within EEPROM.

	Function Name	Call Location	Definition	Pass/Fail
Test: 6B	isEEPROMStructureValid	ensureEEPROMStructureIsValid in config.c	src/main/config/config.c	PASS
Expected function Behavior	Checks if the PG config file within EEPROM is valid by conducting a CRC check.			
Expected Function Output	Returns 0 if the PG config file is invalid. Also prints "EEPROM Structure Not Valid - magic_be (0xBE) in Header File Missing" to the terminal.			
Overall	Success - Function returns 0 (want a 0 here because we want to test the 'fail' branch).			

	Function Name	Call Location	Definition	Pass/Fail
Test: 7B	pgResetAll()	resetConfig -> resetEEPROM in config.c	src/main/pg/pg.c	PASS
Expected function Behavior	Resets all parameter group (PG) configs.			

Expected Function Output	The function will print "Reset All Configs - successful" to terminal after all configs have been reset.
Overall	Success - The function executes correctly, returns back to resetConfig.

	Function Name	Call Location	Definition	Pass/Fail
Test: 8B	validateAndFixConfig()	writeUnmodifiedConfig ToEEPROM -> resetEEPROM in config.c	src/main/config/config.c	PASS
Expected function Behavior	Validates and fixes any broken parameter group (PG) configs.			
Expected Function Output	The function will print "Validate and Fix Config - successful" to terminal after all configs have been validated.			
Overall	Success - Function prints success message.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 9B	suspendRxPwmSignal (currently commented out)	writeUnmodifiedConfig ToEEPROM -> resetEEPROM in config.c	src/main/rx/rx.c	PASS
Expected function Behavior	Suspends PWM/PPM signals. This function is currently commented out because we have no signals to suspend. But we still want to see the execution flow to this function and return successfully.			
Expected Function Output	The function will print "Suspended PWM/PPM Signals - successful" to terminal upon successful return.			
Overall	Success - Function prints success message, returns to calling function writeUnmodifiedConfigToEEPROM.			

	Function Name/ call location	Call Location	Definition Location	Pass/Fail
Test: 10B	writeConfigToEEPROM()	writeUnmodifiedConfigToEEPROM -> resetEEPROM in config.c	src/main/config/config_eprom.c	PASS
Expected function Behavior	Writes PG configs from EEPROM into Flash. This function calls several subroutines which also include stub print statements. The expected output of writeConfigToEEPROM() and its subroutines is detailed below. For more detail refer to the Test Flow 2 flowchart.			
Expected Function Output	<ul style="list-style-type: none"> This function will print "Writing to Flash at Starting Address 0x130000" just prior to calling the subroutine writeSettingsToEEPROM(). Within writeSettingsToEEPROM it will call config_streamer_start which will print "Config File at RAM Address 0x80008328 Total Size Reserved (4096) kB" to show where the PG configs are stored in EEPROM and how much memory space they require. Next, writeSettingsToEEPROM will print "Config Header includes EEPROM_CONF_VERSION = 172 and magic_be = be" to demonstrate we have the correct EEPROM version and magic_be number in the header of the PG config file. Next, writeSettingsToEEPROM begins writing the PG config data from EEPROM to Flash by calling config_streamer_write. This is demonstrated with a series of messages that show the address data is being written from ("Writing to Flash Address 0x130000"), and also which PG config it's currently writing ("Writing PGn 47 Size 0x28 Version 1 Address of Group in RAM 0x0"). See the Test Flow 2 flowchart for more detail. After all PG configs have been written from EEPROM into Flash, writeSettingsToEEPROM will return 'true' if successful. 			

	<ul style="list-style-type: none"> • We verify the write process was successful by checking the return value and printing "Flash Write Complete - successful" if it was successful. • Then, the EEPROM version number in the PG config file header is checked by calling isEEPROMVersionValid. If the version is correct, isEEPROMVersionValid will print "EEPROM Version Valid - successful". • Finally, writeSettingsToEEPROM should print "Write Unmodified Config to EEPROM - successful" after successfully returning to the function it was called from (writeUnmodifiedConfigToEEPROM).
Overall	Success - writeSettingsToEEPROM prints "Write Unmodified Config to EEPROM - successful".

	Function Name	Call Location	Definition	Pass/Fail
Test: 11B	resumeRxPwmSignal (currently commented out)	writeUnfmodifiedConfigToEEPROM -> resetEEPROM in config.c	src/main/rx/rx.c	PASS
Expected function Behavior	Resumes PWM/PPM signals. This function is also commented out because we have no signals to suspend. But we still want to see the execution flow to this function and return successfully.			
Expected Function Output	The function will print "Resumed PWM/PPM Signals - successful" to terminal upon successful return.			
Overall	Success - Function prints success message, returns to calling function writeUnfmodifiedConfigToEEPROM.			

	Function Name/ call location	Call Location	Definition	Pass/Fail
Test: 12B	ensureEEPROMStructureIsValid	Init in init.c	src/main/config/config.c	PASS
Expected function Behavior	After creating a new PG config file in EEPROM, program execution returns here.			
Expected Function Output	Function will return true and print "Reset EEPROM complete - successful" to terminal if new PG config file creation is successful.			
Overall	Success - Function returns true, prints success message.			

	Function Name	Call Location	Definition	Pass/Fail
Test: 13B	isEEPROMVersionValid	init in init.c	src/main/fc/init.c	PASS
Expected function Behavior	Checks if EEPROM config version number in PG config file matches what has been previously defined in config_eeprom.h			
Expected Function Output	Function will return true and print "EEPROM Version Valid - successful" to the terminal if successful.			
Overall	Success - Function returns true, prints success message.			

At this point the system configuration process is complete for Test Flow 2. SPI and Flash have been initialized, a new PG config file was created in EEPROM, and the features and configs have been initialized. Program execution will return to main.c and print the message "Board Initialized: OK"

Functional Testing - Black Box

Tests:

Test 1	Voltage 3.3V	Voltage Supply		
Overview:	Used a voltmeter, expected voltage 3.3 V			
Results:	Got 3.3V	Pass		

Test 2	Pin 0	JTAG_TCLK		
Overview:	IO_0 I/O Multifunctional IO (FPIOA)(Bank 0,Group A)			
	This signal synchronizes the internal state machine operations			
Results:	PASS	Used 20k pull down resistor for strong 0		

Test 3	Pin 1	JTAG_TDI		
Overview:	IO_1 I/O Multifunctional IO (FPIOA)(Bank 0,Group A), tested with JTAG			
	Test Data In - Represents the data shifted into the device's test or programming logic. It is sampled at the rising edge of TCK when the internal state machine is in the correct state			
Results:	PASS	Used 20k pull down resistor for strong 0		

Test 4	Pin 2	JTAG_TDO		
Overview:	IO_2 I/O Multifunctional IO (FPIOA)(Bank 0,Group A), tested with JTAG			
	TDO (Test Data Out) – this signal represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.			
Results:	PASS	Used 20k pull down resistor		

Test 5	Pin 3	JTAG_TMS		
Overview:	IO_3 I/O Multifunctional IO (FPIOA)(Bank 0,Group A),tested with JTAG			
	Test Mode Select – this signal is sampled at the rising edge of TCK to determine the next state			
Results:	PASS	Used 20k pull down resistor for strong 0		

Note: Tests 2-5 are based on the successful handshake with openOCD using the provided configuration file (k210.cfg)

Test 4	GPIO Pin 16			
Overview:	<p>IO_16 is used for boot mode selection.</p> <ul style="list-style-type: none"> - During power-on reset, pull high to boot from FLASH and pull low to enter ISP mode. - In-system programming (ISP) - After reset, IO_0, IO_1, IO_2, and IO_3 are JTAG pins. IO_4 and IO_5 are ISP pins. - Failure of Pin 16 means not able to enter ISP and not being able to flash <p>The pin was tested by physically grounding the pin to GND, after advising the manufacturer manual</p>			
Results:	Pass	The board entered ISP state		

Test 5	Voltage 5 V			
Overview:	Main voltage supply , tested using a voltmeter			
Results:	Voltage is stable at 5V			

Test 7	GND			
Overview:	Main ground, it should be 0V			
Results:	Pass / Used decoupling capacitors just in case			

Test 8	Pin 24			
Overview:	General Purpose IO, tested using an LED that blinks, voltage ref 3.3V			
Results:	Pass			

Test 9	Pin 25			
Overview:	General Purpose IO, tested using an LED that blinks, voltage ref 3.3V			
Results:	Pass			

Test 10	Pin RST			
Overview:	Force RESET, if pulled HIGH resets the board. Pulled the RST HIGH from power supply of 3.3 V			
Results:	Pass	The board reseted and functioned correctly		

References:

- <https://kendryte.com/downloads/>
- <https://github.com/kendryte>
- <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0517a/Cjaeccji.html> - Arm has a good reference for JTAG

Appendix:

Power supply reference and pins

2.3 Power Supplies

Supply	Name	Voltage (V)	Max Current (mA)
I/O 3.3V/1.8V	VDDI00A	3.3/1.8V*1	200
I/O 3.3V/1.8V	VDDI01A	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI02A	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI03B	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI04B	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI05B	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI06C	3.3/1.8V	200
I/O 3.3V/1.8V	VDDI07C	3.3/1.8V	200
I/O 1.8V	VDDI018	1.8	200
OTP 1.8V	VDDOTP	1.8	50
Core 0.9V	VDD	0.9	2000
SoC	VSS	0	–
PLL 0.9V	VDDPLL	0.9	15

Figure 1: K210 power supplies

2.5 Special Pins

IO_16 is used for boot mode selection. During power-on reset, pull high to boot from FLASH and pull low to enter ISP mode. After reset, IO_0, IO_1, IO_2, and IO_3 are JTAG pins. IO_4 and IO_5 are ISP pins.

Figure 2: Special pins for K210

Screenshots/Test Results:

```

eric@eric-vm: ~/betaflight/k210_unit_testing/obj
File Edit View Search Terminal Help
M 0x800b0148
Writing to Flash Address 0x130148
test_src/test_config/test_config.c:0:test_writeSettingsToEEPROM:PASS
test_src/test_config/test_config.c:0:test_config_streamer_write:PASS
Writing to Flash Address 0x8011f360
test_src/test_config/test_config.c:0:test_config_streamer_flush:PASS
test_src/test_config/test_config.c:0:test_config_streamer_finish:PASS

-----
10 Tests 0 Failures 0 Ignored
OK
test_src/test_flash/test_flash.c:0:test_flash_read_id:PASS
test_src/test_flash/test_flash.c:0:test_flash_is_busy:PASS
test_src/test_flash/test_flash.c:0:test_flash_sector_erase:PASS
test_src/test_flash/test_flash.c:0:test_flash_32k_block_erase:PASS
test_src/test_flash/test_flash.c:0:test_flash_chip_erase:PASS
test_src/test_flash/test_flash.c:0:test_flash_write_data:PASS
test_src/test_flash/test_flash.c:0:test_flash_read_data:PASS
test_src/test_flash/test_flash.c:0:test_flash_init:PASS

-----
8 Tests 0 Failures 0 Ignored
OK

```

Figure 3: Unit test results

```

eric@eric-vm: ~/betaflight-rubens_branch
File Edit View Search Terminal Tabs Help

eric@eric-vm: ~/betaflight-rubens_bran... x  eric@eric-vm: ~/betaflight-rubens_branch x

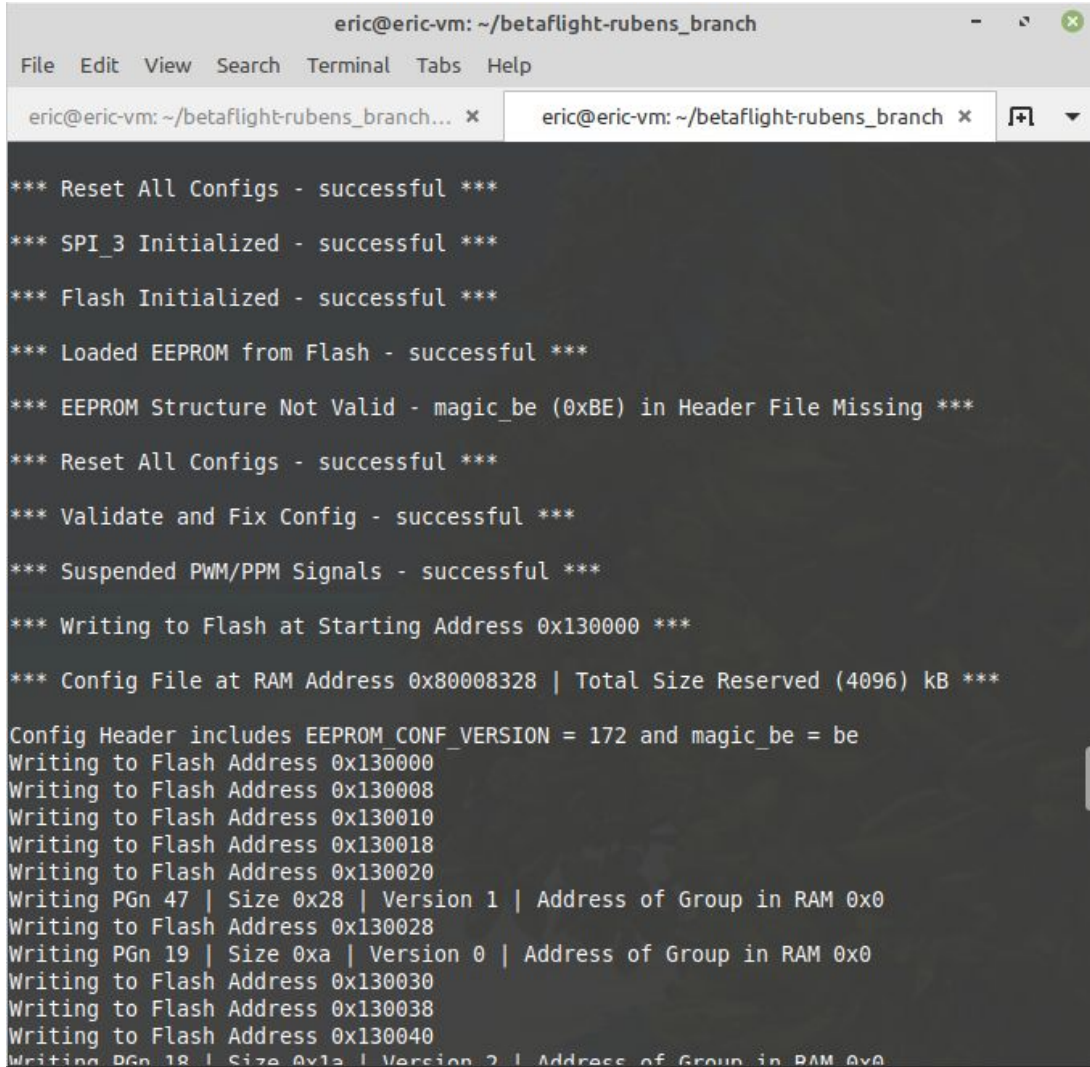
=====

*** Reset All Configs - successful ***
*** SPI_3 Initialized - successful ***
*** Flash Initialized - successful ***
*** Loaded EEPROM from Flash - successful ***
*** EEPROM Structure Valid - successful ***
*** EEPROM Version Valid - successful ***
*** System Configuration Ready - successful ***

Board Initialized: OK

```

Figure 4: Integration test prints - Flow 1: Valid PG config file

A screenshot of a terminal window titled "eric@eric-vm: ~/betaflight-rubens_branch". The terminal displays a series of test results. Most steps are successful, including resetting configs, initializing SPI_3 and Flash, loading EEPROM from Flash, and validating/fixing the config. However, the EEPROM structure is found to be invalid due to a missing magic_be (0xBE) in the header file. The test then proceeds to write the config to flash at address 0x130000. The final output shows the config header details and the start of writing to flash addresses.

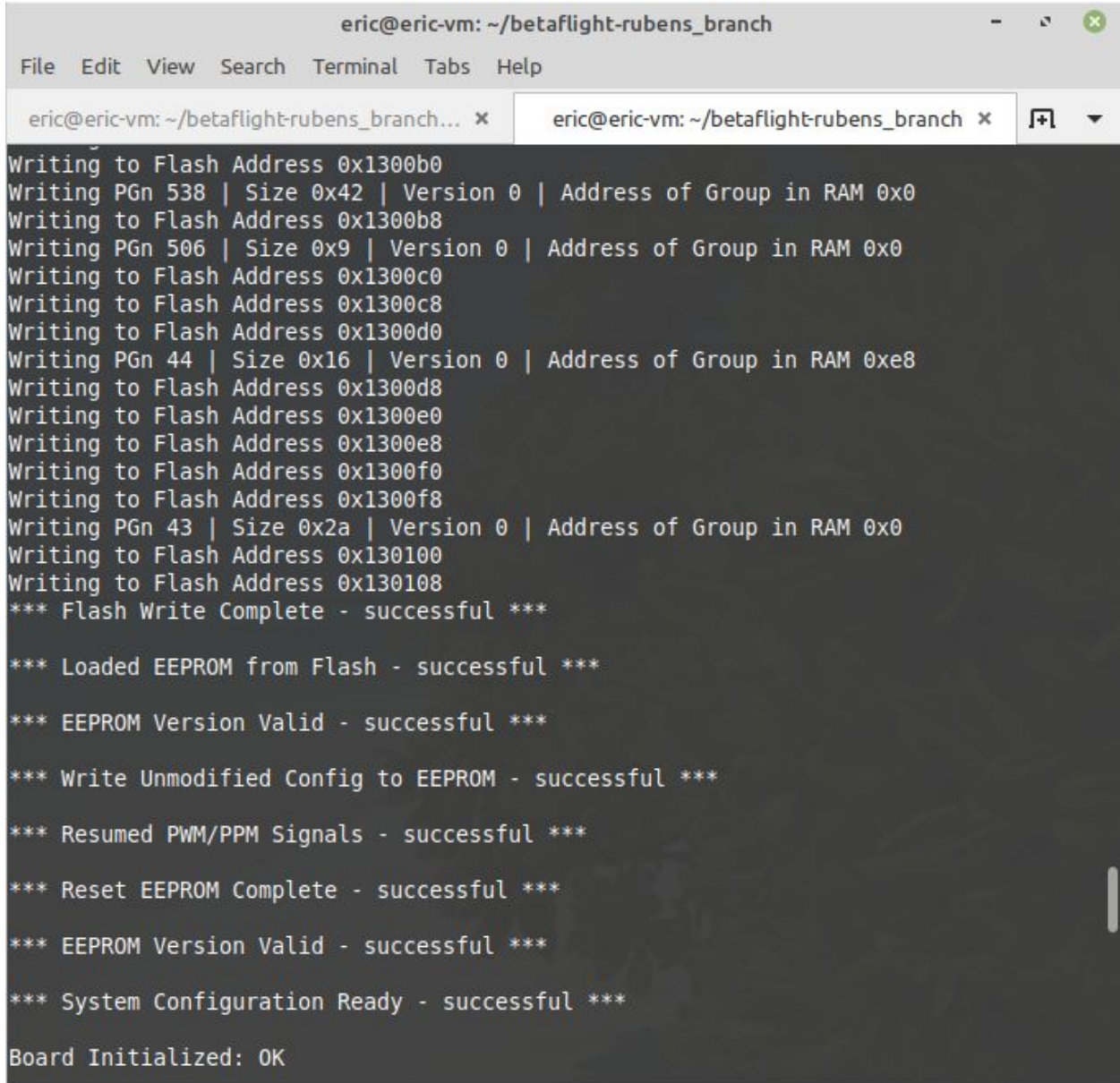
```
eric@eric-vm: ~/betaflight-rubens_branch
File Edit View Search Terminal Tabs Help

eric@eric-vm: ~/betaflight-rubens_branch... x eric@eric-vm: ~/betaflight-rubens_branch x

*** Reset All Configs - successful ***
*** SPI_3 Initialized - successful ***
*** Flash Initialized - successful ***
*** Loaded EEPROM from Flash - successful ***
*** EEPROM Structure Not Valid - magic_be (0xBE) in Header File Missing ***
*** Reset All Configs - successful ***
*** Validate and Fix Config - successful ***
*** Suspended PWM/PPM Signals - successful ***
*** Writing to Flash at Starting Address 0x130000 ***
*** Config File at RAM Address 0x80008328 | Total Size Reserved (4096) kB ***

Config Header includes EEPROM_CONF_VERSION = 172 and magic_be = be
Writing to Flash Address 0x130000
Writing to Flash Address 0x130008
Writing to Flash Address 0x130010
Writing to Flash Address 0x130018
Writing to Flash Address 0x130020
Writing PGn 47 | Size 0x28 | Version 1 | Address of Group in RAM 0x0
Writing to Flash Address 0x130028
Writing PGn 19 | Size 0xa | Version 0 | Address of Group in RAM 0x0
Writing to Flash Address 0x130030
Writing to Flash Address 0x130038
Writing to Flash Address 0x130040
Writing PGn 18 | Size 0x1a | Version 2 | Address of Group in RAM 0x0
```

Figure 5: Integration test prints - Flow 2, Part 1: Invalid PG config file



```
eric@eric-vm: ~/betaflight-rubens_branch
File Edit View Search Terminal Tabs Help

eric@eric-vm: ~/betaflight-rubens_branch... x  eric@eric-vm: ~/betaflight-rubens_branch x

Writing to Flash Address 0x1300b0
Writing PGN 538 | Size 0x42 | Version 0 | Address of Group in RAM 0x0
Writing to Flash Address 0x1300b8
Writing PGN 506 | Size 0x9 | Version 0 | Address of Group in RAM 0x0
Writing to Flash Address 0x1300c0
Writing to Flash Address 0x1300c8
Writing to Flash Address 0x1300d0
Writing PGN 44 | Size 0x16 | Version 0 | Address of Group in RAM 0xe8
Writing to Flash Address 0x1300d8
Writing to Flash Address 0x1300e0
Writing to Flash Address 0x1300e8
Writing to Flash Address 0x1300f0
Writing to Flash Address 0x1300f8
Writing PGN 43 | Size 0x2a | Version 0 | Address of Group in RAM 0x0
Writing to Flash Address 0x130100
Writing to Flash Address 0x130108
*** Flash Write Complete - successful ***

*** Loaded EEPROM from Flash - successful ***

*** EEPROM Version Valid - successful ***

*** Write Unmodified Config to EEPROM - successful ***

*** Resumed PWM/PPM Signals - successful ***

*** Reset EEPROM Complete - successful ***

*** EEPROM Version Valid - successful ***

*** System Configuration Ready - successful ***

Board Initialized: OK
```

Figure 6: Integration test prints - Flow 2, Part 2: Invalid PG config file

Design Flowcharts:

Test Flow 1: Valid config file in EEPROM

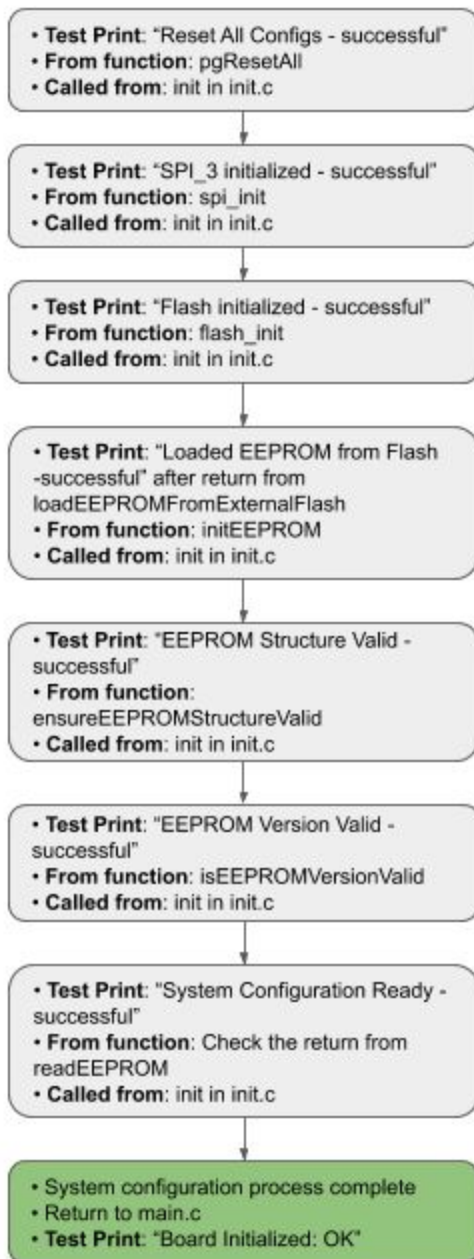
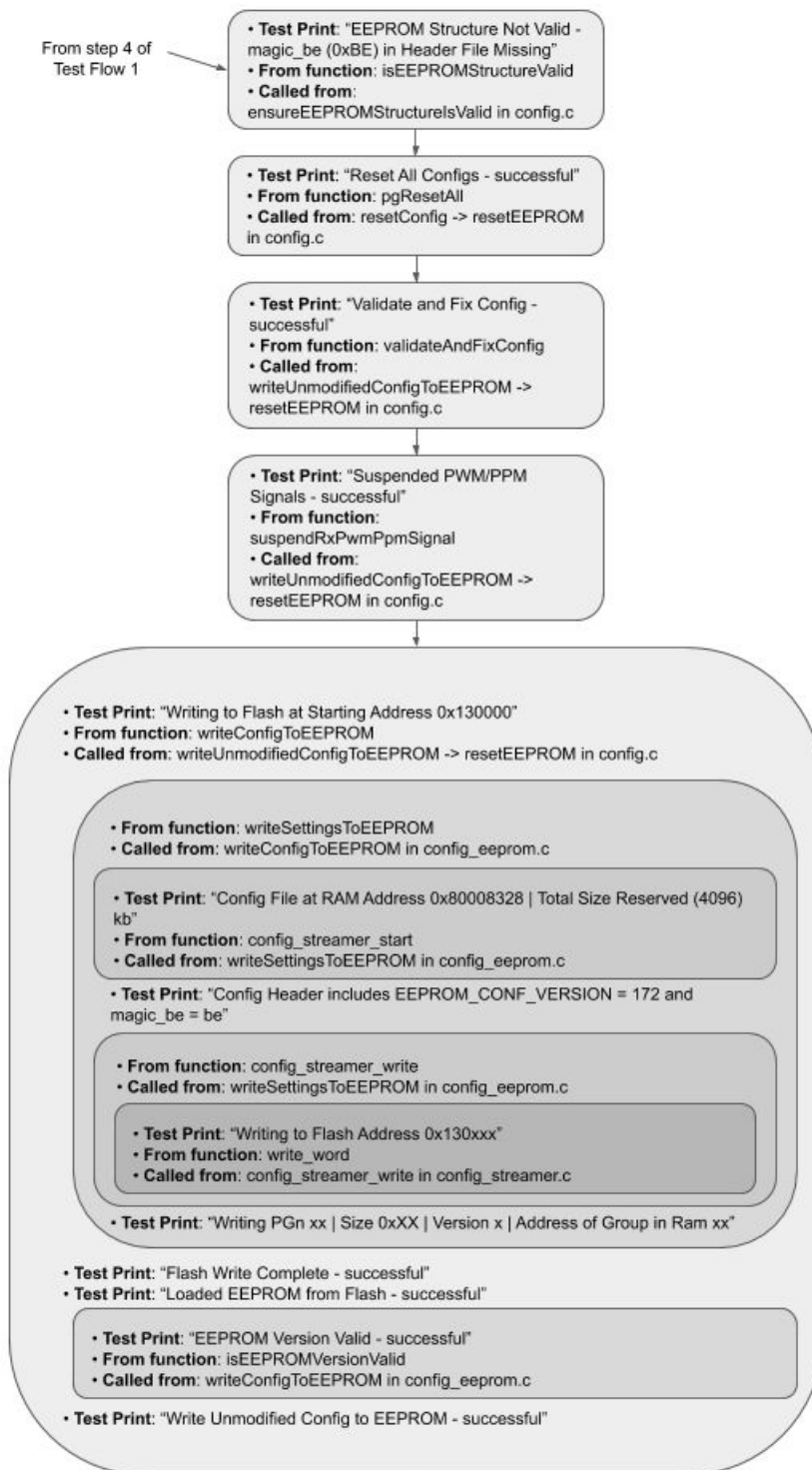


Figure 7: Flowchart 1 maps the print statements that get executed when a valid PG config file exists.

Test Flow 2: Invalid config file in EEPROM



(Figure continued on next page)

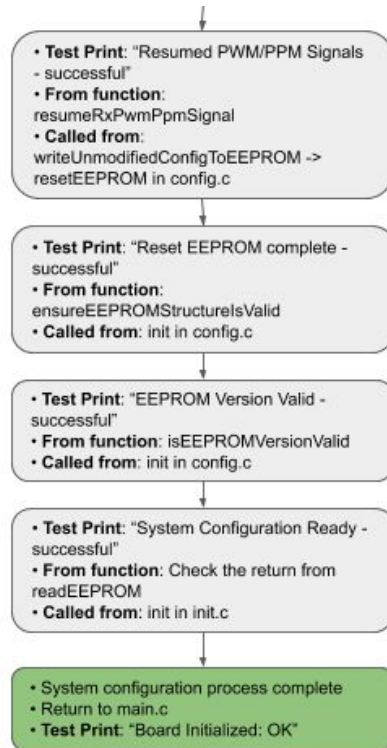


Figure 8: Flowchart 2 maps the print statements that get executed when the PG config file is invalid (or doesn't exist).