Subsystem Message Layout Proposal

Lunar Zebro Software Department

June 1, 2022

1 Introduction

All robotic systems that consist of hardware subsystems have some form of message layout which is used for intersystem communication. The purpose of this document is to review existing message layouts and propose a message layout which is optimised for the in-house developed hardware subsystems.

1.1 Outline

Section 2 will review existing message layouts and previous proposals. Section 3 will list the various in-house developed subsystems, RS-485 physical layer map and a list of commands per hardware subsystem. Section 4 will propose our own message format based on the results of the previous section.

2 Literature review

This section consist of a literature review in which we look at previously used message formats for intersystem communication as well as previous work proposed by the department.

2.1 Hyperion Technologies RS485 Modbuss ASCII message format

Hyperion Technologies uses a relative straightforward message format which is visualised in Figure 1.

1. START [1]: Indicating the start of a message which happens to be defined as $0x3A$ [1], [2]
2. ADDRESS [2]: Indicating which subsystem or component of a subsystem the destination is [1], [2]
3. COMMAND [2]: Indicating what the request- or action to be taken is [1], [2]
4. DATA [8]: Possible data that is needed for the COMMAND [1], [2]
5. CRC [2]: CRC-16 algorithm output such that the receiver can verify data integrity [1], [2]
6. STOP [1]: Indicating the end of a message which happens to be defined as $0x0A$ [1], [2]

Messages are always 16 bytes in length and best case scenario the overhead is 8 bytes (50%). This happens when all 8 bytes in the `DATA` field are used. However, in most cases there is no data needed or only portion of the field is used. The message layout is simple since it does not support variable lengths, does not support for resending the same message when a CRC error is detected and relatively speaking has quite some overhead. It acts as a "fire and forget" system.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
START	ADDRESS		COMMAND		DATA								CRC		STOP

Figure 1: Hyperion Technologies message format used for the BMS and MD

A characteristic of the protocol is the use of ASCII hex to represent the bytes. Every byte is represented by a pair of ASCII codes that represent the byte's two hexadecimal characters. By doing so, only $[0, 9]_{16}$ and $[A, F]_{16}$ are used for data representation. The benefit is that all other codes are free for other uses such as flow-control codes, end-of-file indicators, or network addresses [3]. However, the downside is that the message length becomes twice as large. The message structure and representation is a modified version of Modbus ASCII.

2.2 Message manager format

3 Hardware subsystems- and commands

There are a couple of in-house developed hardware subsystems. Table 1 list each of them along with a short description. The external non-volatile memory is not included because it does not make use of the RS-485 bus but instead uses the Serial Peripheral Interface (SPI) protocol. Also note that communication to the umbilical is not included as of now. This is because the very details of umbilical are not released at the time of writing.

Subsystem	*Description*
SHRIMP (Small High Resolution Independent Modular Photographer)	Tiny camera subsystem that takes 640x480 images and has an Inertial Measurement Unit (IMU) consisting of a gyroscope and magnetometer.
COMMs (Communication subsystem)	Responsible for retrieving command- and data from Earth and send back telemetry- and payload data to the ground station.
Payload	Hardware subsystem that measures various radiation particles.
PPU (Power Processing Unit)	Microcontroller that is responsible for power distribution across the rover as well as looking after any anomalies.

Table 1: List of the different hardware subsystems that are designed or programmed in-house

Now that we have established the different hardware subsystems we can list the commands on a per subsystem basis. Table 2 lists the the different commands that SHRIMP will have to support with eventual return values.

Command	*Description*	*Return*
`RESET`	Soft reset of the entire SHRIMP submodule. This includes the firmware of SHRIMP, the IMU sensor and the camera.	-
`RESET_IMU`	Soft reset only the IMU sensor. This restores registers of the IMU back to default values. The default is to be determined.	-
`RESET_CAMERA`	Soft reset only the camera. This restores registers of the camera back to default values. The default is to be determined.	-
`GET_TEMPERATURE`	Requests and sends back the temperature which is part of the IMU sensor.	Temperature (uint16_t)
`TAKE_PICTURE`	See TODO note	Pixel data (307200 bytes)

Table 2: List of commands that SHRIMP will support

For the TAKE_PI command we can make a seperate command to actuall send it if taking a picture takes too much tim

Table 3 lists the different commands that the payload sensor supports. The Payload consist of a microcontroller and a Floating Gate Dosimeter (FGDOS) chip which holds 2 digital radiation sensors. Please note that some rows in Table 3 have a yellow shade. These commands are only supported during the mission and not during the 2022 NLR test. Please also note that a lot of commands end with either `1` or `2`. Because the FGDOS holds two different sensors one has to explicitely address which of the two sensors the particular command is destined for.

Command	*Description*	*Return*
GET_RUNTIME	Time since the last power cycle has happened in milliseconds.	Runtime (uint32_t) (ms)
GET_MOSFET_FREQ_1	Get the frequency that the MOSFET of FGDOS sensor 1 is switching. Nominal range is $[40, 90]$ KHz	Frequency (uint32_t) (Hz)
GET_MOSFET_FREQ_2	Get the frequency that the MOSFET of FGDOS sensor 2 is switching. Nominal range is $[40, 90]$ KHz	Frequency (uint32_t) (Hz)
GET_MOSFET_REF_FREQ_1	Get the reference frequency that the MOSFET of FGDOS sensor 1 is supposed to switch. Nominal range is $[65, 75]$ KHz	Frequency (uint32_t) (Hz)
GET_MOSFET_REF_FREQ_2	Get the reference frequency that the MOSFET of FGDOS sensor 2 is supposed to switch. Nominal range is $[65, 75]$ KHz	Frequency (uint32_t) (Hz)
GET_TEMP_1	Returns the internal temperature read from sensor 1. Nominal range is $[40, 230]$ digits. Conversion from digits to degrees Celsius is not done by the payload sensor.	Temperature (uint8_t) (digits)
GET_TEMP_2	Returns the internal temperature read from sensor 2. Nominal range is $[40, 230]$ digits. Conversion from digits to degrees Celsius is not done by the payload sensor.	Temperature (uint8_t) (digits)
GET_RECHARGES_1	Number of times that sensor 1 has been recharged. Nominal range is $[0, 15]$ recharges.	Recharges (uint8_t)
GET_RECHARGES_2	Number of times that sensor 2 has been recharged. Nominal range is $[0, 15]$ recharges.	Recharges (uint8_t)
GET_RECHARGE_TARGET_FREQ_1	Returns the target recharge frequency of sensor 1.	Frequency (uint32_t) (Hz)
GET_RECHARGE_TARGET_FREQ_2	Returns the target recharge frequency of sensor 2.	Frequency (uint32_t) (Hz)
GET_RECHARGE_THRESHOLD_FREQ_1	Returns the recharge threshold frequency of sensor 1.	Frequency (uint32_t) (Hz)
GET_RECHARGE_THRESHOLD_FREQ_2	Returns the recharge threshold frequency of sensor 2.	Frequency (uint32_t) (Hz)
GET_MICROCONTROLLER_TEMP	Returns the temperature measured from the internal temperature sensor which sits in the microcontroller that interfaces with the radiation sensor.	Temperature (int8_t)
GET_OPERATIONAL_MODE	Returns the current mode in which the payload operates in.	Mode (uint8_t)
GET_ERROR_CODE	Returns a sequence of bits which indicate whether or not the payload had any error or anomaly.	Error (uint8_t)
RESET_MICROCONTROLLER	Resets the microcontroller that interfaces with the radiation sensor.	-

Table 3: List of commands that payload will support

Table 4 lists the different commands that the PPU will support.

Command	*Description*	*Return*
TURN_MOTORx_OFF	Electrically cuts the power to motor x where $0 \leq x \leq 6$ and where $x = 6$ represents the solar panel motor.	-
TURN_MOTORx_ON	Provide power to motor x where $0 \leq x \leq 6$ and where $x = 6$ represents the solar panel motor.	-
TURN_OBC5V_OFF	Electrically cuts the 5v power rail of the OBC.	-
TURN_OBC5V_ON	Provide power to the 5v power rail of the OBC.	-
TURN_OBC3V3_OFF	Electrically cuts the 3v3 power rail of the OBC.	-
TURN_OBC3V3_ON	Provide power to the 3v3 power rail of the OBC.	-
TURN_SHRIMPx_OFF	Electrically cuts the power to SHRIMP x where x is either 0 or 1.	-
TURN_SHRIMPx_ON	Provide power to SHRIMP x where x is either 0 or 1.	-
TURN_BMS_OFF	Electrically cuts the power to the BMS.	-
TURN_BMS_ON	Provide power to the BMS.	-
TURN_5V_OFF	Electrically cuts the power to the 5v regulator.	-
TURN_5V_ON	Provide power to the 5v regulator.	-
TURN_3V3_OFF	Electrically cuts the power to the 3v3 regulator.	-
TURN_3V3_ON	Provide power to the 3v3 regulator.	-
TURN_COMMS_OFF	Electrically cuts the power to the COMMs subsystem.	-
TURN_COMMS_ON	Provide power to the COMMs subsystem.	-
TURN_PAYLOAD_OFF	Electrically cuts the power to the payload subsystem.	-
TURN_PAYLOAD_ON	Provide power to the payload subsystem.	-
GET_MOTORx_CURRENT	Retrieve the latest measurement of current flowing through MD x where $0 \leq x \leq 6$ and where $x = 6$ represents the solar panel motor.	Current (int16_t) (mA)
GET_MOTORx_VOLTAGE	Retrieve the latest measurement of voltage across MD x where $0 \leq x \leq 6$ and where $x = 6$ represents the solar panel MD.	Voltage (int16_t) (mV)
GET_MOTORx_POWER	Retrieve the latest measurement of power that is consumed by MD x where $0 \leq x \leq 6$ and where $x = 6$ represents the solar panel MD.	Power (int16_t) (mW)
GET_OBC5V_CURRENT	Retrieve the latest measurement of current flowing through the 5v rail of the OBC.	Current (int16_t) (mA)
GET_OBC5V_VOLTAGE	Retrieve the latest measurement of voltage across the 5v rail of the OBC.	Voltage (int16_t) (mV)
GET_OBC5V_POWER	Retrieve the latest measurement of poewr that is consumed by the 5v rail of the OBC.	Power (int16_t) (mW)
GET_OBC3V3_CURRENT	Retrieve the latest measurement of current flowing through the 3v3 rail of the OBC.	Current (int16_t) (mA)
GET_OBC3V3_VOLTAGE	Retrieve the latest measurement of voltage across the 3v3 rail of the OBC.	Voltage (int16_t) (mV)
GET_OBC3V3_POWER	Retrieve the latest measurement of poewr that is consumed by the 3v3 rail of the OBC.	Power (int16_t) (mW)

`GET_SHRIMPx_CURRENT`	Retrieve the latest measurement of current flowing through SHRIMP x where x is either 0 or 1.	Current (int16_t) (mA)
`GET_SHRIMPx_VOLTAGE`	Retrieve the latest measurement of voltage across SHRIMP x where x is either 0 or 1.	Voltage (int16_t) (mV)
`GET_SHRIMPx_POWER`	Retrieve the latest measurement of power that is consumed by SHRIMP x where x is either 0 or 1.	Voltage (int16_t) (mV)
`GET_BMS_CURRENT`	Retrieve the latest measurement of current flowing through the BMS.	Current (int16_t) (mA)
`GET_BMS_VOLTAGE`	Retrieve the latest measurement of voltage across the BMS.	Voltage (int16_t) (mV)
`GET_BMS_POWER`	Retrieve the latest measurement of power that is consumed by the BMS.	Power (int16_t) (mW)
`GET_5V_CURRENT`	Retrieve the latest measurement of current flowing through the 5v regulator.	Current (int16_t) (mA)
`GET_5V_VOLTAGE`	Retrieve the latest measurement of voltage across the 5v regulator.	Voltage (int16_t) (mV)
`GET_5V_POWER`	Retrieve the latest measurement of power that is consumed by the 5v regulator.	Power (int16_t) (mW)
`GET_3V3_CURRENT`	Retrieve the latest measurement of current flowing through the 3v3 regulator.	Current (int16_t) (mA)
`GET_3V3_VOLTAGE`	Retrieve the latest measurement of voltage across the 3v3 regulator.	Voltage (int16_t) (mV)
`GET_3V3_POWER`	Retrieve the latest measurement of power that is consumed by the 3v3 regulator.	Power (int16_t) (mW)
`GET_COMMS_CURRENT`	Retrieve the latest measurement of current flowing through the COMMs subsystem.	Current (int16_t) (mA)
`GET_COMMS_VOLTAGE`	Retrieve the latest measurement of voltage across the COMMs subsystem.	Voltage (int16_t) (mV)
`GET_COMMS_POWER`	Retrieve the latest measurement of power that is consumed by the COMMs subsystem.	Power (int16_t) (mW)
`GET_PAYLOAD_CURRENT`	Retrieve the latest measurement of current flowing through the payload.	Current (int16_t) (mA)
`GET_PAYLOAD_VOLTAGE`	Retrieve the latest measurement of voltage across the payload.	Voltage (int16_t) (mV)
`GET_PAYLOAD_POWER`	Retrieve the latest measurement of power that is consumed by the payload.	Power (int16_t) (mW)

`PING_PPU`	A multifunctioning command which is expected by the PPU to be retrieved every TBD. This is to feed a software-based watchdog timer programmed in the PPU. As a response to this command it will let the OBC know what kind of errors the PPU has come across, if any.	Error (uint32_t)

Table 4: List of commands that PPU will support

It is obvious that there is a lot of similarities between a lot of commands. The vast majority of commands can be parameterized in the following way

1. `TURN_SUBSYSTEM_ON`
2. `TURN_SUBSYSTEM_OFF`
3. `GET_SUBSYSTEM_CURRENT`
4. `GET_SUBSYSTEM_VOLTAGE`
5. `GET_SUBSYSTEM_POWER`

Although some commands are possible to be executed, they may not be practical. For instance, `TURN_3V3_OFF` results in the 3v3 regulator to be switched off. The PPU itself is also powered from the 3v3 regulator. Switching the 3v3 regulator off implies that the PPU will be switched off as well, and so will the OBC. The list may be updated as the functionality of the PPU has not been finalized yet.

4 Message format proposal

This section will list various questions raised by defining commands in the previous section. There is no single message format suitable for all robotic systems. Depending on the answers, a different message format will be proposed and backed up with reaons. This section will not answer those questions but rather list them and reason through them.

The first point of discussion is en- and decoding. Hyperion Technologies uses Modbus ASCII to make their message size about twice as long but only use $[0,9]_{16}$ and $[A,F]_{16}$. By doing so, special characters such as the colon is reserved for the `START` byte and line feed is used for the `STOP` byte. Doubling the message size is not of a huge problem for almost all commands except for `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) and `RECEIVE_DATA` (COMMs) as they have a lot of data. The result of not en- and decoding data is that the `START`- and `STOP` byte are not unique as these particular bytes could be present in the `DATA` field. That does not mean that there is no value in having those bytes in a message protocol. It could still serve as a marker for all subsystems that the first index contains the `START` byte and the last index contains the `STOP` byte. Determining the indices or the length of the message, however, could not solely be done by counting the amount of bytes until the `STOP` byte is found.

The second point of discussion is introducing a `LENGTH` field. Hyperion Technologies uses a static length of 8 bytes for the `DATA` field. Certain commands of Hyperion Technologies do not use the `DATA` field at all and some only do use half of it. The convention is that all the unused parts of the `DATA` field are filled with zeroes. If commands have a fixed length in the `DATA` field (e.g. arguments or fixed length in return value) then one could argue that there is no need for a `LENGTH` field as subsystems and the OBC could agree upon a fixed length for different commands, or just like Hyperion Technologies, a fixed length for the `DATA` field and zero out the unused parts. However, the fixed length then is determined by the command with the largest `DATA` size which most likely is `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) or `RECEIVE_DATA` (COMMs).

The third point of discussion is the `CRC` field and its function. `CRC` is used for the receiver to verify the integrity of the data. This is very common to have in almost any communication protocol. One wants to be sure that the right command is executed with the right parameters, or that the correct payload data is send to GS. The underlying question to consider is what should happen when the receiver detects that data integrity is not intact. A few factors play a role. For all the relative small commands, that is all commands except `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) and `RECEIVE_DATA` (COMMs), there could be a `RESEND` command which will latest send the packet again. This implies that subsystems can not discard data once it has been send but should always buffer the last data in case it is requested again. This works if all commands and data is send in one go. An argument to send all data in one message is to simplify communication while also be able to resend data if requested. A counterargument would be that resending the entire message response of the `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) or `RECEIVE_DATA` (COMMs) commands take quite some time.

4.1 Proposal 1: No length or sequence number

The first proposal is to have a fixed length N for the `DATA` field used for all commands. This implies that messages are not partioned. The advantage is that all subsystems know how many bytes to expect upon each message and communication is relatively simple. However, there are two disadvantages. The first one being that N is determined by the largest message and thus other subsystems who want to a significantly smaller amount of data still have to send so-called dummy bytes to comply with the message format. The second disadvantage is that when a subsystem is requested to resend the result, all those dummy-bytes are send again. This first proposal therefore is simple but not efficient. It is visualised in Figure 2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
START	SUBSYSTEM		COMMAND		DATA								CRC		STOP

Figure 2: Fixed length N for the `DATA` field

4.2 Proposal 2: mutual agreement of variable data size N

The second proposal wants to get rid of dummy bytes for all kind of messages without introducing any new field. This is done by establishing a mutual agreement between the software running on the OBC and the subsystem. A small advantage compared to proposal 1 is that no dummy bytes will be send in the `DATA` field. However, the `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) and `RECEIVE_DATA` (COMMs) commands are very large in size. The image of SHRIMP has a static size, but messages send to GS as well as receiving from GS will very likely be variable in size. This message format does not offer enough flexibility in the message size to be changed during runtime. One could see the resulting size of the message format as well as the structure in Figure 3.

0	1	2	3	4	5	6	...	...	...	...	$N-1$	N	$N+1$	$N+2$	$N+3$
START	SUBSYSTEM		COMMAND		DATA								CRC		STOP

Figure 3: Support for non-flexible length of the data`DATA` field without introducing any new field

4.3 Proposal 3: Variable `DATA` length by adding a `LENGTH` field

The third proposal is to get rid of dummy bytes for all kind of messages. We could do so by introducing a `LENGTH` field which will inform how many bytes the `DATA` field contains. If the receiver detects that data integrity is not intact then it can use the proposed `RESEND` command. But so far this proposed `RESEND` command would only send the latest message back. If data is split up for the `TAKE_PICTURE` (SHRIMP), `SEND_DATA` (COMMs) and `RECEIVE_DATA` (COMMs) commands then one cannot individually request which message should be resend. This may not be a problem. For the `TAKE_PICTURE` (SHRIMP) command this might mean that one or more pixels are not coloured correctly. Depending on the requirements of the Mission, this consequence could be acceptable. For both the `SEND_DATA` (COMMs) and `RECEIVE_DATA` (COMMs) commands this might be more problematic. COMMs could verify data integrity and immediately send the `RESEND` command. If data is send continuously and there is no room for COMMs to send intermediate `RESEND` commands then proposal 4 might be more practical. Proposal 3 is visualised in Figure 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
START	SUBSYSTEM		COMMAND		LENGTH		DATA								CRC		STOP

Figure 4: Support for variable-length `DATA` by introducing a `LENGTH` field

4.4 Proposal 4: Variable `DATA` length with sequence number `INDEX`

The fourth proposal tackles the issue of splitting up data and request a certain frame to be resend if necessary by labeling an index number in the message. The length of each frame could still have a variable length size using the `LENGTH` field. This message format is visualised in Figure 5.

0	1	2	3	4	5	6	7	8	9	...	...	...	...	$N-1$	N	$N+1$	$N+2$	$N+3$
STRT	SUBSYSTEM		COMMAND		IDX	LENGTH		DATA								CRC		STOP

Figure 5: Request a retransmit of a certain frame is now supported by adding a `INDEX` field

References

[1] *HT-MD-200 Motor Driver*, Hyperion Technologies, June 2019, rev 1.0.

[2] *HT-BMS Preliminary Interface Document*, Hyperion Technologies, February 2022, rev 5.

[3] J. Axelson, *Serial Port Complete - Second Edition.* 5310 Chinook Ln., Madison: Lakeview Research LLC, 2007.