LIST OF ACRONYMS/ABBREVIATIONS

|  |  |
| --- | --- |
| OBC | On-Board Computer |
| GS  ADCS  COMMS  EPS  I/O  GPIO  SPI  I2C  UART  CSP  OS  GUI | Ground Station  Attitude Determination and Control Subsystem  Communications Subsystem  Electrical Power Subsystem  Input/Output  General Purpose Input/Output  Serial Peripheral Interface  Inter-Integrated Circuit  Universal Asynchronous Receiver/Transmitter  CubeSat Space Protocol  Operating System  Graphical User Interface |
|  |  |
|  |  |
|  |  |
|  |  |

ON BOARD COMPUTER

“OBC”

The On-Board Computer can be thought of as the brains of the satellite. It’s also known as the Command & Data Handling subsystem, as it not only handles the execution of any commands sent by our Ground Station, but also any data that is collected and sent back for analysis. It also serves another critical function, which is to coordinate between all of the different subsystems of the satellite, maintaining proper execution of periodic and aperiodic tasks to ensure correct operation.

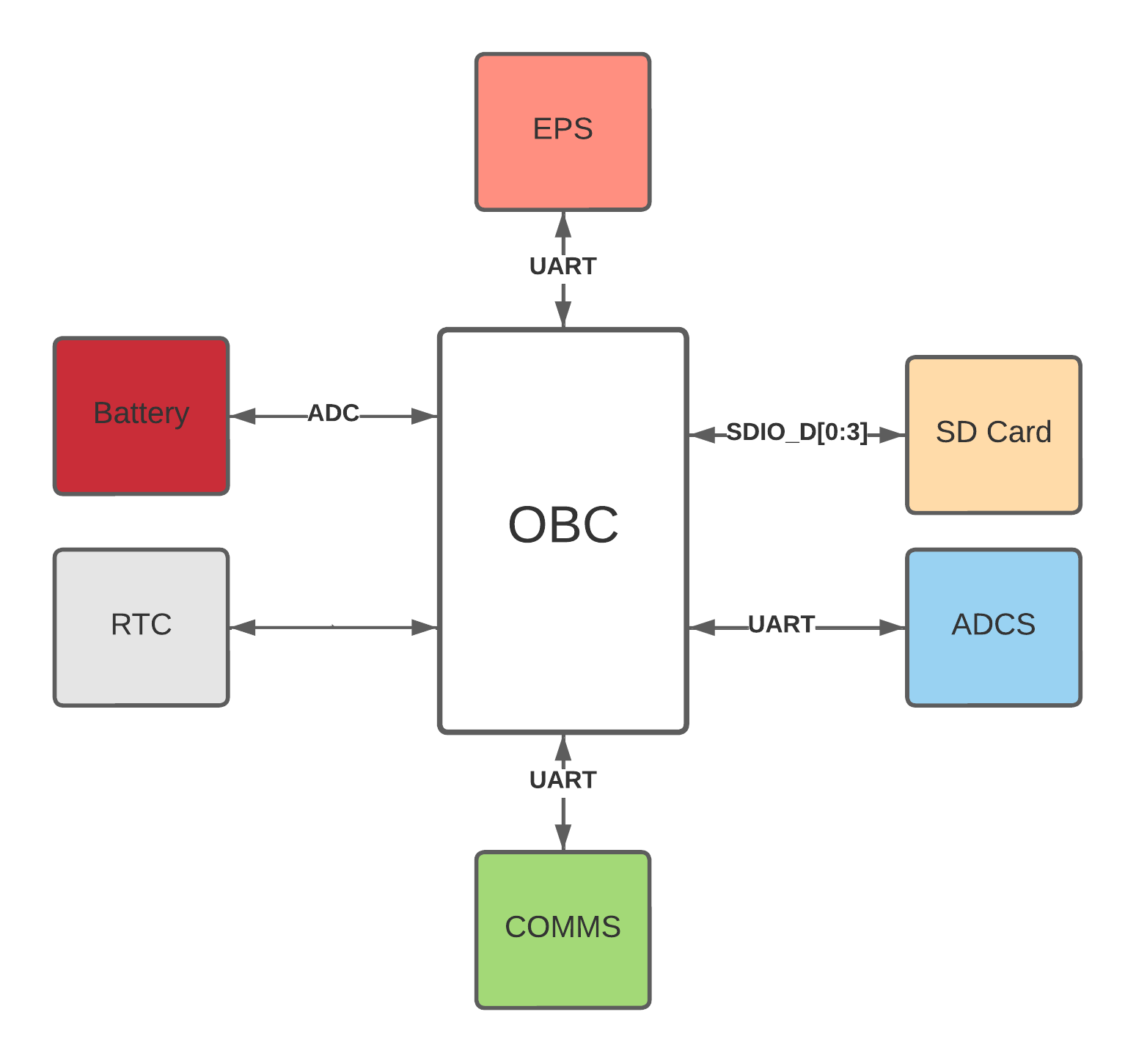
# 1- REQUIREMENTS

The main functions of this subsystem can be summed up as follows:

* **Ensure correct operation of the satellite:** the system should be able to handle any unexpected faults, and should recover from them as swiftly and with as minimal damage as possible. It should also make sure that its main tasks are executed in a timely manner.
* **Process and execute commands:** the system should be able to understand commands sent from our GS and ensure its execution on the relevant subsystems.
* **Perform housekeeping tasks:** the system should periodically log the current state of the satellite along with critical readings.
* **Operating the mass storage memory:** the system should operate the mass storage memory used for logs and configuration storage.
* **Provide data interfaces to all of the other subsystems:** the system should have adequate interfaces to connect to all of the other subsystems by using UART.
* **maintaining UTC time.**

# 2- ARCHITECTURE

The OBC architecture is essentially based on the connectivity between subsystems within the CubeSat. This simply means that the microcontroller’s peripherals are configured according to the data flow within the CubeSat’s computing scheme. This has several benefits:

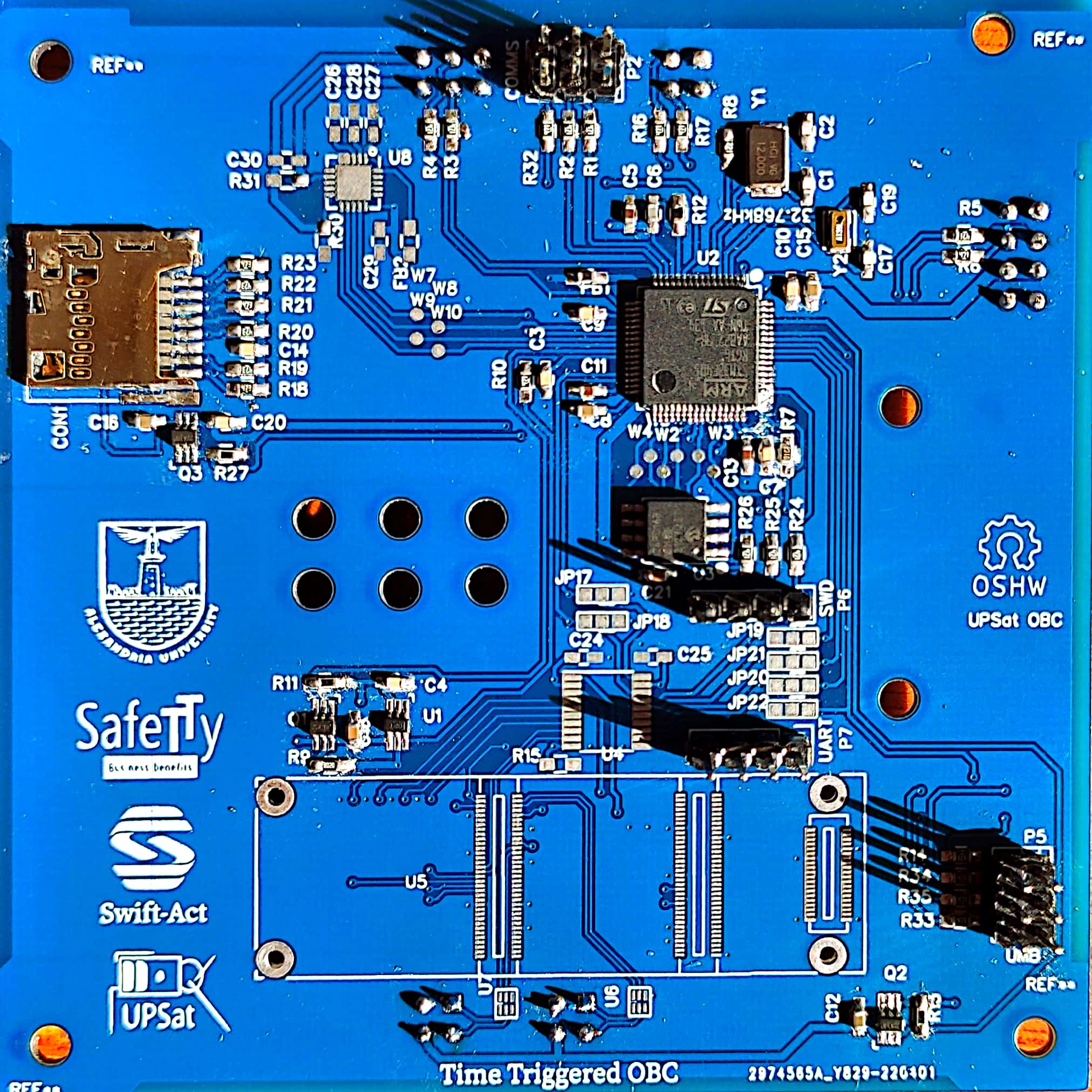


**Figure 0-1: System topology.**

* **No single point of failure:** as there’s no common bus for all of the subsystems, it allows for more leeway to act upon failures in which a single link is down, as the other links would be unaffected.
* **Better performance:** as the subsystems do not need to share the same bus.
* **Flexibility:** as the various modules can have different interfaces than one another while still maintaining proper functionality, as long as the OBC supports it.

# 3- HARDWARE

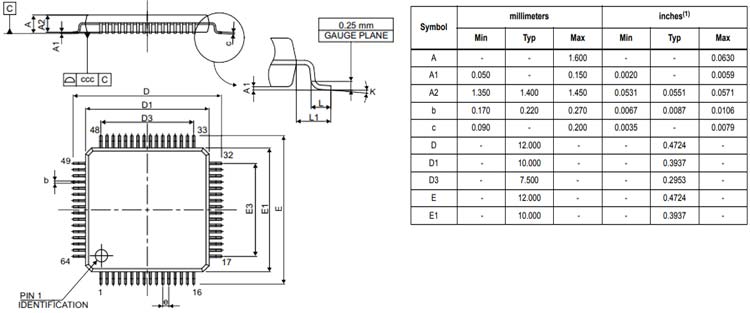
* We are using a STM32F405 with an ARM cortex M4 cpu core that has 1 Mbyte of Flash and 192 Kbytes of SRAM; shown in Figure 0-2, as our On-Board Computer, which provides processing power, memory and adequate interfaces to connect to the rest of our satellite
* The microcontroller’s internal Real Time Clock connected with a coin cell battery.
* An SD card connected with SDIO.
* IS25LP128 128 MBIT Flash memory connected with SPI.



**Figure 0-2: OBC Subsystem**

It fits our desired specifications, including:

* **A powerful yet efficient CPU:** with an ARM cortex M4 cpu core processor, which is more than adequate for our application.
* **A large amount of RAM:** with 1 Mbyte of Flash and 192 Kbytes of SRAM, while not being the absolute fastest it still provides a very large amount that would allow running tasks with room to spare.
* **Low power consumption:** drawing less than 0.5W of power while idle.
* **Large community support:** with available documentation and test results.
* **Commercial availability:** being readily available in most markets.
* **Flexible I/O:** Up to 15 communication interfaces (including 6x USARTs running at up to 10.5 Mbit/s, 3x SPI running at up to 42 Mbit/s, 3x I²C, 2x CAN, SDIO), 12-bit ADCs, two DACs, a low-power RTC, twelve general-purpose 16-bit timers including two PWM timers for motor control.
* **Small size:** which saves valuable space inside of our mechanical structure, where it’s at a premium. The mechanical diagram is shown in Figure 0-3.



**Figure 0-3: STM32F405 mechanical diagram.**

# 4- CUBESAT MODES OF OPERATION

**Table 0-1: Modes of operation.**

|  |  |
| --- | --- |
| *Command* | *Function* |
| Start | Start state. |
| Initialization | Boot up OBC, start initializing all subsystems. |
| Idle | Only maintains critical functionality |
| Nominal | Normal operating mode. |

The satellite has different modes of operation to account for different states, they can be summarized as follows:

* **Start state:** when first powered on, the satellite should not go back to this state after the first time.
* **Initialization sequence:** power up the OBC, and initialize links to all connected subsystems.
* **Safe mode:** when a critical failure occurs, all non-mission critical functions should be halted.
* **Nominal mode:** all subsystems are up and running, payload can be used at this state.

# 5- COMMUNICATIONS PROTOCOL

The OBC is also responsible for the processing of commands and data, adding appropriate metadata to ensure accurate delivery. In this case, we’re using ECSS (European Cooperation for Space Standardization) protocol.

## Command and control module

The CnC (command and control) module, defines the protocol for earth to satellite (and vice versa) communication and inter subsystem communication. It consists of the packet format, header and data definition. Operations are grouped into services, defined by the protocol.

## Requirements

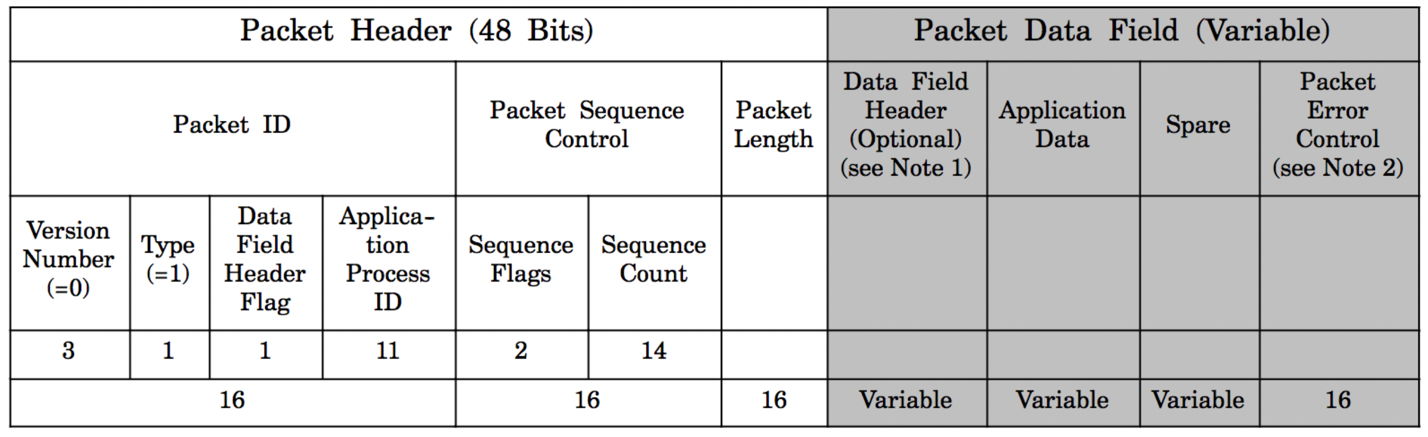
The following requirements were set in order to evaluate ECSS:

* Low protocol overhead.
* Lightweight.
* Highly modular and customized.

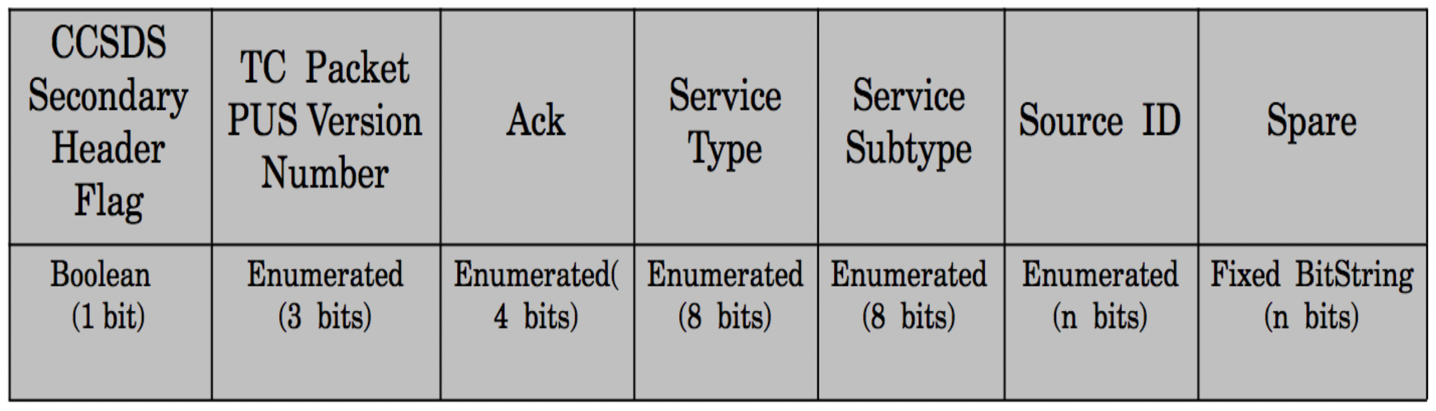
Since all interactions to subsystems and earth uses the protocol, if the protocol overhead isn’t efficient, it leads to power waste and added data traffic.

The CnC module will be used with microcontrollers that have limited processing power and resources, so it needs to be lightweight.

The protocol should be designed in way that allows some degree of customization, so the implementation will be tailored on the resources available and be efficient.



**Figure 0-4: ECSS TC frame header**



**Figure 0-5: ECSS TC data header**

The ECSS-E-70-41A specification is a work of European Cooperation for Space Standardization and is based in previous experiences. For simplicity ECSS-E-70-41A would be refereed as ECSS in this document.

ECSS is a well-defined protocol and the specification is clear and well written. ECSS describes the frame header and a set of services. The header has a lot of optional parameters that makes it easily adapted to the application needs. The services listed are all optional and it’s up to the user to implement them. Moreover, each service has a list of standard and additional features depending again to the application needs.

The header has 2 parts: a) the packet header, which is standard, and is 6 bytes. b) the data field header, which is variable due to different options available. The options are: packet error control (Checksum [27]), timestamp, destination id and spare bits (so the that the frame is in octet intervals).

Depending on the application and the number of distinct application ids, the user can select to have only 1 set of application ids, defining both the source and destination in one byte, If the number of application ids is large than a separate source/destination has to be used.

The header implemented was 12 bytes. The switch between TC/TM and source/destination can be confusing. One feature that the protocol lacks is that it doesn’t have a mechanism for different configurations on runtime, having configuration bits denoting different configurations, such as the existence of a timestamp or a checksum in the packet, would have made the protocol a lot more versatile.

ECSS is packet oriented e.g., mass storage service use of store packets, even though it’s not a huge disadvantage, it can a bit tedious and could lead to inefficiency on the implementation.

For the following reasons, it was decided to use the ECSS protocol.

* ECSS was recommend by QB50.
* ECSS is used by many other cubesats.
* ECSS is based on experience on previous protocol designs, as a result the protocol is highly refined.
* ECSS is highly flexible on the actual implementation and it allowed to customize it according to our needs.

## ECSS Services

The ECSS standard is highly adaptive and provides many different choices. In this section, the design choices for CubeSat are analyzed.

## Services

* The telecommand verification service provide a way to receive a response about the successful or not outcome of a telecommand’s operation. This service is required since for some operations it is critical to know the outcome of the operation.
* The housekeeping & diagnostic data reporting service provide a way to transmit and receive information (housekeeping) that denote the status of the CubeSat. The housekeeping operation is standard in CubeSats.
* The function management service is used for operations that aren’t part for other services operations. In CubeSat the service is used mainly for controlling the power in subsystems and devices and setting configuration parameters in different modules.
* The time management service is providing a way to synchronize time between subsystems and the ground station. This service was added later when the need to synchronize time between ADCS and OBC and the ability to change the time from the ground came up.
* The on-board operations scheduling service provide a way for to trigger events in specific times or continuously with specific intervals with the release of telecommands. This service allows to perform events without having connection with the ground station.
* The large data transfer service provides a way to exchange packets that are larger than the size that is allowed by cutting the original packet in chunks that their size is allowed. In CubeSat it is used for transferring large files.
* The on-board storage and retrieval service provide a way to store and retrieve information in mass storage devices. In CubeSat it is used to store various logs and configuration parameters in the SD card of the OBC.
* The test service provides a simple way to verify that a subsystem is working. It is very similar to the ping program used in IP networks.
* The event reporting service provides a way for a subsystem to report events. It was originally designed for subsystems that didn’t have storage devices to report events that were critical to the CubeSat’s operation to the OBC so that it would store them for later review from a human operator. Not implemented.
* Event-action service uses the event service to generate action when a particular event takes place. Since the event service was removed there wasn’t a way to use the event-action service.

From the 16 services only 8 were decided to with the time management service added later and the event reporting services removed during the implementation phase.

The specification states that any custom services or services subtypes should have an number larger than 128. In CubeSat’s design this rule wasn’t followed and the custom could have any number that it’s not used from the specification. This happens because there is a a large lookup table for every subsystem which defines which services are used, the way it was implemented having 128 service number and above would create a huge lookup table that wouldn’t fit in the microcontroller’s memory.

## Application ids

Application ids are a core concept in ECSS, it is the address of a module that the packet is heading towards, it is very similar to the IP address concept. Using 11 bits for application ids a total of 2047 address can be achieved. Application ids are not confounded only in hardware subsystems but software modules can be given an id.

In CubeSat a total of 6 application ids were used: 4 for each subsystem and 2 for the ground station. The 2 application ids used for the ground station is because there are 2 different paths available and there was a need to differentiate them. The first is the serial connection through the umbilical connector and was used only during testing. The second was through the RF communication and the COMMS subsystem. The software is design of CubeSat is simple enough so there wasn’t a need for more application ids.

## Packet frame

Even if the ECSS standard treats the telecommand and telemetry as packets with different frame structure, the design intention of the packet frame in CubeSat was that both of frames could be as identical as possible. The reason behind that was that the software remains as simple as possible, using the same code for manipulating telecommand and telemetry frames. The simpler design in software would lead in less developing the code and testing it.

The packet header and packet error control are identical in telecommands and telemetry packets. The data field header is designed to be identical with source ID in telecommands and destination ID in telemetry packets and without the optional fields of packet sub-counter and time in telemetry that don’t exist in a telecommand packet.

For routing purposes, the source and destination application ID was added in telecommand and telemetry packets. When the packet is a telecommand the application ID in packet id denotes the subsystem destination and in the data header field the subsystem that the packet originated and vice versa in a telemetry packet. Without the source ID, it would be impossible to know to which subsystem a possible response should be send and without the destination ID where to route the telemetry packet. It was possible to only use the application ID by having application IDs would denote both the source and destination ID but that design would be more obscure leading to confusion during testing.

The maximum length of a normal frame is 210 bytes, by subtracting the headers and error correction it leaves with 198 bytes for application data. This number derives from restrictions in RF communication with the Earth. Because the COMMS subsystem doesn’t use error correction algorithms, if the size is larger than 210 bytes the probability that the packet is received correctly from the ground station, quickly deteriorates.

For the case of handling large files the normal packet size is inefficient and restrictive. For that reason and for subsystem communication only, the length of a packet can be extended to a maximum of 2050 bytes. This transaction happens only for OBC-COMMS communication only. For RF communications if the size is larger than normal, the large data service is used.

The version number and data field header flag of the packet ID have the default values of 0 and 1.

The type equals to 1 if the packet is a telecommand and 0 if it is a telemetry packet.

The application ID uses only the 8 bits for efficiency but for compatibility reasons 11 bits are used in the frame.

The Sequence flags in telecommands or Grouping flag in telemetry packets are used only in standalone mode with default value equal to 3.

The sequence count is a counter that counts the packets that the subsystem has transmitted to another subsystem, there is a different counter for each application id. If a subsystem routes the packet to its indented destination, it doesn’t modify the counter. The counter is uses 8 bits instead 14 bits as the standard for efficiency reasons. For compatibility reasons for the packet frame remains 14 bits. Every system that transmits packets frames needs to implement the counter. Moreover, the counter in CubeSat is not stored in mass storage memory and it is reset to zero in each subsystem reset. By observing when the counter resets to zero in a subsystem it can be deduced that a reset happened to that subsystem.

The packet length is calculated by subtracting from the actual packet size the size of the packet header which is 6 bytes and subtracting 1.

Packet length = Packet size (bytes) - 6 (packet header) - 1

The data field header varies in a telecommand and a telemetry packet. The CCSDS Secondary Header Flag has a default value of 0 in a telecommand and it’s used as padding in a telemetry packet. The Ack is used in a telecommand from the verification service and as padding in a telemetry since the verification service doesn’t work with telemetry packets.

In both telecommand and telemetry packets the Packet PUS Version Number has a default value of 1.

The service type and subtype denote the functionality of the packet and the service associated with the packet.

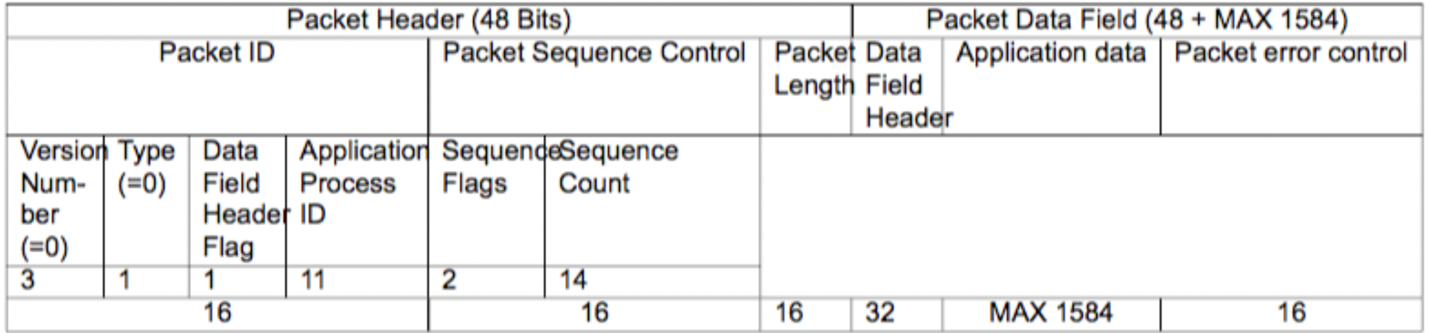
In a telecommand the source ID denotes the application ID of the subsystem that the packet originates from and in a telemetry packet the destination ID denotes the destined subsystem. Both the source and destination ID reside in the same position in a telecommand and telemetry packet.

The Packet error control is implemented as a CRC8 algorithm that occupies 8 bits but for compatibility reasons the frame has 16 bits with the first 8 bits are unused.

The packet sub-counter and time fields in a telemetry packet are not used because there wasn’t any need for them in CubeSat.

Finally, no optional spare fields were added in both telecommand and telemetry packets.

**Table 0-2: ECSS TC data header**



## Services in subsystems

Verification, housekeeping and test service are basic services needed in all subsystems.

Time management is only used in ADCS and OBC since only them require precision time keeping, OBC for the ADCS for the control calculations.

On-board scheduling and On-board storage services are only used in OBC since they require a mass storage device.

Finally, large data transfer is only implemented in COMMS since it is the only capable for RF communications.

## Telecommand verification service

The Telecommand verification service is the 1. In table 0-3 are shown the minimum and additional capabilities offered by the services. The service is used when a telecommand has the values shown in table 0-3 The service doesn’t support telemetry packets.

For CubeSat it was decided that only the minimum capabilities would be used, even if the additional would be definitely helpful they would also complicate the software design. For that reason, only values of 0 and 1 are valid in the ACK field, if other values are present the packet is flagged as invalid and dropped.

Since most of the telecommands are usually finished immediately, the acceptance report means also the competition of the telecommand but the semantics of the acceptance should be considered to be a telecommand.

If the telecommand results in failure, the frame has an error code field. By checking the error code, the ground station operators could find the reason for the failure and make correcting procedures accordingly. The ECSS standard provides some error codes about packet decoding failure listed in table 0-3.

One particular idiosyncrasy of the service is found in the table 0-7 where are listed errors about packet decoding such as error 2 incorrect checksum, that leads reporting acceptance failure about a packet that could have corrupted information including the ACK field.

Table 0-3 Telecommand packet data ACK field settings

|  |  |
| --- | --- |
| Value | Value meaning |
| 0 | none |
| 1 | Acknowledge acceptance |
| 2 | Acknowledge start of execution |
| 4 | Acknowledge progress of execution |
| 8 | Acknowledge completion of execution |

Table 0-4 Telecommand verification service subtypes

|  |
| --- |
| Telecommand acceptance report |
| success Telecommand acceptance report |
| failure Additional capabilities Telecommand execution started report |
| success Telecommand execution started report |
| failure Telecommand execution progress report |
| success Telecommand execution progress report |
| failure Telecommand execution completed report |
| success Telecommand execution completed report |

Table 0-5 Telecommand verification service acceptance report frame

|  |  |
| --- | --- |
| Packet sequence control | 16 bits 16 bits |

Table 0-6 Telecommand verification service acceptance failure frame.

|  |  |
| --- | --- |
| Packet sequence control Error | 16 bits 16 bits 8 bits |

Table 0-7 Telecommand verification service error codes

|  |  |
| --- | --- |
| Value | Value meaning |
| 0 | Illegal APID |
| 1 | incomplete or invalid length packet |
| 2 | incorrect |
| 3 | illegal packet type |
| 4 | illegal packet subtype |
| 5 | illegal or inconsistent application data |

## On-board operations scheduling service

There are two scenarios where the capability for the on-board execution of operations that have been loaded in advance from the ground shall be implemented:

* Those missions that perform operations outside of ground contact because of limited ground station visibility (e.g., LEO spacecraft) or signal propagation delays (e.g., deep-space probes).
* Those missions whose operations concept is to minimize the dependency on the ground segment. Thus, a geostationary telecommunication or meteorological mission can perform all of its routine operations in this manner, even though the spacecraft is permanently in view of a ground station. This approach potentially increases the availability of operational services or mission products, since the continuous availability of the uplink is eliminated.

The simplest form of on-board operations scheduling -that has been implemented-consists of storing time-tagged commands that have been loaded from ground and releasing them to their destination application process(es) when their on-board time is reached.

## Housekeeping

Information indicating the status of the CubeSat, are broadcasted to earth, in specific intervals. WOD is transmitted automatically, so it can be easily gathered by ground stations that don’t have transmit capabilities.

CubeSat has 3 different WODs and each is used for different purposes:

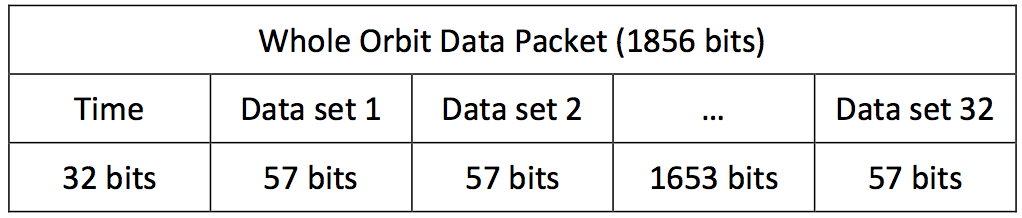
* QB50 WOD.
* Extended WOD.
* CW WOD.

The QB50, extended and CW WOD, is used for understanding the state of CubeSat. It is the last line of defense in the case there isn’t a communication link between ground stations and CubeSat. It is going to be the first indication if CubeSat works correctly or not. The CW WOD is crucial during the first days of operation, in order to track and verify the operation of CubeSat. Since HAM operators around the globe could listen for CW WOD, a global coverage can be obtained.

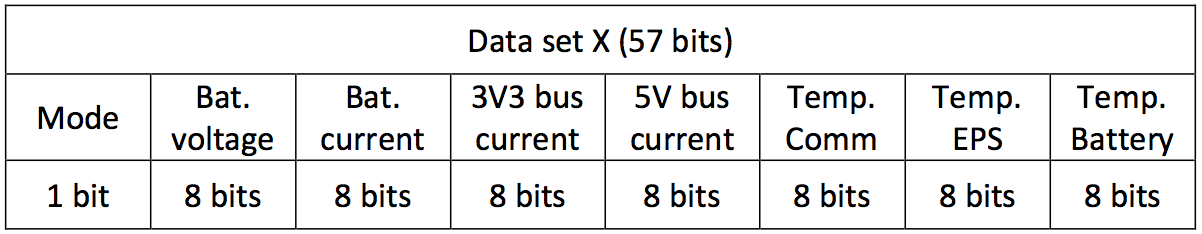
## WOD

In the QB50 requirements, there is WOD. In the frame, it provides historical information. The dataset provides general information. For compatibility with the rest of the missions in QB50, WOD is not encapsulated in ECSS.

**Table 0-8 WOD packet format**

****

**Table 0-9 WOD dataset**

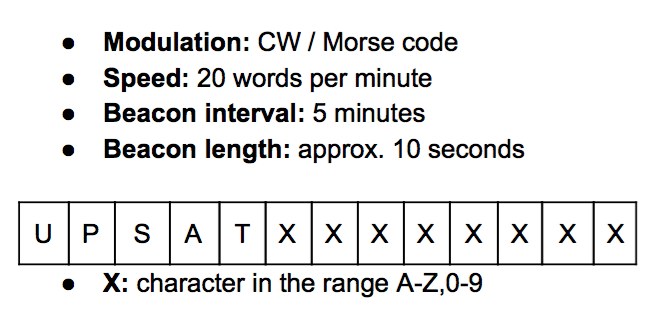
****

## Extended WOD

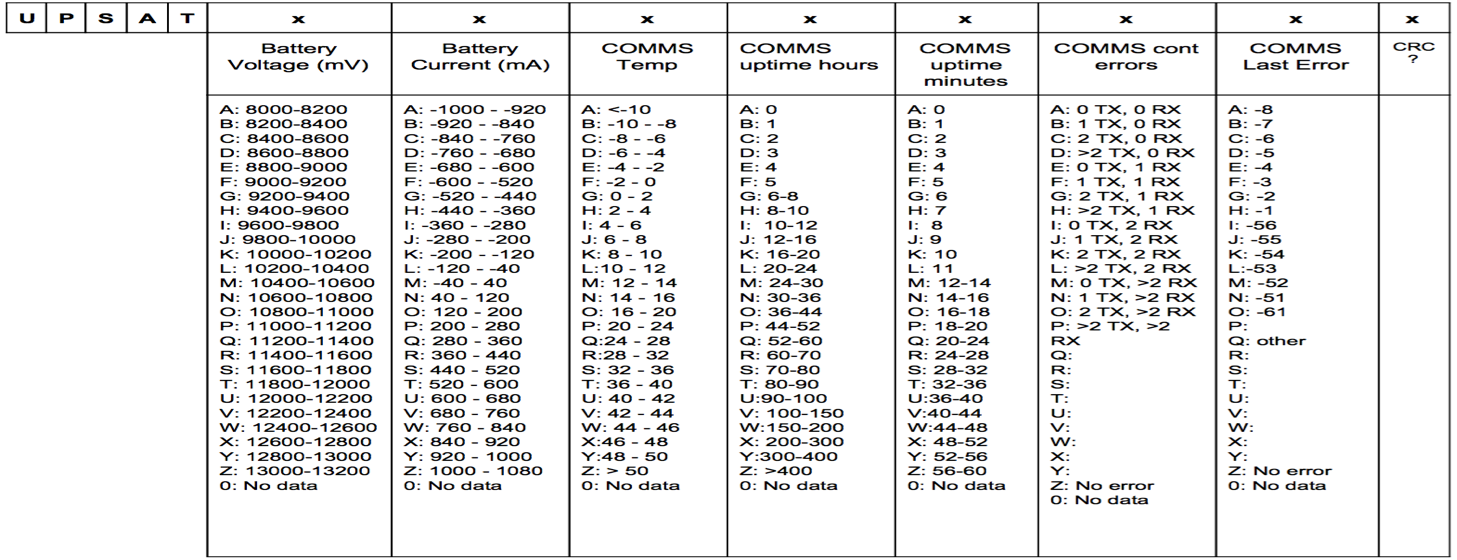
Since WOD offers only basic information, it was decided to add an independent extended WOD. That would provide more information about the state of CubeSat. It was asked of all engineers to supply a set of variables, that would help them understand what happens in the subsystems. A small refactored happened in order to fit all data in a single frame. Extended WOD is encapsulated in a ECSS frame.

## CW WOD

In addition to WOD and extended WOD, a CW WOD was added. The reason was that CW has better chances of receiving it from the ground than the FSK modulated WOD and extended WOD. Moreover, in CW there isn’t a need for complicated demodulation hardware, even a human with proper training can understand it. The disadvantage of CW and the reason that it hosts minimal information, is that it has far lower data rate than FSK and higher consumption due to lower data rate.



**Figure 0-6: CW WOD frame**

****

**Figure 0-7: CW WOD dataset**

## Housekeeping & diagnostic data reporting service

In this service, the design has deviated a lot from the specification. In the specification, each housekeeping structure send a report in specific intervals. There are also functions that implement new structures and modify the intervals that the reports are generated.

For the CubeSat, a different, simpler design was followed: OBC would send a telecommand report parameters request (3,21) and the subsystem would respond with a telemetry parameters report (3,23). OBC handles all the timing and not the service. In the request, there is the structure ID that the OBC wants. The structure ID is most of the times general and the parameters in report are in conjunction with the subsystem that reports it. For WOD variants the OBC gathers the health and extended health reports, forms the WOD variant structures and then transmits it to earth. Some subsystems have structure IDs that are specific to them. Finally. the ground station could also send a request for a parameter report and receive the response.

Even though the request/response mechanism works well, a design that uses intervals as specified in the ECSS standard would have been a better solution for 3 reasons:

* It removes the task of sending requests.
* Minimizes traffic.
* Easier to change intervals.

Table 0-10 Housekeeping service structure IDs

|  |  |
| --- | --- |
| Structure ID name | Structure ID Meaning |
| HEALTH\_REP | Health report |
| EX\_HEALTH\_REP | Extended health report |
| EVENTS\_REP | Events report |
| WOD\_REP | WOD report |
| EXT\_WOD\_REP | Extended WOD report |
| ADCS\_TLE\_REP | ADCS TLE report |
| EPS\_FLS\_REP | EPS flash memory contents report |
| ECSS\_STATS\_REP | ECSS statistics report |

Table 0-11 Housekeeping service request structure id frame

|  |  |
| --- | --- |
| Structure ID | 8 bits |

Table 0-12 Housekeeping service report structure id frame

|  |  |
| --- | --- |
| Structure ID Data | 8 bits 1 - 1584 bits |

## On-board storage and retrieval service

The first design choice was that the file names for the logs should only be numbers and not characters. That has the advantage of lower size overhead when there is ground to CubeSat communication e.g., the file name as a string could be 8 characters long meaning 8 bytes as string when the equivalent number 99999999 only uses 4 bytes. Moreover, file operation could be performed with simple mathematics e.g. The next file name could be found by adding 1 to the current file name.

For the logs storage design, it was decided to use one file per log entry. This approach adds great size overhead since the minimum size a file occupies is 512 bytes but simplifies the actions needed to operate (retrieve, delete, store) with a log entry. A maximum file number of 5000 is defined, that way it is ensured that logs can’t consume all the disk size.

For the logs, on top of the file system an extra layer was added. A circular buffer was used with the head and tail pointers pointing into files. All logs must be placed in sequential manner into files. Using that mechanism, the logs operation is simplified: First when a log entry is deleted it doesn’t actual delete the file, leaving the data intact in case there is a need to retrieve it and saves time by not calling the delete function of the file system. Secondly when a downlink operation happens there is no need to know the actual file name, only the log number should be specified and the file name is found by adding the head pointer file name number and the log number e.g. if the head points to a file with a name 5 the 3rd log that is stored is found by adding 5 and 3 resulting in the file name 8.

Service subtype Enable (15,1) and Disable (15,2) control the power of the SD card, there is no data used. Instead of using the function management for power control of the SD it was decided to use the specification’s mechanism.

Service subtype Downlink (15,9) and the response Content (13,8) provide a way to download a file from CubeSat. The telecommand downlink provides the storage ID, the file name and if batch download is needed, the number of files. Batch download is used for logs only. For batch download, the files are sequential to the file name specified in the telecommand. The response has the storage id, the file name and the file content and if batch download is used, the next file name and file content.

Service subtype Delete (15,11) has multiple functionalities. The first field is the storage ID that the delete function should act. Second field is the mode. Depending on the mode the delete has different functionality. The mode FS reset with a logs storage ID, resets the file system, this is used in the case there is an issue with the file system. The hard delete mode is used with the logs storage ID and deletes all the files in storage ID. Also, when the hard delete is used with the SRAM storage ID, it initializes the static memory region on the OBC to zero. The hard delete mode should be only used when there is an issue with file system. The delete all mode used with logs clears the pointers used and finally the delete to mode used again with logs removes entry logs from the pointers.

Custom service subtype 15 uses the FatFS format function and formats the SD card. This was added in the case the file system gets corrupted and it’s unusable from the OBC.

## Test service

The test service is very simple. It has the 17 service type and only 2 subtypes: perform test (17,1) and report test (17,2). The service doesn’t use any application data.

Table 0-19 On-board storage and retrieval service uplink subtype frame

|  |  |
| --- | --- |
| Store ID File File data | 8 bits 16 bits 1 - 16384 bits |

Table 0-20 On-board storage and retrieval service downlink subtype frame

|  |  |
| --- | --- |
| Store ID File Number of files | 8 bits 16 bits 16 bits |

Table 0-21 On-board storage and retrieval service downlink content subtype frame

|  |  |
| --- | --- |
| Store ID File File data | 8 bits 16 bits 1- 16384 |

Table 0-22 On-board storage and retrieval service subtypes used on CubeSat

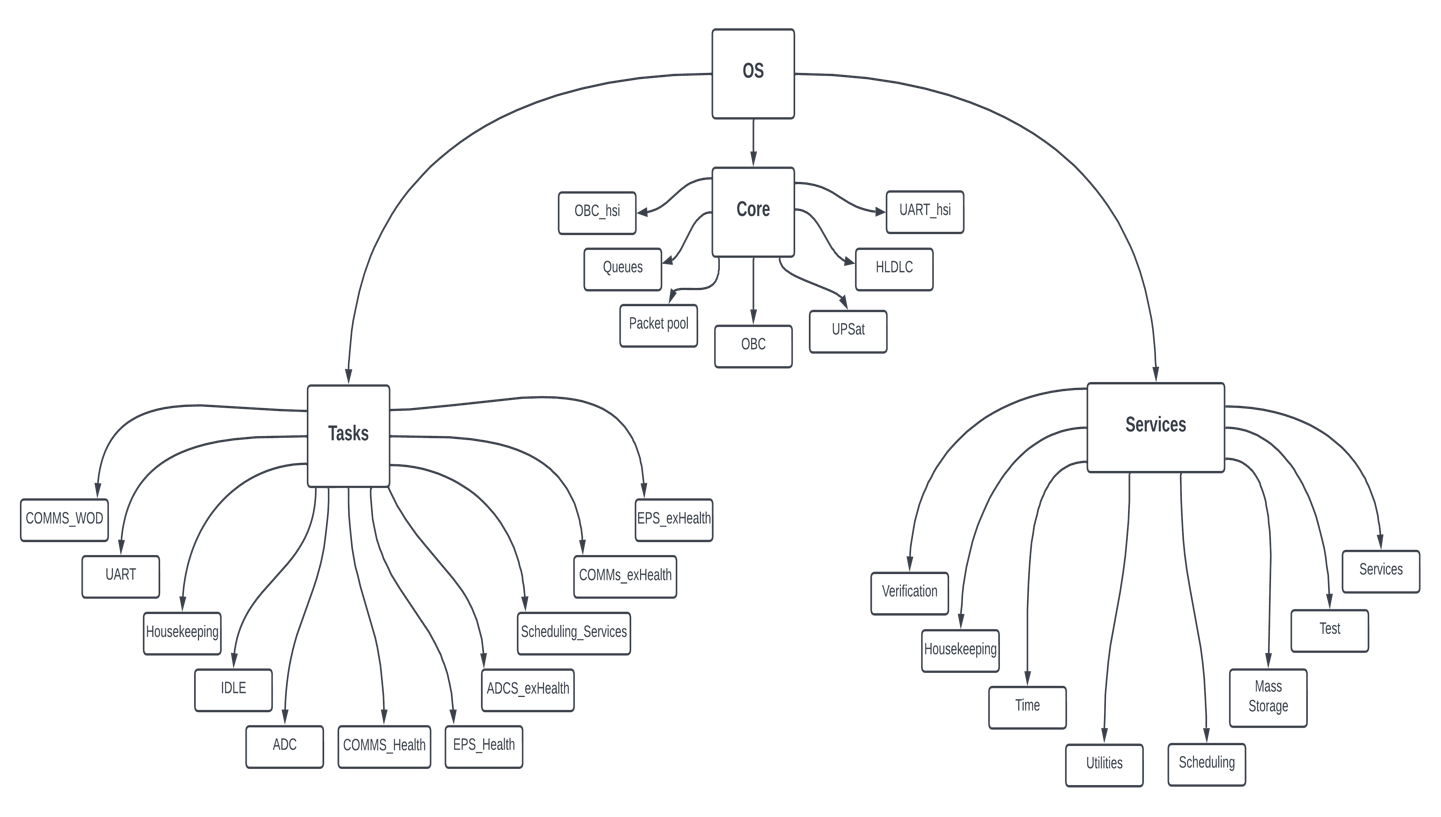
|  |  |
| --- | --- |
| Name Service subtype | Explanation |
| ENABLE | Turns on the SD card |
| DISABLE | Turns off the SD card |
| CONTENT | File contents |
| DOWNLINK | Download a file request |
| DELETE | Delete files |
| REPORT | Storage status report request |
| CATALOGUE REPORT | Storage status report response |
| UPLINK | File upload |
| FORMAT | Formats the SD card |
| LIST | List file information request |
| CATALOGUE LIST | List file information response |

Table 0-23 Store IDs

|  |  |
| --- | --- |
| Name | ID |
| WOD LOG | 9 |
| EXT WOD LOG | 10 |
| EVENT LOG | 11 |
| FOTOS | 12 |
| SCHS | 13 |
| SRAM | 14 |

# 6- Implementation

In this chapter, the implementation of the ECSS services and the OBC software is discussed.



**Figure 0-8: Project Organization**

## HLDLC

Protocols like SPI and $I\^2C$ it is possible to transfer multiple byte with one transaction. That way a transaction signifies a packet transfer. UART on the other hand transfers a byte at the time thus leaving no way to know when a packet starts or stops. For that reason, for UART subsystem communication, there are 3 options to pass the ECSS frames:

* ASCII characters.
* Binary protocol with length.
* HLDLC algorithm.

Use ASCII and have unique in the frame delimiters like ‘,’ and newline for frame endings. The use of ASCII is a good choice because it is easier to implement, it is easier for a human to debug since the data can be read. Unfortunately, ASCII requires more processing power in order to convert strings to numbers, requires more time to transfer and need more memory than a binary protocol. A good example for an ASCII protocol is the NMEA, which is used in all modern GPS receivers.

Use common delimiters but have a length field with the number of bytes in the packet, in the beginning of each packet and use that to calculate when the packet ends. This approach would require a timeout to discern different packets in the case there is an error. The advantage of that approach is that the implementation is easy and quick. The main disadvantage is the error handling is very poor in the case there are failures in the transfer.

The method that was eventually used, uses an algorithm that modifies the original data so there are unique delimiters. The disadvantage it has is that it could add up 2x times size overhead of the original data and it cannot be used when a fixed packet length is required. The main advantages are that is fairly easy to implement, it doesn’t use much processing resources, O(n) time is needed and makes fault tolerant mechanisms easy to implement. A similar mechanism is used for CSP when the transfer is over UART. During testing the overhead that was observed in most packets was minimum.

The HLDLC is very easy: There is the frame boundary byte, which is 0x7E that signifies the start and stop of every packet and the control escape byte 0x7D. In the case there is the frame boundary byte or the control escape byte inside the data frame, a control escape byte is inserted and the 5th bit is inverted e.g. if there is the 0x14 0x7E 0x55 0x7D 0x14 byte frame the frame is modified into 0x7E 0x14 0x7D 0x5E 0x55 0x7D 0x5D 0x14 0x7E .

The HLDLC module is very simple, it has 2 functions: one for adding HLDLC in a packet and one to deframe the HLDLC. The functions only need the buffer that has the data and the buffer that stores the modified data, plus the size of the original buffer that returns the size of the modified data. The deframe function checks for HLDLC integrity. The return code is SATR-EOT when it was successful or SATR-ERROR when there was an error.

Listing 0-1 HLDLC module functionsh

SAT\_returnState HLDLC\_deframe(uint8\_t \*buf\_in, uint8\_t \*buf\_out, uint16\_t \*size);

SAT\_returnState HLDLC\_frame(uint8\_t \*buf\_in, uint8\_t \*buf\_out, uint16\_t \*size);

## Packet Pool

Since malloc is prohibited, there was a need for a memory storing mechanism for ECSS packets.

A fixed array of the packet structure was used. An array with the same number of elements denotes the status of the corresponding packet, a status of free or active.

There are 4 operations for the packet pool:

* Initialize.
* get packet.
* free packet.
* idle.

The initialize function must be called before the packet pool can be used. It initializes all packet structures payload data pointers with the data arrays allocated. If any other operation is performed before the Initialize function is called, will result in error.

The get packet function returns a free packet from the packet pool. The function iterates the array until a free packet is found, changes the flag to active and returns the address of the structure. If there isn’t a free packet it returns NULL. The software module calling the get packet function should check for a NULL pointer. It takes the size needed for a parameter.

The free packet function returns a packet to the pool. It takes the address of the packet that it frees as a parameter, iterates the packet array and when the address is found, the status changed to free. If the address is not found in the array nothing happens.

Since this code is used in all subsystems that have different memory constraints, preprocessor statements are used to modify the number of packets and their data size.

The maximum size of a data payload is around 2k bytes. The maximum packet that can be transferred through COMMS is 210 bytes, for more than that the large transfer data service is used.

There are 3 groups of data sizes that can be distinguished:

* Telecommands that are up to 80 data bytes.
* Housekeeping packets that can reach the maximum of 210 bytes.
* Mass storage service operations that can reach to 2kbytes.

Since only COMMS and OBC is using the 2k data sizes and EPS has strict memory constraints, it was decided to have 2 types of packets: one normal with maximum size of 210 bytes and the extended that reaches 2k bytes of data payload. For simplification 2 types are used instead of 3.

A more elaborate scheme for the packet pool memory model was considered, with allocated data blocks of 210 bytes and for an extended data the packet would occupy the blocks required but it has 2 main disadvantages:

* Possible fragmentation would slow packet processing.
* The algorithm for handling of the block allocation would be more complex and would require more time for testing and development.

A note about thread safe implementation: the free function doesn’t require mutex protection because each packet can be freed once and the operation is atomic. The get packet was a candidate but there isn’t a problem for all subsystems since the function is not used in ISRs and on the OBC the function is only used in the serial task.

Listing 0-2 packet pool module functions

tc\_tm\_pkt \* get\_pkt(uint16\_t size);

SAT\_returnState free\_pkt(tc\_tm\_pkt \*pkt);

uint8\_t is\_free\_pkt(tc\_tm\_pkt \*pkt);

SAT\_returnState pkt\_pool\_INIT();

void pkt\_pool\_IDLE(uint32\_t tmp\_time);

## Queues

When a packet is about to be shipped in another subsystem, the pointer of the packet structure is pushed in a queue according to the destined subsystem. The OBC has 3 queues responding to to each subsystem the OBC connects. The other subsystems have only 1 queue since they connect only to the OBC. The export packet function checks if the queue is not empty and if the UART peripheral is available and if that’s the case, it pops the structure’s pointer and then sends it.

The queue functions are defined in the core folder. The queue module functions are the basic used in queues: with push a packet structure is added in the queue, pop gets a packet from the queue if it has one, size returns the number of packets in the queue and the peak gets a packet with removing it from the queue and finally the idle was developed for fault tolerance but it wasn’t used.

Listing 0-3 Queue module functions

SAT\_returnState queuePush(tc\_tm\_pkt \*pkt, TC\_TM\_app\_id app\_id);

tc\_tm\_pkt \* queuePop(TC\_TM\_app\_id app\_id);

uint8\_t queueSize(TC\_TM\_app\_id app\_id);

tc\_tm\_pkt \* queuePeak(TC\_TM\_app\_id app\_id);

void queue\_IDLE(TC\_TM\_app\_id app\_id);

## Peripheral modes

In this section, the operation modes of the microcontroller’s peripheral will be discussed while focusing in the UART. Peripherals like SPI and I^2^C have similar functionality. The UART and most of the STM32F4’s peripherals have 3 modes of operation:

* blocking.
* interrupt.
* DMA.

The first mode is very simple, the function constantly checks a flag, probably a SFR bit. When the bit changes state, it signals an event like a new character is received or a character has finished transmission.

The interrupt mode uses the built-in hardware functions in order to free the CPU from constantly checking a SFR. When an event happens, the CPU stop the normal operation and jumps to an ISR function that handles the event. That way CPU cycles are only used for the processing of the event.

Finally, when the DMA mode is used, the events are processed without the use of the CPU. In particular usually a buffer address is configured along with the size of the data. During receive the buffer is filled automatically and in transmission the data are taken from the buffer. When the transmission is finished or when the data received are reached the size defined an ISR is triggered signalling the events end.

The blocking mode has simpler operation but it wastes CPU cycles for constantly checking the SFR. This is great reduced with the interrupt mode but there is some overhead. The DMA mode is the most efficient mode of operation with no overhead but while the other modes don’t have a restriction on the number of characters, the DMA is only efficient when used with a predefined fixed number of characters.

The Standard Peripheral provides 3 set of functions: blocking, interrupt and DMA. The only difference in the provided functions is in the blocking mode where a timeout parameter is added. It is worth noting that in all 3 different types of microcontrollers used in CubeSat have the almost the same implementation. This simplifies and reduces the time needed for the implementation of the ECSS handling.

DMA mode was used for the reception and the transmission.

## ECSS services

The services folder contains all the code related to the services implemented in the ECSS document. Each service is implemented in a module that is uncoupled with each other; The services module contains all the configuration parameters for the services. The service utilities contain functions that are helpful for the ECSS services. If a service need information that is specific to a subsystem it uses the function that is defined in the platform folder in the respective file e.g., housekeeping for the housekeeping service. The subsystem-id file has all the application ids of CubeSat.

A single-entry point for all incoming packets is used for a better modular design and is denoted by the -app. Each function that belongs to a module start with the service name e.g., mass-storage-app.

## CubeSat module

The CubeSat module has a collection of functions that are used from all subsystems together with with the ECSS services.

Listing 0-4 CubeSat module functions

SAT\_returnState import\_pkt(TC\_TM\_app\_id app\_id, struct uart\_data \*data);

SAT\_returnState export\_pkt(TC\_TM\_app\_id app\_id, struct uart\_data \*data);

SAT\_returnState test\_crt\_heartbeat(tc\_tm\_pkt \*\*pkt);

SAT\_returnState firewall(tc\_tm\_pkt \*pkt);

The import and export packet functions are used for processing incoming packets and transmitting packets through the UART to their corresponding subsystem.

The import and export packet function takes the application ID that the function operates for and the respectively data that are packed into the uart-data structure.

The import packet functions check if there is a new packet, it is deframed from HLDCL encapsulation, unpacket into a packet taken from the packet pool and if those process are finished without error the packet is routed into the respectively services handler or forwarded to another subsystem. The verification service handler is called then and finally if the packet is for that subsystem it is returned to the packet pool since all processing is finished. If the packet is for another subsystem the packet is freed when the export packet function finishes.

The export packet function checks if the UART peripheral is not transmitting another packet which in that case isn’t anything more to do until the packet is finished transmitting. After that it checks if there is any packet in the queue, if there is the packet is packed from the packet structure into a temporary buffer array, HLDLC encapsulation, transmit ion from the UART and finally the packet is returned to the packet pool.

At first the export packet was called directly when a new packet was created and needed to forward it to its destination. That created 2 issues: First in the case the export packet was used inside the import packet e.g., for a telecommand response, the import had to wait for the export function to send it. If the UART was already used from another packet, it had to wait for that operation to finish as well. That could lead in packet loss if a new packet arrived while the import function was in use. Also, it created unwanted coupling between the import and export module. Secondly for the OBC which is multithreaded, it created race conditions.

The solution of this problem was the use of queues. Each time a new packet is generated the pointer to the structure is pushed in the queue of the destined subsystem. When the export packet is called, the packet pointer is retrieved from the the queue. This uncouples the 2 functions. The race conditions are solved by the way that the export function is called from only one place, so the pop function of the queue is atomic. The push function which can be used from all the threads is solved through the use of queues.

Listing 0-5 import function

SAT\_returnState import\_pkt(TC\_TM\_app\_id app\_id, **struct** uart\_data \*data)

{

tc\_tm\_pkt \*pkt;

uint16\_t size = 0;

SAT\_returnState res;

SAT\_returnState res\_deframe;

res = recieve\_packet(app\_id,data);

**if**( res == SATR\_EOT ) {

size = data->uart\_size;

res\_deframe = HLDLC\_deframe(data->uart\_unpkt\_buf, data->deframed\_buf, &size);

**if**(res\_deframe == SATR\_EOT) {

pkt = get\_pkt(size);

**if**((!C\_ASSERT(pkt != **NULL**)) == **true**) { **return** SATR\_ERROR; }

**if**((res = unpack\_pkt(data->deframed\_buf, pkt, size)) == SATR\_OK) {

stats\_inbound(pkt->type, pkt->app\_id, pkt->dest\_id, pkt);

route\_pkt(pkt); }

**else** {

stats\_dropped\_upack();

pkt->verification\_state = res;

}

verification\_app(pkt);

TC\_TM\_app\_id dest = 0;

**if**(pkt->type == TC) { dest = pkt->app\_id; }

**else** **if**(pkt->type == TM) { dest = pkt->dest\_id; }

**if**(dest == SYSTEM\_APP\_ID) {

free\_pkt(pkt);

}

}

**else** {

stats\_dropped\_hldlc();

}

}

**return** SATR\_OK;

}

Listing 4.12 export function

SAT\_returnState export\_pkt(TC\_TM\_app\_id app\_id, **struct** uart\_data \*data) {

tc\_tm\_pkt \*pkt = 0;

uint16\_t size = 0;

SAT\_returnState res = SATR\_ERROR;

/\* Checks if the tx is busy \*/

**if**((res = uart\_tx\_check(app\_id)) == SATR\_ALREADY\_SERVICING) { **return** res; }

/\* Checks if that the pkt that was transmitted is still in the queue \*/

**if**((pkt = queuePop(app\_id)) == **NULL**) { **return** SATR\_OK; }

stats\_outbound(pkt->type, pkt->app\_id, pkt->dest\_id, pkt);

pack\_pkt(data->uart\_pkted\_buf, pkt, &size);

res = HLDLC\_frame(data->uart\_pkted\_buf, data->framed\_buf, &size);

**if**(res == SATR\_ERROR) { **return** SATR\_ERROR; }

**if**((!C\_ASSERT(size > 0)) == **true**) { **return** SATR\_ERROR; }

send\_packet(app\_id, data->framed\_buf, size);

free\_pkt(pkt);

**return** SATR\_OK;

}

## Service module

The service module provides most of the information related to the ECSS services. Here all significant definitions of the ECSS specification used by software are found: services types and subtypes names are defined, the SAT-returnState that holds all states of the software, supporting type definitions of each service, definitions, packet structure and assertion macro definition along with configuration definitions like the maximum size of a packet.

The supporting type definitions of each service could had been define in each service module and maybe that design was better in terms of software separation design but it was decided to be in the service module in one place as a way for the developer or used to quickly find information about the service inputs. Moreover, a global definition file that holds the most important definitions would save the developer from searching in multiple files for key parameters.

## Error codes

The ECSS module are using one enumeration for status codes that could happen during the process of a packet. The status codes are used for debugging, logging and in the verification service in the case of a telecommand failure. The enumeration plays a key part in error handling, since all functions in the ECSS module return the enumeration. The design intend was to have 1 status code for all the failures in order to easily identify the source of error, this happened in the most part but unfortunately not in the extend that was wished for.

There are status codes used for different modules. The first 6 (0-5) are defined in the ECSS standard. The 6th OK means that everything happened as planned. ERROR provides a generic name for failure. Scheduling service uses the 16-29 codes. The status codes of the FatFS are shifted between number 30-50. The final 5 provide specific errors.

## ECSS packet structure

The ECSS packet structure holds packets in that form when they are process, this structure plays the prime role since most all functionality in the ECSS module revolves to a packet. Most of the field of the structure are named after the counterparts in the ECSS specification and hold the same amount of information in bytes. The only difference is the source or destination ID which the name is different according the type of the packet but it was decided to both share the same field and the structure member was named dest-id from destination ID since most of the times that it was used in the code was when the variable held the destination ID and it was easier to remember.

The only structure member that doesn’t belong to the ECSS specification is the verification-state that is a supporting variable for the verification service and holds the verification state that the packet is in. If the state was in another variable it would only make the code more complicated.

Listing 4.13 Assertion preprocessor macro definition and calling example

#define C\_ASSERT(e) ((e) ? (true) : (tst\_debugging(\_\_FILE\_ID\_\_, \_\_LINE\_\_, #e)))

if(!C\_ASSERT(\*size <= UART\_BUF\_SIZE) == true) { return SATR\_ERROR; }

Listing 4.14 ECSS module packet structure definition

typedef struct {

/\* packet id \*/

//uint8\_t ver; /\* 3 bits, should be equal to 0 \*/

//uint8\_t data\_field\_hdr; /\* 1 bit, data\_field\_hdr exists in data = 1 \*/

TC\_TM\_app\_id app\_id; /\* TM: app id = 0 for time packets, = 0xff for idle packets. should be 11 bits only 8 are used though \*/

uint8\_t type; /\* 1 bit, tm = 0, tc = 1 \*/

/\* packet sequence control \*/

uint8\_t seq\_flags; /\* 3 bits, definition in TC\_SEQ\_xPACKET \*/

uint16\_t seq\_count; /\* 14 bits, packet counter, should be unique for each app id \*/

uint16\_t len; /\* 16 bits, C = (Number of octets in packet data field) - 1, on struct is the size of data without the headers. on array is with the headers \*/

uint8\_t ack; /\* 4 bits, definition in TC\_ACK\_xxxx 0 if its a TM \*/

uint8\_t ser\_type; /\* 8 bit, service type \*/

uint8\_t ser\_subtype; /\* 8 bit, service subtype \*/

/\*optional\*/

//uint8\_t pckt\_sub\_cnt; /\* 8 bits\*/

TC\_TM\_app\_id dest\_id; /\*on TC is the source id, on TM its the destination id\*/

uint8\_t \*data; /\* pkt data \*/

/\*this is not part of the header. it is used from the software and the verification service,

\*when the packet wants ACK.

\*the type is SAT\_returnState and it either stores R\_OK or has the error code (failure reason).

\*it is initialized as R\_ERROR and the service should be responsible to make it R\_OK or put the corresponding error.

\*/

SAT\_returnState verification\_state;

}tc\_tm\_pkt;

## Assertions

Used directly from the JPL’s 10 rules, assertions are used whenever possible. It is defined as a preprocessor macro definition that calls the test-debugging function.

Most checks are for NULL pointers and wrong ranges in parameters. The checks that are using assertions are only for parameters that are invalid and denote wrong behavior. There are also used for fault containment.

Assertions are defined as a preprocessor macro, when the e expression is false the tst-debugging is called which takes the filename, the line in the file that the assertions is placed and finally the expression. These parameters help to identify where the error happened. The assertion code is taken directly from the JPL 10 rules.

## Service utilities module

The service utilities module contains a collection of functions that complementary to the use of the ECSS modules.

The cnv functions converts from an uint8-t to the corresponding type (16

* 32 bits) and vice versa. Using that functions all subsystem share the same endianness type. It was designed for packet conversion from the structure to the 8-bit array used for transmission and from the raw 8-bit array to the packet structure.

There are 2 ways for converting a variable in C language to 8 bits and vice versa:

* Using unions.
* Using shifts and bit masking operations.

The union approach has the advantage of better code clarity.

The checksum is used for calculating the error checking byte of the ECSS packet. It is used when the packet is received for error checking and for calculating it when its for transmission. The checksum is a simple XOR based algorithm.

The STM32 that is used in all subsystems have a hardware peripheral for calculating the checksum but a software implementation was preferred even though the hardware peripheral would be more efficient. This was primary for fault tolerance issues, in the case the hardware was destroyed from the space radiation, it would render all subsystem communications useless since the checksum would fail. If there is a hardware failure in the ALU of the microcontroller that would be used from the software checksum, all operations on the microcontroller would fail, disabling the microcontroller.

The sys-data-init is used in initialization and initializes the state of the ECSS packet sequence counters for all the application IDs.

The create packet functions crt-pkt initializes a packet structure.

The unpack packet function gets a packet in a raw 8-bit array and it fills a packet structure while making the necessary checks. The pack packet function gets the information in a packet structure and fills an 8-bit array in order to transmit the data through the UART.

Listing 4.15 Service utilities module functions

SAT\_returnState checkSum(const uint8\_t \*data, const uint16\_t size, uint8\_t \*res\_crc);

SAT\_returnState unpack\_pkt(const uint8\_t \*buf, tc\_tm\_pkt \*pkt, const uint16\_t size);

SAT\_returnState pack\_pkt(uint8\_t \*buf, tc\_tm\_pkt \*pkt, uint16\_t \*size);

SAT\_returnState crt\_pkt(tc\_tm\_pkt \*pkt, TC\_TM\_app\_id app\_id, uint8\_t type, uint8\_t ack, uint8\_t ser\_type, uint8\_t ser\_subtype, TC\_TM\_app\_id dest\_id);

void cnv32\_8(const uint32\_t from, uint8\_t \*to);

void cnv16\_8(const uint16\_t from, uint8\_t \*to);

void cnv8\_32(uint8\_t \*from, uint32\_t \*to);

void cnv8\_16(uint8\_t \*from, uint16\_t \*to);

void cnv8\_16LE(uint8\_t \*from, uint16\_t \*to);

void cnvF\_8(const float from, uint8\_t \*to);

void cnv8\_F(uint8\_t \*from, float \*to);

void cnvD\_8(const double from, uint8\_t \*to);

void cnv8\_D(uint8\_t \*from, double \*to);

## Test service module

The implementation of the test service module is very simple, making the perfect candidate for a more through code examination. When a telecommand is received with (17,1) it replies with a telemetry packet of (17,2) with no data.

The test-app function serves as an entry point for the packet in the route. Most of the function’s code has to do with checks. The test-crt-pkt is a complimentary function

Listing 4.16 Test service module functions

SAT\_returnState test\_app(tc\_tm\_pkt \*pkt) {

tc\_tm\_pkt \*temp\_pkt = 0;

if(!C\_ASSERT(pkt != NULL && pkt->data != NULL) == true) { return SATR\_ERROR; }

if(!C\_ASSERT(pkt->ser\_subtype == TC\_CT\_PERFORM\_TEST) == true) { return SATR\_ERROR; }

test\_crt\_pkt(&temp\_pkt, pkt->dest\_id);

if(!C\_ASSERT(temp\_pkt != NULL) == true) { return SATR\_ERROR; }

route\_pkt(temp\_pkt);

return SATR\_OK;

}

SAT\_returnState test\_crt\_pkt(tc\_tm\_pkt \*\*pkt, TC\_TM\_app\_id dest\_id) {

\*pkt = get\_pkt(PKT\_NORMAL);

if(!C\_ASSERT(\*pkt != NULL) == true) { return SATR\_ERROR; }

crt\_pkt(\*pkt, SYSTEM\_APP\_ID, TM, TC\_ACK\_NO, TC\_TEST\_SERVICE, TM\_CT\_REPORT\_TEST, dest\_id);

(\*pkt)->len = 0;

return SATR\_OK;

}

## Telecommand verification service module

The telecommand verification service allows to see the outcome of a telecommand operation. If the operation is critical or the operator wants confirmation the acknowledgement flags are set in the packet header.

The implementation in order to be simple and efficient was straightforward. The verification state was added in the packet structure. There each service is responsible to place the state of the operation. During the packet’s allocation from the packet pool the state is set to initialized, in order to differentiate from other states.

After the processing of the telecommand is finished, the verification-app is called. It checks the acknowledgment flags in the telecommand frame and if a verification flag exists, it sends the corresponding telemetry packet. If the verification state in the telecommand flag indicates a failure, the verification state is used for the failure reason.

Moreover, the use of the verification state could be used to store more information for the status of the packet. The use of preprocessor macros that automatically add an error or a general state in the process could simplify the development.

During the implementation, the first design of export packet made the inline processing of the verification and the implementation of extra stages impossible without adding overhead that could interrupt the primary process. With the later addition of queues, the design could have been simpler. One different design would have a table with all packets needing verification and the verification service module residing in a different or in the idle task, checking for a change in the state. That way more verification states could have been used.

The implementation follows more or less the same scheme as the test service module.

## Event reporting service module

Table 0-24 Event service frame

|  |  |
| --- | --- |
| Subsystem ID Event ID Event time Data | 8 bits 8 bits 32 bits 32 bits |

Since the only storage medium is on the OBC, there wasn’t a way to store event logs in other subsystems. For that reason, it was decided that all events are to be sent to the OBC for storage. There is always the issue that some events won’t be stored for various reasons like power resets or packet loss but it is better to have more information and lose some than the opposite. An implication of using the OBC for event storage is that only the most critical events are send in order not to create too much traffic for the OBC to handle.

At first the UART was used for displaying debug messages. After the subsystems integration and since only ECSS packets could be forwarded to the umbilical UART, the debug messages were encapsulated in event service packets. The ASCII messages were used only for early debugging and there were replaced later.

The next design had a fixed packet format. For the implementation, each event had it’s own function and the subsystem’s developer was responsible for defining the subsystem’s events.

For design simplicity, the event-app had a pre-processor ifdef that checked the subsystem and if it was the OBC, it stored the event, in all subsystems it forwarded the packet to the OBC.

Listing 4.17 Event service module boot event example function

SAT\_returnState event\_boot(const uint8\_t reset\_source, const uint32\_t boot\_counter) {

tc\_tm\_pkt \*temp\_pkt = 0;

if(event\_crt\_pkt(&temp\_pkt, EV\_sys\_boot) != SATR\_OK) { return SATR\_ERROR; }

temp\_pkt->data[10] = reset\_source;

cnv32\_8(boot\_counter, &(temp\_pkt->data[11]));

for(uint8\_t i = 15; i < EV\_DATA\_SIZE; i++) { temp\_pkt->data[i] = 0; }

if(SYSTEM\_APP\_ID == OBC\_APP\_ID) {

event\_log(temp\_pkt->data, EV\_DATA\_SIZE);

} else {

route\_pkt(temp\_pkt);

}

return SATR\_OK; }

## Housekeeping & diagnostic data reporting service module

The most common task for a cubesat is housekeeping: in regular intervals data from each subsystem is gathered, stored and transmitted to earth. The data provide an insight to the state of the cubesat. For CubeSat there are 2 housekeeping functions, WOD and extended WOD.

The mechanism for WOD formation is that the OBC sends a telecommand request in each subsystem with the structure ID, the subsystems respond with the data, the OBC gathers all the data, stores it and forwards it to the COMMS subsystem, in order to transmit them to Earth. Each structure ID and application ID form a unique set of data.

There are 2 distinct functions of the housekeeping service:

* Requesting and retrieving data.
* Storing housekeeping data (OBC).

Since each subsystem has its own set of data related to different structure IDs, the housekeeping service calls the subsystem dependent functions that are in the platform folder. The functions in the module are used only for checking input parameters and calling the functions in the platform folder.

## OBC Housekeeping

OBC sends the WOD and extended WOD in 1 minute interval. First it sends the requests to EPS and COMMS with 1 second delay between them and with the health report structure ID. When a response arrives, the data are temporary stored in memory. If a response doesn’t come the respective data are left with 0 value. A mechanism for re-requesting the data wasn’t implemented. After 29 seconds OBC forms the WOD report with the responses from the subsystems, stores it and it sends it to COMMS for transmission. After the WOD comes the extended WOD. The same mechanism is used, only this time requests are send to all 3 subsystems and the structure ID is extended health.

Since the WOD requires 31 historic values resulting in data logged 31 minutes ago, the persistent RAM region of the OBC was used. The SRAM was used instead of the SD card because it minimizes the data access time and improves reliability since the delete function of the FatFS is not invoked. For storage, a circular buffer was implemented in the SRAM containing 32 sets of values.

## Mass storage service module

The mass storage service was the most difficult and complex module in CubeSat. It has the most lines of code and functions in services.

There are 2 main reasons that shaped the design of the module: the restraints of data sizes send through the RF channel and the limitations from the design and documentation of the FatFS library and the FAT file system specification.

Logs come from various sources like housekeeping and events. There are implemented as a circular buffer so in the case of a overflow the newest log overwrites the oldest one. The logs are stored for downloading when there is link with the ground station. After the logs are downloaded, they should be deleted so in the next link, the operator doesn’t re download them. Configuration files are files that are unique and they store values for parameters. Large unique files are the files that are larger than the normal data size used in mass storage and unique.

In CubeSat there are 4 specific types:

* Event logs.
* WOD logs.
* Extended WOD logs.
* Scheduling service configuration files.

It was decided that each type should have its own store ID and stores should be implemented with folders. Each store has different set of properties.

the normal and extended WOD. The logs share the same properties:

* Fixed size of records.
* Implemented as circular buffers.
* Unique data in entries.
* No need for modification after the log entry.
* Able to search, download and deleted.

Each log is implemented as a separate file. This happens mainly due to the FatFS and its limitations. The Filenames are implemented as numbers only, stored in ASCII number characters. With the addition of folders, it gives unique log entries. Numbers are used because it is easier and more efficient for processing them in the microcontroller than ASCII strings. e.g. the log 4916 occupies only 2 bytes in RAM as an integer in contrast of using 5 bytes as an ASCII string. Again, with integers file comparisons are a lot faster than as strings.

There was a though of using timestamps as the filenames of logs. The timestamp would had been the time of the log creation. The unique file name would have been achieved by using the correct time resolution. The reason that it wasn’t used was that with 1 millisecond resolution and 8 characters for the timestamp, the timer would rollover in ~51 days, which was way lower than the mission specifications. When the timer rolled over the result would had been to have logs that would be impossible to figure if they were created before or after the roll over. Moreover, it was difficult to guarantee that all the times the OBC would had correct time.

The circular buffer mechanism was used because of the nature of the log operations. First of all, having the circular buffer ensures that the logs won’t overrun and fill all the SD card. Moreover, since communication with the ground is limited, makes the downlink of a very large log number impossible.

## LIFE OF A PACKET

After the analysis of each part, it is imperative to give the reader an overview of the work flow, the best way is by describing the life of an incoming packet.

There are 4 steps for processing an incoming packet:

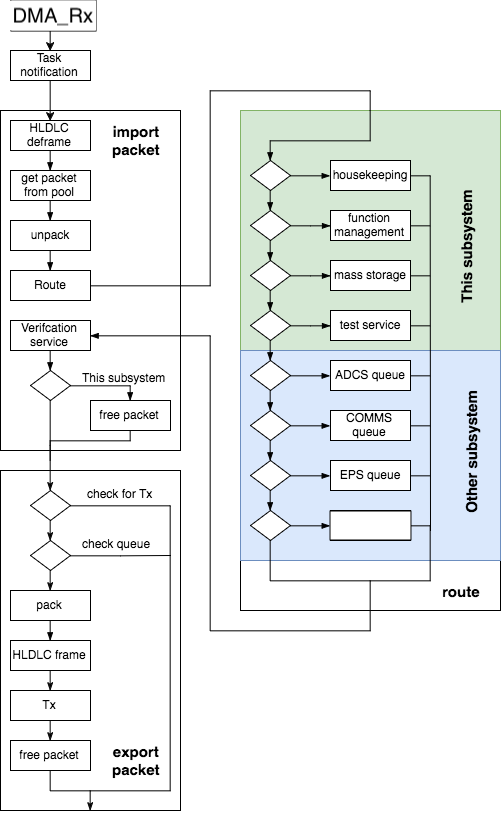
I. First the packet is received byte per byte from the UART DMA. If the packet is encapsulated in a valid HLDLC frame, the packet is stored from the DMA to a temporary buffer.

II. When the import function is called, the packet goes through the HLDLC deframe, a new packet is allocated from the packet pool and then the ecss-unpack gets the packet from the array, performs all necessary checks and places it in the packet structure. If the unpack receives a valid packet, then the route function is called. If there was a failure the verification service is called. Finally, if the packet was destined for that subsystem the packet’s state returns to free and it is available for reuse from the packet pool.

III. Route directs the packet to the correct service or the correct queue if the packet is indented for another subsystem. It is highly probable that the service used, generates a response, then the route is called again and the packet is placed in the correct queue for transmission. If there was an error in the route the packet is freed and the function returns the error.

IV. The verification module checks if the incoming packet needed an acknowledgement and if that’s the case, if the packet has all the necessary information the response is routed back.

The life of an outgoing packet is a lot simpler. The export function checks the queue for packet, if the UART is available, the packet is popped from the queue, it is packed to an 8-bit array from the packet structure, then is encapsulated in a HLDLC frame and finally is transmitted.



**Figure 0-9: The life of a packet**

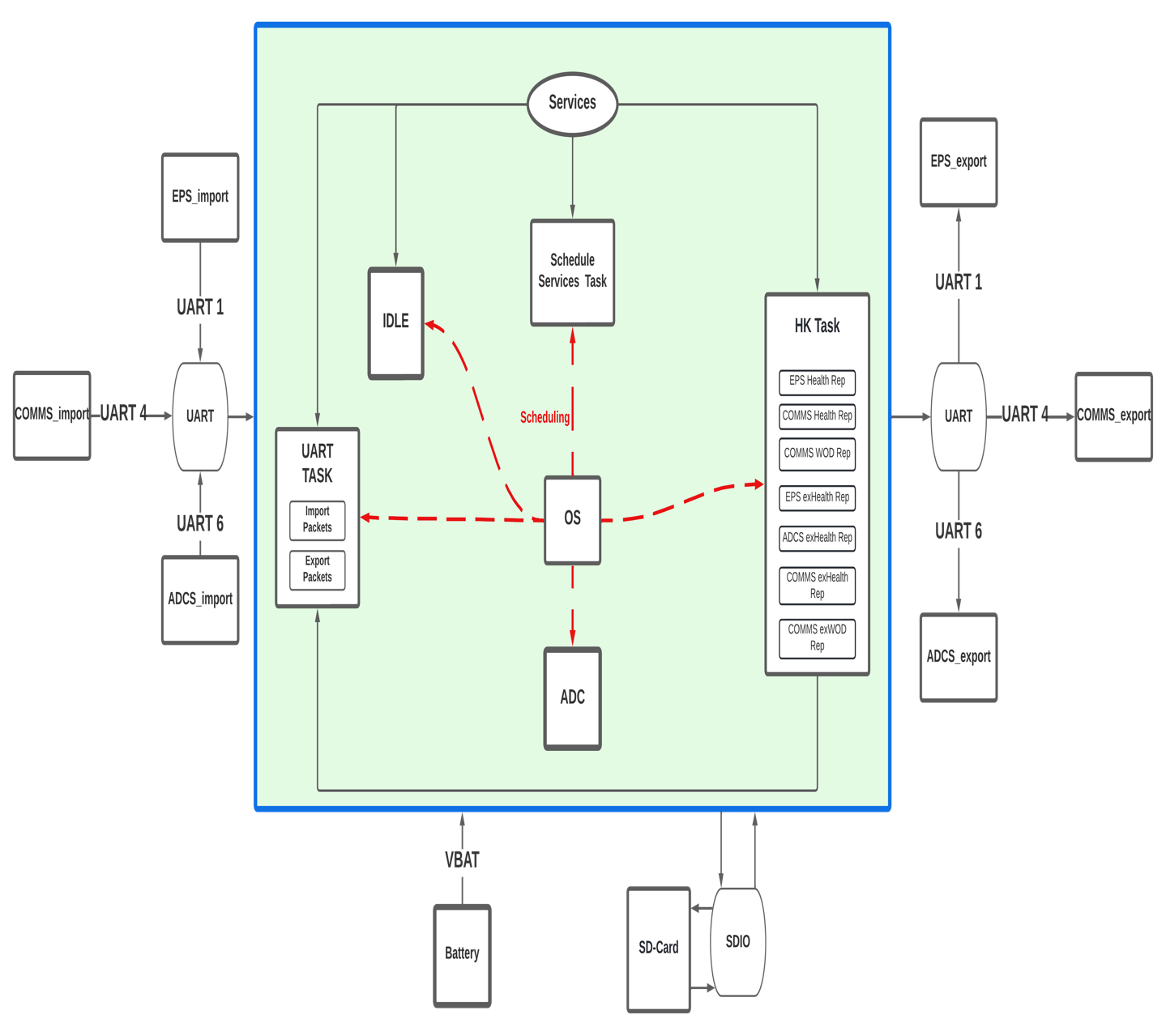
# 7- On-board computer Software

In this section, the software implementation for the OBC will be discussed.

## Operating System

We do this using Time Triggered‘ (TT) architectures that incorporate patented run-time monitoring techniques.

## Static Design:



There are 11 tasks:

* UART task.
* IDLE task
* Scheduling task.
* EPS Health Report task.
* EPS exHealth Report task.
* COMMS Health Report task.
* COMMS exHealth Report task.
* COMMS WOD Report task.
* COMMS EXT WOD Report task.
* ADCS exHealth Report task.
* ADC task.

**UART task**: It handles incoming and outgoing packets. This task is the most important since the first hard real-time requirement is to handle as soon as possible incoming packets so there aren’t any packet losses.

**IDLE Task**

* Check the sanity of the packet pool.
* Check the sanity of the queues.

**ADC Task**: Start and get the results of the A/D converter that is connected to the OBC battery.

EPS Health Report task, EPS exHealth Report task, COMMS Health Report task, COMMS exHealth Report task, COMMS WOD Report task, COMMS EXT WOD Report task and ADCS exHealth Report task are Housekeeping tasks handle all the housekeeping activities. First the housekeeping service module initialization is called which sets the buffer that is allocated for the reserved outgoing housekeeping packet. Then the task calls continuously the hk-sch function which sends housekeeping requests to the subsystems that form the normal and extended WOD. After each packet request is put in the queues , the UART task is notified and switched in order to send the requests.

**Scheduling Task**:

First, the scheduling service init function is called which loads the scheduling packets stored in the SD-card.

Then the update function takes the stored time-tagged commands(packets) that have been loaded from ground and release them to their destination application process(es) when their on-board time is reached.

**Housekeeping Tasks**:

First, the housekeeping service module initialization is called which sets the buffer that is allocated for the reserved outgoing housekeeping packet. Then the tasks are called periodically, sending housekeeping requests to the subsystems that form the normal and extended WOD. After each packet request is put in the queues, the UART task is notified and switched in order to send the requests.

EPS Health Request task, COMMS Health Request task: Create the health request packet and route the packet to the subsystem’s queue.

COMMS WOD Report task: Take the health report parameters received from the subsystems and create a packet, then store it on the SD-card and route it to COMMS queue.

EPS exHealth Request task, COMMS exHealth Request task, ADCS exHealth Request task: Create the extended health request packet and route the packet to the subsystem’s queue.

COMMS EXT WOD Report task: Take the extended health report parameters received from the subsystems and create a packet, then store it on the sdcard and route it to COMMS queue.

## Real time clock

The OBC has a coin li-on battery that supplies the RTC peripheral and 4kbytes of the SRAM when the main power is off. The RTC is part of the STM43F4 microcontroller but runs independently.

The RTC peripheral is used for storing the time without the need for the microcontroller running. It has a time drift that is larger than what the mission profile requires. For that reason, the time needs to be regular updated either from the ground station or from the GPS in the ADCS subsystem. Using the RTC simplifies the design.

The battery also powers a 4K bytes portion of the SRAM. This part stores configuration parameters of the OBC. Having the battery, the data remain after a reset.

This data is not stored in the SD or the flash memory because storing the data would take more time, that could lead to normal operation interaction. Moreover, continuous writing of the data could lead to file system corruption.

The event buffer was intended to be temporary stored in this portion of the memory and write the events in the SD when there was enough data, during the idle time of the CPU.

## Dynamic design

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task  Group | Task | Offset (tick) | Period (mS) | Period (tick) |
| TTRD-19A  related tasks | Watchdog Update | 0 | 10 | 1 |
| HEARTBEAT\_SW\_Update | 0 | 1000 | 100 |
| ADC1\_Update | 0 | 500 | 50 |
| PROCESSOR\_TASK\_Update | 0 | 1000 | 100 |
| Services  Tasks | EPS\_Health\_Rep\_task\_Update | 1 | 200 | 20 |
| EPS\_exHealth\_Rep\_task\_Update | 4 | 200 | 20 |
| ADCS\_exHealth\_Rep\_task\_Update | 5 | 200 | 20 |
| COMMS\_Health\_Rep\_task\_Update | 2 | 200 | 20 |
| COMMS\_WOD\_Rep\_task\_Update | 3 | 200 | 20 |
| COMMS\_exHealth\_Rep\_task\_Update | 6 | 200 | 20 |
| COMMS\_EXT\_WOD\_Rep\_task\_Update | 7 | 200 | 20 |
| SCHEDULE\_SERVICES\_Update | 2 | 1000 | 100 |
| Vbat Task | ADC1\_vbat\_Update | 5 | 200 | 20 |
| IDLE Task | IDLE\_Update | 1 | 1000 | 100 |
| OBC UART  Task | UART\_Update | 0 | 100 | 1 |

Table 0-25 OBC Dynamic Design

## SECURE DIGITAL INPUT/OUTPUT INTERFACE (SDIO)

SDIO is an interface designed as an extension for the existing SD card standard, to allow connecting different peripherals to the host with the standard SD controller.

It provides an interface between the APB2 peripheral bus and MultiMediaCards (MMCs),

SD memory cards, SDIO cards and CE-ATA devices.

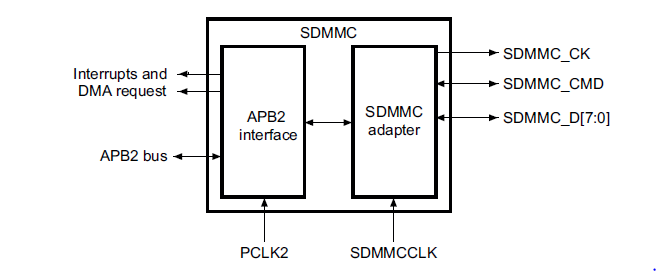
## SDIO FEATURES

* Data transfer up to 50 MHz for the 8 bit mode
* Data and command output enable signals to control external bidirectional drivers.
* Allows card to interrupt host.
* Operational Voltage range: 2.7-3.6V

## SDIO FUNCTIONAL DESCRIPTION

The SDIO consists of two parts:

* The SDIO adapter block provides all functions specific to the MMC/SD/SD I/O card such as the clock generation unit, command and data transfer.
* The APB2 interface accesses the SDIO adapter registers, and generates interrupt and DMA request signals.



**Figure 0-10:** SDIO block diagram