CUBESPACE

Bootloader Application Manual

Bootloader Interface

DOCUMENT NUMBER	CS-DEV.AMNL.BL-01
VERSION	5.00
DATE	22/08/2024
PREPARED BY	O. GRIES
REVIEWED BY	LV, FL, KM
APPROVED BY	L. Visagie
DISTRIBUTION LIST	External

Revision History

VERSION	AUTHOR(S)	DATE	DESCRIPTION
1.0	K. Malan	11/01/2022	First Draft
1.01	J. Jacobson	26/04/2022	Updated to include Bootloader and CubeToolbox
2.00	O. Gries	17/05/2022	Added Auto-Discovery, CubeToolbox remove write-unprotect/erase step, Updated upgrade setup command, added various sections for more info.
2.01	O. Gries	07/06/2022	CubeComputer bootloader upgrades, and default configuration. Auto-Discovery, allow for none expected.
2.02	O. Gries	08/02/2023	Added paragraph describing bootloader exit timeout.
3.00	O. Gries	23/03/2023	New template. Migrate file transfer protocol to bulk data transfer protocol.
4.00	O. Gries	07/09/2023	Migrate to new base-bootloader.
4.01	O.Gries	26/09/2023	Removed comms interface locking from base-bootloader.
4.02	O.Gries	02/10/2023	Added integration testing section. Added appendix for how to test config corruption recovery.
4.03	L. Visagie	07/06/2024	Added appendix with details on changing bootloader configuration via the CubeSupport application
5.00	O. Gries	20/08/2024	Added CubeComputer redundant config to FDIR section. Added flash ECC error handling to FDIR section. Added ECC simulation testing to integration testing section.

Reference Documents

The following documents are referenced in this document.

[1] N/A cube-common-1-base-bootloader-6-api-reference.html

[2] CS-DEV.FRM.CA-01 CubeProduct Firmware Reference Manual, Ver 7.00, 23/03/2023

[3] CS-DEV.UM.CU-01 CubeSupport HMI, Ver 1.00

[4] CS-DEV.MNL.BL-03 Bootloader Upgrade Guide

List of Acronyms/Abbreviations

ACP	ADCS Control Program
ADCS	Attitude Determination and Control System
CAN	Controller Area Network
COTS	Commercial Off The Shelf
CSS	Coarse Sun Sensor
CVCM	Collected Volatile Condensable Materials
DUT	Device Under Test
ECC	Error Correction Code
EDAC	Error Detection and Correction
EHS	Earth Horizon Sensor
EM	Engineering Model
EMC	Electromagnetic Compatibility
EMI	Electromagnetic Interference
ESD	Electrostatic Discharge
FDIR	Fault Detection, Isolation, and Recovery
FM	Flight Model
FSS	Fine Sun Sensor
GID	Global Identification
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GYR	Gyroscope
I2C	Inter-Integrated Circuit
ID	Identification
LTDN	Local Time of Descending Node
LEO	Low Earth Orbit
MCU	Microcontroller Unit
MEMS	Microelectromechanical System
MTM	Magnetometer
MTQ	Magnetorquer
NDA	Non-Disclosure Agreement
OBC	On-board Computer
OTP	One-Time Programmable

PCB	Printed Circuit Board
RTC	Real-Time Clock
RWA	Reaction Wheel Assembly
RW	Reaction Wheel
SBC	Satellite Body Coordinate
SOFIA	Software Framework for Integrated ADCS
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
SSP	Sub-Satellite Point
STR	Star Tracker
TC	Telecommand
TCTLM	Telecommand and Telemetry (protocol)
TID	Total Ionizing Dose
TLM	Telemetry
TML	Total Mass Loss
UART	Universal Asynchronous Receiver/Transmitter

Table of Contents

Table of Tables

Table of Figures

1 Introduction

This manual describes all aspects of interfacing to the CubeSpace bootloader, both on the ground and in orbit.

This manual explains how to upgrade a CubeProduct using software tools provided by CubeSpace, as well as how to perform upgrades in an embedded environment for integration with the OBC. The CubeSpace bootloader is a simple, bare-metal, flash utility application. The bootloader does not interface with any peripherals, other than those required by the communications interfaces. The bootloader implementation and API is common for all CubeProducts, however multiple firmware binaries are produced to cater for different MCU's. The bootloader application is written such that an upgrade to itself should not be required. Therefore, the bootloader is not expected to change in orbit, and cannot upgrade itself. Upgrading the CubeSpace bootloader is only possible via the manufacturer ROM bootloader and is not expected to be used in orbit.

This document references CubeSpace components as follows:

- CubeComputer – The CubeADCS Core processing unit (ADCS OBC).
- CubeProduct(s) – All active processing CubeSpace components, including CubeComputer.
- Node(s) – Sensor(s) and/or wheel(s) referred to in the context of being connected to the CubeComputer. Nodes are thus also referred to as CubeProducts, generically speaking.
- CubeADCS – The integrated ADCS system containing all the above and all associated harnessing and mechanical mounting hardware.

Detailed Information of TCTLM data structures can be found in the API reference [1].

Throughout this document, all direct references to TCTLM messages and/or the parameters within, as shown in [1], appears in `This Font` for example "`Halt`".

1.1 Document Applicability

This manual applies to the software and firmware versions as stated in Table 1

Table 1: Document Applicability

ELEMENT	VERSION	NOTES
Software Bundle	4.5.0	Or later.
CubeSupport	8.1	Or later.
base-bootloader	1.7	Or later. See Table 2.

1.2 Bootloader Applicability

Unless otherwise stated, the information in this document is applicable to all bootloaders as used in CubeProducts.

Although all the bootloader applications share the same API, different application binaries are required for certain CubeProducts to cater for different MCU's and communications interfaces.

The bootloader applicability for each CubeProduct is listed in Table 2.

The bootloaders can be found in the software bundle in the following directories:

a) <software-bundle>/firmware/cube-common-1-computer-*[1]/base-bootloader
b) <software-bundle>/firmware/cube-common-1-node/base-bootloader-52
c) <software-bundle>/firmware/cube-common-1-node/base-bootloader-R5

Table 2: Bootloader applicability

CUBEPRODUCT	BOOTLOADER
CubeComputer	base-bootloader (cube-common-1-computer-*[1])
CubeSense Sun	base-bootloader-52 (cube-common-1-node)
CubeSense Earth	base-bootloader-52 (cube-common-1-node)
CubeMag Deploy	base-bootloader-52 (cube-common-1-node)
CubeMag Compact	base-bootloader-52 (cube-common-1-node)
CubeWheel	base-bootloader-52 (cube-common-1-node)
CubeNode (All derivations)	base-bootloader-52 (cube-common-1-node)
CubeStar	base-bootloader-R5 (cube-common-1-node)
CubeAuriga	base-bootloader (cube-common-1-auriga)

1.3 Upgrade Scenarios

Different applications may be uploaded to the bootloader, however under normal circumstances, only the control program is expected to be uploaded. The ability to upload different applications is reserved for future expansion of each CubeProduct's functionality. Therefore, for all intents and purposes, the user should only be concerned about the control program firmware and configuration files while referring to this document. All references to applications other than the bootloader can be interpreted as referring to the control program application.

For ease of use, different upgrade scenarios are given so that the user may skip forward to the chapter that is relevant to them.

On the ground:

1. CubeSpace provides a new version of a control program for any of the CubeProducts and so the control program needs to be updated.
    a. Refer to chapter 5.
2. CubeSpace provides a new version of a control program configuration for any of the CubeProducts and now the configuration needs to be updated,
    a. Refer to chapter 5.

In orbit / via OBC:

1. CubeSpace provides a new version of a control program for any of the CubeProducts and so the control program needs to be updated.

[1] The wildcard "*" refers to all hardware versions of CubeComputer.

   a. Refer to section 2.7.
2. CubeSpace provides a new version of a control program configuration for any of the CubeProducts and now the configuration needs to be updated.
   a. Refer to section 2.7.

2 Functions and Features

This chapter describes the functionality of the bootloader application.

The sole purpose of the bootloader is to read/write flash memory. The bootloader does not use any peripherals other than those required for communications. This implies that within the CubeADCS system, each CubeProduct has its own bootloader programmed that is responsible for programming/upgrading the firmware of only that CubeProduct.

2.1 Configuration

The bootloader application differs from other applications within the Gen 2 system, in that it does not have a configuration file associated with it. Rather, the bootloader configuration is only accessible via TCTLM.

2.1.1 Default Configuration

A default configuration is embedded in the bootloader binary and is never modified by the application. Upon first boot, the bootloader assumes the default configuration. Modifications to the configuration are stored in a separate location in flash. Once the configuration is modified from the default via TCTLM, the modified configuration becomes the first source of configuration.

The configuration is CRC protected. Upon boot, if the configuration fails CRC validation, the bootloader reverts to using the default configuration. This event will be indicated by the `Config Init` flag in the `Errors` telemetry.

Note that the CubeComputer bootloader stores a redundant configuration (see section 2.11.1) and will therefore only assume the default configuration if both the primary and redundant configuration are invalid/corrupt.

In the event of the user's configuration becoming corrupt, the user must be prepared to communicate with the CubeProduct using the default values to, at a minimum, restore the configuration to the desired values. Information on how to artificially induce configuration corruption so that the recovery implementation can be tested, refer to Appendix: Testing Configuration Corruption Recovery.

If the bootloader reverts to using the default configuration, the configuration of the communications interfaces for the control program application may be affected as well, if the control program is configured to use the bootloader's configuration (see section 2.1.4).

Table 3: Default Configuration

PARAMETER / FIELD	DESCRIPTION	DEFAULT VALUE
`Serial Number`	Serial number string. This value can remain the default value if the OTP memory contains a valid serial number string. This is used in conjunction with the "OtpOverride" item.	XXxxxxx
`Backoff`	How long the bootloader will wait before jumping to application.	5000 [ms]
`OTP Override`	If the serial number string in OTP memory is invalid, this can be set to TRUE. The bootloader will then assume the serial number string set in the configuration.	FALSE
`UART1 Baud`	Baud rate of UART1 interface.	`Baud921600`
`UART1 RS485 Transceiver`	Is a RS485 transceiver populated on UART1 interface. If the transceiver is not populated, a TRUE value has no effect, therefore the default is TRUE. However, this item is left as an option to cater for the possibility that, if an	TRUE

PARAMETER / FIELD	DESCRIPTION	DEFAULT VALUE
	RS485 transceiver is not populated, and the MCU cannot drive the DE pin, the DE pin can be ignored by setting this to FALSE.	
UART1 RS485 Address	RS485 address of UART1 interface.	1
UART1 Timeout	Timeout of complete message reception on UART1 interface.	200 [ms]
UART2 Baud	Only applicable to CubeComputer. Baud rate of UART2 interface.	`Baud921600`
UART2 RS485 Transceiver	Only applicable to CubeComputer. Is a RS485 transceiver populated on UART2 interface. If the transceiver is not populated, a TRUE value has no effect, therefore the default is TRUE. However, this item is left as an option to cater for the possibility that, if an RS485 transceiver is not populated, and the MCU cannot drive the DE pin, the DE pin can be ignored by setting this to FALSE.	TRUE
UART2 RS485 Address	Only applicable to CubeComputer. RS485 address of UART1 interface.	1
UART2 Timeout	Only applicable to CubeComputer. Timeout of complete message reception on UART1 interface.	200 [ms]
CAN1 Address	Address of CAN1 interface.	1
CAN1 CSP Enabled	Use the Cubesat Space Protocol (CSP) on CAN1 interface.	FALSE
CAN1 Timeout	Timeout of complete message reception on CAN1 interface.	200
CAN2 Address	Only applicable to CubeComputer. Address of CAN2 interface.	1
CAN2 CSP Enabled	Only applicable to CubeComputer. Use the Cubesat Space Protocol (CSP) on CAN2 interface.	FALSE
CAN2 Timeout	Only applicable to CubeComputer. Timeout of complete message reception on CAN2 interface.	200
I2C Address	I2C address.	83 (decimal) (0xA6 Write) (0xA7 Read)
I2C Timeout	Timeout of complete message reception on I2C interface.	200 [ms]

2.1.2 Modifying Configuration

The bootloader configuration can be requested/modified using the `Config` telemetry/telecommand.

The modified (non-default) configuration is stored in a page in flash and requires that the previous modified configuration be erased before the new non-default configuration is programmed. Therefore, the user must ensure that the CubeProduct has a stable power supply when the telecommand is sent. However, the configuration is only expected to be modified once, on the ground.

If the new non-default configuration is successfully programmed, the bootloader will immediately issue a soft reset for the new configuration to take effect on the next boot.

Please see appendix 6 for details on how to change configuration via the CubeSupport application.

2.1.3 CubeWheel Backoff Configuration

The considerations mentioned in this section are applicable to CubeWheel models CW0017, CW0057 and CW0162. If the user is unsure of the hardware version, the information in this section may be applied regardless, with no negative influence on the behaviour of more recent hardware versions.

If the CubeWheel is running in its control program and is spinning at a high speed, and a reset occurs, the wheel may experience braking while the bootloader is running. The bootloader is agnostic to which CubeProduct it is running on, and therefore cannot interface with the motor driver to prevent this behaviour.

To minimize the effect of braking, the bootloader backoff configuration should be set to 1000 milliseconds.

This configuration is set in production and should not require user input, however, should the CubeWheel revert to using the default configuration as explained in 2.1.1, the user must be prepared to set the backoff configuration.

The above considerations are only applicable if the CubeWheel(s) is not integrated in a CubeADCS bundle but rather commanded by the client OBC. For CubeWheels that are part of a CubeADCS bundle, the CubeComputer will manage the backoff configuration of all connected CubeWheels, and there is no input required by the user.

2.1.4 Configuration Shared with Other Applications

The bootloader places certain configuration items, relating to the communications interfaces, in a section of memory where the application that is started from the bootloader can access them. Those applications can then adopt the same communications configuration as the bootloader. Whether or not an application adopts the configuration from the bootloader is configurable for each application (it is not a bootloader configuration item). By default, all applications will adopt the bootloader configuration.

This reduces the complexity of managing configuration for multiple applications on the same CubeProduct.

The configuration items that get passed on to other applications are:

- UART1 Baud
- UART1 RS485 Address
- UART1 RS485 Transceiver
- UART2 Baud
- UART2 RS485 Address
- UART2 RS485 Transceiver
- CAN1 Address
- CAN1 CSP Enabled
- CAN2 Address
- CAN2 CSP Enabled
- I2C Address

2.2 Communication Interfaces

All communication interfaces on a given CubeProduct are always enabled. The bootloader will listen to all interfaces for messages.

2.2.1 UART1

This interface is present on all CubeProducts.

This interface can use the point-to-point UART or RS485 protocol described in [2].

The port configuration is 8N1, with configurable baud rate.

2.2.2 UART2

This interface is only present on CubeComputer.

This interface can use the point-to-point UART or RS485 protocol described in [2].

The port configuration is 8N1, with configurable baud rate.

2.2.3 CAN1

This interface is present on all CubeProducts.

This interface can use the Cubesat Space protocol (CSP) or the CubeSpace protocol, described in [2].

The baud rate is always 1Mbps.

Note that for CubeComputer, this port is used for internal communication with CubeProducts in the CubeADCS system and is not the primary slave communications port. For all other CubeProducts, this is the primary slave communications port.

2.2.4 CAN2

This interface is only present on CubeComputer.

This interface can use the Cubesat Space protocol (CSP) or the CubeSpace protocol, described in [2].

The baud rate is always 1Mbps.

Note that this is the primary slave communications port for CubeComputer used for communicating with the client OBC.

2.2.5 I2C

This interface is present on all CubeProducts.

The I2C port uses 7-bit addressing:

- Read: ((address << 1) | 0x01)
- Write: (address << 1)

The I2C port supports speeds up 400kbps.

The I2C port uses clock-stretching and is required for stable communications, particularly during file uploads.

The I2C port has an SCL low timeout of 340 milliseconds.

The I2C port does not support repeated start conditions.

2.3 Identification

The bootloader application is common for all CubeProducts. The only differences in the application between CubeProducts, are the microcontroller and the communication interfaces. The CubeComputer is the only CubeProduct with unique communications interfaces, all other CubeProducts have the same communications interfaces and in the same hardware configuration. Therefore, only the bootloader on

CubeComputer has knowledge of which CubeProduct it is. For this reason, the bootloader implementation is agnostic to CubeProduct type.

The `Identification` telemetry will return `NodeTypeInvalid` as the value for `Node type identifier`, for all CubeProducts other than for CubeComputer. Therefore, the true source of identification for a given CubeProduct, while running in the bootloader, is the `Serial Number` telemetry.

Note that the bootloader API definition does not implement the “common-framework-*.xml” API module, which implements the same `Identification` telemetry. This is done so that the bootloader application remains independent of the common framework and will not be affected by compatibility issues in future. However, at the time of this writing, both implementations of the `Identification` command use Id=128, and the same structure. Therefore, the same telemetry can be used for all other applications. This telemetry is consistent across legacy CubeProducts and is not expected to change.

2.4 Reset

The `Reset` telecommand can be used to command the bootloader to reset itself.

If the `Soft` reset option is specified, the bootloader will only reset if the `AppState` is idle.

If the `Hard` reset option is specified, the bootloader will always reset.

In both cases, the reset will occur in the communications handler, and the user will not receive a response if the reset takes place.

Note that the bootloader API definition does not implement the “common-framework-*.xml” API module, which implements the same `Reset` command. This is done so that the bootloader application remains independent of the common framework and will not be affected by compatibility issues in future. However, at the time of this writing, both implementations of the `Reset` command use Id=1, and the same structure. Therefore, the same command can be used for all other applications.

2.5 Backoff

The bootloader implements a backoff timer to allow for automatic exit. The timer is started on boot. When the timer expires, the bootloader will attempt to jump to the default application, see section 2.8. The duration of the backoff is configurable. The minimum allowed backoff is 1000 milliseconds to maintain the ability to halt the bootloader.

The backoff period can be interrupted by sending the `Halt` telecommand before the backoff period has expired. Once the `Halt` telecommand is received the bootloader will not exit until it is commanded to do so.

The backoff period is automatically interrupted if the bootloader receives a command to read/write data to flash. However, best practice is to manually command the bootloader to halt before performing any operations.

2.6 File Table

The bootloader stores a “file table” in a separate location in flash memory that tracks the location, file type, and version information of files that have been uploaded using the process in section 2.7. Each file has its own entry in the file table. The information of each file entry can be queried as described in section 2.7.2.

When the user requests the `File Info` telemetry, the bootloader will perform a CRC validation of the target file for that iteration. If the CRC validation fails, the file will be marked as corrupt in the returned information for that file. When a file is marked as corrupt, the information of the file is presented in the `FileInfo` telemetry, but the `File Corrupt` flag will be set.

Configuration files are not only written by the bootloader. The control program may also modify its own configuration. Therefore, the CRC value for the configuration may change. The last four bytes of a configuration file contains the current CRC of the file. The bootloader validates against this value, rather than the CRC that was valid at the time that the file was uploaded to the bootloader.

The CRC value that the bootloader validates against for a given file is always the CRC returned when requesting file information. Therefore, if the bootloader detects a change in the CRC value of a configuration file, it will update the entry in the file table for that file and re-write the file table to flash.

2.7 Firmware Programming

Firmware is programmed by uploading CubeSpace (.cs) files to the bootloader. For information on CubeSpace files, see [2].

Example code is provided as part of libcubeobc (included in the software bundle) for uploading files to the bootloader as well as for requesting file information and deleting file entries.

The `Write File Setup` telecommand is used to initiate the upload. This telecommand only requires the metadata extracted from the CubeSpace file. The metadata contains information about the file. Most importantly, it contains the address the file should be written to, and the file size.

The bootloader will only disallow an upload if the address range, indicated in the metadata of the file is invalid. There are no checks done by the bootloader to ensure that the file being uploaded is in fact the correct file for that CubeProduct. Additionally, the bootloader does not validate firmware binary files against configuration files. Therefore, it is left to user discretion to ensure that the correct files are uploaded, and that a firmware binary file is always paired with a configuration file with matching versions and program type. The user should always use firmware and configuration files from the same software bundle, which ensures that the address that each file is written to is correct, and that the configuration is where the firmware expects it to be.

2.7.1 File Upload Sequence

The file upload procedure requires that the file table be re-written to flash once the upload is complete, therefore the user must ensure that the CubeProduct has a stable power supply before performing the operation.

The file upload sequence is as follows:

1. Ensure that the `App State` is idle by requesting the `State` telemetry.
2. Send the `WriteFileSetup` telecommand, containing the metadata extracted from the file to be uploaded.
3. Poll the `State` telemetry.
    a. Until `App State` transitions to `StateBusyWaitFrame`.
    b. Allow for up to 30 seconds for the state to transition. This allows time for the required flash region to be erased.
    c. The poll backoff should be at least 200 milliseconds, however a longer backoff is preferred to allow the application more processing time.
    d. Abort if the `Result` parameter is non-zero, or if the state does not transition in time. Request the `Errors` telemetry to get more detailed information on the error.
4. Upload the file data using the Bulk Data Transfer protocol, described in [2].
5. Confirm that the file is present in the file table by requesting file information, as described in section 2.7.2.

2.7.2 File Information

The bootloader stores a file table (see section 2.6) that tracks the files that have been uploaded. Each entry in the file table contains information about the file. This information is extracted from the file metadata when the file is uploaded. The index of the entry in the file table is referred to as the file handle. Entries within the file table are always ordered in ascending order of the address where the file is in flash. Therefore, when a new file is uploaded, the file handles of existing files may change as the new file entry is inserted.

The file table can be queried via telemetry, one entry at a time. The bootloader stores a "file-info-index", which determines which file information is returned when the `File Info` telemetry is requested. The index starts from 0(zero) (after boot) and automatically increments every time the `File Info` telemetry is requested and wraps back to 0(zero) once all valid files have been queried.

The user can reset the file-info-index back to 0(zero) at any time by sending the `Reset File Info Index` telecommand.

A typical request for file information is as follows:

1. Send the `Reset File Info Index` telecommand.
2. Request `File Info` until `Empty`==TRUE or `Last`==TRUE.

Note that requesting `File Info` triggers a CRC calculation over the target file. This CRC calculation is performed in the communications handler before the response, and so for large files, the response to the telemetry request will be delayed. A timeout of 500 milliseconds should be sufficient for all cases.

2.7.3 Delete File Entry

An entry in the file table can be deleted by sending the `Delete File Entry` telecommand and specifying the file handle of the entry to be deleted. This command will delete the entry in the file table but will not erase the region of flash that the file occupies. Therefore, the file contents will remain in flash, but the bootloader will no longer be able to locate it.

This command requires that the file table be re-written to flash, therefore the user must ensure that the CubeProduct has a stable power supply before performing the operation.

2.8 Jump-To-Application

This section describes how to exit the bootloader and execute a programmed application by using the `JumpTo*` commands.

2.8.1 Default Application Target

The default application target is the file handle of the binary file the bootloader should jump to when the backoff timer expires. The default value for the default application target is 0(zero). The user can query and modify the default application target using the `Default application target` telemetry/telecommand.

The default application target is configurable but does not form part of the bootloader configuration items. It is rather stored within the file table, as it is expected to change more often than the bootloader configuration.

Note that when the default application target is changed, the file table will need to be re-written to flash, therefore the user must ensure the CubeProduct has a stable power supply for this operation.

2.8.2 Jump To Default Application

The `Jump To Default Application` telecommand can be used to command the bootloader to jump to whatever the default application target is set to. This is the preferred method under normal operating conditions. The

file pointed to by the default application target must not be corrupt. If the file is corrupt, the bootloader will not exit.

2.8.3 Jump To Application

The `Jump To Application` telecommand can be used to command the bootloader to jump to a specific application. The target file handle must be a binary file and must not be corrupt, otherwise the bootloader will not exit.

2.8.4 Jump To Address

The `Jump To Address` telecommand can be used to command the bootloader to jump to a specific flash address. This telecommand should not be used under normal operating conditions and serves only as a backdoor should it be required.

2.9 Raw Memory Access (Internal Use)

The bootloader exposes telemetry/telecommands to read/write flash memory, outside of the nominal use case of uploading firmware files. They are listed in the API definition (see [1]) as part of the “InternalUse” group. This functionality should only be used in a debugging scenario, however, it is recommended that client sub-systems that interface with the CubeADCS implements this functionality, should the need arise later. Refer to chapter 6 for details on testing such an implementation.

2.9.1 Write Memory

The `Write Memory Setup` telecommand is used to initiate a memory write. The sequence of writing memory is identical to uploading files, apart from the setup command.

The procedure to write to memory is as follows:

1. Ensure that the `App State` is idle by requesting the `State` telemetry.
2. Send the `Write Memory Setup` telecommand, containing the desired parameters.
3. Poll the `State` telemetry.
    a. Until `App State` transitions to `StateBusyWaitFrame`.
    b. If `Auto Erase` was set to TRUE in the setup, allow enough time for the erase operation to complete. This varies depending on the write size. A general rule is to allow for 500 milliseconds per 8KB of data. If `Auto Erase` was set to FALSE in the setup, the state should transition within 1 second.
    c. The poll backoff should be at least 200 milliseconds, however a longer backoff is preferred to allow the application more processing time.
    d. Abort if the `Result` parameter is non-zero, or if the state does not transition in time. Request the `Errors` telemetry to get more detailed information on the error.
4. Upload the data using the Bulk Data Transfer protocol, described in [2].

Note that the bootloader will not allow itself to be overwritten; there are both software and hardware write-protection implemented to ensure that this is not possible. However, the bootloader configuration and file table can be overwritten, therefore care must be taken when testing the implementation.

2.9.2 Read Memory

The `Read Memory Setup` telecommand is used to initiate a memory read.

The procedure to read memory is as follows:

1. Ensure that the `App State` is idle by requesting the `State` telemetry.

2. Send the `Read Memory Setup` telecommand, containing the desired parameters.
3. Poll the `State` telemetry.
    a. Until `App State` transitions to `StateBusyWaitFrame`.
    b. Allow for up to 1 second for the state to transition.
    c. The poll backoff should be at least 100 milliseconds.
    d. Abort if the `Result` parameter is non-zero, or if the state does not transition in time. Request the `Errors` telemetry to get more detailed information on the error.
4. Download the data using the Bulk Data Transfer protocol, described in [2].

2.9.3 Erase Memory

The `Erase Memory Setup` telecommand is used to initiate a memory erase. Note that the size of the erase is specified in bytes. The size is converted to page numbers internally. The page size of each CubeProduct may vary, and are not specified in this document, as any use of this functionality should be under the guidance of CubeSpace. Also, testing of this functionality on the ground does not require knowledge of page sizes.

The procedure to erase memory is as follows:

1. Ensure that the `App State` is idle by requesting the `State` telemetry.
2. Send the `Erase Memory Setup` telecommand, containing the desired parameters.
3. Poll the `State` telemetry.
    a. Until `App State` transitions to `StateIdle`.
    b. Allow enough time for the erase operation to complete. This varies depending on the write size. A general rule is to allow for 500 milliseconds per 8KB of data.
    c. The poll backoff should be at least 200 milliseconds. However, a longer backoff is preferred to allow for more processing time.
    d. Abort if the `Result` parameter is non-zero, or if the state does not transition in time. Request the `Errors` telemetry to get more detailed information on the error.

2.10 Manufacturer Option Bytes (Internal Use)

The `Option Bytes` telemetry/telecommand exposes the microcontroller manufacturer option bytes. These parameters are vital to the functionality of the CubeProduct.

The `Option Bytes` should only be modified when upgrading the CubeComputer bootloader. Information on how to upgrade the CubeComputer bootloader is provided in [4].

Therefore, this document will not provide information on how to set these parameters. The user should implement the telemetry request to allow for confirmation of the parameters.

Additionally, the `Commit Option Bytes` telecommand is used in the procedure to set these option bytes and should not be commanded by the user.

Both telecommands require a `MagicNumber` input from the user for the command to be processed. Information on this number is provided in [4]. If either telecommand is sent with the incorrect `MagicNumber`, a `SequenceError` nack will be returned.

2.11 Fault Detection, Isolation and Recovery

All error conditions are reported by the `Errors` telemetry.

The errors encountered by the bootloader can be categorized as either initialization errors or runtime errors.

Runtime errors never impede further operations of the bootloader, they are isolated to the current operation, and are reset when the next operation begins.

Initialization errors apply only to the initialization of the HAL layer, the communications interfaces, and the configuration validation.

Initialization of the HAL layer consists of manufacturer HAL setup, clock setup, and interrupt setup. Failure to initialize the HAL layer is only reported. The bootloader will attempt normal operation with no deviation to the program flow. Behaviour of such a failure will only be apparent during operation.

Initialization of each communications interface is isolated to that interface. i.e., a failure on one interface does not affect the functioning of another interface. If an interface fails initialization, the bootloader does not attempt to service that interface, and it will not be usable. The bootloader will not re-attempt the initialization. Therefore, the only way to recover would be for the user to power cycle the unit or to issue a reset command on a functional interface.

Initialization of the configuration consists of a CRC validation. If the validation fails, the default configuration is used, and the error is reported. For CubeComputer, there is a redundant (backup) configuration, therefore both the primary and redundant configuration must be invalid before the default configuration will be used. The failure only affects the configuration values, and the bootloader will not deviate from the normal program flow.

2.11.1 Redundant Configuration (CubeComputer only)

The CubeComputer bootloader stores two copies of the user configuration, in separate flash pages. These pages are referred to as the primary configuration and the redundant configuration.

The bootloader will always attempt to use the primary configuration first, if validation fails, then the redundant configuration is attempted, if validation fails as well, then the default configuration, embedded in the binary, is used.

During normal operation, the primary configuration and the redundant configuration will mirror each other. When the user sets the configuration using the `Config` telecommand, first the primary config will be erased and programmed, and only if that is successful, the redundant configuration is then erased and programmed, with the same values. If programming of the primary configuration fails, the error is reported via the `Commit File Table` flag of the `Errors` telemetry. If the programming of the redundant configuration fails, it is reported via the `Redundant Cfg Write` flag in the `Warnings` telemetry.

Whether or not the primary or redundant configuration is corrupt is reported by the `Warnings` telemetry. The following flags indicate configuration corruption:

- Flag `Primary Cfg CRC`
- Flag `Primary Cfg ECCD` (see section 2.11.2)
- Flag `Redundant Cfg CRC`
- Flag `Redundant Cfg ECCD` (see section 2.11.2)

If both the primary and redundant configuration are corrupt, in addition to the above flags being set, the `Errors` telemetry will report the error via the `Config Init` flag.

The user should request the `Warnings` telemetry every time it is detected that the bootloader is running, as well as after the configuration has been set.

To gain the users attention, the bootloader will not automatically jump to the control program at the end of backoff, if any of the above-mentioned `Warnings` flags are set. However, the user is still able to command the bootloader to jump.

2.11.2 Flash ECC Error Handling (CubeComputer only)

The microcontroller implements ECC correction/detection in hardware. The implementation offers automatic **detection and correction** of single bit errors (referred to as ECCC), and **only detection** of double bit errors (referred to as ECCD).

This type of flash corruption can occur due to a write to flash being interrupted by an unexpected reset (watchdog, power, hard reset), or a radiation event causing bit flips in flash.

The bootloader mitigates the risks of ECCD errors by storing a redundant user configuration (see section 2.11.1), as well as embedding a redundant executable binary.

All references to the bootloader binary, refer to the combined primary and redundant binary, unless there is an explicit reference to either the primary or redundant.

The flash layout of the CubeComputer is shown in Figure 1.

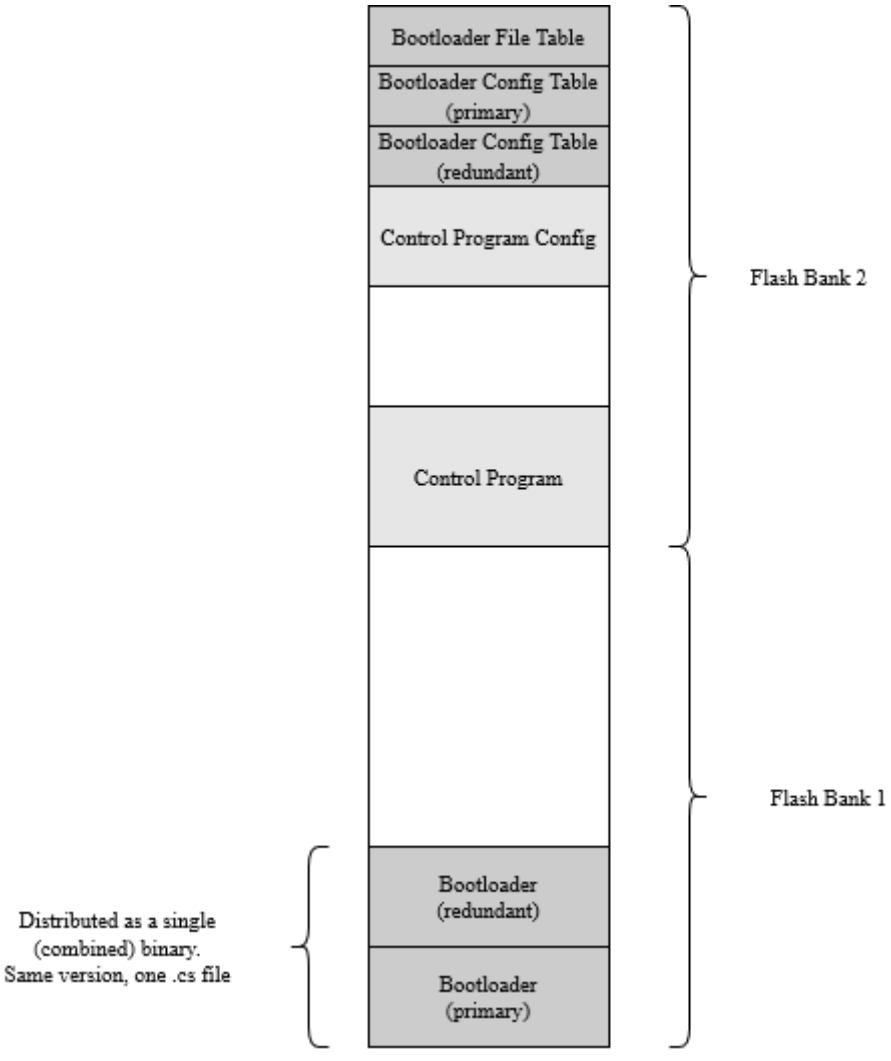

Figure 1: CubeComputer flash layout

The bootloader binary is write-protected and will never be erased/written, unless a bootloader upgrade is performed. Therefore, the purpose of the redundant binary is to protect against ECCD errors induction by radiation.

The bootloader configuration can be written to at any time, and the redundant configuration protects against ECCD errors.

The bootloader file table is written to when a file is uploaded, or when the default jump target is set. There is not a redundant file table because the file table is not critical.

All ECC errors are reported via the `Warnings` telemetry.

A summary of the actions taken in the event of ECC errors are shown in Figure 2 and Table 4.

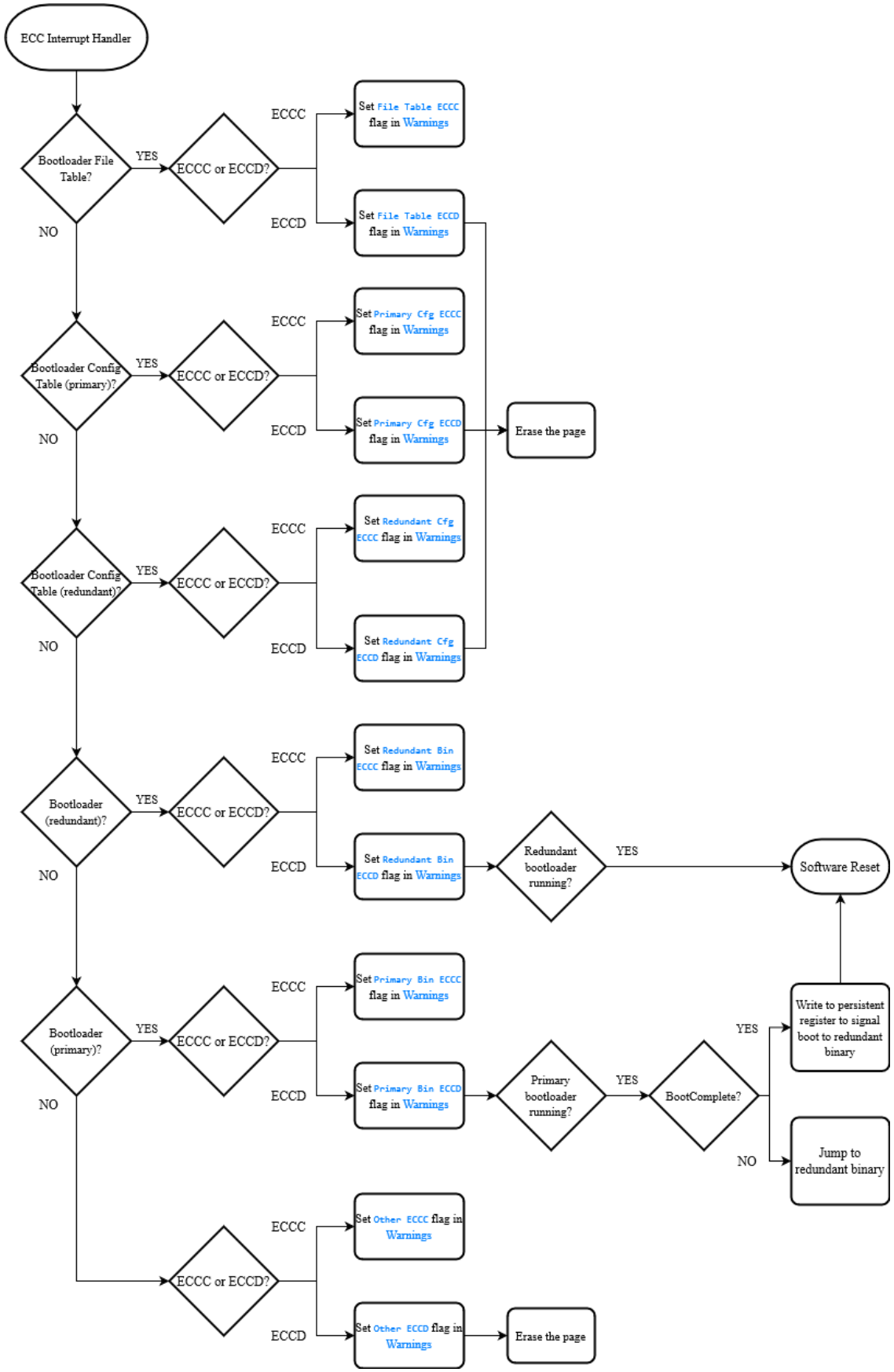

Figure 2: ECC Interrupt Handler Program Flow

Table 4: Summary of action taken in the event of ECC errors

FLASH REGION	ECC ERROR TYPE	INTERNAL ACTION
Bootloader (primary)	ECCC	Set the `Primary Bin ECCC` flag in `Warnings`.
	ECCD	Set the `Primary Bin ECCD` flag in `Warnings`. Jump to the redundant binary. There are two ways that the jump can occur. These are detailed bellow.
Bootloader (redundant)	ECCC	Set the `Redundant Bin ECCC` flag in `Warnings`.
	ECCD	Set the `Redundant Bin ECCD` flag in `Warnings`. If the primary binary is executing, there is no further action. If the redundant binary is executing, it implies that the primary binary is already corrupt. Since this scenario is highly unlikely, the only action taken is to perform a software reset.
Bootloader Config Table (primary)	ECCC	Set the `Primary Cfg ECCC` flag in `Warnings`.
	ECCD	Set the `Primary Cfg ECCD` flag in `Warnings`. Immediately erase the page. Normal operation continues. The `Primary Cfg ECCD` flag will be cleared after the CubeComputer is reset, and the `Primary Cfg CRC` flag will be set, because the primary config was erased. The redundant configuration will be used if it is valid, otherwise the default configuration is used.
Bootloader Config Table (redundant)	ECCC	Set the `Redundant Cfg ECCC` flag in `Warnings`.
	ECCD	Set the `Redundant Cfg ECCD` flag in `Warnings`. Immediately erase the page. Normal operation continues. The `Redundant Cfg ECCD` flag will be cleared after the CubeComputer is reset, and the `Redundant Cfg CRC` flag will be set, because the redundant config was erased. The primary configuration will be used if it is valid, otherwise the default configuration is used.
Bootloader File Table	ECCC	Set the `File Table ECCC` flag in `Warnings`.
	ECCD	Set the `File Table ECCD` flag in `Warnings`. Immediately erase the page. This only erases the file table, and not the files themselves. The effect of erasing the file table is that the `Jump To Application` and `Jump To Default Application` commands will no longer work because the bootloader thinks the files do not exist. Likewise, the bootloader will not automatically jump to the default app at the end of backoff. The user can still command the bootloader to jump to the control program by using the `Jump To Address` command, however the user must request the required value of `Address` from CubeSpace before attempting the jump.
Other (includes control program regions)	ECCC	Set the `Other ECCC` flag in `Warnings`.
	ECCD	Set the `Other ECCD` flag in `Warnings`. Immediately erase the page. If this occurs, it is most likely the control program binary, or the control program config that is corrupt. In either case, the effect of erasing the page is not critical. If a page of the control program binary is erased, and the bootloader jumps to it, a hard fault will occur, and eventually a watchdog reset. If the control program config is erased and the bootloader jumps to the control program, the program will assert and immediately reset.

It can be noted from the above Figure 2 and Table 4, that ECCC errors are only reported, and that for all ECCD errors that occur outside of the bootloader binary, the corrupt page is immediately erased. If the bootloader fails to erase the page, it will set the `Erase ECCD` flag in the `Warnings` telemetry.

The details of handling ECCD errors that occur within the bootloader binary are as follows. All ECC errors are handled in a Non-Maskable-Interrupt handler, and the code for this interrupt handler is contained within the flash. Therefore, if the ECC error occurs within the interrupt handler itself, the bootloader will experience infinite recursion. There is no way to combat this, as there is always some code that needs to handle the errors. Some mitigation has been put in place to allow for two program paths that can perform the jump to the redundant binary. The two code paths are as follows:

1. The code to perform the jump is placed at the start of program execution. A “BootComplete“ flag is used to indicate if this code has run to completion, meaning, the code is reachable and not corrupt, and can be relied on to perform the jump. If the primary image is executing and an ECCD error is detected in the primary image, the interrupt handler will check the BootComplete flag; if the boot completed (BootComplete=TRUE), then the handler will write to persistent registers to signal that the redundant binary must be booted, then it performs a software reset. On the next boot, the persistent register is read, and if the signal to boot the redundant image is detected, it will be jumped to. This is the preferred method as it allows operation to continue as normal.
2. The code to perform the jump, at the start of program execution, is not reachable. i.e. The ECCD error occurs before or during the execution of that code (BootComplete=FALSE). In this scenario, it is not possible to boot the redundant binary after a software reset. Therefore, the redundant image must be jumped to from the interrupt handler. This scenario has consequences, in that if we jump to the redundant binary from the interrupt handler, the microcontroller thinks the interrupt is still active. The effects of this have been nullified in the redundant binary, so that it can function as normal, however the control program can no longer be executed, because it uses an RTOS that does not allow initialization within an interrupt context. In this scenario, when the redundant binary boots, it will detect that the interrupt from the primary binary is still active and will signal this to the user by setting the `Redundant Bin Int Lock` flag in the `Warnings` telemetry. In this scenario, it is still safe to jump to the control program, as it will simply assert and reset back to the bootloader. A bootloader upgrade must be performed to restore normal operation.

The user can detect that the redundant binary is running via the `Redundant Bin Running` flag in the `Warnings` telemetry. If `TRUE`, the redundant binary is running. If `FALSE`, the primary binary is running.

The recommended action that should be taken by the user in the event of any of the conditions mentioned in this section, are listed in Table 5. The table references the flags in the `Warnings` telemetry.

Table 5: Recommended user action for ECC-related errors

WARNING FLAG	RECOMMENDED ACTION
Redundant Bin Running	The redundant binary is running. This implies that the primary binary is corrupt (check Primary Bin CRC and Primary Bin ECCD). The primary binary can only be recovered by performing a bootloader upgrade. If the bootloader is not upgraded, and the redundant binary also becomes corrupt, there is no way to recover. The user must assess the risk of performing a bootloader upgrade, compared to the risk of the redundant binary becoming corrupt (likely due to radiation).
Redundant Bin Int Lock	The redundant binary is running. This implies that the primary binary is corrupt (check Primary Bin CRC and Primary Bin ECCD). The user should perform the same action as for Redundant Bin Running, except that a bootloader upgrade is immediately required because the control program will be unusable.
Primary Bin CRC	This implies that the redundant binary is running, because only the redundant binary performs a CRC check of the primary binary. If this flag is set, it is highly likely that the Primary Bin ECCD flag will also be set. The user should perform the same action as for Redundant Bin Running.
Primary Bin ECCD	This implies that the redundant binary is running, because if an ECCD error occurs in the primary binary, it immediately jumps to the redundant binary. The redundant binary then performs a CRC check of the primary binary, which would have triggered this ECCD error. The user should perform the same action as for Redundant Bin Running.
Primary Bin ECCC	A single ECC error is automatically corrected in hardware, and the program can execute as normal. The user does not need to act. It is highly unlikely that another bit flip will occur in the same location, but in that scenario, an ECCD error will occur, and the redundant bootloader will be executed. It is riskier to do a bootloader upgrade, to correct the ECCC error, than for this to escalate to an ECCD error.
Redundant Bin CRC	This implies that the primary binary is running, because only the primary binary performs a CRC check of the redundant binary. If this flag is set, it is highly likely that the Redundant Bin ECCD flag will also be set. In this scenario, the redundant binary can no longer be relied on. The redundant binary can only be recovered by performing a bootloader upgrade. If the bootloader is not upgraded, and the primary binary also becomes corrupt, there is no way to recover. The user must assess the risk of performing a bootloader upgrade, compared to the risk of the primary binary becoming corrupt (likely due to radiation).
Redundant Bin ECCD	This implies that the primary binary is running, because if an ECCD error occurs in the redundant binary, and the redundant binary is running, there is no way to recover. When the primary binary boots, it performs a CRC check of the redundant binary, which would have triggered this ECCD error. The user should perform the same action as for Redundant Bin CRC.
Redundant Bin ECCC	A single ECC error is automatically corrected in hardware, and the program can execute as normal. The user does not need to act. It is highly unlikely that another bit flip will occur in the same location, but in that scenario, an ECCD error will occur, see Redundant Bin ECCD. It is riskier to do a bootloader upgrade, to correct the ECCC error, than for this to escalate to an ECCD error.
Primary Cfg CRC	The primary config is corrupt. If the redundant config is not corrupt, the user should request the Config telemetry and then set the Config using the returned values. If the redundant config is also corrupt, then the bootloader is using the default config, and the user will need to use the default config communications settings to set the desired config.
Primary Cfg ECCD	The primary config is corrupt. The user should perform the same action as for Primary Cfg CRC.
Primary Cfg ECCC	No action required.
Redundant Cfg CRC	The redundant config is corrupt. If the primary config is not corrupt, the user should request the Config telemetry and then set the Config using the returned values. If the primary config is also corrupt, then the bootloader is using the default config, and the user will need to use the default config communications settings to set the desired config.
Redundant Cfg ECCD	The redundant config is corrupt. The user should perform the same action as for Redundant Cfg CRC.

`Redundant Cfg ECCC`	No action required.
`File Table ECCD`	The file table is corrupt. The user can confirm this by requesting `File Info` and noting that `Empty`=`TRUE`. The user will need to upload the control program files to restore the entries of the file table.
`File Table ECCC`	No action required.
`Other ECCD`	Most likely, one of the control program files are corrupt. The user can confirm this by requesting the `File Info` telemetry, and checking if `File Corrupt`=`TRUE`, for each file. The user will need to upload the corrupt file to restore it.
`Other ECCC`	No action required.
`Erase ECCD`	This flag will only be set if an ECCD error occurs outside of the bootloader binary, and the attempt to erase the faulty page failed. In this scenario, the user should contact CubeSpace, providing the `Warnings` telemetry response.

3 Integration Testing

This chapter describes the key areas to focus on during integration testing. They are:

- Stability of communication interface(s)
- File upload implementation
- Recovery from configuration corruption
- Recovery from ECC errors (CubeComputer only)

The main area of focus is stability of the communication interface(s). The bootloader is a bare-metal application and may be overwhelmed by excessive communication. The amount of communication with the bootloader can be managed when it is known that the bootloader is running, however, if an unexpected reset occurs, the bootloader may need to process a large amount of communication that is intended for the control program application. The bootloader is not expected to fail in this case and will jump to the control program at the end of the backoff period, however, this scenario must be tested during integration, with all subsystems active.

The core functionality of the bootloader is to program firmware and configuration files. The user must test their implementation thoroughly, after the CubeProduct is integrated in the system. This includes the procedure to reset to the bootloader and halting it. The file upload procedure is a communication intensive task on its own, and the bootloader may have difficulty processing unrelated communication while a file upload is in progress. It is recommended that all non-related communication with the CubeProduct is stopped while an upload is in progress.

The bootloader configuration is not expected to change in orbit, and the bootloader does not write to the flash region where the configuration is stored, unless it is commanded to set a new configuration. Therefore, corruption of the configuration is highly unlikely. However, the user must ensure that the desired configuration can be recovered should corruption occur. See Appendix: Testing Configuration Corruption Recovery, or for CubeComputer see section 3.1.

3.1 Flash ECC Error Recovery Testing (CubeComputer only)

The behaviour of ECC error handling is described in section 2.11.2. This section describes how the user can test all important cases of ECC errors.

The bootloader exposes a telecommand, `Simulate ECC Error`, which allows the user to simulate ECC errors without corrupting the flash. This section describes how to induce flags in the `Warnings` telemetry. The user can reference Table 5 for the required action to take for each warning.

The required parameters of `Simulate ECC Error`, to induce specific warning flags, are listed in Table 6. In all cases, the value of `MagicNumber` should be 777777777.

Table 6: Simulated ECC Error Values

WARNING FLAG	EccSim PARAMETERS	COMMENTS
Redundant Bin Running	ADDR_ECC = 0x00000 – 0x14FFF BK_ECC = FALSE ECCD = TRUE ECCC = FALSE Boot Incomplete = TRUE (Redundant Bin Int Lock = TRUE) Boot Incomplete = FALSE (Redundant Bin Int Lock = FALSE)	Induce an ECCD error within the primary binary. The range of ADDR_ECC is anywhere within the primary binary. The value of Boot Incomplete can be either TRUE or FALSE depending on how the user wants the primary bootloader to jump to the control program (see section 2.11.2).
Redundant Bin Int Lock	ADDR_ECC = 0x00000 – 0x14FFF BK_ECC = FALSE ECCD = TRUE ECCC = FALSE Boot Incomplete = TRUE	Induce an ECCD error within the primary binary. This is the same as for Redundant Bin Running, except Boot Incomplete must be TRUE, to signal that the redundant binary must be jumped to from the interrupt handler.
Primary Bin CRC	N/A	It is not possible to simulate the CRC error.
Primary Bin ECCD	Same as Redundant Bin Running.	It is not possible to set this flag via simulation, because there is not actually an ECCD error. Therefore, when the jump to the redundant binary happens, the flag is lost. During an actual ECCD error, the redundant binary would CRC the primary binary, which would induce this error so that this flag would be set. For simulation, it is sufficient to note that Redundant Bin Running = TRUE.
Primary Bin ECCC	ADDR_ECC = 0x00000 – 0x14FFF BK_ECC = FALSE ECCD = FALSE ECCC = TRUE Boot Incomplete = N/A	Induce an ECCC error within the primary binary. The range of ADDR_ECC is anywhere within the primary binary.
Redundant Bin CRC	N/A	It is not possible to simulate the CRC error.
Redundant Bin ECCD	ADDR_ECC = 0x15000 – 0x29FFF BK_ECC = FALSE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the redundant binary. The range of ADDR_ECC is anywhere within the redundant binary. The primary binary must be running for this simulation. If the redundant binary is running and these parameters are used, the bootloader will software reset.
Redundant Bin ECCC	ADDR_ECC = 0x15000 – 0x29FFF BK_ECC = FALSE ECCD = FALSE ECCC = TRUE	Induce an ECCC error within the redundant binary. The range of ADDR_ECC is anywhere within the redundant binary.

	Boot Incomplete = N/A	
Primary Cfg CRC	ADDR_ECC = 0xFE000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the primary config. The page will be erased. This works in tandem with the Primary Cfg ECCD flag. At first only the Primary Cfg ECCD flag will be set. A soft reset must be commanded for the CRC validation to happen to set this flag.
Primary Cfg ECCD	ADDR_ECC = 0xFE000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the primary config. The page will be erased.
Primary Cfg ECCC	ADDR_ECC = 0xFE000 BK_ECC = TRUE ECCD = FALSE ECCC = TRUE Boot Incomplete = N/A	Induce an ECCC error within the primary config.
Redundant Cfg CRC	ADDR_ECC = 0xFD000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the redundant config. The page will be erased. This works in tandem with the Redundant Cfg ECCD flag. At first only the Redundant Cfg ECCD flag will be set. A soft reset must be commanded for the CRC validation to happen to set this flag.
Redundant Cfg ECCD	ADDR_ECC = 0xFD000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the redundant config. The page will be erased.
Redundant Cfg ECCC	ADDR_ECC = 0xFD000 BK_ECC = TRUE ECCD = FALSE ECCC = TRUE Boot Incomplete = N/A	Induce an ECCC error within the redundant config.
File Table ECCD	ADDR_ECC = 0xFF000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error within the file table. The page will be erased.
File Table ECCC	ADDR_ECC = 0xFF000 BK_ECC = TRUE	Induce an ECCC error within the file table.

	ECCD = FALSE ECCC = TRUE Boot Incomplete = N/A	
Other ECCD	ADDR_ECC = 0x00000 BK_ECC = TRUE ECCD = TRUE ECCC = FALSE Boot Incomplete = N/A	Induce an ECCD error in the first page of the control program binary. The page will be erased.
Other ECCC	ADDR_ECC = 0x00000 BK_ECC = TRUE ECCD = FALSE ECCC = TRUE Boot Incomplete = N/A	Induce an ECCC error in the first page of the control program binary.

The user can, and is encouraged to, make use of the Simulate ECC Error telecommand in any situation (e.g. during a bootloader upgrade, when the bootloader is in flash bank 2). Additionally, the user is encouraged to test power failures during flash write sequences, such as persisting configuration. It is unlikely that an ECC error will be induced is such cases, so the extent of this kind of testing is up to user discretion.

4 Appendix: State Machine Diagram

In the following state machine diagram, all states transition to the idle state on any error. Only the non-error transitions to idle are shown.

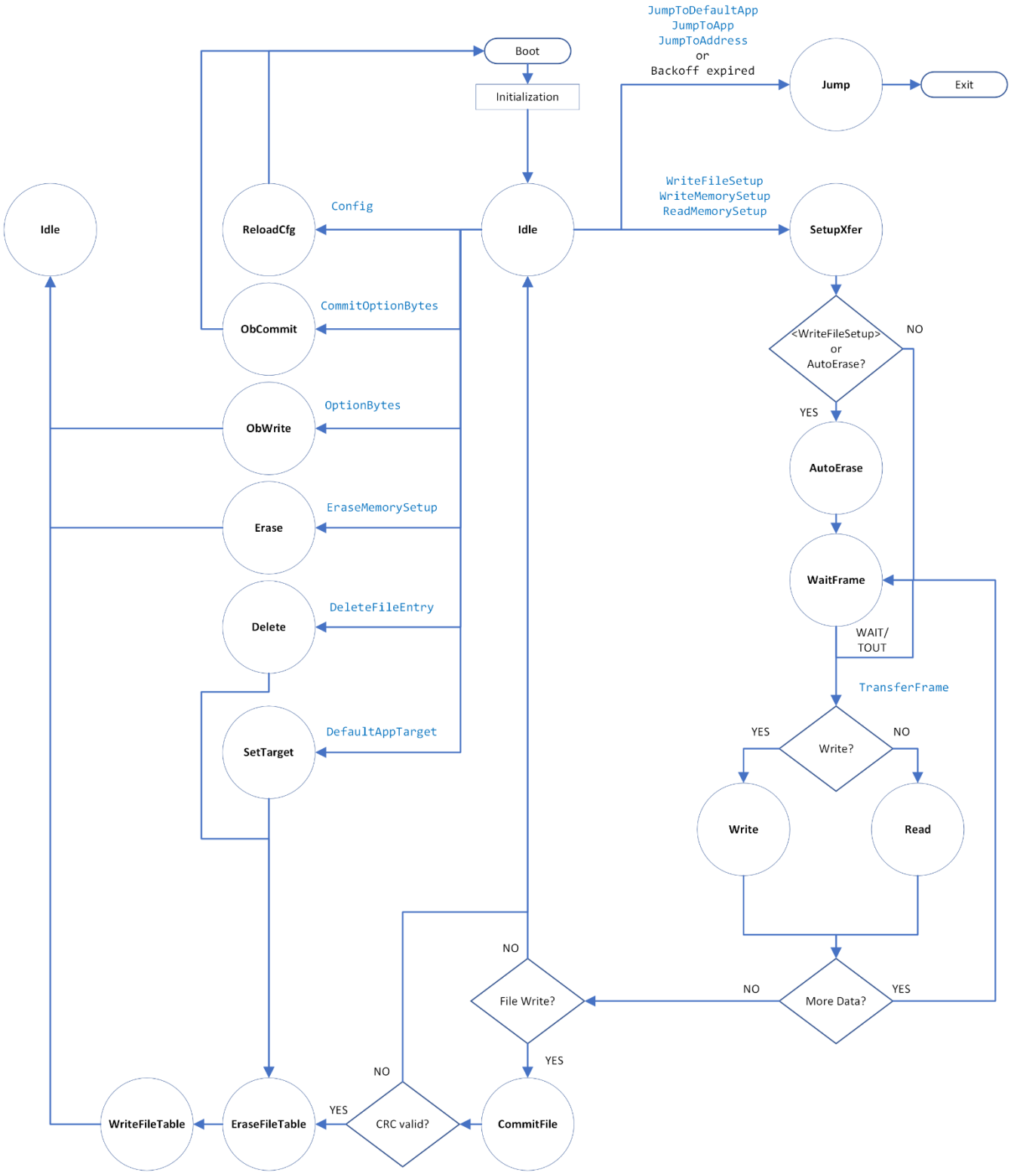

Figure 3: State Machine Diagram

5 Appendix: Bootloader Interface via CubeSupport

This appendix describes how to upload firmware and configuration files to the bootloader using the CubeSupport application, refer to [3] for more information on the CubeSupport application. The implementation of the CubeSupport application follows section 2.7 and 2.8.

This appendix assumes that the CubeSupport application is currently connected to the bootloader application, see [3] for information on how to connect.

Navigate to the “Firmware/Config Upload” tab. The user interface (UI) is shown in Figure 4.

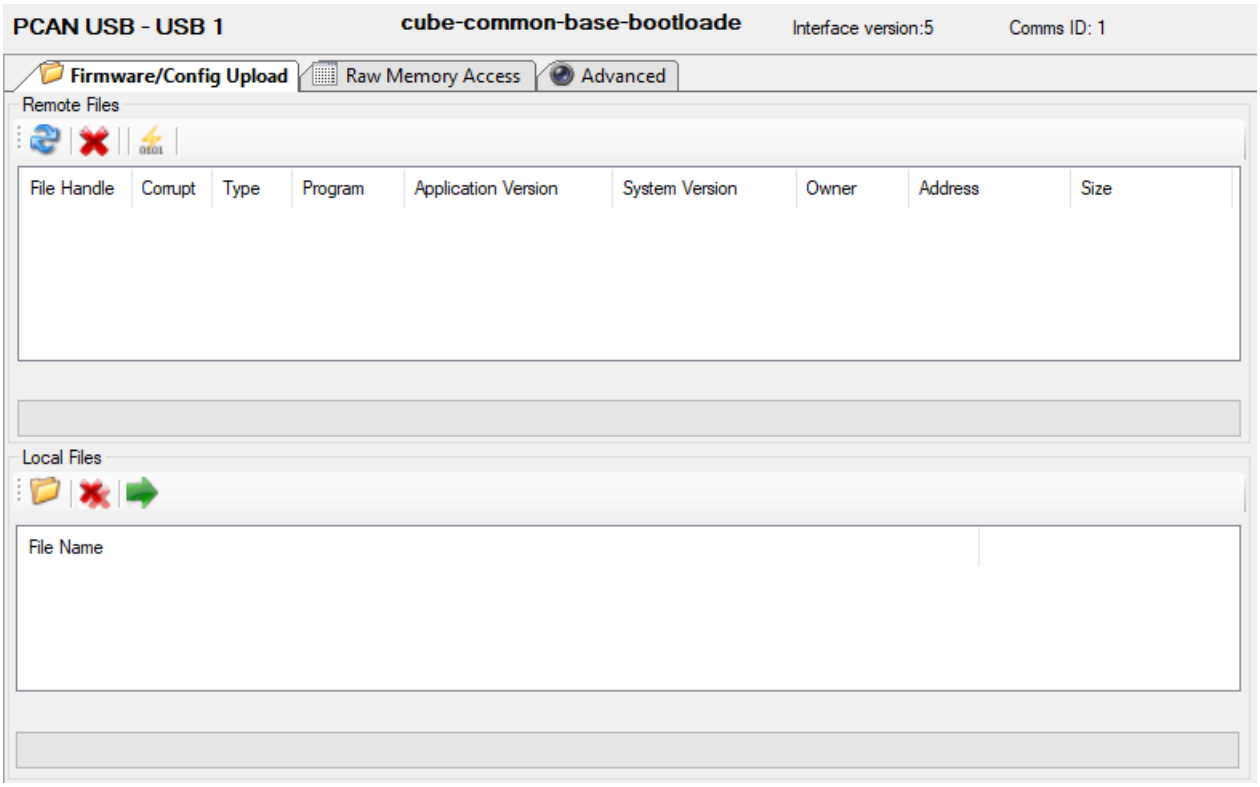

Figure 4: Firmware/Config Upload UI

The local files group is used to load local CubeSpace (.cs) files and stage them for upload. Once staged, the files that the user wants to upload can be selected and then uploaded. The progress bar at the bottom of the group shows the progress of the upload of each individual file. When the batch of uploads is complete a message box appears indicating a successful upload of all files, or individual message box(s) for each failed upload.

The remote files group is used to view files already uploaded. The refresh button can be used to request file information for all files. The file list is automatically refreshed after every batch of file uploads. Once the user has confirmed that the required files are present, the user can then select the binary file of the application to jump to, then click the jump button.

The user may hover the mouse cursor over the buttons to see the tool-tip text on what the button is used for.

6 Appendix: Modifying bootloader configuration via CubeSupport

As described in 2.1.1 and 2.1.2, the default communications configuration of a CubeProduct can be modified to suit customer needs and/or to achieve the correct result in an integrated system. This is conveniently done using the CubeSupport application, by connecting to CubeProduct while the base bootloader is running. Please consult [3] for details.

From the Advanced tab, the Configuration set of settings can be requested and modified (see Figure 5).

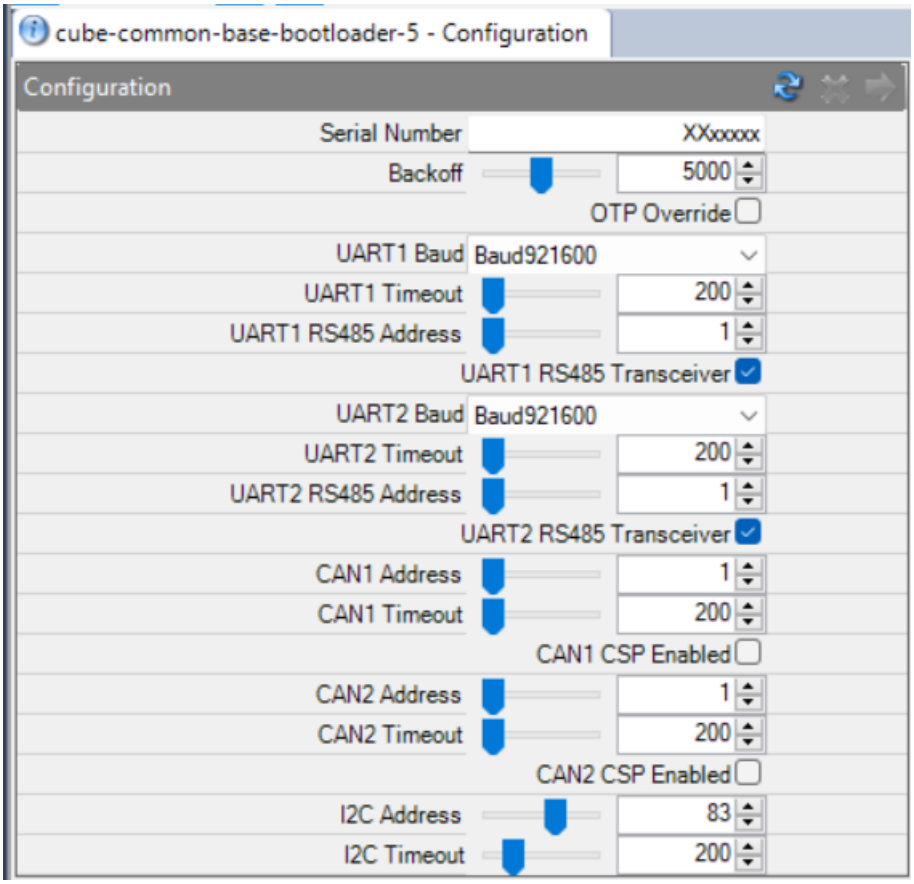

Figure 5: Bootloader configuration

The serial number should not have to be modified, since all CubeProducts have a “one-time programmable” (OTP) serial number which should already be programmed. For the same reason, the OTP override setting can be left unchecked.

The UART, CAN and I2C configuration settings can further be adjusted as needed. Note that not all settings may apply to a certain CubeProduct. CAN2 is only available on the CubeComputer, for instance, and not all CubeProducts offer an I2C interface. For these situations, the relevant settings will have no effect.

Since changing the configuration for the bootloader requires reprogramming of flash memory, the user must ensure that the CubeProduct has a stable power supply when the telecommand to change the configuration is sent.

Once the new, non-default configuration is successfully programmed, the bootloader will immediately issue a soft reset for the new configuration to take effect on the next boot.

7 Appendix: Testing Raw Memory Access Implementation

The procedure to test the user's implementation of the raw memory access functionality involves uploading a firmware binary file using raw memory access, instead of the normal file upload procedure, and reading the file back to compare with the uploaded file. Once the read/write functionality is confirmed to work, the user can then test the erase functionality by erasing the file and reading it back to confirm the flash is erased.

The user should not reset the bootloader between steps unless instructed to do so.

Step 1: Upload a firmware file using the normal procedure.

Follow section 2.7 to upload a firmware binary CubeSpace (.cs) file. The user may also upload the configuration file, jump to the application, and ensure that the programmed application is working. This will create a file entry for the uploaded file.

Step 2: Read back the uploaded file.

Follow section 2.7.2 to request file information for the file(s) that was uploaded in step 1. Note the address and the size of the file. Use the address and size to read the file back, by following section 2.9.2.

Step 3: Compare the read data to the file uploaded.

The data that was read back should match the uploaded file data, without the metadata. Therefore, the user should compare the read data with the data portion of the CubeSpace file that was uploaded. For information on CubeSpace files, refer to [2].

Step 4: Write the file back to flash.

The user can either use the data read in **step 2**, or the data portion of the original CubeSpace file. Write this data to the address where it was read from in **step 2**, by following section 2.9.1. Note that this does not create an entry in the file table.

Step 5: Confirm the file is not seen as corrupt by the bootloader.

The bootloader is not "aware" that the data written in **step 4** is a file. However, the bootloader should not have been reset during this process, and the entry in the file table for this file should still be present, from **step 1**, and deemed not to be corrupt. Now, If the data written in **step 4** is correct, the CRC validation for the file entry created in **step 1** should pass. The user can confirm this by requesting file information. If the user had previously jumped to the application after **step 1**, the user may attempt to jump to it again.

Step 6: Erase the file data.

Use the address and size to erase the file data by following section 2.9.3. The user can then either read back the file and confirm all data is erased or issue a soft reset and confirm that the file is now corrupt by requesting file information.

8 Appendix: Testing Configuration Corruption Recovery

This section is **not** applicable to CubeComputer. To test configuration corruption for CubeComputer, follow section 3.1.

This appendix describes how to artificially induce configuration corruption, such that the bootloader will revert to using the default configuration. Thus, the user may test their implementation of the configuration recovery to the desired values.

The testing process involves using the erase memory feature (see 2.9.3) to erase the current configuration.

Step 1: Ensure that the current configuration is not the same as the default configuration.

This can be done using the CubeSupport app. The most easily noticeable configuration item would be the serial number string.

Step 2: Erase the configuration.

Use the `Erase Memory Setup` command to erase the configuration. If using the CubeSupport app, this command is only in the "Advanced" tab. The setup parameters should be as follows:

- CubeAuriga:
  `Address` = 134316032 (0x8018000)
  `Size` = 1
- CubeStar: (R5 bootloader)
  `Address` = 136298496 (0x81FC000)
  `Size` = 1
- All other CubeProducts (52 bootloader)
  `Address` = 134737920 (0x807F000)
  `Size` = 1

Step 3: Confirm that the erase was successful.

Request the `State` telemetry. The parameters should be as follows:

- `App State` = `StateIdle`
- `Previous App State` = `StateBusyErase`
- `Error Code` = 0. If this is non-zero, report to CubeSpace, and retry step 2.
- All other parameters are don't care.

Step 4: Command a soft reset.

This will cause the bootloader to reload the configuration. At this point the CRC of the configuration will fail, and the bootloader will assume the default configuration.

Step 5: Confirm the configuration is now set to the default values.

Request the `Config` telemetry.

Step 6: Recover the desired configuration.

This is implementation specific.

The most noticeable difficulty with recovering the configuration is that the CubeProduct may no longer be using the expected protocol. Implementing multiple protocols to handle this one scenario may not be desired, therefore it is recommended that the configuration recovery be implemented using hard-coded raw bytes that have been encoded using the default protocol on a given interface. It is recommended that the

encoded bytes of the `Identification` telemetry request be stored, as this can be used as a sign-of-life, and confirmation that the loss of communication is indeed caused by configuration corruption. The desired configuration should be stored as the encoded bytes of the `Config` command, using the default protocol. These raw bytes can then be written directly to the interface peripheral. See [2] for information on the communication protocols.