QB50

FP7-284427

WP250: Satellite Control Software

Recommendation for Flight Software implementation

Issue 1 – version 0

Prepared by:
Louis Masson

Checked by:
Stéphane Billeter
Yann Voumard
Florian George

Approved by:
Muriel Richard

Swiss Space Center EPFL
Lausanne
Switzerland

RECORD OF REVISIONS

ISS/REV	Date	Modifications	Created/modified by
0/0	23rd January 2014	First issue	L. Masson
0/1	3rd February 2014	Added service handling descriptions, flowcharts, and details to the generation and decoding of CCSDS packets.	L. Masson
0/2	27th March 2014	Modification of the document to propose an on-board implementation the ECSS standard tailored to QB50 rather than SwissCube.	L. Masson
1/0	30th April 2014	Various updates taking into account requests for clarifications. Added Florian George as a contributor.	L. Masson

TERMS AND ABBREVIATIONS

Ack	Acknowledgement
ADCS	Attitude Determination and Control Subsystem
ADS	Antenna Deployment Subsystem
APID	Application Process ID
CCSDS	Consultative Committee for Space Data Systems
CDMS	Command and Data Management Subsystem
COM	Communication [subsystem]
CRC	Cyclic Redundancy Code
CUC	CCSDS Unsegmented Code
ECSS-PUS	European Cooperation on Space Standardization – Packet Utilisation Services
EPS	Electrical Power Subsystem
GS	Ground Station
GS Manager	Ground Station Manager
HK	House-Keeping
MCU	MicroController Unit
PL	PayLoad [subsystem]
SID	Structure Identification
TC	Telecommand
TM	Telemetry

INTRODUCTION

SwissCube was a successful satellite project in that it was capable of assuming its technical functionalities and was capable of successfully interacting with the GS by receiving telecommands from the ground and replying with the appropriate telemetry. SwissCube has been operational for more than 4 years, and has still not been subject to a software related fault. To this day, it is still possible to interact with the flight software on board of SwissCube. This goes to show that the software that has been developed for SwissCube is a good example of how to implement some of the services defined in the ECSS-E-70-41A standard, as well as a good starting point for the development of the flight software of a new CubeSat.

The flight software of SwissCube was written to run on the MSP430 family of microcontrollers by Texas Instruments, and more specifically on the MSP430F1611. All of the subsystems are using this MCU, thus it is easy to share code for all of the subsystems. The CDMS was initially intended to handle commands sent from the GS as well as take care of the housekeeping for the whole satellite. However, due to complications, this subsystem was left offline on-board the flight model of SwissCube and couldn't exercise its role. This specific role was taken on by the EPS subsystem, in addition to its other responsibilities.

The goal of this document is to explain how the ECSS standard should be implemented on a CubeSat, by using the flight software of SwissCube as an example. The flight software of SwissCube was initially intended to support service types 1, 3, 4, 8 and 15 on top of the custom service 128. This document will exclusively describe the software of the EPS of SwissCube. The reason for this is because all of the top-level behavioural decisions are undertaken by this subsystem exclusively, including CCSDS packet decoding and generation, since the CDMS is unused in the final design of SwissCube. From this point on, we may consider the other subsystems (i.e. COM, ADCS, PL and ADS) as "black boxes" that respond to basic commands sent from the EPS through the I^{2}C bus.

Please note that it is advised to have read the ECSS-E-70-41A standard document before proceeding with this document, as a basic knowledge of the ECSS-PUS is required to understand some of the concepts explained. Furthermore, this document is only an overview of the flight software of SwissCube, thus details will be left out for the sake of readability and simplicity.

1 TOP-LEVEL SOFTWARE DESCRIPTION

This chapter covers the main characteristics of SwissCube's EPS flight software. The following sections will offer a basic overview of the source files and their roles within the project.

Please note that all of the source code for all of the subsystems is in the same folder. The root of SwissCube's flight software folder contains all of the source and header files that are common amongst all of the subsystems of the satellite, and that the source and header files containing the main loop and the main functionalities of each subsystem are saved in folders on the root named after the concerned subsystem (e.g. */eps* or */adcs*).

1.1 Definition headers

The following files simply contain macro and constant definitions that are used throughout the whole flight software, and withhold no function prototypes. These header files are common to the flight software of all of the subsystems (i.e. they can be included in any of the subsystems' source code), so the same definitions and macros are used throughout the whole flight software of SwissCube.

These files will not be analysed in detail in the following chapters, but they should be consulted by the reader for a better understanding of the main source code of the EPS flight software.

apids.h	Defines the APIDs (Application Process ID) used in the ECSS-PUS defined telecommand and telemetry packets for the SwissCube mission. Each subsystem is attributed one APID.
errors.h	Defines all the possible error codes that functions, or telecommands, may encounter during their execution.
functions.h	Defines the service type 8 function IDs for each of the subsystems of SwissCube. These IDs are used in order to identify functions with telecommand and telemetry packets.
hk.h	Defines the housekeeping database structure types and their characteristics (e.g. size), as well as some miscellaneous macros and structures to allow easy manipulation of these databases from any point of the flight software. Each subsystem has their own database, and the EPS has access to one additional database : the statistics database containing the minimum and maximal values since last statistics reset of several housekeeping parameters.
modes.h	Defines the different modes of the satellite. On the flight model of SwissCube, only two modes are defined : a *nominal* mode and a *safe* mode.
sids.h	Defines the SIDs (Structure IDs) that are used by the service type 3 functionalities (i.e. housekeeping parameter reports). Each SID identifies the contents of a given housekeeping report telemetry packet, so that it can be correctly interpreted from the GS.
storeids.h	Defines the Store IDs (not to be confused with SID) in which packets are stored for an indefinite amount of time before the GS sends a telecommand requesting the downlink of the packet store contents. There are two packet storages : the Acknowledgement storage (not used in the final version of SwissCube) and the

	Housekeeping archiving storage. This pertains to the On-board storage and retrieval service (service type 15).
subsystems.h	Several parameters in the housekeeping database describe the status of several subsystems on a single byte, with each subsystem being represented by a specific bit (e.g. enable/disable status of the subsystems). This header file defines the position, in the form of a bitmask, of each of the subsystem's bit for easy manipulation of these variables
types.h	Defines the custom types used throughout the software for coding convention consistency. Also contains macros for big endian to little endian (and vice versa) conversions and manipulations.
vc.h	Defines the IDs of the virtual channels used within AX.25 transfer frames, and are coded on 3 bits. This allows for up to eight virtual channels to run on a particular physical data channel.

Table 1: Definition headers of the general SwissCube flight software

1.2 Libraries

The following header and source files are functional libraries that allow the flight software to drive several hardware components, or manipulate abstracts concepts (such as CCSDS TM/TC packets, databases, checksums, etc.). As in the case of the previously presented header files, these libraries are common to all of the subsystems' flight software.

In the following chapters of this document, only the CCSDS library (ccsds.c and ccsds.h) will be looked into, as the main focus of this document is to explain the inner workings of the TM/TC and data management of SwissCube.

adc12_s3.c adc12_s3.h	This library holds the functions that allow the flight software to manipulate the MSP430F1611's 12-bit A/D converter channels, such as initialisation functions or conversion functions.
ccsds.c ccsds.h	The actual CCSDS packet handling is done by this library. It contains the type definitions for telemetry and telecommand structures, macros for packet manipulations, as well as functions for generating telemetry source packets.
crc.c crc.h	Cyclic redundancy checks (CRC) are used in order to ensure that the various communication packets flowing through the satellite aren't corrupted during their transmission. This library offers functions for initialising the CRC polynomial look-up table as well as functions for generating a CRC code from a given buffer.
i2c.c i2c.h	The internal communication of the satellite is done through the I^2C bus. The master (EPS) is able to communicate with and coordinate the other subsystems of SwissCube through this bus. This library handles the low level functions necessary to initialise and drive the I^2C transceiver of the MSP430F1611 (common to all subsystems).
time.c time.h	This library handles the on-board clock of each subsystem, and offers basic function for initialising the MSP430F1611's timers, and for getting the current time.

watchdog.c	This library contains functionalities for handling the MSP430F1611's watchdog timer. Its most important feature is a function for determining whether or not the watchdog timer has been triggered (used during the start up of a subsystem in order to know if it has been shut down due to a critical error), as well as macros for resetting the watchdog timer.
watchdog.h	

Table 2: Functionality libraries of the general SwissCube flight software

1.3 EPS source code

These last files are exclusive to the EPS, and contain the core source code of this subsystem. The main() function, taking care of all the initialisations and in which the main software loop takes place is contained in the eps.c source file.

Please note that the software that will be analysed in the following chapters of this document will pertain mostly to these source files.

eps_beacon.c	This source file handles the generation of the beacon signal, and includes the eps.h header file. It's the part of the EPS' flight software pertaining exclusively to the beacon.
eps_payload.c	This source file handles the control of the PL subsystem from the EPS, and contains the main function for payload image service handling to be called periodically in the program's mainloop. Same as eps_beacon.c, it includes the eps.h header file.
eps.c	This library is the main source code for the EPS subsystem. The main loop of the EPS is executed within the eps.c source file, and it effectively constitutes the central decision making process of the satellite. Service types 3, 8 and 15 are handled here.
eps.h	

Table 3: Source code of the EPS flight software

2 FUNCTIONAL OVERVIEW

This chapter will go over the main details of the software's operations. It will describe in particular the actions taken, grouped in steps, by the software's main loop. A look will then be cast on the software interruptions and their roles.

2.1 Main loop

Figure 1 is a flowchart of the main() function of the EPS flight software, in which the hardware and library initialisations are made and the main loop of the function is executed. A summary of the role of each of these major steps can be found further.

Between each major step of the main loop, the CLRWDT_EPS macro (defined in watchdog.h) is inserted in order to clear the watchdog timer and to avoid any system reset. It goes without saying that if the software is stuck for any reason in one of the major steps of the main loop, the watchdog timer will not be cleared in time and the EPS will be reset.

Measurements: handled by the *makeAllMeasures()* function (eps.c), this step handles the on-board measurements of voltages, currents and temperatures with the help of the 12-bit A/D converter of the MSP430F1611 and updates the housekeeping database of the EPS accordingly.

Scheduling: handled by the *timedEvents()* function (eps.c), this step handles pre-programmed scheduled events that take place a certain time after the deployment of the satellite in orbit. These scheduled events pertain mainly to the antenna deployment (see next step) and the initialisation of the on-board beacon.

Antenna deployment: handled by the *antennaDeployment()* function (eps.c), this step starts by checking if an antennaDeployment is under way (3 attempts of antenna deployment are pre-programmed in the previous scheduling step). If that is the case, it will check if the length of time the antenna deployment has been running has exceeded a pre-defined interval, in which case the antenna deployment mode is set to inactive and the ADS subsystem is turned off.

Time diffusion: handled by the *timeDiffusion()* function (eps.c), this step communicates to all of the active subsystems the current on-board time of the EPS, which is the master of the I^2C bus.

Telecommand retrieval: handled by *retrieveTelecommand()* (eps.c), this step retrieves a telecommand packet from the COM subsystem (if such a packet has been received), decodes it, takes the necessary actions associated with the telecommand, and then generates a telemetry source packet to the COM subsystem so that it can be downlinked to the GS. This step is only brought up if there is no measurement currently taking place on-board the PL subsystem. For more details, see chapter 3.

Payload service: handled by *handleImageService()* (eps_payload.c), this step handles the management of the PL subsystem by the EPS through the I^2C bus.

Watchdog handler: handled by *watchdog()* (eps.c), this step takes care of the watchdog functionality of the EPS. All of the subsystems are periodically polled (each time this step is brought up) by the EPS through the I^2C bus. If one of the subsystems cannot reply within a pre-defined length of time, the EPS will power cycle this subsystem.

Beacon update: handled by *Beacon_Update()* (eps_beacon.c), this step takes care of the beacon signal generation, and is only called in the case where the beacon signal needs to be updated.

The multiple tasks are for all intents and purposes run concurrently, in the sense that each of the tasks in the loop aren't time consuming and do not block the software ; the main loop keeps running rapidly and frequently. The housekeeping measurement task, the antenna deployment task, and the payload handling task are all asynchronous. Each call of these functions checks if the feature they are handling needs starting, is in progress, or needs processing of completion. For example, the housekeeping task when first called will initiate the conversions on the AD converter. The subsequent calls will check if the data is ready by checking if the AD converter is still busy and will immediately return if it isn't the case. As soon as the data is ready, the task will log the new housekeeping parameters into the housekeeping structures in the memory. The next call of the function will reinitiate the conversions on the AD converter. Thus the tasks are run concurrently, as housekeeping measurements can be made all the while a picture is being taken for example.

2.2 Software interruptions

In addition to the normal flow of the EPS flight software, a few interrupt routines are also used in order to respond to exterior asynchronous events. Most of the interrupt enabled pins (P1.0 through P1.5) are connected to the overcurrent signal lines of each subsystem's current limitation: this will notify the EPS of a latch-up event, and will allow it to know that the subsystem has been shutdown (the housekeeping database is updated accordingly).

The interrupt service routine which handles all of the software interrupts is Port1_ISR(). The software execution enters this function only when one of the pins of Port 1 detect a rising edge. Which pin has provoked the interrupt is determined within the function, and proper action is taken in consequence.

The interrupt pins of Port 1 are configured as follows :

Port 1 / Pin 0: Beacon current limiter

Port 1 / Pin 1: COM current limiter

Port 1 / Pin 2: CDMS current limiter

Port 1 / Pin 3: PL current limiter

Port 1 / Pin 4: ADCS current limiter

Port 1 / Pin 5: ADS current limiter

Port 1 / Pin 6: EPS emergency SAFE line

Additionally, an interrupt service routine triggered by a timer is used to generate the Morse message of the satellite's beacon.

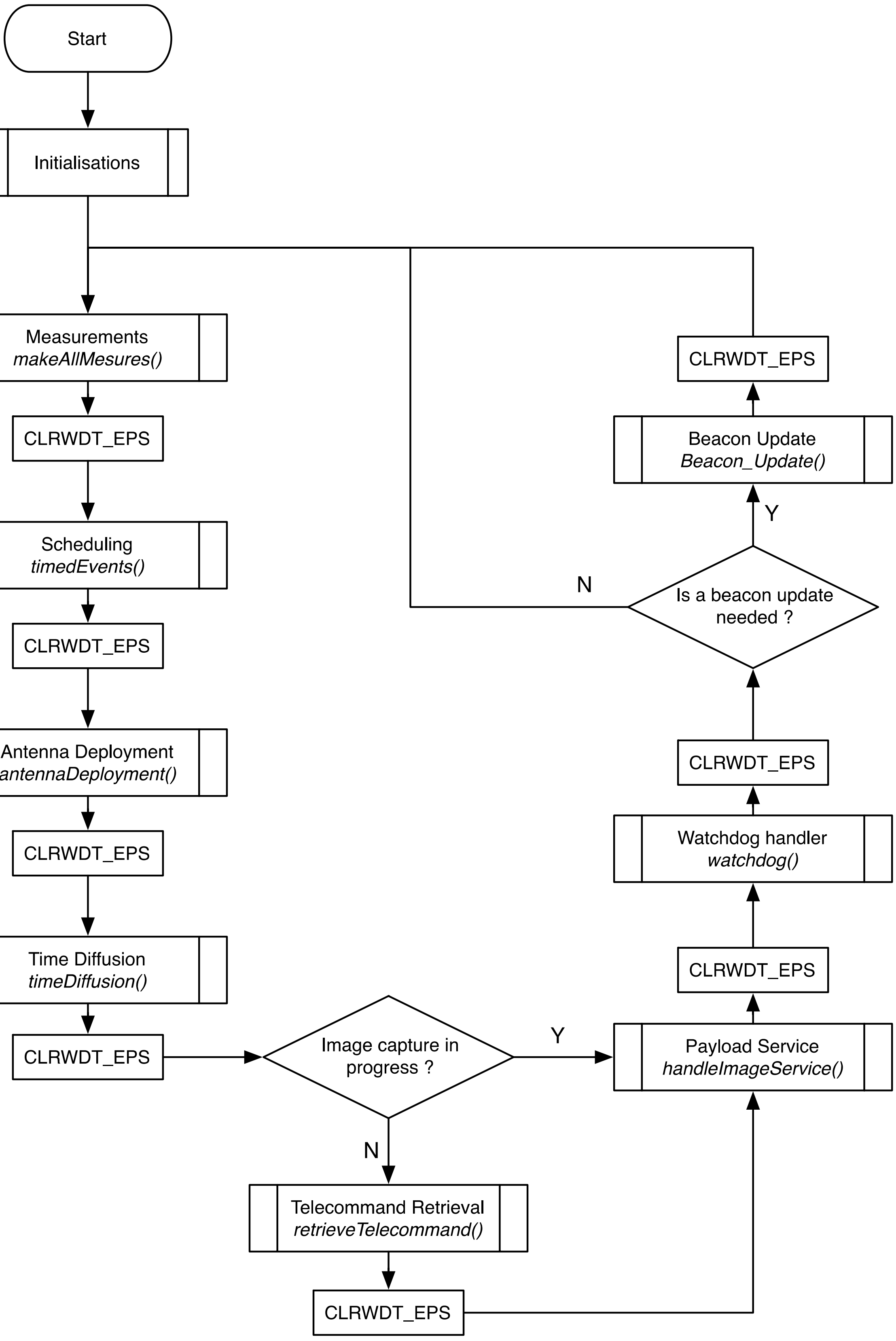

Figure 1: Flowchart of the main function of the EPS flight softwar

3 TM/TC HANDLING

This chapter will serve as an overview of the TM/TC packet handling. The first section will describe how the telecommand and telemetry source packets are represented in memory. The following sections will describe how TC packets are retrieved and how TM packets are generated, as well as how the different services supported by SwissCube are handled within the EPS flight software.

3.1 TM/TC structure definitions

This first section covers how the TC and TM packets are represented within the code of SwissCube. As a reminder, the structure of the telecommand and telemetry source packets from the ECSS standard can be found below.

Packet Header (48 Bits)							Packet Data Field (Variable)		
Packet ID				Packet Sequence Control		Packet Length	Telecommand Data Field Header	Application Data	Packet Error Control
Version Number	Type	Data Field Header Flag	APID	Sequence Flags	Sequence Count				
3	1	1	11	2	14				
16				16		16	24	Variable	16

Table 4: Telecommand packet structure

CCSDS Secondary Header Flag	TC Packet PUS Version Number	Ack	Service Type	Service Subtype
Boolean (1 bit)	Enumerated (3 bits)	Enumerated (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)

Table 5: Telecommand packet data field structure

Packet Header (48 Bits)							Packet Data Field (Variable)		
Packet ID				Packet Sequence Control		Packet Length	Telemetry Data Field Header	Source Data	Packet Error Control
Version Number	Type	Data Field Header Flag	APID	Grouping Flags	Source Sequence Count				
3	1	1	11	2	14				
16				16		16	64	Variable	16

Table 6: Telemetry source packet structure

Spare	TM Source Packet PUS Version Number	Spare	Service Type	Service Subtype	Time
Fixed BitString (1 bit)	Enumerated (3 bits)	Fixed BitString (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Absolute Time (40 bits)

Table 7: Telemetry source packet data field structure

Structures containing the different parts of the packets are defined as types, and are used to constitute the packets themselves which represented as structures defined as types. These definitions are held with the ccsds.h header file. Please note that the custom types (INT8U, INT16U, etc.) are defined with the types.h header file.

The CCSDS packet header is the same for both telemetry source packets and telecommand packets. The structure type withholding all of the information that is contained within this packet header is defined as follows :

CcsdsPacketHeader:

Packet ID	(INT16U)
PacketSequenceControl	(INT16U)
PacketLength	(INT16U)

The data field header of the telecommand packet is defined in figure 3. The structure type containing the complete information of this data field header can be found below. Please note that FlagVersionAck contains the first 3 entries of the data field header, which makes up for 8 bits : CCSDS Secondary Header Flag (1 bit), TC PUS Version Number (3 bit), and the Acknowledgment parameters (4 bit).

TelecommandDataFieldHeader:

FlagVersionAck	(INT8U)
ServiceType	(INT8U)
ServiceSubtype	(INT8U)

Thus the structure type for a complete telecommand packet is as follows :

Telecommand:

Header	(CcsdsPacketHeader)
DataFieldHeader	(TelecommandDataFieldHeader)

The data field header of the telemetry source packet is defined in figure 5. The structure type containing the complete information of this data field header can be found below. The SpareVersionSpare corresponds to the 3 first entries of the data field header, which makes up for 8 bits : 1 spare bit, TM Source Packet PUS Version Number (3 bit), 4 spare bits.

TelemetryDataFieldHeader:

SpareVersionSpare	(INT8U)
ServiceType	(INT8U)
ServiceSubtype	(INT8U)
C1	(INT8U)
C2	(INT8U)
C3	(INT8U)
C4	(INT8U)
F1	(INT8U)

The C1, C2, C3, C4 and F1 variables are the bytes of the CCSDS Unsegmented Code (CUC) time representation format, as defined in the ECSS standard. The representation of time can be found by multiplying each of these values by there corresponding scale, and adding this length of time to the reference time aboard the spacecraft (usually the launch date or the time of system boot in-orbit) :

$$\text{Time} = \text{Reference Time} + C1 * 2^{24} + C2 * 2^{16} + C3 * 2^{8} + C4 * 2^{0} + F1 * 2^{-8}$$

C1	C2	C3	C4	F1
2^{24}	2^{16}	2^{8}	2^{0}	2^{-8}

Table 8: CCSDS Unsegmented Code (CUC) time representation format

Thus the structure type for a complete telemetry source packet is as follows :

Telemetry:

Header	(CcsdsPacketHeader)
DataFieldHeader	(TelemetryDataFieldHeader)

3.2 Telemetry source packet generation

Two functions defined in the *ccsds.c* source file allow the software to create telemetry source packets :

CCSDS_GenerateTelemetryPacket()

CCSDS_GenerateTelemetryPacketWithTime()

These functions take as argument the telemetry buffer array and its size, in which the telemetry source packet will be written. As the length of telemetry source packet is variable depending on its content, the pointer of the size of the telemetry is passed as argument, so that its value can be written by the function. Note that the telemetry buffer is initialised with a size corresponding to the maximum size of a telemetry source packet and the telemetry buffer size is initialised to this maximum size before calling the packet generation function. The arguments relevant to the creation of the telemetry source packet are the APID, the service type and subtype, and the telemetry data buffer with its size.

It is important to note that the *CCSDS_GenerateTelemetryPacketWithTime()* creates a packet in which the user can specify a custom timestamp for the packet, and is mainly used by archive downlink telemetry source packets in which the time of the first entry of the archive is relevant, and not the time of the creation of the packet.

In contrast, *CCSDS_GenerateTelemetryPacket()* creates a packet with the timestamp automatically set to the time of the generation of the telemetry source packet. In effect, this function calls the *CCSDS_GenerateTelemetryPacketWithTime()* function by passing the current time as argument, after having determined the current on-board time of the satellite through the *Time_Get()* function of the *time.c* source file.

Before actually writing on the telemetry buffer, these functions make the following checks on the contents of the telemetry source packet :

- Verifies if the *sourceData* pointer on the telemetry data buffer is not NULL.
- Verifies if the *telemetryBuffer* pointer on the buffer intended to be written with the telemetry source packet is not NULL.
- Verifies if the *telemetryBufferSize* pointer on the variable containing the telemetry buffer size is not NULL.
- Verifies if the total length of the *sourceData* buffer (stored in *sourceDataLength*) does not exceed the maximum size of telemetry data defined for the SwissCube mission.
- Verifies if the total length of the *telemetryBuffer* buffer (stored in *telemetryBufferSize*) is high enough to contain the data and the CCSDS telemetry source packet header.

If any of these verifications fail, the function returns an error code and doesn't generate a telemetry source packet.

3.3 Telecommand retrieval and interpretation

Telecommand retrieval on the side of the EPS subsystem is done periodically. The *retrieveTelecommand()* function is called in the main loop of the software as long as there is no image capture in process (as to avoid sending telecommands to the payload while it is currently busy). A time of last telecommand retrieval is stored as a static variable in the function, and if the time since the last retrieval is higher than a certain threshold (defined by the INTERVAL_REQUEST_COM_TELECOMMAND macro), the EPS will proceed with the request through the I^{2}C bus of the oldest telecommand packet that has been received by the COM subsystem. If the reply of the COM contains a telecommand packet, the *decodeCCSDSTelecommand()* function is called.

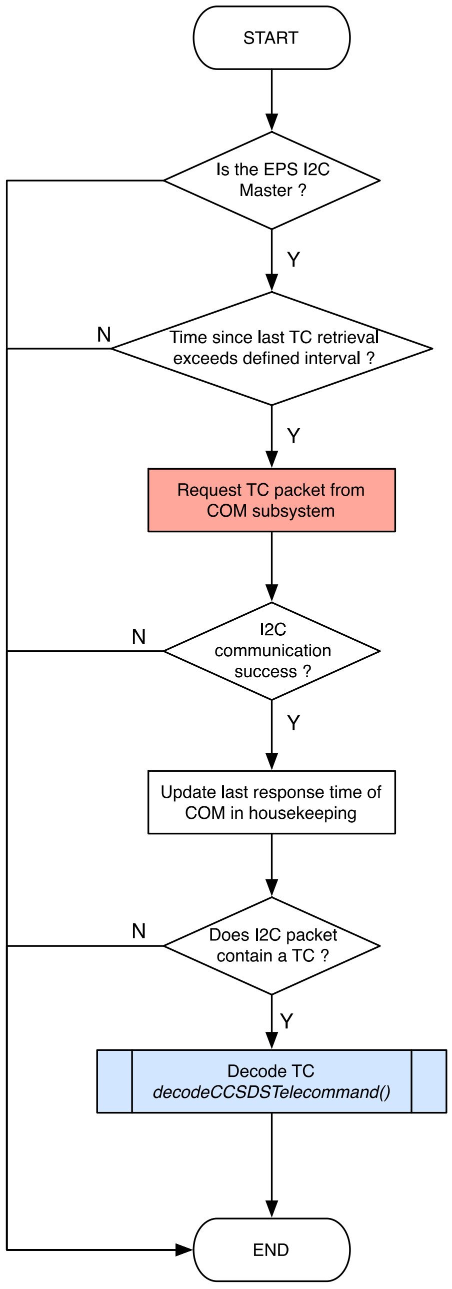
START
Is the EPS I2C Master ?
Y
N
Time since last TC retrieval exceeds defined interval ?
Y
Request TC packet from COM subsystem
N
I2C communication success ?
Y
Update last response time of COM in housekeeping
N
Does I2C packet contain a TC ?
Y
Decode TC
decodeCCSDSTelecommand()
END

Figure 2: Flowchart of the *retrieveTelecommand()* function - Red processes imply the use of the I2C bus, blue processes imply the downlink of data

The *decodeCCSDSTelecommand()* function is called in order to interpret the service to which the received telecommand belongs to. Depending on the service of the telecommand packet, the appropriate function is called in order to decode the telecommand packet. The following flowchart illustrates how this was implemented on SwissCube. Keep in mind that decoding functions for service type 3, 11, and 15 should also implemented, which isn't the case in this flowchart since SwissCube had a reduced TM/TC dictionary due to development time and hardware constraints.

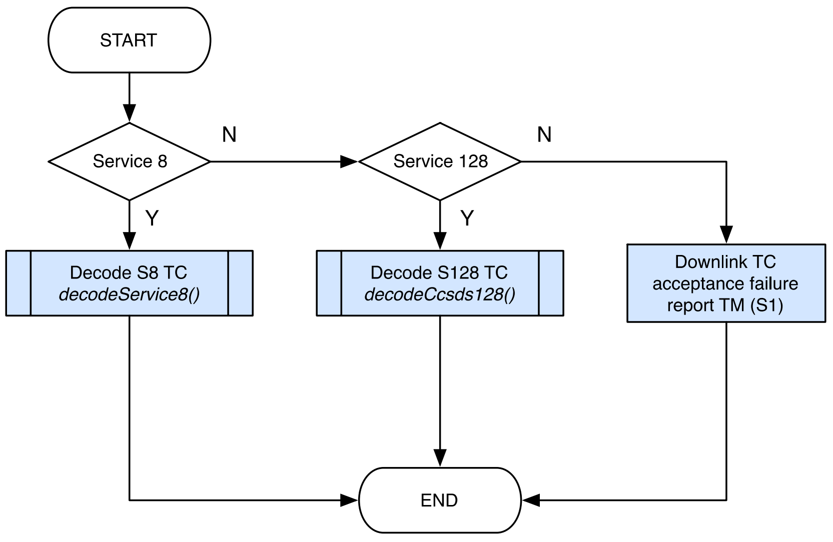

Figure 3: Flowchart of the *decodeCCSDSTelecommand()* function - Blue processes imply the downlink of data

The following macros are used in order to interpret and decode CCSDS telecommand packets on the EPS (*ccsds.h*), by allowing the user to extract the specific fields of a packet. :

PACKET_LENGTH(packet)

PACKET_ID(packet)

PACKET_PEC(packet)

PACKET_APID(packet)

PACKET_SEQUENCE_CONTROL(packe)

PACKET_SEQUENCE_COUNT(packet)

PACKET_SERVICE_TYPE(packet)

PACKET_SERVICE_SUBTYPE(packet)

TC_ACK(packet)

TC_DATA(packet)

TC_DATA_LENGTH(packet)

TM_TIME(packet)

TM_DATA(packet)

TM_DATA_LENGTH(packet)

3.4 Service handling

The following subsections will go over the handling of the services intended to be offered by SwissCube, and of how they should be implemented in a cubesat using the same simplified TM/TC dictionary. Note that service types 11 and 15, were not implemented as such in SwissCube due to time and hardware constraints, but the examples that follow will illustrate how they should be implemented on the functional level.

Service 1 : Telecommand verification service			
		1	Acceptance success report
		2	Acceptance failure report
		3	Execution started success report
		4	Execution started failure report
		7	Completion success report
		8	Completion failure report
Service 3 : Housekeeping & diagnostic data reporting service			
		25	Housekeeping parameter report
Service 8 : Perform management service			
1	Perform function		
Service 11 : On-board operations scheduling			
1	Enable release of telecommands		
2	Disable release of telecommands		
3	Reset telecommand schedule		
4	Insert telecommand in schedule		
5	Delete telecommands		
6	Delete telecommands over time period		
7	Time-shift selected telecommands		
15	Time-shift all telecommands		
17	Report command schedule as summary	13	Summary schedule report
Service 15 : On-board storage and retrieval service			
1	Enable storage in packet stores		
2	Disable storage in packet stores		
		8	Packet Store contents report
Service 128 – 255 : Custom services			

Table 9: Complete SwissCube TM/TC dictionary (right side: TM packets / left side: TC packets)

3.4.1 Service type 1 : Telecommand verification service

The telecommand verification service is handled during the decoding and execution of incoming telecommand packets in the scope of other services. Two functions defined in the *eps.c* file, as evidenced below, are used to either create a success report or a failure report.

```
INT8U sendTelecommandReport_Success(INT8U* telecommand, INT8U reportType)
INT8U sendTelecommandReport_Failure(INT8U* telecommand, INT8U reportType, INT8U err)
```

The *reportType* parameter of these functions define whether the report concerns the acceptance of the telecommand, the start of its execution, or its completion. A pointer to the telecommand is passed as argument, since the report needs to include the header of the concerned telecommand. Finally, the *err* parameter in the failure report function is used by the satellite to report the nature of the error (illegal service type or subtype, illegal APID, I^2C error, etc.).

3.4.2 Service type 3 : Housekeeping & diagnostic data reporting service

The only subservice implemented in this service is the housekeeping parameter report subservice. This telemetry source packet contains housekeeping parameters that are defined by the SID given at the beginning of the packet's data field. It is important to note that this telemetry source packet is intended to be sent in a *periodic* or *filtered* manner, as described in the ECSS standard, and thus does not require a request from the ground segment to be transmitted (unless service type 15 is used to store the telemetry source packets of this service).

The function that allows the software of SwissCube to generate service 3 / subservice 25 telemetry source packets is the following :

```
static INT8U generateSendHkPacket  (INT8U* messageBuffer,
                                    INT8U messageBufferLength,
                                    INT8U* hkBuffer,
                                    INT8U hkBufferLength,
                                    INT8U sid)
```

The *messageBuffer* array and the corresponding *messageBufferLength* size of the array are the parameters in which the message containing the telemetry source packet will be sent to the COM subsystem through the I^2C bus. The *hkBuffer* array and the corresponding *hkBufferLength* size of the array contain the housekeeping parameters that need to be downlinked. The *sid* parameter simply states the SID that will be downlinked to the ground station.

As service type 15 wasn't implemented on SwissCube due to time and hardware constraints, the downlink of archives of data was taken care of by functions in service type 3, where arrays of data would be downlinked in a defined SID. For an illustration on how this should have been implemented instead of in service type 3, please refer to section 3.4.5.

The following figure contains a description of the function dependencies pertaining to the generation of service 3 telemetry source packets.

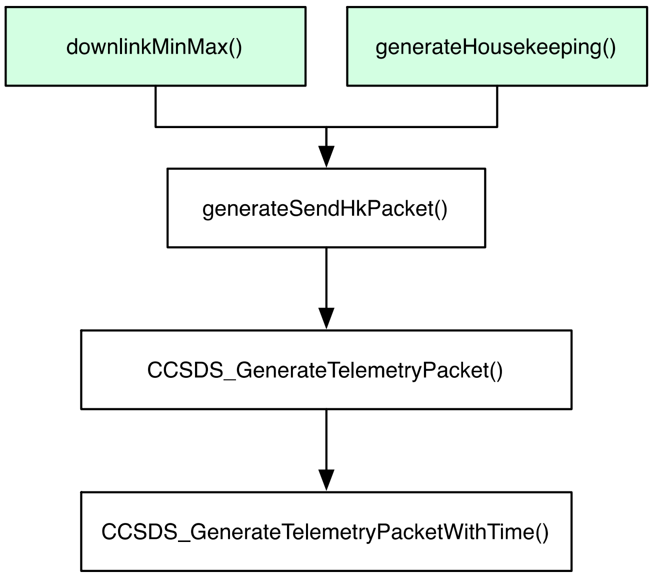
downlinkMinMax()
generateHousekeeping()
generateSendHkPacket()
CCSDS_GenerateTelemetryPacket()
CCSDS_GenerateTelemetryPacketWithTime()

Figure 4: Function dependencies for housekeeping parameters report generation

Note that minimum and maximum values in the case of SwissCube are part of the housekeeping data, and not the statistics data which should've been handled by service type 4 intended for statistics parameters such as min/max, mean and standard deviation values. The reason for this is because the minimum and maximum values of the temperature, voltage and current alone do not justify the need for a whole service for their handling. They were therefore put together with the rest of the housekeeping data (functions for resetting the statistics are accessed through service type 8). If a more complex statistics generation is required by the mission, the implementation of service type 4 is strongly recommended.

3.4.3 Service type 8 : Function management service

The flowchart presented in figure 6 depicts how service 8 telecommand packets are handled by SwissCube's EPS, in the *decodeService8()* function. The processes highlighted in red are the tasks making use of the I^2C bus, and the processes highlighted in blue imply the downlink of data and thus the use of the I^2C bus as well in order to relay the packet to the COM subsystem. Two parameters are important for the handling of the incoming telecommand packet by the function.

The *acceptanceError* parameter (initialised at *ERR_SUCCESS*) is an error flag containing the status of the acceptance of the telecommand packet. If an error is detected in the packets format (e.g. illegal parameters), this error flag is set to a value describing the nature of the error. It is otherwise left untouched.

The *completionError* parameter (initialised at *ERR_SUCCESS*) is an error flag containing the status of the completion of the request contained within the telecommand packet. If an error has occurred during the execution of the request (e.g. error on the I^2C bus), this error flag is set to a value describing the nature of the error. It is otherwise left untouched.

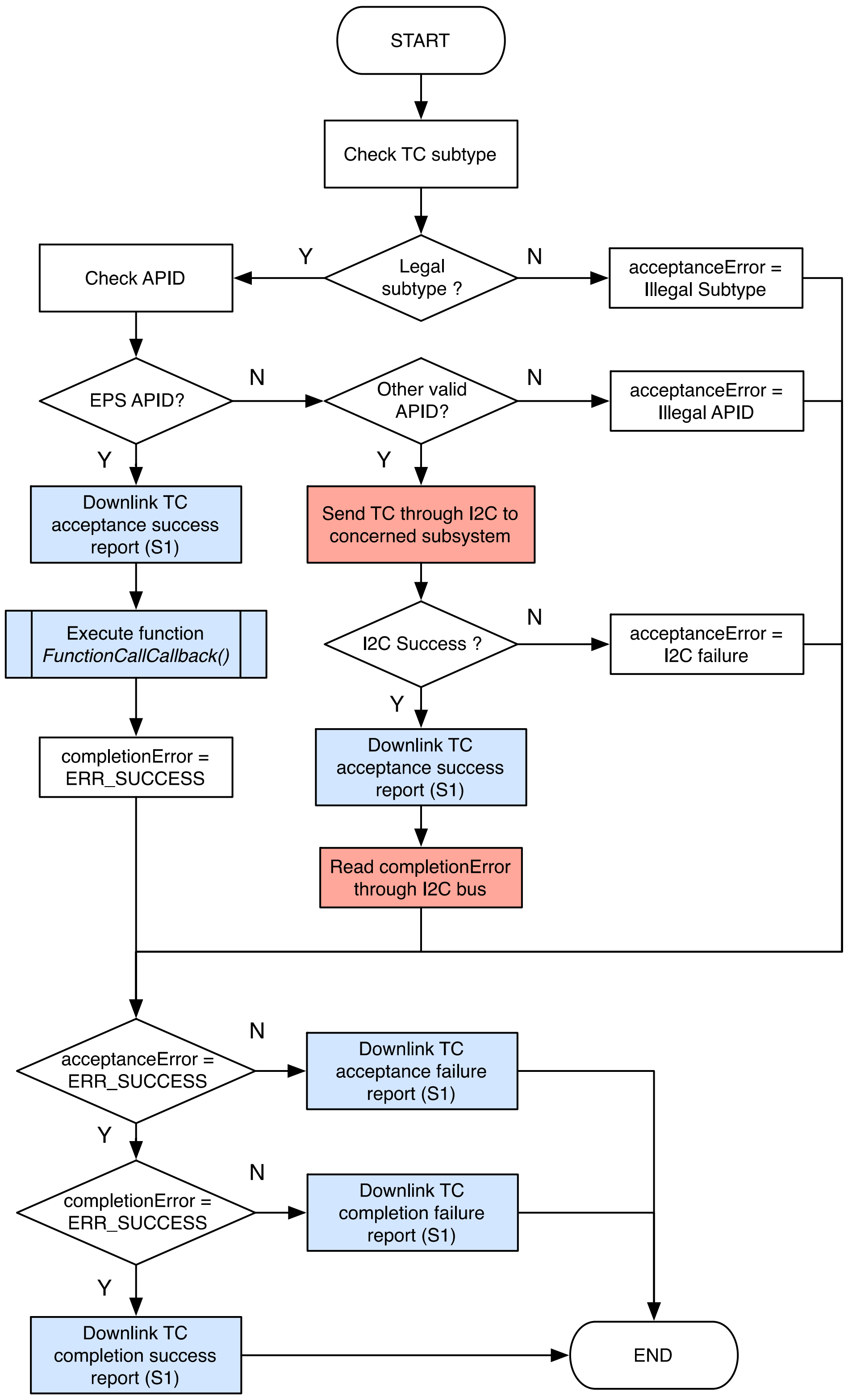
START
Check TC subtype
Legal subtype ?
Y
N
Check APID
acceptanceError = Illegal Subtype
EPS APID?
N
Other valid APID?
N
acceptanceError = Illegal APID
Y
Y
Downlink TC acceptance success report (S1)
Send TC through I2C to concerned subsystem
Execute function FunctionCallCallback()
I2C Success ?
N
acceptanceError = I2C failure
Y
completionError = ERR_SUCCESS
Downlink TC acceptance success report (S1)
Read completionError through I2C bus
acceptanceError = ERR_SUCCESS
N
Downlink TC acceptance failure report (S1)
Y
completionError = ERR_SUCCESS
N
Downlink TC completion failure report (S1)
Y
Downlink TC completion success report (S1)
END

Figure 6: Flowchart of the *decodeService8()* function - Blue processes imply downlink of data and red processes imply use of I^2C bus

The list of functions that can be accessed through the function management service is listed in a header file by the name of *functions.h* for convenience. The *FunctionCallCallback()* function handles the service 8 requests by communicating the request to the targeted subsystem through the I^2C bus and generating telemetry source packets as reply.

3.4.4 Service type 11 : On-board operations scheduling service

The implementation of this service is straightforward : scheduled telecommands need to be stored in the memory of the controller supporting this service (such as a Command and Data Management Subsystem) along their proper time tags. On the basis of an interruption, or of a simple poll initiated by a timer, scheduled telecommands that are due are released to the proper subsystem.

The use of a linked list or of a fixed size array are both possible solutions for the storage of the scheduled telecommands, however the use of dynamic allocations (in the case of a linked list) is greatly discouraged for software robustness reasons. However, some Real Time Operating Systems such as µC-OS/II by Micrium offer robust alternatives to the standard dynamic allocation operated through the *malloc()* function.

3.4.5 Service type 15 : On-board storage and retrieval service

If the number of ground stations used to interact with the satellite are few and don't allow for a high coverage of the satellite, the use of service type 15 should be considered. There are two reasons why this would be useful :

- For the downlink of data archives : service type 3 housekeeping reports could be periodically logged into multiple entry packet stores that act as circular buffers (older entries are overwritten by newer entries).
- For lowering satellite power consumption : instead of having the APIDs supporting service type 3 constantly downlink housekeeping reports, the newly generated telemetry source packets could be stored in a single entry packet store (thus overwriting the previous entry) for retrieval on request of a ground station.

Service type 15 implies rerouting telemetry source packets to packet stores (data placeholders in the on-board computer's memory, such as arrays) of a given SID[1], dedicated to this given type of telemetry source packet, when storage in packet stores is enabled. This can be enabled or disabled through the (15,1) and (15,2) telecommands individually for each packet store; when disabled, telemetry source packets would not be rerouted to their designated packet store and would be downlinked as soon as they are generated.

In order to retrieve these telemetry source packets, a (15,9) request for the downlink of a given packet store must be sent to the satellite from the ground station. Note that the downlinked packets don't necessarily need to be encapsulated in a (15,8) packet and can be downlinked as is, with the type and subtype they were generated with.

Whether or not a telemetry source packet is rerouted to a packet store, the number of entries of a packet store, and the size of an entry in the packet store (normally the maximum size of the telemetry

[1] Store ID, not to be confused with the Structure IDs used in service type 3

source packet intended to be stored in it), are entirely up to the developer. However an ICD containing a rigorous description of the Store IDs and their corresponding Structure IDs needs to be created.

The following figure is an illustration of how this service is intended to be implemented for the downlink and archiving of certain housekeeping parameters (such as quaternions, battery voltages and temperatures) on CubETH, another satellite being developed by the Swiss Space Center in collaboration with other Swiss universities.

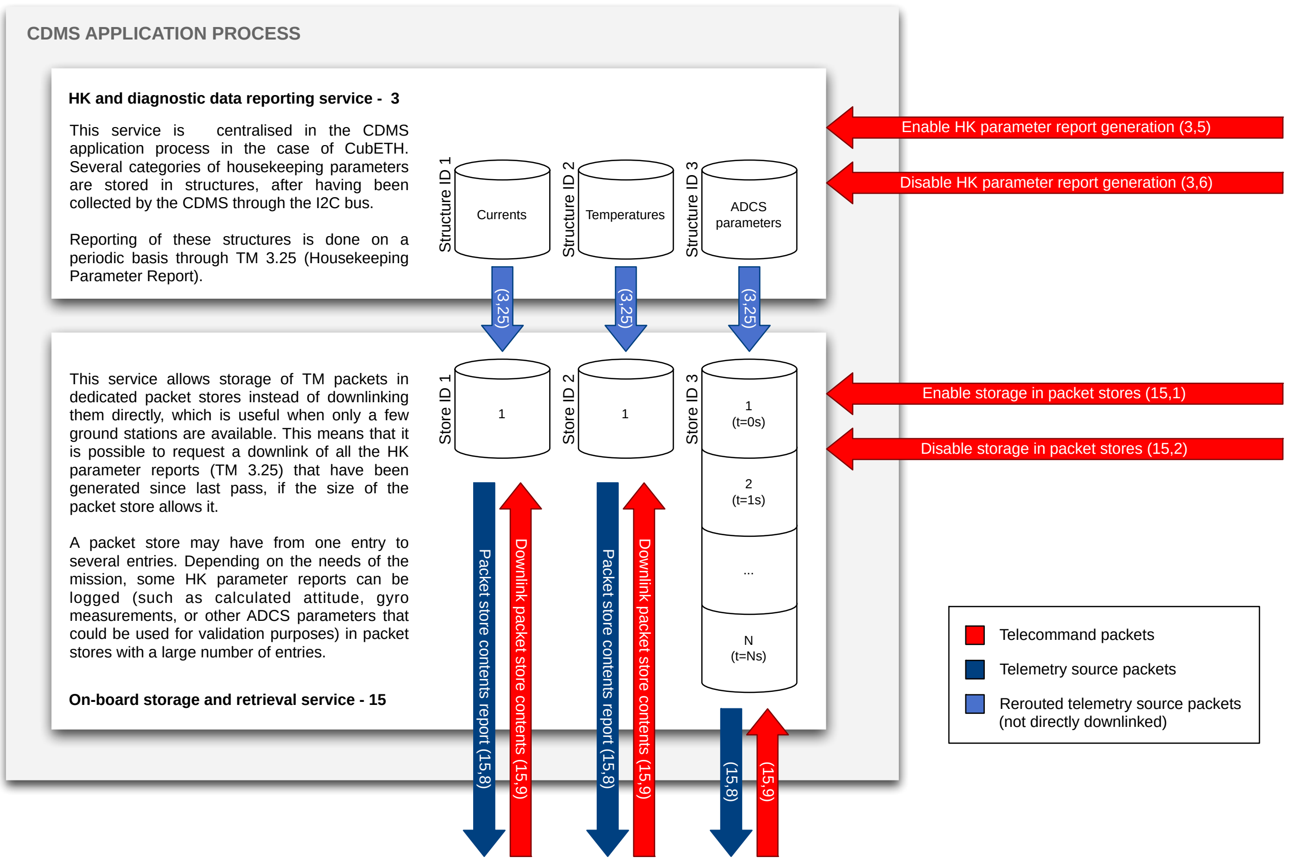
CDMS APPLICATION PROCESS
HK and diagnostic data reporting service - 3
This service is centralised in the CDMS application process in the case of CubETH. Several categories of housekeeping parameters are stored in structures, after having been collected by the CDMS through the I2C bus.
Reporting of these structures is done on a periodic basis through TM 3.25 (Housekeeping Parameter Report).
Structure ID 1
Currents
Structure ID 2
Temperatures
Structure ID 3
ADCS parameters
(3.25)
(3.25)
(3.25)
Enable HK parameter report generation (3,5)
Disable HK parameter report generation (3,6)
This service allows storage of TM packets in dedicated packet stores instead of downlinking them directly, which is useful when only a few ground stations are available. This means that it is possible to request a downlink of all the HK parameter reports (TM 3.25) that have been generated since last pass, if the size of the packet store allows it.
A packet store may have from one entry to several entries. Depending on the needs of the mission, some HK parameter reports can be logged (such as calculated attitude, gyro measurements, or other ADCS parameters that could be used for validation purposes) in packet stores with a large number of entries.
On-board storage and retrieval service - 15
Store ID 1
1
Store ID 2
1
Store ID 3
1 (t=0s)
2 (t=1s)
...
N (t=Ns)
Enable storage in packet stores (15,1)
Disable storage in packet stores (15,2)
Packet store contents report (15,8)
Downlink packet store contents (15,9)
Packet store contents report (15,8)
Downlink packet store contents (15,9)
(15,8)
(15,9)
Telecommand packets
Telemetry source packets
Rerouted telemetry source packets (not directly downlinked)

Figure 5: Example of how service type 15 is used for housekeeping telemetry source packets in CubETH

3.4.6 Service type 128 : SwissCube payload service (Example)

The following flowchart is an example of how the handling of service type 128 telecommands could be handled. This example depicts how service 128 telecommand packets are handled by SwissCube's EPS, in the *decodeCcsds128()* function. The processes highlighted in blue imply the downlinking of data, and thus the use of the I^{2}C bus in order to send packets to the COM subsystem.

Since this custom service is specifically tailored for SwissCube, not much detail will be put into the sub-processes that relate to each specific telecommand of the service. However, it is important to note that all of these sub-processes handle by themselves the generation of failure or success acknowledgement reports, as well as the downlink of other telemetry source packets.

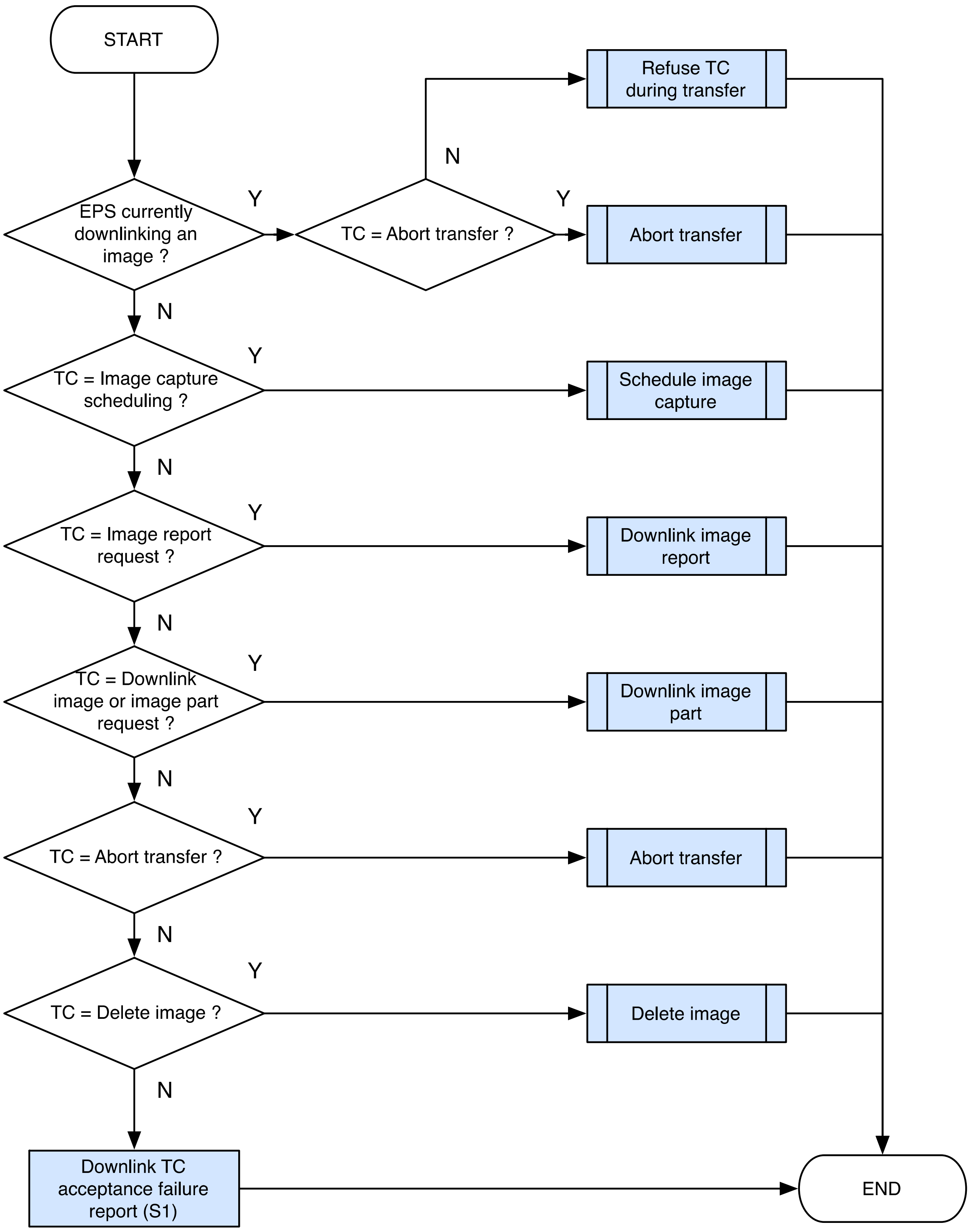
START
EPS currently downlinking an image ?
Y
N
TC = Abort transfer ?
Y
N
Refuse TC during transfer
Abort transfer
TC = Image capture scheduling ?
Y
Schedule image capture
N
TC = Image report request ?
Y
Downlink image report
N
TC = Downlink image or image part request ?
Y
Downlink image part
N
TC = Abort transfer ?
Y
Abort transfer
N
TC = Delete image ?
Y
Delete image
N
Downlink TC acceptance failure report (S1)
END

Figure 7: Flowchart of the *decodeCcsds128()* function - Blue processes imply downlink of data

CONCLUSION

This document gives the reader a first hand example of the implementation of the ECSS standard with the example of SwissCube's flight software, and should help the reader get started on the development of the software architecture of a new cubesat.

However, even though lessons can be taken from of the flight software of SwissCube, the source code of this cubesat must be taken with a grain of salt : some of the ECSS standard services are not implemented explicitly in the source code, and are implemented implicitly instead.

- The On-board storage and retrieval service (type 15) is implicitly supported by SwissCube through the downlink of housekeeping archives, the downlink of archives is done through a service type 8 function call and not the proper service type 15 telecommand.
- The statistics reporting (min/max values), which is normally handled in the Parameter statistics reporting service (service type 4), but is handled in the Housekeeping & diagnostic data reporting service (service type 3) in the case of SwissCube.
- The way the housekeeping parameter reports are downlinked is not compliant with the ECSS standard. These reports (service 3, subservice 25) are usually generated either periodically or in a filtered manner (for instance, if the parameter change exceeds a threshold), while in the case of SwissCube they are downlinked after a service type 8 telecommand with a specific function.

It must be noted however that the reasons for these differences with the pure ECSS standard are due to time and technical constraints. Initially, a CDMS (Command & Data Management Subsystem) was intended to handle the services tied to housekeeping and statistics (service types 3 and 4), as well as packet storage and retrieval (service type 15). Towards the end of the development of SwissCube, the decision was taken to remove this subsystem from the final design because of flaws in its design. The EPS was intended to take on the responsibilities of the CDMS but, because of its lack of memory and processing power, it was decided to use these shortcuts in the standard and to use service type 8 as a risk-reduction measure.

This document illustrates how the ECSS standard should be implemented on a CubeSat's flight software. However, the shortcuts to the standard in SwissCube mentioned in the document must remain at the process level (i.e. how the packets are handled within the satellite), and not at the data level (i.e. format of the data contained in the packets). Despite these shortcuts, SwissCube could be controlled and monitored with a fully ECSS-compliant MCS, since no modifications to the data format of the packets of the ECSS standard were modified.

Appendix A downlinkMinMax() – eps.c

```
/*-----------------------------------------------

* downlinkMinMax()

* -----------------------------------------------

* Input  :

*  none

*

* Output :

*  error : ERR_SUCCESS when no error has occured, and error ID otherwise

*

* Description:

*  This function makes use of generateSendHkPacket() in order to prepare a

*  housekeeping packet containing the statistic min / max values.

*

*  NB: This function is exclusively accessed from the ground segment.

*

* -----------------------------------------------*/


static INT8U downlinkMinMax(void)
{
        INT8U messageBuffer[sizeof(minMaxHK)+32]; // HK + CCSDS + I2C with margin
        return generateSendHkPacket(messageBuffer, sizeof(messageBuffer), (INT8U*)&minMaxHK,
sizeof(minMaxHK), SID_EPS_MM_RT);
}

static INT8U generateAndSendArchiveHk(INT8U *dataBuffer, INT8U dataBufferLength, INT32U time)
{
        INT8U messageBuffer[254]; // HK + CCSDS + I2C with margin
        INT8U packetLength = sizeof(messageBuffer)-6; // Max packet size in I2C message (+FCS)
        INT8U err;

        // Generate packet with custom time
        err = CCSDS_GenerateTelemetryPacketWithTime(&messageBuffer[4], &packetLength, APID_EPS,
CCSDS_T3_HK_DIAGNOSTIC_REPORTING, CCSDS_ST_S3_HK_REPORT, dataBuffer, dataBufferLength, time);
        if(err == ERR_SUCCESS)
        {
                // Send packet to COM
                messageBuffer[0] = I2C_TYPE_FUNC_CALL_REQUEST;
                messageBuffer[1] = FCT_COM_SEND_PKT;
                messageBuffer[2] = SC_VC_HK_RT;
                messageBuffer[3] = 0; // Spare

                packetLength = 4 + packetLength; // I2C bytes
                err = i2cWriteAndCheckError(I2C_ADDR_COM, messageBuffer, &packetLength);
        }

        return err;
}
```

Appendix B generateHousekeeping() – eps.c

```
/*-----------------------------------------------
* generateHousekeeping()
* -----------------------------------------------
* Input  :
*  none
*
* Output :
*  error : ERR_SUCCESS when no error has occured, and error ID otherwise
*
* Description:
*  This function serves the purpose of downlinking the housekeeping of each
*  subsystem at a time. The CCSDS packets are built then sent to the COM sub-
*  system by using the generateSendHkPacket() function.
*
*  The first housekeeping SID to be downlinked is the EPS housekeeping. Once
*  it has been transferred, the function creates and downlinks another house-
*  keeping report packet for each subsystem that is online. In order to do so,
*  it sends a housekeeping report request on the I2C bus to the enabled and
*  powered up subsystems, then builds and sends the packet by using the
*  generateSendHkPacket() function. The subsystems that are prompted for HK
*  and for which their SID is downlinked are the COM, ADCS, and PL subsystems.
*
*  NB: This function is exclusively accessed from the ground segment.
*
* -----------------------------------------------*/

static INT8U generateHousekeeping(void)
{
   INT8U err = ERR_SUCCESS;
   INT8U dataRead = 0;
   INT16U i;

   // I2C Messages with COM (Message type : Function call request)
   // Sizes : 1 = message type (I2C_TYPE_FUNC_CALL_REQUEST)
   //         1 = function ID (FCT_COM_SEND_PKT)
   //             1 = virtual channel
   //             * = size of the CCSDS packet that is sent
   //                         CCSDS fields, 1 SID, HK size
   INT8U messageBuffer[255]; // Maximum size
   INT8U ssHk[128];
```

```
    // Update the housekeeping
    // Copy housekeeping
    {
        INT16U *sourcePtr = (INT16U*)&housekeeping;
        INT16U *destPtr = (INT16U*)ssHk;
        INT32U *dwordPtr = (INT32U*)ssHk;

        _DINT();
        for(i=0 ; i<((sizeof(housekeeping)+1)/2) ; i++)
                *(destPtr++) = *(sourcePtr++);
        _EINT();

        // Last report times, little-endian -> big-endian
        dwordPtr[0] = INT32U_LE2BE(dwordPtr[0]);
        dwordPtr[1] = INT32U_LE2BE(dwordPtr[1]);
        dwordPtr[2] = INT32U_LE2BE(dwordPtr[2]);
        dwordPtr[3] = INT32U_LE2BE(dwordPtr[3]);
    }

    // Generate and send EPS HK
    generateSendHkPacket(messageBuffer, sizeof(messageBuffer), ssHk, HK_EPS_SIZE, SID_EPS_COMPLETE_RT);

    // Generate COM' housekeeping packet if COM is powered up
    if(ST_COM(housekeeping))
    {
        // HACK: Small wait for COM flushing
        for(i=0 ; i<0x3FFF ; i++)
                P5OUT &= ~BIT0;
        CLRWDT_EPS;

        // Retrieve HK from Subsystem
        messageBuffer[0] = I2C_TYPE_HK_REQUEST;
        // Request the HK
        err = I2C_MasterWrite(I2C_ADDR_COM, messageBuffer, 1);
        if(err == ERR_SUCCESS)
        {
                // Read the HK
                err = I2C_MasterRead(I2C_ADDR_COM, ssHk, &dataRead);
                if(err==ERR_SUCCESS && ssHk[0]==I2C_TYPE_HK_REPORT)
                        generateSendHkPacket(messageBuffer,   sizeof(messageBuffer),   ssHk+2,   dataRead-4,
SID_COM_RT);
        }
    }

    // Generate ADCS' housekeeping packet if ADCS is powered up
    if(ST_ADCS(housekeeping))
    {
        // HACK: Small wait for COM flushing
        for(i=0 ; i<0x3FFF ; i++)
                P5OUT &= ~BIT0;
        CLRWDT_EPS;

        // Retrieve HK from Subsystem
        messageBuffer[0] = I2C_TYPE_HK_REQUEST;
        // Request the HK
        err = I2C_MasterWrite(I2C_ADDR_ADCS, messageBuffer, 1);
        if(err == ERR_SUCCESS)
        {
                // Read the HK
                err = I2C_MasterRead(I2C_ADDR_ADCS, ssHk, &dataRead);
                if(err==ERR_SUCCESS && ssHk[0]==I2C_TYPE_HK_REPORT)
                        generateSendHkPacket(messageBuffer,   sizeof(messageBuffer),   ssHk+2,   dataRead-4,
SID_ADCS_RT);
        }
    }

    // Generate Payload's housekeeping packet if Payload is powered up
    if(ST_PL(housekeeping))
    {
        // HACK: Small wait for COM flushing
        for(i=0 ; i<0x3FFF ; i++)
                P5OUT &= ~BIT0;
        CLRWDT_EPS;

        // Retrieve HK from Subsystem
        messageBuffer[0] = I2C_TYPE_HK_REQUEST;
        // Request the HK
        err = I2C_MasterWrite(I2C_ADDR_PAYLOAD, messageBuffer, 1);
        if(err == ERR_SUCCESS)
        {
                // Read the HK
                err = I2C_MasterRead(I2C_ADDR_PAYLOAD, ssHk, &dataRead);
                if(err==ERR_SUCCESS && ssHk[0]==I2C_TYPE_HK_REPORT)
                        generateSendHkPacket(messageBuffer,   sizeof(messageBuffer),   ssHk+2,   dataRead-4,
SID_PL_RT);
        }
    }

    return err;
}
```

Appendix C generateSendHkPacket() – eps.c

```
/*-----------------------------------------------
* generateSendHkPacket()
* -----------------------------------------------
* Input  :
*  *messageBuffer : I2C message to be sent to the COM subsystem
*  messageBufferLength : length of the I2C message buffer.
*  *hkBuffer : buffer containing the housekeeping parameters to be downlinked
*  hkBufferLength : length of the downlinked housekeeping parameters
*  sid : structure ID corresponding to the downlinked housekeeping data
*
* Output :
*  error : ERR_SUCCESS when no error has occured, and error ID otherwise
*
* Description:
*  This function is used in order to generate a housekeeping packet from a
*  given housekeeping buffer identified by an SID. The packet is created by
*  using the CCSDS_GenerateTelemetryPacket() function of the ccsds library.
*  Once the packet created, it is sent to the COM by using the
*  i2cWriteAndCheckError() function.
*
* -----------------------------------------------*/

static INT8U generateSendHkPacket(INT8U* messageBuffer,  INT8U messageBufferLength,
                                                         INT8U* hkBuffer,
                                                         INT8U hkBufferLength,
                                                         INT8U sid)
{
   INT8U err;
   INT8U packetLength;
   INT16U i;

   // HACK: Copy HK to add SID
   INT8U hackBuffer[128];
   hackBuffer[0] = sid;
   for(i = 0; i < hkBufferLength; i++)
        hackBuffer[i + 1] = *(hkBuffer++);

   // Packet at offset 4, Type + Func ID + VC + Spare before
   packetLength = messageBufferLength-6; // Max packet size in I2C message (+FCS)
   // Warning SID before housekeeping!
```

```
    err        =        CCSDS_GenerateTelemetryPacket(&messageBuffer[4],       &packetLength,       APID_EPS,
CCSDS_T3_HK_DIAGNOSTIC_REPORTING, CCSDS_ST_S3_HK_REPORT, hackBuffer, hkBufferLength+1);
    if(err == ERR_SUCCESS)
    {
         // Send packet to COM
         messageBuffer[0] = I2C_TYPE_FUNC_CALL_REQUEST;
         messageBuffer[1] = FCT_COM_SEND_PKT;
         messageBuffer[2] = SC_VC_HK_RT;
         messageBuffer[3] = 0; // Spare

         packetLength = 4 + packetLength; // I2C bytes
         err = i2cWriteAndCheckError(I2C_ADDR_COM, messageBuffer, &packetLength);
    }

    return err;
}
```

Appendix D CCSDS_GenerateTelemetryPacket() – ccsds.c

```
// Generates a complete telemetry packet with the values and data specified

INT8U CCSDS_GenerateTelemetryPacket(INT8U *telemetryBuffer, INT8U *telemetryBufferSize, INT16U apid, INT8U
serviceType, INT8U serviceSubtype, INT8U* sourceData, INT8U sourceDataLength)

{

   INT32U time;

   Time_Get(&time);

   return  CCSDS_GenerateTelemetryPacketWithTime(telemetryBuffer,  telemetryBufferSize,  apid,  serviceType,
serviceSubtype, sourceData, sourceDataLength, time);

}
```

Appendix E CCSDS_GenerateTelemetryPacketWithTime() – ccsds.c

```
// Generates a complete telemetry packet with the values and data specified
INT8U  CCSDS_GenerateTelemetryPacketWithTime(INT8U  *telemetryBuffer,  INT8U  *telemetryBufferSize,  INT16U
apid, INT8U serviceType, INT8U serviceSubtype, INT8U* sourceData, INT8U sourceDataLength, INT32U time)
{
   INT8U i;
   INT16U chk;                                          // CRC syndrome
   INT16U sequenceCount;                       // Sequence Count of packet
   INT8U packetLength;                         // Total length of packet

   INT16U packetLengthFieldValue;       // Value of Packet Length header field

   INT8U *dataPtr;                                      // For source data copy
   INT8U *dataEndPtr;

   // Get sequence count
   sequenceCount = CCSDS_GetSequenceCount();

   if(sourceData == NULL)
        return ERR_MISSING_PARAMETER; // No data
   if(telemetryBuffer == NULL)
        return ERR_MISSING_PARAMETER; // No telemetry buffer
   if(telemetryBufferSize == NULL)
        return ERR_MISSING_PARAMETER; // No telemetry buffer size

   // Total packet length must be <= than 251 octets (otherwise it cannot be transmitted in a I2C message)
   if(sourceDataLength > (0xFB - TM_NONDATA_SIZE))
        return ERR_CCSDS_TOO_MUCH_DATA; // Data too big

   // Total packet length (not value of packet length field !)
   packetLength = TM_NONDATA_SIZE + sourceDataLength;

   // Test if telemetry buffer big enough
   if(*telemetryBufferSize < packetLength)
        return ERR_CCSDS_BUFFER_TOO_SMALL; // Buffer too small



   // Packet Header (Packet ID, Packet Sequence Control and Packet Length) (48 bits)
   // Packet ID (16 bits)
   telemetryBuffer[0] = 0x08 | ((INT8U)(apid >> 8) & 0x07);     //  Version  number  (3  bits)  =  0,  Type  (1
bit) = 0, Data Field Header Flag (1 bit) = 1, APID (11 bits) -> 3 MSB
   telemetryBuffer[1] = (INT8U)(apid);          // APID (11 bits) -> 8 LSB
   // Packet Sequence Control (16 bits)
   telemetryBuffer[2] = 0xC0 | ((INT8U)(sequenceCount >> 8) & 0x3F);    // Grouping Flag (2 bits), Source
Sequence Count (14 bits) -> 6 MSB
   telemetryBuffer[3] = (INT8U)(sequenceCount);                                      // Source Sequence
Count (14 bits) -> 8 LSB

   // Packet Length (16 bits)
   packetLengthFieldValue = packetLength-7;
   telemetryBuffer[4] = (INT8U)(packetLengthFieldValue >> 8);   // Packet Length (16 bits) -> 8 MSB
   telemetryBuffer[5] = (INT8U)(packetLengthFieldValue);        //                         -> 8 LSB



   // Packet Data Field (Telemetry Data Field Header, Source Data and Packet Error Control) (Variable)
   // Telemetry Data Field Header (56 bits)
   telemetryBuffer[6] = 0x10;   // Spare (1 bit) = 0, PUS Version Number (3 bits) = 1, Spare (4 bits) = 0
   telemetryBuffer[7] = serviceType;            // Service Type (8 bits)
   telemetryBuffer[8] = serviceSubtype;         // Service Subtype (8 bits)

   // Absolute Time (40 bits, 5 octets)
   // CUC with 4 octets of coarse time and 1 octet of fine time
   telemetryBuffer[9] =  (INT8U)(time>>28); // C1
   telemetryBuffer[10] = (INT8U)(time>>20); // C2
   telemetryBuffer[11] = (INT8U)(time>>12); // C3
   telemetryBuffer[12] = (INT8U)(time>>4);       // C4
   telemetryBuffer[13] = (INT8U)(time<<4);       // F1 // TODO: Seems to be always 0



   // Copy Source Data (variable length)
   dataPtr = (telemetryBuffer+TM_HEADERS_SIZE);
   dataEndPtr = dataPtr+sourceDataLength;
   while(dataPtr < dataEndPtr)
        *(dataPtr++) = *(sourceData++);



   // Packet Error Control (16 bits)
   chk = 0xFFFF; // Init syndrome
   // Compute CRC (all packet data except FCS field)
   for(i=0 ; i<packetLength-2 ; i++)
        chk = CRC_Optimized(telemetryBuffer[i], chk); // Optimized CRC

   // Fill CRC field
   telemetryBuffer[packetLength-2] = (INT8U)(chk >> 8);
   telemetryBuffer[packetLength-1] = (INT8U)(chk);
```

```
    // Return number of bytes written
    *telemetryBufferSize = packetLength;

    return ERR_SUCCESS;
}
```

Appendix F retrieveTelecommand() – eps.c

```
static void retrieveTelecommand(void)
{
   static INT32U lastTelecommandTime = 0;
   INT32U time;
   INT8U err;
   INT8U message[255];
   INT8U messageLength = 0;

   // Only retrieve telecommands when master
   if(!I2C_IsMaster())
        return;

   // Only retrieve telecommand every interval
   Time_Get(&time);
   if((time-lastTelecommandTime) < INTERVAL_REQUEST_COM_TELECOMMAND)
        return;

   // Prepare Packet that is send to COM
   message[0] = I2C_TYPE_FUNC_CALL_REQUEST;
   message[1] = FCT_COM_GET_TELECOMMAND;

   // Send packet to COM
   err = I2C_MasterWrite(I2C_ADDR_COM, message, 2);
   if(err == ERR_SUCCESS)
   {
        // Read possible errors
        err = I2C_MasterRead(I2C_ADDR_COM, message, &messageLength);
        if(err == ERR_SUCCESS)
        {
                // Update Last Response Time of COM
                _DINT();
                _NOP();
                Time_IntGet(&housekeeping.LR_COM);
                _EINT();

                // Telecommand available
                if(message[0] == I2C_TYPE_FUNC_CALL_REPORT)
                {
                        decodeCCSDSTelecommand(message+2);
                }
        }
```

```
    }


    // HACK: COM overflow test
    /*
    {
        INT16U i;
        for(i=0 ; i<2 ; i++)
        {
                INT8U length = 250;
                err  =  CCSDS_GenerateTelemetryPacket(message+4,  &length,  APID_EPS,  128,  7,  message,  220);
//5ms
                if(err == ERR_SUCCESS)
                {
                        message[0] = I2C_TYPE_FUNC_CALL_REQUEST;
                        message[1] = FCT_COM_SEND_PKT;
                        message[2] = SC_VC_PL;
                        message[3] = 0; // Spare
                        err = I2C_MasterWrite(I2C_ADDR_COM, message, length+4);
                        if(err == ERR_SUCCESS)
                        {
                                err = I2C_MasterRead(I2C_ADDR_COM, message, &length);
                                P5OUT ^= BIT4; // HACK:
                                if(err == ERR_SUCCESS)
                                {
                                        if(message[0]    ==    I2C_TYPE_ERROR_REPORT    &&    message[2]    ==
ERR_COM_TX_OVERFLOW)
                                                P5OUT ^= BIT5; // HACK:
                                }
                        }
                }
        }
    }
    */


    Time_Get(&lastTelecommandTime);
}
```

Appendix G decodeCCSDSTelecommand() – eps.c

```
static void decodeCCSDSTelecommand(INT8U* telecommand)
{
   INT8U serviceType = PACKET_SERVICE_TYPE((Telecommand*)telecommand);
   switch(serviceType)
   {
   // Function Management Service (8)
   case CCSDS_T8_FUNCTION_MANAGEMENT:
        decodeService8(telecommand);
        break;
   // Payload Image Service (128)
   case CCSDS_T128_IMAGE_SERVICE:
        decodeCcsds128(telecommand);
        break;
   // Illegal Service Type
   default:
        sendTelecommandReport_Failure(telecommand, CCSDS_ST_S1_ACCEPTANCE_FAILURE, CCSDS_ERR_ILLEGAL_TYPE);
        break;
   }
}
```

Appendix H decodeService8() – eps.c

```
static void decodeService8(INT8U* telecommand)
{
    Telecommand* tc = (Telecommand*)telecommand;
    INT16U i;
    INT8U messageBuffer[254]; // Also used for returnBuffer
    INT8U acceptanceError = ERR_SUCCESS;
    INT8U completionError = ERR_SUCCESS;

    if(PACKET_SERVICE_SUBTYPE(tc) == CCSDS_ST_S3_PERFORM_FUNCTION)
    {
         INT8U* tcData = TC_DATA(tc);
         INT8U* params = tcData+1;
         INT16U apid = PACKET_APID(tc);
         INT8U functionID = *tcData;
         INT8U paramsLength = TC_DATA_LENGTH(tc) - 1;
         INT8U returnBufferLength = 0;

         if(apid == APID_EPS)
         {
                 // Function for EPS
                 // Send acceptance report (needs to be before execution of TC)
                 sendTelecommandReport_Success(telecommand, CCSDS_ST_S1_ACCEPTANCE_SUCCESS);
                 completionError  =  FunctionCallCallback(functionID,  params,  paramsLength,  messageBuffer,
&returnBufferLength);
         }
         else if(apid == APID_COM || apid == APID_CDMS || apid == APID_ADCS || apid == APID_PAYLOAD )//test
if valid APID
         {
                 // Function for other subsystem
                 messageBuffer[0] = I2C_TYPE_FUNC_CALL_REQUEST;
                 messageBuffer[1] = functionID;
                 for(i=0 ; i<paramsLength ; i++)
                         messageBuffer[2+i] = *(params+i);

                 acceptanceError = I2C_MasterWrite(apid, messageBuffer, 2+paramsLength);
                 if(acceptanceError == ERR_SUCCESS)
                 {
                         INT8U dataLength = 0;

                         // Send acceptance report (needs to be before execution of TC)
                         sendTelecommandReport_Success(telecommand, CCSDS_ST_S1_ACCEPTANCE_SUCCESS);
```

```
                        // Read function call result
                        completionError = I2C_MasterRead(apid, messageBuffer, &dataLength);
                        if(completionError == ERR_SUCCESS)
                                completionError = checkI2CReport(messageBuffer);
                }
        }
        else
        {
                acceptanceError = CCSDS_ERR_ILLEGAL_APID;
        }
    }
    else
    {
        acceptanceError = CCSDS_ERR_ILLEGAL_SUBTYPE;
    }

    // Send Telecommand Verification Reports
    if(acceptanceError == ERR_SUCCESS)
    {
        // Acceptance success report already sent (need to be before execution of TC)

        // Completed reports
        if(completionError == ERR_SUCCESS)
                sendTelecommandReport_Success(telecommand, CCSDS_ST_S1_COMPLETED_SUCCESS);
        else
                sendTelecommandReport_Failure(telecommand, CCSDS_ST_S1_COMPLETED_FAILURE, completionError);
    }
    else
    {
        // Acceptance failure report
        sendTelecommandReport_Failure(telecommand, CCSDS_ST_S1_ACCEPTANCE_FAILURE, acceptanceError);
    }
}
```

Appendix I sendTelecommandReport_Success() – eps.c

```
/*----------------------------------------------
* sendTelecommandReport_Success()
* ----------------------------------------------
* Input  :
*   *telecommand : buffer containing the concerned telecommand packet
*   reportType : contains the success type (received, completed, progress, etc)
*
* Output :
*   error : ERR_SUCCESS when no error has occured, and error ID otherwise
*
* Description:
*   This function is called by decodeService8() every time a success message
*   needs to be downlinked to the ground station in a telemetry packet, after
*   a successful step in the execution of a service 8 function.
*
* ----------------------------------------------*/

INT8U sendTelecommandReport_Success(INT8U* telecommand, INT8U reportType)
{
   // Only send ack if requested
   INT8U tcAckField = TC_ACK((Telecommand*)telecommand);
   if((reportType == CCSDS_ST_S1_ACCEPTANCE_SUCCESS) && !(tcAckField & BIT3))
        return ERR_SUCCESS;
   if((reportType == CCSDS_ST_S1_STARTED_SUCCESS) && !(tcAckField & BIT2))
        return ERR_SUCCESS;
   if((reportType == CCSDS_ST_S1_COMPLETED_SUCCESS) && !(tcAckField & BIT0))
        return ERR_SUCCESS;

   {
        INT8U err;
        // CCSDS Source Data
        INT8U success[TM_S1_SUCCESS_SIZE];
        // CCSDS Packet length
        INT8U packetLength = TM_NONDATA_SIZE + TM_S1_SUCCESS_SIZE;

        // Packet Sequence Control (direct copy from telecommand)
        success[0] = telecommand[0];
        success[1] = telecommand[1];

        // Telecommand Packet ID (direct copy from telecommand)
        success[2] = telecommand[2];
        success[3] = telecommand[3];

        // Send packet to COM with function call
        temporaryBuffer[0] = I2C_TYPE_FUNC_CALL_REQUEST;
        temporaryBuffer[1] = FCT_COM_SEND_PKT;
        temporaryBuffer[2] = SC_VC_ACK_RT;
        temporaryBuffer[3] = 0; // Spare

        // Generate CCSDS telemetry packet
        err      =      CCSDS_GenerateTelemetryPacket(temporaryBuffer+4,      &packetLength,      APID_EPS,
CCSDS_T1_TELECOMMAND_VERIFICATION, reportType, success, sizeof(success));
        if(err == ERR_SUCCESS)
        {
                // TODO: Handle error
                err = I2C_MasterWrite(I2C_ADDR_COM, temporaryBuffer, 4+packetLength);
                if(err == ERR_SUCCESS)
                {
                        // Read possible errors (TODO: Handle error)
                        INT8U dataLength = 0;
                        // TODO: Handle error
                        err = I2C_MasterRead(I2C_ADDR_COM, temporaryBuffer, &dataLength);
                        if( err == ERR_COM_TX_OVERFLOW ) {
                                comBusy = 1;
                        }
                }
        }

        return err;
    }
}
```

Appendix J sendTelecommandReport_Failure() – eps.c

```
/*-----------------------------------------------
* sendTelecommandReport_Failure()
* -----------------------------------------------
* Input  :
*   *telecommand : buffer containing the concerned telecommand packet
*   reportType : contains the success type (received, completed, progress, etc)
*   err : describes the nature of the failure
*
* Output :
*   none
*
* Description:
*   This function is called by decodeService8() every time a failure message
*   needs to be downlinked to the ground station in a telemetry packet, after
*   an unsuccesful step in the execution of a service 8 function.
*
* -----------------------------------------------*/

// Sends a telecommand failure report
// Report type can be any failure subtype of service type 1
// Returns encountered errors
INT8U sendTelecommandReport_Failure(INT8U* telecommand, INT8U reportType, INT8U err)
{
   // CCSDS Source Data
   INT8U failure[TM_S1_FAILURE_SIZE];
   // CCSDS Packet length
   INT8U packetLength = TM_NONDATA_SIZE + TM_S1_FAILURE_SIZE;

   // Packet Sequence Control (direct copy from telecommand)
   failure[0] = telecommand[0];
   failure[1] = telecommand[1];

   // Telecommand Packet ID (direct copy from telecommand)
   failure[2] = telecommand[2];
   failure[3] = telecommand[3];

   // Error Code
   failure[4] = 0x00;
   failure[5] = err;

   // Send packet to COM with function call
```

```
    temporaryBuffer[0] = I2C_TYPE_FUNC_CALL_REQUEST;
    temporaryBuffer[1] = FCT_COM_SEND_PKT;
    temporaryBuffer[2] = SC_VC_ACK_RT;
    temporaryBuffer[3] = 0;

    // Generate CCSDS telemetry packet
    err        =        CCSDS_GenerateTelemetryPacket(temporaryBuffer+4,        &packetLength,        APID_EPS,
CCSDS_T1_TELECOMMAND_VERIFICATION, reportType, failure, sizeof(failure));
    if(err == ERR_SUCCESS)
    {
         // TODO: Handle error
         err = I2C_MasterWrite(I2C_ADDR_COM, temporaryBuffer, 4+packetLength);
         if(err == ERR_SUCCESS)
         {
                 // Read possible errors (TODO: Handle error)
                 INT8U dataLength = 0;
                 // TODO: Handle error
                 err = I2C_MasterRead(I2C_ADDR_COM, temporaryBuffer, &dataLength);
                 if( err == ERR_COM_TX_OVERFLOW ) {
                         comBusy = 1;
                 }
         }
    }
    return err;
}
```