CUBESPACE

CubeProduct Firmware Reference Manual

Low-Level Firmware Interface

DOCUMENT NUMBER	CS-DEV.FRM.CA-01
VERSION	7.02
DATE	14/02/2024
PREPARED BY	C. LEIBBRANDT
REVIEWED BY	C.L.
APPROVED BY	C. Leibbrandt
DISTRIBUTION LIST	External

Revision History

VERSION	AUTHORS	DATE	DESCRIPTION
1.00	O. Gries	07/12/2021	First draft
2.00	O. Gries	30/07/2021	Cubesat Space Protocol
3.00	F. Lotter	10/09/2021	Fixed RS485 byte layout
4.0A	O. Gries	14/02/2022	New Template, CAN protocol, Error Logging, Config
5.00	C. Leibbrandt	07/03/2022	Incorporating CubeToolbox details originally in Quick start guide
6.00	L. Visagie, O. Gries	18/11/2022	Add details for unsolicited event encoding, telemetry log encoding and common download telemetry and telecommand sequence
6.01	O. Gries	08/02/2023	Add Nack for invalid passthrough target. Update reference to Bootloader Application Manual.
7.00	O. Gries	23/03/2023	New template. Add bootloader file uploads to bulk data transfer section. Remove cube-toolbox sections.
7.01	O. Gries	05/06/2023	Added RS485 TCTLM passthrough details.
7.02	O.Gries	14/02/2024	Migrated file uploads to bootloader to be compatible with new base bootloader. Added section for uploading files to CubeComputer control program.

Reference Documents

The following documents are referenced in this document.

[1]	CS-DEV-AMNL.BL-01	Bootloader Application Manual, Ver 3.00, 23/03/2023
[2]	N/A	common-framework-enums-1.xml
[3]	N/A	cube-common-1-base-bootloader-5-api-reference.html
[4]	N/A	cube-computer-5-control-program-8-api-reference.html

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
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
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 document contains low-level descriptions of the functionality and interfaces that are common to all CubeADCS components. The relevance of this document will be highlighted in the component's specific documentation. Items that are specific to a CubeADCS component can be found in the documentation for that component. Specific documents will reference this document where it is applicable.

2 Communication Protocols

This section describes the communication protocols that may be used by CubeADCS components.

All CubeADCS components act as slaves in the described protocols.

Communication flow is shown in Figure 1 and Figure 2.

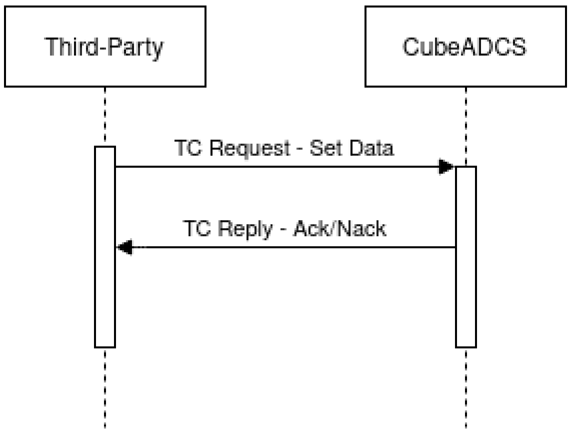


Figure 1: Telecommand Communication Flow

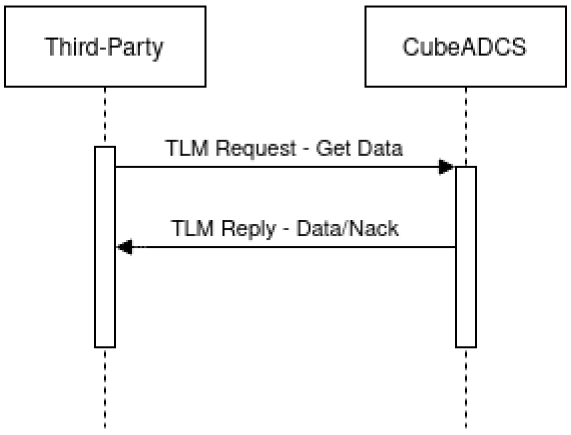


Figure 2: Telemetry Communication Flow

2.1 TCTLM ID's

TCTLM IDs are unique integer values that identify a Telecommand or Telemetry type. These IDs are mapped differently for each CubeADCS component. Refer to the specific components' documentation for a list of its supported TCTLM's.

Valid TCTLM ID ranges are shown in Table 1.

Table 1: Valid TCTLM IDs

TCTLM	ID RANGE
TC	0 – 127
TLM	128 - 254

2.2 Errors

Protocol transport errors are common between all protocols and will be reported in the "***error byte***" field in the event of a "***Nack***" message.

All possible error byte values are shown in Table 2.

Table 2: Communication Protocol Error Byte Values

VALUE	NAME	DESCRIPTION
0	No Error	No Error / Ack
1	Invalid ID	The TCTLM ID provided is invalid
2	Incorrect Length	The payload length is invalid for the provided TCTLM ID
3	Invalid Parameters	One or more parameters parsed from the payload is invalid
4	CRC Error	[TBD]
5	Not Implemented	The TCTLM for the provided ID is not implemented
6	Firmware Busy	The firmware is busy and cannot process the request
7	Command Sequence Error	The command is not valid for the current firmware state
8	Internal Error	The TCTLM was unable to be processed due to an internal error. This is a general error, and the cause will vary for each TCTLM. The exact cause, if applicable, will be listed in each CubeProducts specific documentation.
9	Pass-through timeout	Routing of a pass-through message to the target node timed out. This is only applicable to CubeComputer.
10	Pass-through target	An attempt was made to send a pass-through message, but the pass-through target is invalid (not set).

2.3 UART

The UART protocol is a serial point-to-point protocol. It requires a single master/slave pair on a single and exclusive bus.

2.3.1 Special Characters

The UART protocol makes use of special characters to segment and parse packets. The characters are described in Table 3.

Table 3: UART protocol special characters

NAME	ACRONYM	VALUE	DESCRIPTIOM
Escape	ESC	0x1F	Used as a precursor to any special characters or data bytes that have a value of 0x1F.
Start of Message	SOM	0x7F	Indicates the start of a master-to-slave message.
Start of Reply	SOR	0x07	Indicates the start of a slave-to-master Ack reply.

NAME	ACRONYM	VALUE	DESCRIPTIOM
Start of Nack	SON	0x0F	Indicates the start of a slave-to-master Nack reply.
Start of Unsolicited Event message	SOE	0x2F	Indicates the start of slave-to-master unsolicited event message (CubeComputer only).
Start of Unsolicited Telemetry message	SOU	0x4F	Indicates the start of slave-to-master unsolicited telemetry message (CubeComputer only).
Start of Pass-through message	SOMP	0x7E	Indicates the start of a master-to-slave(node) pass-through message (CubeComputer only).
Start of Pass-Through reply	SORP	0x06	Indicates the start of a slave(node)-to-master pass-through Ack reply (CubeComputer only).
Start of Pass-through Nack	SONP	0x0E	Indicates the start of a slave(node)-to-master pass-through Nack reply (CubeComputer only).
End of Message	EOM	0xFF	Indicates the end of a message or reply.

The escape character is used to distinguish protocol bytes from payload/data bytes. Therefore, any byte in the payload with a value of 0x1F must be “escaped”, including the TCTLM ID. i.e. 0x1F becomes 0x1F1F.

2.3.2 Frame Structure

The UART protocol frame structures are shown in the tables following.

Table 4: UART Protocol Frame Structure - Telecommand

HEAD		ID	PAYLOAD	TAIL	
ESC	*SOM*	*TC ID*	*Data*	*ESC*	*EOM*
0x1F	0x7F	0 to 127	[bytes]	0x1F	0xFF

Table 5: UART Protocol Frame Structure – Telecommand Ack

HEAD		ID	PAYLOAD	TAIL	
ESC	*SOR*	*TC ID*	*Error Byte*	*ESC*	*EOM*
0x1F	0x07	0 to 127	0x00	0x1F	0xFF

Table 6: UART Protocol Frame Structure – Telecommand Nack

HEAD		ID	PAYLOAD		TAIL	
ESC	*SON*	*TC ID*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x0F	0 to 127	> 0	[TBD]	0x1F	0xFF

Table 7: UART Protocol Frame Structure – Telemetry Request

HEAD	ID	TAIL

ESC	*SOM*	*TLM ID*	*ESC*	*EOM*
0x1F	0x7F	128 to 254	0x1F	0xFF

Table 8: UART Protocol Frame Structure – Telemetry Response

HEAD		ID	PAYLOAD	TAIL	
ESC	*SOR*	*TLM ID*	*Data*	*ESC*	*EOM*
0x1F	0x07	128 to 254	[bytes]	0x1F	0xFF

Table 9: UART Protocol Frame Structure – Telemetry Nack

HEAD		ID	PAYLOAD		TAIL	
ESC	*SON*	*TLM ID*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x0F	128 to 254	> 0	[TBD]	0x1F	0xFF

2.3.3 Frame Structure Pass-through

This section is only applicable to CubeComputer. Read through section 5.2 for information regarding pass-through communication.

The UART protocol frame structures for pass-through communication are shown in the tables following.

Table 10: UART Protocol Frame Structure – Pass-through Telecommand

HEAD		ID	PAYLOAD	TAIL	
ESC	*SOMP*	*TC ID*	*Data*	*ESC*	*EOM*
0x1F	0x7E	0 to 127	[bytes]	0x1F	0xFF

Table 11: UART Protocol Frame Structure – Pass-through Telecommand Ack

HEAD		ID	PAYLOAD	TAIL	
ESC	*SORP*	*TC ID*	*Error Byte*	*ESC*	*EOM*
0x1F	0x06	0 to 127	0x00	0x1F	0xFF

Table 12: UART Protocol Frame Structure – Pass-through Telecommand Nack

HEAD		ID	PAYLOAD		TAIL	
ESC	*SONP*	*TC ID*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x0E	0 to 127	> 0	[TBD]	0x1F	0xFF

Table 13: UART Protocol Frame Structure – Pass-through Telemetry Request

HEAD	ID	TAIL

ESC	*SOMP*	*TLM ID*	*ESC*	*EOM*
0x1F	0x7E	128 to 254	0x1F	0xFF

Table 14: UART Protocol Frame Structure – Pass-through Telemetry Response

HEAD		ID	PAYLOAD	TAIL	
ESC	*SORP*	*TLM ID*	*Data*	*ESC*	*EOM*
0x1F	0x06	128 to 254	[bytes]	0x1F	0xFF

Table 15: UART Protocol Frame Structure – Pass-through Telemetry Nack

HEAD		ID	PAYLOAD		TAIL	
ESC	*SONP*	*TLM ID*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x0E	128 to 254	> 0	[TBD]	0x1F	0xFF

2.4 RS485

The RS485 protocol is a serial multidrop protocol. It allows for a single master to communicate with multiple receivers on a multidrop bus.

2.4.1 Special Characters

The RS485 protocol makes use of special characters to segment and parse packets. The characters are described in Table 16.

Table 16: RS485 Protocol Special Characters

NAME	ACRONYM	VALUE	DESCRIPTION
Escape	ESC	0x1F	Used as a precursor to any special characters or data bytes that have a value of 0x1F.
Start of Message	SOM	0x80	Indicates the start of a master-to-slave message.
Start of Reply	SOR	0x08	Indicates the start of a slave-to-master Ack reply.
Start of Nack	SON	0x10	Indicates the start of a slave-to-master Nack reply.
Start of Pass-through message	SOMP	0x81	Indicates the start of a master-to-slave(node) pass-through message (CubeComputer only).
Start of Pass-Through reply	SORP	0x09	Indicates the start of a slave(node)-to-master pass-through Ack reply (CubeComputer only).
Start of Pass-through Nack	SONP	0x11	Indicates the start of a slave(node)-to-master pass-through Nack reply (CubeComputer only).
End of Message	EOM	0xFF	Indicates the end of a message or reply.

The escape character is used to distinguish protocol bytes from payload/data bytes. Therefore, any byte in the payload with a value of 0x1F must be "escaped", including the TCTLM ID. i.e. 0x1F becomes 0x1F1F.

2.4.2 Frame structure

The RS485 protocol differs from the UART protocol in that there is now a source and destination address in the packet for routing data.

Valid address ranges are shown in Table 17.

Table 17: RS485 Protocol Valid Address Ranges

ADDRESS	VALID RANGE
Source	1-255
Destination	0-255

A destination address, 0 (zero), is reserved for broadcast messages.

The RS485 protocol frame structures are shown in the tables following.

Table 18: RS485 Protocol Frame Structure - Telecommand

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	*SOM*	*TC ID*	*SRC*	*DST*	*Data*	*ESC*	*EOM*
0x1F	0x80	0 - 127	1 - 255	0 - 255	[bytes]	0x1F	0xFF

Table 19: RS485 Protocol Frame Structure – Telecommand Ack

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	SOR	TC ID	SRC	DST	Error Byte	ESC	EOM
0x1F	0x08	0 - 127	1 - 255	0 - 255	0x00	0x1F	0xFF

Table 20: RS485 Protocol Frame Structure – Telecommand Nack

HEAD		ID	ADDRESSING		PAYLOAD		TAIL	
ESC	*SON*	*TC ID*	*SRC*	*DST*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x10	0 - 127	1 - 255	0 - 255	> 0x00	[TBD]	0x1F	0xFF

Table 21: RS485 Protocol Frame Structure – Telemetry Request

HEAD		ID	ADDRESSING		TAIL	
ESC	*SOM*	*TLM ID*	*SRC*	*DST*	*ESC*	*EOM*
0x1F	0x80	128 - 254	1 - 255	0 - 255	0x1F	0xFF

Table 22: RS485 Protocol Frame Structure – Telemetry Response

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	*SOR*	*TLM ID*	*SRC*	*DST*	*Data*	*ESC*	*EOM*

0x1F	0x08	128 - 254	1 - 255	0 - 255	[bytes]	0x1F	0xFF

Table 23: RS485 Protocol Frame Structure – Telemetry Nack

HEAD		ID	ADDRESSING		PAYLOAD		TAIL	
ESC	*SON*	*TLM ID*	*SRC*	*DST*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x10	128 - 254	1 - 255	0 - 255	> 0x00	[TBD]	0x1F	0xFF

2.4.3 Frame structure Pass-through

This section is only applicable to CubeComputer. Read through section 5.2 for information regarding pass-through communication.

The UART protocol frame structures for pass-through communication are shown in the tables following.

Table 24: RS485 Protocol Frame Structure – Pass-through Telecommand

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	*SOMP*	*TC ID*	*SRC*	*DST*	*Data*	*ESC*	*EOM*
0x1F	0x81	0 - 127	1 - 255	0 - 255	[bytes]	0x1F	0xFF

Table 25: RS485 Protocol Frame Structure – Pass-through Telecommand Ack

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	SORP	TC ID	SRC	DST	Error Byte	ESC	EOM
0x1F	0x09	0 - 127	1 - 255	0 - 255	0x00	0x1F	0xFF

Table 26: RS485 Protocol Frame Structure – Pass-through Telecommand Nack

HEAD		ID	ADDRESSING		PAYLOAD		TAIL	
ESC	*SONP*	*TC ID*	*SRC*	*DST*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x11	0 - 127	1 - 255	0 - 255	> 0x00	[TBD]	0x1F	0xFF

Table 27: RS485 Protocol Frame Structure – Pass-through Telemetry Request

HEAD		ID	ADDRESSING		TAIL	
ESC	*SOMP*	*TLM ID*	*SRC*	*DST*	*ESC*	*EOM*
0x1F	0x81	128 - 254	1 - 255	0 - 255	0x1F	0xFF

Table 28: RS485 Protocol Frame Structure – Pass-through Telemetry Response

HEAD		ID	ADDRESSING		PAYLOAD	TAIL	
ESC	*SORP*	*TLM ID*	*SRC*	*DST*	*Data*	*ESC*	*EOM*

0x1F	0x09	128 - 254	1 - 255	0 - 255	[bytes]	0x1F	0xFF

Table 29: RS485 Protocol Frame Structure – Pass-through Telemetry Nack

HEAD		ID	ADDRESSING		PAYLOAD		TAIL	
ESC	*SONP*	*TLM ID*	*SRC*	*DST*	*Error Byte*	*Error Index*	*ESC*	*EOM*
0x1F	0x11	128 - 254	1 - 255	0 - 255	> 0x00	[TBD]	0x1F	0xFF

2.5 I2C

All CubeProducts act as slave devices on the external I2C bus, with 7-bit addressing. The 7-bit address for each CubeProduct is configurable.

All CubeProducts implement clock stretching to slow communication.

2.5.1 Frame Structure

The I2C protocol frame structures are shown in the tables following.

Table 30: I2C Protocol Frame Structure - Telecommand

HEAD			ID	PAYLOAD	TAIL
I2C start	*7-bit address*	*I2C write bit*	*TC ID*	*Data*	*I2C stop*
-	[7-bit address]	0	0 to 127	[bytes]	-

There is no direct Ack or Nack response to telecommands, because CubeProducts cannot initiate I2C communication as slaves. The Ack or Nack status of a preceding telecommand can be queried by requesting a specific telemetry frame – the `<TelecommandAcknowledge>` telemetry frame.

An I2C telemetry request is performed by first issuing an I2C write transaction to set the telemetry ID, followed by an I2C read transaction to retrieve the telemetry response. The total transaction may not exceed 300 milliseconds. If the timeout is exceeded, an I2C NACK is generated.

Table 31: I2C Protocol Frame Structure – Telemetry request

HEAD			ID	TAIL
I2C start	*7-bit address*	*I2C write bit*	*TLM ID*	*I2C stop*
-	[7-bit address]	0	0 to 127	-

Table 32: I2C Protocol Frame Structure – Telemetry response

HEAD			PAYLOAD	TAIL
I2C start	*7-bit address*	*I2C read bit*	*Data*	*I2C stop*
-	[7-bit address]	1	[bytes]	-

2.6 CAN

The CubeSpace CAN TCTLM protocol is a transport layer on top the CAN V2.0B hardware layers.

An 8-byte CAN packet of the CAN V2.0B hardware layers will be referred to as a single CAN frame from here onwards.

The protocol allows for multiple CAN frames to be chained to form a single TCTLM message, up to a maximum payload length of 256 bytes.

The protocol uses the 29-bit identifier for arbitration, routing, and message-type identification. TCTLM messages that require more than a single CAN frame, will be marked as such in the 29-bit identifier. In this scenario, a byte is reserved in each CAN frame to indicate the frame number. The structure of the 29-bit identifier is shown in Table 33.

Table 33: CAN Protocol 29-bit Identifier Structure

	MESSAGE TYPE	TCTLM ID	SOURCE ADDRESS	DESTINATION ADDRESS
Bit Index	28..24	23..16	15..8	7..0
Mask	0x1F	0xFF	0xFF	0xFF

Table 34: CAN Protocol Message Types

TYPE	VALUE	DESCRIPTION
Telecommand	1	Telecommand with data.
Telecommand Ack	2	Ack of telecommand.
Telecommand Nack	3	Nack of telecommand.
Telemetry Request	4	Telemetry request of data.
Telemetry Response	5	Telemetry response with data.
Telemetry Nack	6	Nack of telemetry request.
Telecommand Extended	7	Telecommand that requires multiple CAN frames.
Telemetry Response Extended	8	Telemetry response that requires multiple CAN frames.
Unsolicited Event	9	Unsolicited event message (sent as multiple CAN frames). Only applicable to CubeComputer.
Unsolicited Telemetry – First	10	First frame of an unsolicited telemetry message. Only applicable to CubeComputer.
Unsolicited Telemetry – Body	11	Neither the first nor the last frame of an unsolicited telemetry message. Only applicable to CubeComputer.
Unsolicited Telemetry – Last	12	Last frame of an unsolicited telemetry message. Only applicable to CubeComputer.

2.6.1 Extended Message Types

Extended message types are messages that require multiple CAN frames. Extended messages reserve the most significant byte of each CAN frame to indicate the number of frames remaining in the transaction, which starts at the total number of required frames, less one, and decrements to zero.

Hence, the number of required frames, and number of remaining frames, can be calculated as:

$$N = \frac{L}{7} + 1 \qquad N \overset{\text{def}}{=} Number\ of\ Frames$$

$$L \overset{\text{def}}{=} Length\ Data$$

$$k \overset{\text{def}}{=} Frame\ Number$$

$$n = N - k \qquad n \overset{\text{def}}{=} Frames\ Remaining$$

The position of the "*Frames Remaining*" field in the last frame is dependent on the length of the data. The parameter is always placed in the most significant byte. Hence, if two bytes are to be sent in the last frame the "*Frames Remaining*" parameter should be placed in the third byte.

The general structure of an extended message is shown in Table 35.

Table 35: CAN Protocol Extended Message Structure

FRAME NUMBER	29-BIT IDENTIFIER	DLC	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
$k_{1..N}$	--	8	n	D_6	D_5	D_4	D_3	D_2	D_1	D_0
$k = N$	--	$L - ((N-1) * 7) + 1$	n	n	n	n	n	n	n	D_0
				D_6	D_5	D_4	D_3	D_2	D_1	

2.6.2 *Message Structure*

This section describes the message structure for each of the CAN message types.

2.6.2.1 *Telecommand*

The following example makes use of the following parameters:

Message Type:	1
TCTLM ID:	12
Source Address:	16
Destination Address:	17
L:	6

Table 36: CAN Protocol Message Structure - Telecommand

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	1	12	16	17	6	$D_{0..5}$

2.6.2.2 *Telecommand Ack*

The following example makes use of the following parameters:

Message Type:	2
TCTLM ID:	12
Source Address:	17

Destination Address:	16
L:	N/A

Table 37: CAN Protocol Message Structure – Telecommand Ack

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	2	12	17	16	0	N/A

2.6.2.3 *Telecommand Nack*

The following example makes use of the following parameters:

Message Type:	3
TCTLM ID:	12
Source Address:	17
Destination Address:	16
L:	N/A

Table 38: CAN Protocol Message Structure – Telecommand Nack

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	3	12	17	16	0	N/A

2.6.2.4 *Telemetry Request*

The following example makes use of the following parameters:

Message Type:	4
TCTLM ID:	128
Source Address:	16
Destination Address:	17
L:	N/A

Table 39: CAN Protocol Message Structure – Telemetry Request

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	4	128	16	17	0	N/A

2.6.2.5 *Telemetry Response*

The following example makes use of the following parameters:

Message Type:	5
TCTLM ID:	128
Source Address:	17
Destination Address:	16
L:	6

Table 40: CAN Protocol Message Structure – Telemetry Response

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	5	128	17	16	6	$D_{0..5}$

2.6.2.6 Telemetry Nack

The following example makes use of the following parameters:

Message Type:	6
TCTLM ID:	128
Source Address:	17
Destination Address:	16
L:	N/A

Table 41: CAN Protocol Message Structure – Telemetry Nack

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*
1	6	128	17	16	0	N/A

2.6.2.7 Telecommand Extended

The following example makes use of the following parameters:

Message Type:	7
TCTLM ID:	12
Source Address:	16
Destination Address:	17
L:	18
N:	3

Table 42: CAN Protocol Message Structure – Telecommand Extended

FRAME NUMBER	29-BIT IDENTIFIER	DATA

	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*	
1	7	12	16	17	8	$D_7 = 2$	$D_{0..6}$
2	7	12	16	17	8	$D_7 = 1$	$D_{0..6}$
3	7	12	16	17	5	$D_4 = 0$	$D_{0..3}$

2.6.2.8 *Telemetry Response Extended*

The following example makes use of the following parameters:

Message Type:	8
TCTLM ID:	128
Source Address:	17
Destination Address:	16
L:	18
N:	3

Table 43: CAN Protocol Message Structure – Telemetry Response Extended

FRAME NUMBER	29-BIT IDENTIFIER				DATA		
	Type	*ID*	*Src*	*Dst*	*DLC*	*Bytes*	
1	8	12	17	16	8	D7 = 2	D0..6
2	8	12	17	16	8	D7 = 1	D0..6
3	8	12	17	16	5	D4 = 0	D0..3

2.7 Cubesat Space Protocol

"Cubesat Space Protocol (CSP) is a small protocol stack written in C. CSP is designed to ease communication between distributed embedded systems in smaller networks, such as Cubesats."

CubeSpace implements CSP as a connectionless, master/slave interface, which follows very closely to our internal protocol. Whereby, CubeSpace components act as slave devices and never transmit unsolicited packets. CSP is effectively a wrapper for our internal protocol, less the framing data, which is now handled by CSP.

2.7.1 *Network Services*

The only CSP service currently supported by CubeSpace components is "Ping".

2.7.2 *TCTLM*

CubeSpace CSP TCTLM is bound to port 8.

The address of a particular device is configurable.

A device listens for incoming messages on port 8. Upon reception of a complete CSP packet, the data portion is parsed and inspected. The device will then reply immediately to the source address with data, Ack, or Nack.

2.7.3 *Message Types*

The message types and their associated integer values are shown in Table 44.

Table 44: CSP Message Types

TYPE	VALUE
Telecommand	0x01
Telecommand Ack	0x02
Telecommand Nack	0x03
Telemetry Request	0x04
Telemetry Response	0x05
Telemetry Nack	0x06

2.7.4 *Internal Packet Structure*

The internal packet structure refers to the “data” portion of a CSP packet. This is the unwrapped CubeSpace packet. The structure of each of the packets is shown below.

The data portion of each packet (if relevant) can be found in a component’s specific reference manual or it’s API documents. The data portion corelates to the TCTLM ID.

The internal packet structure for each message type is shown in the tables following.

Table 45: CSP Internal Packet Structure - Telecommand

TELECOMMAND		
B0	*B1*	*B2 - BN*
Message Type	TCTLM ID	Data
0x01	0 - 127	[bytes]

Table 46: CSP Internal Packet Structure – Telecommand Ack

TELECOMMAND ACK		
B_0	*B_1*	*B_2*
Message Type	TCTLM ID	Error Byte
0x02	0 - 127	0

Table 47: CSP Internal Packet Structure – Telecommand Nack

TELECOMMAND NACK

B_0	B_1	B_2	B_3
Message Type	TCTLM ID	Error Byte	Error Index
0x03	0 - 127	1 - 7	[TBD]

Table 48: CSP Internal Packet Structure – Telemetry Request

TELEMETRY REQUEST	
B_0	B_1
Message Type	TCTLM ID
0x04	128 - 255

Table 49: CSP Internal Packet Structure – Telemetry Response

TELEMETRY RESPONSE		
B_0	B_1	B_2 - B_N
Message Type	TCTLM ID	Data
0x05	128 - 255	[bytes]

Table 50: CSP Internal Packet Structure – Telemetry Nack

TELEMETRY NACK			
B_0	B_1	B_2	B_3
Message Type	TCTLM ID	Error Byte	Error Index
0x06	128 - 255	1 - 7	[TBD]

2.7.5 TCTLM Pass-through

This section is only applicable to CubeComputer.

Read through section 5.2 for information regarding pass-through communication.

Pass-through TCTLM is bound to port 48.

3 Events

Events are defined as time-stamped occurrences of significant firmware state changes. This section only applies to the CubeComputer, as other CubeProducts do not log events.

The CubeComputer will store events to non-volatile memory and can also optionally output events to the OBC as unsolicited messages, if configured to do so. “Unsolicited” in this context refers to the fact that the communications message will be transmitted to the OBC without a prior request message.

Events fall in one of the following classes:

Table 51: Event classes

CLASS	CLASS IDENTIFIER	DETAILS
Critical	3	A condition occurred that severely compromises system functionality. It is not possible to automatically recover from a critical error condition.
Major Warning	2	A condition occurred that mildly compromises system functionality. It is not possible to automatically recover from a major warning event.
Minor Warning	1	A minor error occurred but it does not affect system functionality. The ADCS will automatically recover from such events
Information	0	A notification message for system changes. No reaction is required.

3.1 Event Decoding

A single event is encoded using 24 bytes, in the following format:

Table 52: Event entry encoding.

	COUNTER	UPTIME	UNIX TIME (S)	MILLISECONDS	IDENTIFIER	EVENT DATA
Byte offset	0	4	8	12	14	16
Byte length	4	4	4	2	2	8
Data type	32-bit unsinged int	32-bit unsinged int	32-bit unsinged int	16-bit unsinged int	Compound type (see **Table 53**)	[TBD]

Table 53: Event identifier encoding.

	EVENT CLASS	EVENT SOURCE	EVENT TYPE
Bit range	15:14	13:9	8:0
Data type	Enumeration from **Table 51**	AbstractNode enumeration value (see [2] for possible values)	Event type

Refer to the CubeADCS documentation for a list of its supported event types, and how the event data is mapped per event.

3.2 Unsolicited Event Message Format

The CubeComputer may be configured to output unsolicited events on UART or CAN. The frame structure for both is given in this section.

3.2.1 UART

Table 54: UART Protocol Frame Structure – Unsolicited Event Messages

HEAD		PAYLOAD	TAIL	
ESC	*SOE*	*Event (see* Table 52*)*	*ESC*	*EOM*
0x1F	0x2F	[24 bytes]	0x1F	0xFF

3.2.2 CAN (CubeSpace Protocol)

Unsolicited events are output using the CubeSpace CAN protocol as extended messages, with ID equal to 255. The destination ID [dst] used for events is configurable. The default destination ID is 240 (decimal).

Table 55: CAN Protocol Frame Structure – Unsolicited Event Messages

	29-BIT IDENTIFIER	DATA	

FRAME NUMBER	*Type*	*ID*	*Src*	*Dst*	*DLC*	*B_7*	*$B_6 – B_0$*
1	9	255	[src]	[dst]	8	3	Event bytes 0-6
2	9	255	[src]	[dst]	8	2	Event bytes 7-13
3	9	255	[src]	[dst]	8	1	Event bytes 14-20
4	9	255	[src]	[dst]	3	0	Event bytes 21-23

3.2.3 CAN (CSP)

Unsolicited events make use of the CSP broadcast address. The CSP destination port for unsolicited events is configurable. The default destination port is 58 (decimal).

The internal structure of the unsolicited CAN-CSP message (the “data” portion of the CSP packet) is simply the event message from Table 52.

4 Telemetry Logging

This section is only applicable to CubeComputer.

Telemetry is logged autonomously, and in a periodic, synchronous manner. The log does not capture telemetry requests, but rather autonomously captures telemetry parameters as they would be, had the telemetry been requested at that time.

Only a sub-set of telemetry is logged. In general, telemetry that changes internally as a result of ADCS operation is selected for logging. This set of logged telemetry is identified via the CubeComputer control program API xml file. Telemetries that will be logged have a non-zero “LogId” field. This set is not configurable and is defined by CubeSpace.

Telemetry logging is synchronized to the ADCS control loop.

Each log-able telemetry is either “fast” or “slow” and cannot be both. Fast telemetry is captured once per ADCS control loop iteration (5Hz). Slow telemetry is captured every fifth ADCS control loop iteration (1Hz). Whether a telemetry is fast or slow, is not configurable and is defined by CubeSpace, however the configuration can be queried using the `<TelemetryLogInclusionMasks>` telemetry.

A single log entry contains all telemetry captured over five ADCS control loop iterations. Therefore, an entry is logged at 1Hz, and includes 5x fast telemetry and 1x slow telemetry.

4.1 Inclusion Masks

Inclusion masks are multi-byte bitmasks used as a method of selecting telemetries. Although the set of telemetries that are logged onboard the ADCS is fixed, it is possible to retrieve logged telemetry as a subset. An example would be if the OBC wants to retrieve a log of only magnetometer measurements. The inclusion masks offer a method of identifying which telemetries to include in the retrieved log.

The selection is always done by referencing the log ID associated with a telemetry, rather than the telemetry ID itself.

The position of each bit in the mask represents a log ID. The bit position of a log ID is one less than the log ID. E.g., the bit position for log ID 1 is bit 0, the bit position for log ID 30 is bit 29.

All masks are 5 bytes long, allowing for a maximum of 40 telemetries, however it may be the case that not all bit positions are used. The `<TelemetryLogInclusionMasks>` frame can be requested to determine which telemetries (bit positions) are used.

The bit position for a given log ID is as follows:

$$idx = \frac{logId - 1}{8}$$

$$bitPos = (logId - 1)\ mod\ 8$$

Then, the telemetry mask as:

$$mask[idx]\ \&\ (0b1 \ll bitPos)$$

4.2 Telemetry Encoding

Raw telemetry log data follows the general format:

Table 56: Telemetry log raw data format

	INCLUSION MASK	ENTRY 0	ENTRY 1	ENTRY 2	...	ENTRY N
Byte offset	0	5	5 + z	5 + (2 x z)		5 + (n x z)
Byte length	5	z	z	z		z
Data type	Byte array	bytes	bytes	bytes		bytes

A single entry is encoded in the following format:

Table 57: Telemetry log entry format

	METADATA				TELEMETRY DATA
	Counter	Uptime	Unix time (s)	Milliseconds	
Byte offset	0	4	8	12	14
Byte length	4	4	4	2	variable
Data type	32-bit unsigned int	32-bit unsigned int	32-bit unsigned int	16-bit unsigned int	bytes

The first 5 bytes of the raw data will always contain the inclusion mask. The mask describes which telemetries are present in the "Telemetry data" portion of each entry. Note that there is only one inclusion mask for a set of raw data. This is because the format of each entry is constant for each raw data set. The data itself will vary, but the format will remain constant.

For the case where all telemetry is included, the format of telemetry data for each entry is as follows:

Table 58: Telemetry log telemetry data format example - all included

	TELEMETRY 1	TELEMETRY 2	TELEMETRY 3	TELEMETRY 4	...	TELEMETRY N
Log ID	1	2	3	4		n
Byte offset	0	a	a + b	a + b + c		...
Byte length	a	b	c	d		z

Note that telemetry is always ordered in ascending order of log ID.

Following from the above format, consider the case where log ID 3 is not included in the mask, the format of telemetry data for each entry would be as follows:

Table 59: Telemetry log telemetry data format example - exclusion

	TELEMETRY 1	TELEMETRY 2	TELEMETRY 4	TELEMETRY 5	...	TELEMETRY N
Log ID	1	2	4	5		n
Byte offset	0	a	a + b	a + b + d		...
Byte length	a	b	d	e		z

An important aspect of this encoding scheme is that, although the raw data will represent the user's request, the format of the data is contained within the data itself. The benefit of this is that the implementation of the decoder remains independent of any other aspect of the telemetry log, and unsolicited telemetry, as well as the user's request.

4.3 Log Return Interval

The log return interval is a user specified parameter that determines the rate of telemetry returned when downloading the log, as well as for unsolicited telemetry. This parameter is not to be confused with the internal logging frequency, which is always constant.

For example, specifying a log return interval of 200 milliseconds will result in the downloaded telemetry log data containing telemetry captured every 200 milliseconds. Specifying a log return interval of 2 seconds will result in the downloaded log data containing telemetry captured once every 2 seconds (skipping telemetry captured in between).

The log return interval does not change the telemetry encoding format described in section 4.2, however, a log return interval of 200 milliseconds does have implications. The remainder of this section will explain these implications.

Recall that a single log entry is logged at 1Hz and contains fast telemetry captured at 5Hz and slow telemetry captured at 1Hz. Therefore, one log entry will contain 5 instances of the same type of fast telemetry and 1 instance of slow telemetry.

The encoding scheme is designed so that the decoder implementation can remain independent of knowledge of which telemetries are fast and which are slow. This independence means there is no knowledge of the relative frequency of different entries to each other. i.e., the decoder implementation cannot assume there are 5 telemetries of type x and then 1 of type y. This design leads to the following implications:

1. A single log entry (1Hz) is split-up to appear as 5 entries (5Hz) in the raw data. This results in multiple entries in the raw data having the same metadata, because they belong to the same log entry.
2. Any slow telemetry included will be repeated 5 times in each of the split-up entries in 1., with the exact same data.

It is thus advised to only include fast telemetries when requesting telemetry logs with a log return interval of 200 milliseconds.

4.4 Unsolicited Telemetry Message Format

The CubeComputer may be configured to output unsolicited telemetry on UART or CAN. The message structure for both is given in this section.

Unsolicited telemetry is encoded in the same way as described in section 4.2, therefore the same decoder implementation used for downloaded log data can be used for unsolicited telemetry.

4.4.1 Configuration

Configuration consists of selecting which telemetries to output as unsolicited telemetry, as well as the frequency it should be output.

Telemetry is selected using an inclusion mask as described in section 4.1. The frequency of the output is determined by a return interval as described in section 4.3. The configuration may be modified using the `<UnsolicitedTlmSetup>` TCTLM.

Each communication interface can be configured independently.

The fastest rate at which unsolicited telemetry can be output is 1Hz. This is due to unsolicited telemetry being closely linked to the logging functionality. Unsolicited telemetry is encoded as a sub-set of a full log entry, as per configuration, and log entries are generated at 1Hz. However, if the selected interval is sub-second, all the fast telemetry captured within the second will be present in the unsolicited raw data, in the same way as for downloaded log data.

Due to strict timing requirements, the maximum allowed size of unsolicited telemetry data is limited for each interface. The user is required to calculate the total expected size of the unsolicited data for the desired configuration. If the configuration would result in a total size greater than the limit, the `<UnsolicitedTlmSetup>` telecommand will return an `<InvalidParam>` NACK. The limit for each interface is as follows:

UART: 5825 bytes

CAN (All): 1000 bytes

Unsolicited telemetry is output at the end of the ADCS control loop iteration within which the log entry is generated (every fifth iteration). Due to strict timing requirements, it is required that at least 70 milliseconds remain before the start of the next ADCS control loop iteration, for unsolicited telemetry to be output.

4.4.2 UART

Table 60: UART Protocol Frame Structure – Unsolicited Telemetry Messages

HEAD		PAYLOAD	TAIL	
ESC	*SOU*	*Encoded telemetry (see section 4.2)*	*ESC*	*EOM*
0x1F	0x4F	[bytes]	0x1F	0xFF

4.4.3 CAN (CubeSpace protocol)

Unsolicited telemetry is output using the CubeSpace CAN protocol as extended messages, with ID equal to 255. The destination ID [dst] used for unsolicited telemetry is configurable. The default destination ID is 241 (decimal).

The unsolicited telemetry protocol differs slightly from the standard communication protocol, in that the last byte of each CAN packet **does not** contain the number of packets remaining. Rather, flow control is achieved by using a different message type, allocated in the extended ID. This is due to unsolicited data requiring more than 255 packets to be transmitted. The different message types used for unsolicited telemetry are listed in Table 61.

Table 61: CAN (CubeSpace protocol) Unsolicited Telemetry Message Types

MESSAGE TYPE	VALUE (DECIMAL)	DESCRIPTION

First	10	Indicates that the packet is the first of the transmission. If the packet is both first and last, it will have the “Last” message type.
Body	11	Indicates that the packet is between the first and last packets of the transmission.
Last	12	Indicates that the packet is the last of the transmission. If the packet is both first and last, it will have the “Last” message type.

An example of unsolicited telemetry frame structure for a data size of 27 bytes is shown in Table 62.

Table 62: CAN (CubeSpace protocol) Unsolicited Telemetry Frame Structure

FRAME NUMBER	29-BIT IDENTIFIER				DATA	
	Type	*ID*	*Src*	*Dst*	*DLC*	*B_7 – B_0*
1	10	255	[src]	[dst]	8	Telemetry bytes 0-7
2	11	255	[src]	[dst]	8	Telemetry bytes 8-15
3	11	255	[src]	[dst]	8	Telemetry bytes 16-23
4	12	255	[src]	[dst]	3	Telemetry bytes 24-26

4.4.4 CAN (CSP)

Unsolicited telemetry makes use of the CSP broadcast address. The CSP destination port for unsolicited telemetry is configurable. The default destination port is 59 (decimal).

The size of CSP packets is limited internally to 512 bytes. The total size of unsolicited telemetry data may exceed this limit. Therefore, flow control is achieved by use of different source ports for each of the packets in the transmission. The different source ports used for unsolicited telemetry are listed in Table 63.

Table 63: CAN (CSP) Unsolicited Telemetry Source Ports

SOURCE PORT	VALUE (DECIMAL)	DESCRIPTION
First	60	Indicates that the packet is the first of the transmission. If the packet is both first and last, it will use the “Last” source port.
Body	61	Indicates that the packet is between the first and last packets of the transmission.
Last	62	Indicates that the packet is the last of the transmission. If the packet is both first and last, it will use the “Last” source port.

5 Firmware Operational Sequences

5.1 Bulk Data Transfer

The procedure to transfer data to and from all CubeProducts is the same.

The transfer consists of two stages:

1. Transfer setup
2. Transfer data (upload or download)

The transfer setup consists of a single telecommand used to specify certain characteristics about the data to be transferred and is unique to the type of data.

Depending on the type of data being transferred, a CubeProduct may expose a status telemetry, which is not required to perform the transfer, but provides additional information on the state and/or result of the transfer.

5.1.1 Transfer Setup

5.1.1.1 CubeSense Sun Images

Images can be downloaded in different resolutions to reduce transfer time. The maximum resolution is 1024x1024 bytes.

To start the download:

1. Send `<ImageTransferSetup>`
   Specify `<LocSelection>` as the same location as the image capture location.
   Specify `<ImageSize>` as the desired resolution.
   Specify `<Direction>` as `<Download>`.

2. Wait 100 milliseconds for the setup to complete
   Note that the CubeSense Sun does not provide a transfer status telemetry.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.2 CubeSense Earth Images

Images are always 80x64x2 bytes.

To start the download:

1. Send `<ImageTransferSetup>`
   Specify `<DataSelect>` as the desired data format.
   Specify `<Direction>` as `<Download>`

2. Wait 100 milliseconds for the setup to complete.
   Note that the CubeSense Earth does not provide a transfer status telemetry.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.3 CubeStar Images

TBD

5.1.1.4 CubeADCS Event Log

The event log can be downloaded from both the bootloader and control program running on the CubeComputer.

It is possible to only download a sub-set of events from the log. Selection of this sub-set is referred to as filtering. The event log transfer setup specifies various parameters that define the filter.

The filter consists of three stages:

1. Entry span: which entries should be searched and passed to the next stage of the filter. The entry span is determined by the `<filterType>` and its corresponding parameters, described in Table 64.
2. Class: which event classes should be included in the download.
3. Source: which event sources should be included in the download.

The different filter types for stage 1. of the filter are described in Table 64.

Table 64: CubeADCS event log download filter types

FILTER TYPE	CORRESPONDING PARAMETERS	DESCRIPTION
<FilterNone>	N/A	All entries are passed to the next stage of the filter.
<FilterTimeSpan>	<startTimeUnix> <endTimeUnix>	Only Entries that were logged between and at the specified start time and end time are passed to the next stage of the filter.
<FilterTimeNextX>	<startTimeUnix> <numEntries>	Only the specified number of entries logged following the specified start time are passed to the next stage of the filter.
<FilterFirstX>	<numEntries>	Only the specified number of entries following the first (oldest) logged entry are passed to the next stage of the filter.
<FilterLastX>	<numEntries>	Only the specified number of entries preceding the last (latest) logged entry are passed to the next stage of the filter.
<FilterCounterNextX>	<writeCounter> <numEntries>	Only the specified number of entries logged following, and including, the entry with a write counter value matching the specified write counter, are passed to the next stage of the filter.

To start the download:

1. Send `<EventLogTransferSetup>`
   With the desired parameters for filtering.

2. Wait for the read queue state to transition to download:
   Get `<EventLogStatus>`
   Repeat until `<readQueueState>` == `<EvtReadQDownload>`.

The poll backoff should be at least 10 milliseconds.
Allow for up to 2 seconds for the state to change.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.5 CubeADCS Telemetry Log

The telemetry log can be downloaded from both the bootloader and control program running on the CubeComputer.

It is possible to only download a sub-set of telemetries from the log. Selection of this sub-set is referred to as filtering. The telemetry log transfer setup specifies various parameters that define the filter.

The filter consists of three stages:

1. Entry span: which entries should be searched and passed to the next stage of the filter. The entry span is determined by the `<filterType>` and its corresponding parameters, described in Table 65.
2. Interval: the frequency of telemetry to include in the data.
3. Inclusion bitmask: which telemetry types to include in the data.

The different filter types for stage 1. of the filter are described in Table 65.

Table 65: CubeADCS telemetry log download filter types

FILTER TYPE	CORRESPONDING PARAMETERS	DESCRIPTION
<FilterNone>	N/A	All entries are passed to the next stage of the filter.
<FilterTimeSpan>	<startTimeUnix> <endTimeUnix>	Only Entries that were logged between and at the specified start time and end time are passed to the next stage of the filter.
<FilterTimeNextX>	<startTimeUnix> <numEntries>	Only the specified number of entries logged following the specified start time are passed to the next stage of the filter.
<FilterFirstX>	<numEntries>	Only the specified number of entries following the first (oldest) logged entry are passed to the next stage of the filter.
<FilterLastX>	<numEntries>	Only the specified number of entries preceding the last (latest) logged entry are passed to the next stage of the filter.
<FilterCounterNextX>	<writeCounter> <numEntries>	Only the specified number of entries logged following, and including, the entry with a write counter value matching the specified write counter, are passed to the next stage of the filter.

To start the download:

1. Send `<TelemetryLogTransferSetup>`
   With the desired parameters for filtering.

2. Wait for the read queue state to transition to download:
   Get `<TelemetryLogStatus>`
   Repeat until `<readQueueState>` == `<TlmReadQDownload>`.
   The poll backoff should be at least 10 milliseconds.

Allow for up to 2 seconds for the state to change.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.6 CubeADCS Image Log

Images captured by nodes can be stored internally on the CubeComputer and downloaded at a later stage, or an image can be captured and streamed directly via the CubeComputer without storing it internally on the CubeComputer.

Images can be downloaded from both the bootloader and control program of the CubeComputer.

5.1.1.6.1 Stored image download

Each stored image is assigned a `<FileHandle>`, used as a reference to that image. To obtain the handle of the target file to download, request `<ImageFileInfo>` until the information matches the desired image and extract the `<FileHandle>`.

To start the download:

1. Send `<ImageTransferSetup>`
   With `<OpCode>` == `<Download>`.
   Specify `<FileHandle>` as the target file.
   All other parameters are unused for the `<Download>` operation.

2. Wait for the state to transition to download:
   Get `<ImageTransferStatus>`
   Repeat until `<State>` == `<StateBusyDownload>`.
   The poll backoff should be at least 50 milliseconds.
   Allow for up to 5 seconds.
   Abort if timeout condition reached.
   Abort if `<ErrorCode>` != 0.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.6.2 Direct image download

Images that are downloaded directly (without storing) via the CubeComputer, are a data stream and do not have an assigned file handle, rather images are downloaded by specifying the `<NodeType>` of the node to capture the image from.

To download an image directly, the target node must be powered on and must not be active in the system. When the CubeComputer bootloader is the running application, it is sufficient to ensure that Auto-Discovery has completed and to power on the target node. When the control program is the running application, the target node's power state must be set to `<PowerOnPass>`, which ensures the node is powered but prevents the node from being sampled for ADCS control. Note that the `<PowerOnPass>` power state only powers on the node, therefore the node bootloader will execute and remain running for the duration of its configured backoff period. Any attempt to download an image directly while the target node bootloader is running, will fail.

To start the download:

1. Send `<ImageTransferSetup>`
   With `<OpCode>` == `<CaptureDownload>`.
   Specify `<NodeType>` as the target node.
   All other parameters are unused for the `<CaptureDownload>` operation.

2. Wait for the state to transition to download:
   Get `<ImageTransferStatus>`
   Repeat until `<State>` == `<StateBusyDownload>`.
   The poll backoff should be at least 50 milliseconds.
   Allow for up to 5 seconds.
   Abort if timeout condition reached.
   Abort if `<ErrorCode>` != 0.

3. Proceed to download the data by following section 5.1.2.1.

5.1.1.7 CubeADCS File Uploads

This section describes how to setup a transfer of firmware and configuration file to the CubeComputer control program. Note that only files that target the expected nodes can be uploaded to the CubeComputer control program. Files that target the CubeComputer itself can only be uploaded to the bootloader, see section 5.1.1.8.

Firmware binaries and config files are provided as CubeSpace (.cs) files. For information on these files see chapter 7. This procedure will require extracting the metadata from a CubeSpace file.

Before proceeding with a file upload, request `<File Transfer Status>` to confirm that the `<State>` is `<Idle>`.

To start the upload:

1. Send `<File Transfer Setup>` (see Table 66 and *File Transfer Setup Command Format* in [4]):

Table 66: `<File Transfer Setup>` **Telecommand to Set Up a File Upload**

PARAMETER / FIELD	VALUE	COMMENT
<Op Code>	<Upload>	Upload a file.
<File>	0	Not used. This is determined from File Metadata.
<Node>	0	Not used. This is determined from File Metadata.
<Serial Number Integer>	0	Not used. This is determined from File Metadata.
<Program>	0	Not used. This is determined from File Metadata.
<File Size>	0	Not used. This is determined from File Metadata.
<Force Port>	0	Not used.
<File Meta Data>	As extracted from the .cs file to be uploaded	Note: The size allocated for File Metadata in the telemetry structure may be larger than the meta data present in the in the .cs file. Zero-padding is required to end of the allocated size.

2. Confirm that the transfer has started:
   Get `<File Transfer Status>`
   Repeat until `<State>` == `<Busy>`
   The poll backoff time should be at least 10 milliseconds.
   Allow for up to 500 milliseconds.

   Abort if `<ErrorCode>` is non-zero, or if timeout condition is reached:

   Send `<File Transfer Setup>`, with `<Op Code>` == `<Cancel>`.

3. Confirm the setup parameters are correct:
   Get `<File Transfer Setup>`
   Proceed if the returned parameters match the setup parameters from step 1.

4. Confirm expected start conditions:
   Get `<File Transfer Status>`
   Proceed if (`<Data Remain>` == `<File Size>`, `<Error Code>` == 0).
   Note: `<File Size>` will be set to the size of the wrapped file, and not the size of the .cs file. This is equivalent to the size of the .cs file less the size of the meta data. The first two bytes of the .cs file contains the size of the meta data.

5. Proceed to upload the file data by following section 5.1.2.2. Note that the meta data is excluded from the upload and only the wrapped file data should be transferred.

5.1.1.8 *Bootloader File Uploads*

Firmware binaries and config files are provided as CubeSpace (.cs) files. For information on these files see chapter 7. This procedure will require extracting the metadata from a CubeSpace file.

Before proceeding with a file upload, request `<State>` to confirm that the `<AppState>` is `<Idle>`.

To start the upload:

1. Send `<Write File Setup>` (see Table 67and *Write File Setup Setup Command Format* in [3]):

Table 67: `<Write File Setup>` **Telecommand to Set Up a File Upload**

PARAMETER / FIELD	VALUE	COMMENT
<File Meta Data>	As extracted from the .cs file to be uploaded	Note: The size allocated for File Metadata in the telemetry structure may be larger than the meta data present in the in the .cs file. Zero-padding is required to end of the allocated size.

2. Confirm that the transfer has started:
   Get `<State>`
   Repeat until `<AppState>` == `<BusyWaitFrame>`
   The poll backoff time should be at least 200 milliseconds.

Allow for up to 30 seconds.

Abort if `<Result>` is non-zero, or if timeout condition is reached.

3. Proceed to upload the file data by following section 5.1.2.2. Note that the meta data is excluded from the upload and only the wrapped file data should be transferred.
4. Upon any failure, request the `<Errors>` telemetry to get details of the failure condition.

5.1.2 Data Transfer

Bulk data transfer is performed by the sequential transfer of individual frames. Each frame can have a maximum size of 256 bytes. The size of each frame may vary depending on the data type and the transfer setup.

The data flow is controlled by a master/slave agreement of what the current frame number is. The master being the transfer initiator and the slave being the CubeProduct. Frame numbering starts from 0 (zero) and increments. The frame number is initialized to 0xFFFF prior to any frames being processed.

Data is transferred using three TCTLM:

- `<TransferFrame>`: Used by the master to specify the current frame number.
- `<FrameInfo>`: Provides information about the current frame on the slave.
- `<Frame>`: Contains the data of the current frame.

All transfers implement a timeout of 1 second between each frame.

The response timeout for each of the TCTLM depends on the interface, and the amount of traffic on the interface. If the only traffic on the interface is to the CubeProduct, the timeout can be as low as 20 milliseconds. The lower the timeout, the more retry attempts are possible if needed. None of the TCTLM are blocking. The timeout should be increased if lower timeouts do not work well for the OBC implementation.

See section 5.1.2.3 on error handling for these TCTLM during the transfer.

5.1.2.1 Download

When downloading; for some data types it is not feasible to calculate the transfer size prior to performing the transfer. For this reason, the transfer procedure is designed to support an arbitrary size of data. The last frame in the transfer is indicated by the `<FrameLast>` flag of the `<FrameInfo>` telemetry.

The procedure for bulk downloads is as follows:

1. Initialize the locally stored current frame number to 0 (zero)

2. Set the current frame number:
   Send `<TransferFrame>`
   Specify `<NextFrame>` as the locally stored current frame number.
   Allow for 3 retries if a `<TctlmBusy>` nack is returned.
   The retry backoff should be at least 5 milliseconds.

3. Wait for the CubeProduct to prepare the frame:
   Get `<FrameInfo>`

Repeat until `<FrameNumber>` == `<NextFrame>` from step 2.
The poll backoff should be at least 10 milliseconds.
Allow for up to 1 second.
If `<FrameLast>` == true, this is the last frame, continue until step 6. then exit.
Abort if `<FrameError>` == true, or timeout reached.

4. Request the prepared frame:
   Get `<Frame>`

5. Calculate the checksum of `<FrameBytes>` over `<FrameSize>` number of bytes and compare to the `<CheckSum>` field of `<FrameInfo>` from step 3.
   Abort if the checksum is invalid.
   See Appendix A: Bulk Data Transfer Checksum Calculation.

6. Copy `<FrameSize>` number of bytes from `<FrameBytes>` into local data buffer.

7. Increment the locally stored current frame number and repeat from step 2.

Note: the retry in step 2. Is required due to an edge case in the protocol, where it is possible that `<FrameInfo>` is requested and contains the correct `<FrameNumber>`, signalling that the frame has been processed, but the CubeProduct has not yet signalled that the operation is complete at the process level by the time the master is ready to request the next frame.

5.1.2.2 Upload

The procedure for bulk uploads is as follows:

1. Initialize the locally stored current frame number to 0 (zero)

2. Extract the next frame from local data buffer:
   Copy the frame data into the `<FrameBytes>` field of `<Frame>`.
   Set `<Framesize>` to the size of the frame.
   Send `<Frame>`.

3. Verify that the frame has been received uncorrupted using the checksum:
   Get `<FrameInfo>`
   Calculate checksum of `<FrameBytes>` over `<FrameSize>` and compare with the `<CheckSum>` field.
   Abort if the checksum is invalid.
   Step 2. can be retried if the checksum fails, keeping in mind that there is a 1 second timeout between frames.
   See Appendix A: Bulk Data Transfer Checksum Calculation.

4. Signal to the CubeProduct that the frame is ready by setting the current frame number:
   Send `<TransferFrame>`
   Specify `<NextFrame>` as the locally stored current frame number.

5. Wait for the CubeProduct to process the frame:
   Get `<FrameInfo>`
   Repeat until `<FrameNumber>` == `<NextFrame>` from step 4.
   The poll backoff should be at least 10 milliseconds.
   Allow for up to 1 second.
   If `<FrameLast>` == true, the CubeProduct will not accept any more frames, exit.
   Abort if `<FrameError>` == true, or timeout reached.

6. Increment the locally stored current frame number and repeat from step 2.

If the transfer had to be aborted due to an error or timeout, the user should request the status telemetry (if any) associated with the data type.

5.1.2.3 Transfer Error Handling

This section describes how to handle communication errors that may occur during the transfer of data.

5.1.2.3.1 No response

How to handle a lack of response is detailed in Table 68.

Table 68: Bulk Data transfer - no response handling

TCTLM	HANDLING
<TransferFrame>	Retry the command immediately. If the previous command was received successfully, and only the response was unsuccessful, the CubeProduct will already have the frame number specified, and will have already started processing the frame: • If the frame has not been fully processed, a <TctlmBusy> nack response with be returned on the retry. In this case, continue as usual by polling <FrameInfo> until the frame is processed. • If the frame has already been processed by the time the retry is sent, a <InvalidParam> nack response will be returned on the retry, because the frame number is now behind by one. In this case, confirm the frame number in <FrameInfo>, and continue as usual to extract the <Frame>. If the previous command was not received successfully the retry should return an ack. In this case continue as usual. The retry attempts should not exceed 1 second in duration, which is the timeout of the internal process, in which case the transfer will have been aborted internally. If the data type being downloaded exposes a status telemetry, query it to inspect for errors.
<FrameInfo>	Retry the request immediately. The retry attempts should not exceed 1 second in duration, which is the timeout of the internal process, in which case the transfer will have been aborted internally. If the data type being downloaded exposes a status telemetry, query it to inspect for errors. This telemetry only returns state information and does not perform any processing; therefore, it will only return an ack.
<Frame>	Retry the request immediately.

	The retry attempts should not exceed 1 second in duration, which is the timeout of the internal process, in which case the transfer will have been aborted internally. If the data type being downloaded exposes a status telemetry, query it to inspect for errors. This telemetry only returns state information and does not perform any processing; therefore, it will only return an ack.

5.1.2.3.2 Nack response

During the transfer, only the `<TransferFrame>` telecommand can generate a nack response. The possible nack types and condition they might occur are listed in Table 69.

Table 69: Bulk data transfer NACK conditions

NACK TYPE	CONDITION(s)
<TctlmBusy>	• The CubeProduct service handling the data busy with a task unrelated to the transfer. This is only possible if the protocol is not followed correctly, and the transfer has not been set up. • Transfer setup is not yet complete. If the data type being transferred exposes a status telemetry, this should be queried to see state information. • The current frame is still being processed. The user should only send <TransferFrame> once the <FrameInfo> telemetry indicates the correct frame number, which indicates that the processing of the current frame is complete. If <FrameNumber> is indeed correct, retry sending the command. • The <FrameNumber> in <FrameInfo> has been updated to indicate the frame has been processed, but the internal process has not yet signalled it is complete internally. This is an unlikely, yet possible, and unavoidable, race condition. In this case retry sending the command. To summarize: if the protocol is followed correctly, and the <FrameNumber> is correct, the command should be retried.
<Sequence>	• The transfer setup did not complete successfully and therefore the transfer state is idle. If the data type exposes a status telemetry it can be queried to confirm the state. • An error occurred while processing the frame and the CubeProduct aborted the transfer. If the data type exposes a status telemetry it can be queried to confirm the state. Alternatively, the <FrameNumber> of <FrameInfo> will return to 0xFFFF if the transfer state is idle. • Timeout has been reached since the last <TransferFrame> and the CubeProduct aborted the transfer. If the data type exposes a status telemetry it can be queried to confirm the state. Alternatively, the <FrameNumber> of <FrameInfo> will return to 0xFFFF if the transfer state is idle.
<InvalidParam>	The set frame number is invalid. i.e., The <FrameNumber> being sent is not the increment of the frame number the CubeProduct has stored. This suggests that the protocol is not being followed. The <NextFrame> parameter was not incremented sequentially from the previous frame number. The first <NextFrame> is not 0 (zero).

5.1.2.4 Cancel Transfer

The bulk data transfer procedure does not implement an explicit cancellation scheme; however, all transfers are protected by a 1 second timeout. Therefore, to cancel an ongoing transfer, simply cease any further transmission.

5.2 Telecommand and Telemetry Pass-through

This section is only applicable to CubeComputer.

Pass-through refers to a communication pipe between the CubeComputer slave interface and the nodes connected in the system. This enables the full set of TCTLM of each node to be exposed via the CubeComputer slave interface.

Pass-through is available for all communication protocols.

5.2.1 Target Node Selection

The target node for pass-through communication is not embedded in the communication protocol. Selection of the target node is achieved using a CubeComputer specific telecommand. This allows the pass-through communication to adhere to the standard communication protocols.

To set the target node, send the `<PassThrough>` telecommand, specifying `<TargetNode>` as the target node.

Note that setting `<TargetNode>` to `<NodeInvalid>` effectively disables pass-through. This has different implications depending on the protocol and will be covered in their respective sections.

5.2.2 Timeouts

Once CubeComputer receives a pass-through message, the message is routed to the target node. The timeout of this communication is hard coded to 200 milliseconds. Adjust the timeout of pass-through communication accordingly.

If communication with the node times out, the CubeComputer will return a "Pass-through timeout" nack, as listed in Table 2. Note that this nack will be returned as a pass-through message and not a standard message, even though the pass-through was not successful.

5.2.3 UART/RS485 Pass-through

Pass-through messages are differentiated from standard messages by use of different start-of-message protocol characters. The characters used for UART pass-through messages are listed in Table 3. The characters used for RS485 pass-through messages are listed Table 16.

The rest of the frame structure following the start-of-message characters is identical to standard messages, therefore pass-through communication can be parsed in the same manner as if communication was directly with the target node.

Note that when using RS485, the destination address should be set to that of CubeComputer, and not the address of the node.

The UART frame structures for pass-through messages are shown in section 2.3.3.

The RS485 frame structures for pass-through messages are shown in section 2.4.3.

If pass-through is disabled, pass-through messages received will not be routed.

5.2.4 I2C Pass-through

Pass-through messages are differentiated from standard messages by use of a different 7-bit slave address. This address is 63 (decimal) and is hard-coded. The CubeComputer only listens to messages on this address if pass-through is enabled.

Apart from the address, all aspects of pass-through communication are identical to standard communication as described in section 2.5.

5.2.5 CAN Pass-through

Pass-through messages are differentiated from standard messages by means of a different destination address. This address is 235 (decimal) and is hard-coded. The CubeComputer only listens to messages on this address if pass-through is enabled.

Apart from the address, all aspects of pass-through communication are identical to standard communication as described in section 2.6.

5.2.6 Cubesat Space Protocol (CSP) Pass-through

Pass-through messages are differentiated from standard messages by means of a different destination port. This port is 48 and is hard-coded.

Apart from the destination port, all aspects of pass-through communication are identical to standard communication as described in section 2.7.

If pass-through is disabled, pass-through messages received will not be routed.

6 Error Logging

All CubeSpace CubeProducts make use of a common error logging system.

This system is not a comprehensive, verbose, logging system, and serves primarily as a debugging tool for CubeSpace internal development as well as for customer support. Although, error logs are exposed over all component's API's and can be queried at any time.

An error log entry is created when the component's software deems that a condition has occurred that is severe enough that it cannot continue operating safely or deterministically. The device will reset and upon start up, attempt to write the error-code associated with the error to non-volatile memory.

An error log entry consists of the following:

- Unix Timestamp
- Error Code

The Error Code is a 32-bit unsigned integer that is generated at compile time and contains information that allows the CubeSpace team to locate the exact line of code that triggered the error.

The Error Code is not intended to be human readable.

If any error logs are present, forward them to CubeSpace so we may investigate the cause.

6.1 Configuration

There are two configuration items associated with the error logging system. These can either be modified in the component's config file and uploaded or they can be modified and persisted via TCTLM.

These parameters control whether error logging is enabled or disabled as well as the desired behaviour if the error log is full.

6.2 Reading Error Logs

Error Logs can only be read out one at a time.

Reading error logs consists of three parts:

1. Setting the index reference
2. Setting the index
3. Reading the log entry

Setting the index reference to **head** sets the index origin to the first(oldest) error log entry. Therefore, an index of 0(zero) will target the first entry in the log. An index of 1 will target the second entry in the log, and so forth.

Setting the index reference to **tail** sets the index origin to the last(latest) error log entry. Therefore, an index of 0(zero) will target the last entry in the log. An in index of 1 will target the second-last entry in the log, and so forth.

After setting the indexing parameters the targeted error log entry can then be queried.

6.3 Clear Error Logs

Clearing the error log via TCTLM will erase all log entries. A Soft-Reset is required after this operation.

6.4 Common Causes

Error conditions can be characterised as one of the following:

- Configuration
- Fatal Component Failure
- Volatile Runtime Corruption

6.4.1 Low-Level Initialization

If the component fails to initialize any of the hal, driver, or service layers, an error condition is generated.

6.4.2 Configuration Error

Error conditions may occur if certain configuration parameters are not compatible or are invalid. These errors typically occur on config items relating to hardware configurations that are not intended to change after launch. This allows for definitive misconfiguration to be quickly identified and corrected.

6.4.3 Critical Component Failure

If a component experiences failure of a critical external component that is fundamental to its operation, it will generate an error condition.

6.4.4 Volatile Runtime Corruption

These conditions are random and may occur due to environmental changes. As an example, cosmic radiation may cause a bit-flip, leading to an out-of-bounds array index.

7 CubeSpace Files (.cs)

CubeSpace utilizes a file-wrapping scheme to allow for finer control over how certain files are used. A CubeSpace file is produced by inserting meta-data to the start of the original file and saving the wrapped file with a file extension "**.cs**". The original file extension (before wrapping) will be shown as the suffix of the filename.

Files in the Software Bundle provided as CubeSpace Files, as well as their original extensions are shown in Table 70.

Table 70: CubeSpace File Transformations

ORIGINAL FILE TYPE	ORIGINAL FILE EXTENSION	CUBESPACE FILE SUFFIX	CUBESPACE FILE EXTENSION
Application Binary	.bin	-bin	.cs
Application Config Binary	.cfg.bin	-cfg-bin	.cs

The structure of a .cs file is shown in Figure 3 and Table 71.

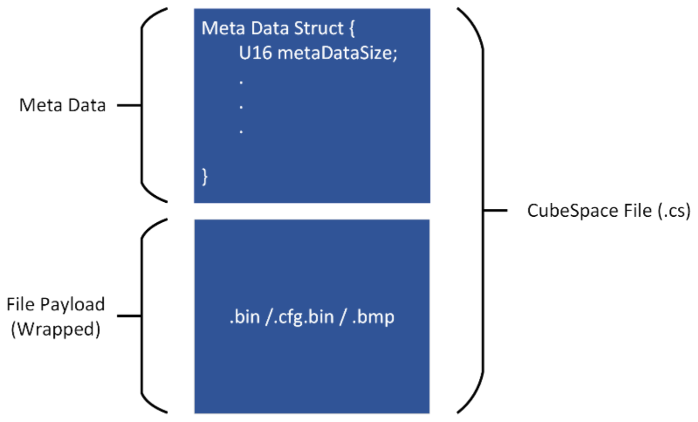


Figure 3: CubeSpace .cs file layout

Table 71: Structure of a .cs File with prepended Metadata

FILE OFFSET [BYTES]	SIZE [BYTES]	PARAMETER / FIELD
0	2	metaDataSize [nr of Bytes]. This is the total size of the meta data, which includes the two bytes of this field.
0	<metaDataSize>	Meta Data
<metaDataSize>	Total File Size (.cs) - <metaDataSize>	File Payload (Wrapped).

To extract the meta data:

1. Read the first 2 bytes of the .cs file which contains the size of the meta data (in number of bytes) stored as a U16.
2. Read <metaDataSize> number of bytes starting at offset 0 of the .cs file.

7.1 Usage – Bootloader

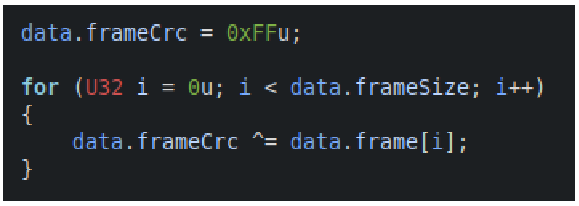
The CubeComputer can store application binaries and config binaries onboard, and can upgrade connected sensors independently, without user intervention.

These files first need to be uploaded to the CubeComputer. CubeSpace Files allow the CubeComputer to accurately validate and associate files uploaded to it. The user does not need to specify which file to target during an upload. CubeComputer will discern this from the meta-data of the CubeSpace File.

See the Bootloader Application Manual [1] for more information on bootloader usage.

Appendix A: Bulk Data Transfer Checksum Calculation

A C implementation of the bulk transfer checksum of frame data is shown below.



```c
data.frameCrc = 0xFFu;

for (U32 i = 0u; i < data.frameSize; i++)
{
    data.frameCrc ^= data.frame[i];
}
```

Figure 4: Bulk Transfer Checksum Calculation