Bus Manager

Lunar Zebro

Software Department

TU Delft

Bus Manager

Lunar Zebro

by

Software Department

in partial fulfilment of the requirements for documentation for:
TU Delft, The Netherlands

Project Director:	prof. dr. ir. C. J. M. Verhoeven,	TU Delft
System Engineer:	Pranav Gurumallappa,	TU Delft
Operations Manager:	Maneesh Verma,	TU Delft
Head of Software:	dr. A. Noroozi,	TU Delft
Vice Head of Software:	Y. Klaassens,	TU Delft

TU Delft

Contents

1 Introduction

Because RS485 is a half-duplex protocol, software on both ends of the physical data line should follow the same set of rules defined in the protocol and Message Format (see Chapter 4). The Bus Manager is responsible for transferring commands and data between subsystems while adhering to this format. This implies checking for CRC errors and sending a retransmission request or NACK when it needs to do so. The Bus Manager acts between the Application Layer and the Physical Layer of the OSI model.
Figure 1.1 shows that there are 5 different RS485 buses inside the rover. The Bus Manager also prevents any race condition on its assigned bus and makes sure that multiple apps cannot communicate with their subsystem at the same time.

Not sure about this yet

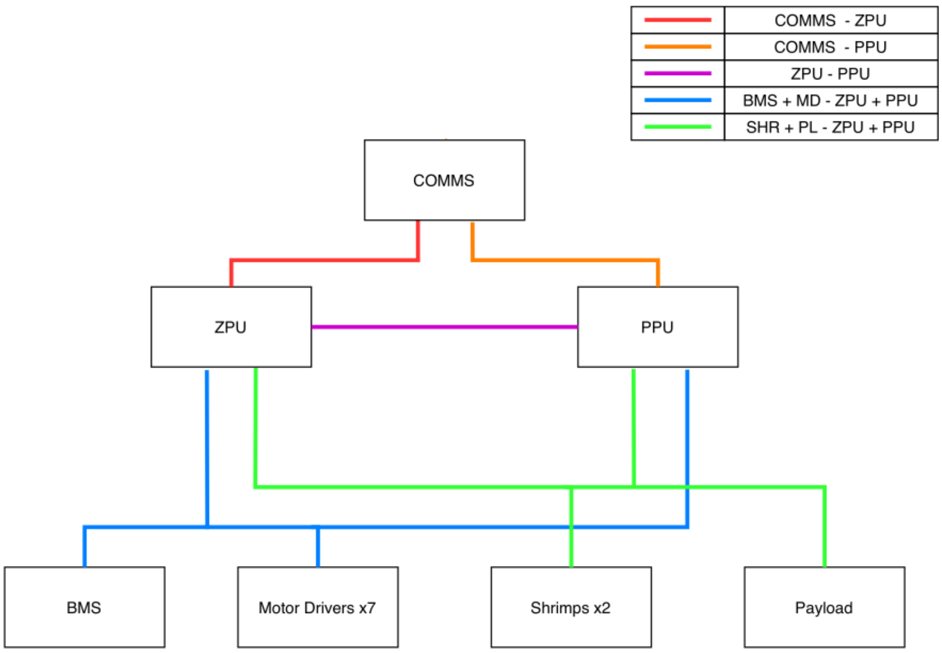

Figure 1.1: Schematic view of the physical RS485 busses layout

It is a bit awkward and impractical to uniquely identify RS485 buses based on the colours used in Figure 1.1. This is why we map the colours of Figure 1.1 to a unique bus name and number. Table 1.1 shows this mapping. From now on in this document as well as in code, we will refer to bus names rather than bus colours.

Bus name	Bus colour
Bus0	
Bus1	
Bus2	
Bus3	
Bus4	

Table 1.1: Bus legend mapping bus names to colours

1.1. Background information and History

1.1.1. First version (1.0)

Bus Manager is a concept that was first introduced in February 2021 as part of Max' thesis [1]. Between February 2021 and August of 2021, the foundational layer of Bus Manager was laid which includes, but is not limited to, the Message Format. Unit tests have shown that this first revision of Bus Manager used to work.

1.1.2. Second version (2.0)

In the second revision of Bus Manager, a few modifications have been made. One of these changes is, for instance, that the Bus Manager won't send an initial message to request if an x amount of bytes are available for the receiver to receive. Instead, it will just send the command which includes the length as part of the Message Format.

Clear up this sentence

The second revision also uses a semaphore to synchronise bus access between bus managers, because the idea was for each subsystem app to have its own bus manager.

1.1.3. Third version (2.1)

The third version of the Bus Manager uses a more centralised approach. The Bus Managers live on the OBC in their own app, which coordinates bus access for all buses. Apps need to communicate with the Bus Managers via inter-thread communication. This avoids the need for using shared semaphores and decreases the dependencies between the subsystem apps. For further information about the design, see Chapter 2.

1.2. List of features of Bus Manager 2.1

Now that the reader is aware of the overall functionality of the Bus Manager, we can list a set of features that characterises Bus Manager 2.1 (subject to change).

- **Instances**. Each bus has a separate instance of the Bus Manager that manages the access to that bus. Avoids the need for a more complex solution for all buses.
- **Timeout**. When a Bus Manager sends any type of message it waits for a response. A configurable timeout limit can be set to signal an error if the response is not received within that limit.
- **Retries**. When too many timeouts have happened and **<tbd>** retries are exceeded, then the application layer of this particular Bus Manager is informed about the failure.
- **Portability**. Because of the extensive use of Bus Manager both on the masters (OBC, PPU) and slaves (subsystems), it is of extra interest to design and implement Bus Mananger in such a way that it is easy to port over to subsystems which run different hardware architectures. Reimplementing it means that subsystem designers need to be fully aware of the protocol- and Message Format and this is a source of error and requires extra testing and validating. Bus Manager should be easily portable and be abstract in nature.
- **Sustainability**. Bus Manager shall comply to the Lunar Zebro Software guidelines. These include, but are not limited to: the code style guide and GoogleTest Unit Tests. The Git repository follows the same skeleton as all the other software modules.

2
Design

2.1. General design

The high-level functionality of the bus manager is displayed in Table 2.1.

Code	Function
BM_F1	Configure RS-485 buses
BM_F2	Manage access to RS-485 buses Not sure
BM_F3	Translate subsystem commands to serial bus messages and vice versa using the message protocol
BM_F4	Check correctness of the received messages

Table 2.1: High-level functions of the bus manager.

When the rover starts up, the bus managers configure and initialise the physical buses once (**BM_F1**). The programmer can use the public interfaces to send messages to the bus and receive messages from the bus (BM_F3). The bus manager checks if all received messages are correct and if they are not, it retries or signals an error (BM_F4).

2.2. Detailed design

This section describes the design decisions in more detail.

2.2.1. Initialisation

There are a total of 5 bus manager instances on the OBC, one for each bus. In addition, each subsystem has its own bus manager instance running in its firmware. Each instance is initialised only once. This initialisation configures the serial bus using the correct settings. This configuration can be different for different devices. The bus manager abstracts away from this configuration so the programmer needs to implement this lower-level functionality!

2.2.2. Interfacing (read/write)

A diagram of the steps involved in the interfacing process can be found in Figure 2.1. The bus managers on the OBC exclusively use this functionality. The subsystems also use it, but only after first reading from the bus (see Section 2.2.3).
After writing each packet, the bus manager reads the responses. These can be either an ACK or a REPLY. For each REPLY received, the bus manager sends an ACK response.

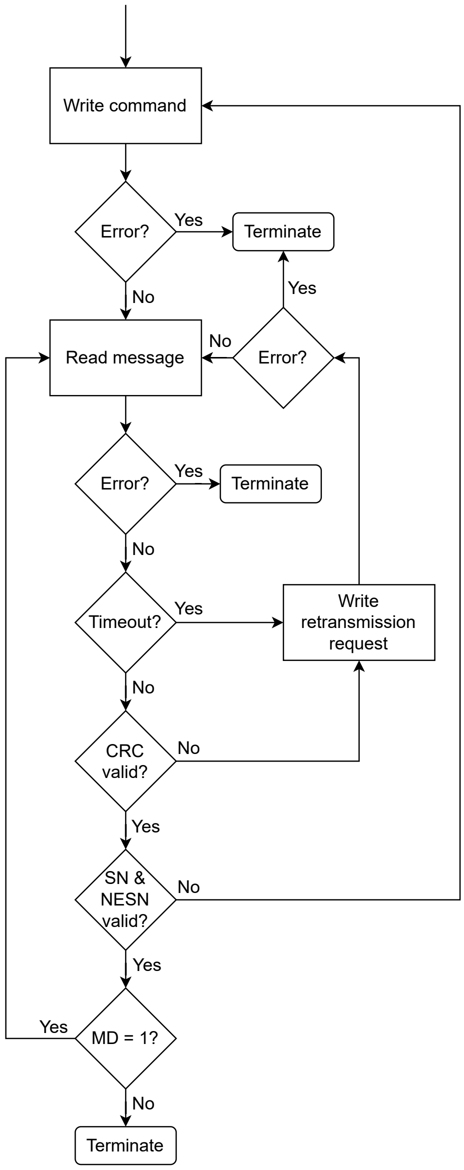

Figure 2.1: High-level design diagram for serial writing using the bus manager

1. **Write** the command to the serial bus.
2. **Check for errors** after writing.
3. **Read** the next incoming message.
4. **Check for errors** after reading:
   - Check for valid CRC
   - Check for read timeout
5. **Send a retransmission request** and go to step 3 if an error occurred.
6. **Check SN/NESN** message fields.
7. **Retransmit the command** and go to step 3 if SN and/or NESN are invalid.
8. **Send an ACK** packet to the serial bus.
9. Go to step 3 if the MD field is set to 1.

2.2.3. Subsystem interfacing

The subsystem interfaces a bit differently, since it actively needs to listen for commands from the OBC or PPU. So it first reads from the bus, then uses the interfacing procedure described in Section 2.2.2.

3 Implementation

The implementation consists of two classes.

- Packet_t class: represents a message protocol packet.
- Bus_manager_t class: represents a bus manager for a single bus.

The bus manager class uses the packet class extensively to form packets for writing to the serial bus. Both classes are discussed below in more detail.

3.1. Packet_t class

The packet class has the same instance variables as the fields of the messages that we use in the messaging protocol:

- preamble: preamble field of the packet.
- source: source field of the packet.
- destination: source field of the packet.
- pdu: PDU field of the packet. Implemented as a struct with payload and header fields.
- crc: CRC field of the packet.

One of the reasons to have this class is that packets can be easily converted to and from raw bytes using a member function. The same is true for calculating the CRC of the packet.

3.2. Bus_manager_t class

3.2.1. Members

The bus manager has three public members:

- init: implements the initialisation design from Section 2.2.1.
- interface: implements the interfacing design from Section 2.2.2.
- read_from_bus: used by subsystems to read from the bus before interfacing.

There are also some members that should be used to **interface to the specific serial bus code of the platform** (to be implemented by the programmer):

- configure_serial: should contain platform-specific code to configure the serial bus.
- write_to_serial: should contain platform-specific code to write to the serial bus.
- read_from_serial: should contain platform-specific code to read from the serial bus.

3.2.2. Errors

The bus manager class uses a `Result_t` struct to indicate errors. It has an error code and error details. The errors should be forwarded to the app that sent the command. In Table 3.1 we can see when which error occurs and what the details are.

Situation	**Error code**	**Error details**	**Fwd to app?**
Bus manager is already initialised	ALREADY_INITIALISED	–	Yes
Serial configuration failed	SERIAL_CONFIGURE_ERROR	configure error code	Yes
Serial write failed	SERIAL_WRITE_ERROR	write error code	Yes
Serial read failed	SERIAL_READ_ERROR	read error code	Yes
Serial read timeout	SERIAL_TIMEOUT_ERROR	–	No
CRC invalid	INVALID_CRC	crc	No
Too many retransmissions	RETRY_LIMIT_REACHED	NACK	Yes
Too many retransmission requests	RETRY_LIMIT_REACHED	previous error code	Yes
MD=1 but no more message expected	UNEXPECTED_MULTIPLE_DATA	–	Yes
NACK received	NACK	–	No
Bus manager not initialised when interfacing	NOT_YET_INITIALISED	–	Yes
Received packet destination not equal to subsystem	INVALID_PACKET_DESTINATION	previous error code	Yes
First received packet invalid (subsystem)	UNKNOWN_PACKET_DESTINATION	–	Yes

Table 3.1: Situations where errors occur.

4
Message Format

The message format described in this section is derived from the Bluetooth Low Energy (BLE) protocol but customised to the specific needs for Lunar Zebro and its subsystems. The level of abstraction allows this message format to be used with all subsystems onboard. Lets now describe from a top-bottom approach how this message format looks like. Figure 4.1 shows how the message format looks like. Each message starts with a preset preamble (PRMBL) that is equal to $0x3A_{16}$. The source (SRC) field indicates the source address of the message whereas the destination (DST) field indicates the destination address of the message. The Packet Data Unit (PDU) (PDU) field is what we will describe in greater detail in the next paragraph. The CRC (CRC) field holds the CRC calculation over all the previous fields.

0	1	2	3	4	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	$N-1$	N	$N+1$	$N+2$	$N+3$
PRMBL	SRC	DST	PDU												CRC		

Figure 4.1: Generic message format

The PDU (PDU) field itself is split up in a header (HEADER) field consisting of two bytes which holds data-link layer information and a payload (PAYLOAD) field. This payload field is the data of interest being transferred. The format of the (PAYLOAD) field depends on the message type which is defined in the (HEADER) field. Figure 4.2 shows the message format of the (PDU) field.

0	1	2	3	4	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	$N-2$	$N-1$	N
HEADER		PAYLOAD															

Figure 4.2: The PDU consists of a header- and a payload part

The (HEADER) field, which are the first two bytes of the (PDU), contain information that the data-link layer use. Figure 4.3 shows the bitrepresentation of the (HEADER) field. Since the (HEADER) field is 2 bytes in size means that there are 16 bits in this field.

0	1	2	3	4	5	6	7
TYPE		NESN	SN	MD	RFU		

8	9	10	11	12	13	14	15
LENGTH							

Figure 4.3: Header field of the PDU in detail

Table 4.1 list the description of each bitfield.

Field	Bit(s)	Decription
TYPE	[0..1]	Defines the type of the message 00_2: Command 01_2: Retransmission 10_2: Reply 11_2: N(ack)
NESN	2	Next Expected Sequence Number See “Subsystem Message Format Proposal” on Teams
SN	3	Sequence Number See “Subsystem Message Format Proposal” on Teams
MD	4	More Data 0_2: No more data is coming. Last message of the device 1_2: More data is coming. This is not the last message
RFU	[5..7]	Reserved for Future Use
LENGTH	[8..15]	Length l of the payload section $0 \leq l \leq 255$ are valid values

Table 4.1: Bitfield description of the HEADER field

As mentioned earlier in this Chapter, the (PAYLOAD) section depends on the type of the message. the following few sections will list how the (PAYLOAD) field looks like for each type of message.

4.1. Command-type PDU

A command is a message type that always goes from the host (OBC, PPU) to the controller (other subsystems). It tells the subsystem to perform a certain functionality. A command may require an argument or additional data. This is where the DATA/PARAM field comes in.

0	1	2	3	4	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	⋯	$N-2$	$N-1$	N
COMMAND		DATA/PARAM															

Figure 4.4: The PAYLOAD section of the PDU when the message type is a command

Table 4.2 lists the descriptions of each bit field regarding the (PAYLOAD) field.

Field	Byte(s)	Decription
COMMAND	[0..1]	Defines which command of the subsystem should be executed. The “Subsystem Message Format Proposal” document hosts a list of commands for each subsystem.
DATA/PARAM	[2..N]	Additional data or parameters concerning the command $0 \leq l \leq 254$ is a valid range and it depends on each command of the subsystem. See the subsystem specific docuementation to see how each command devides the (DATA/PARAM) field

Table 4.2: Bitfield description of the PAYLOAD field

4.2. Retransmission-type PDU

When a host or controller calculates the CRC over the PDU section and detects a mismatch, it can request a retransmission by sending a retransmission-type message. The **NESN** and **SN** have to reflect a retransmission request as well. More about these bits in “Subsystem Message Format Proposal” on Teams. There is no need to supply any data for a retransmission request. Hence why **LENGTH** should be equal to zero and the (PAYLOAD) section is empty.

4.3. Reply-type PDU

A reply-type message is send by the controller to the host as a response of a command-type message. Not all commands will return something to the host. This means that the reply-type message is optional and depends on the subsystem- and command. Figure 4.5 shows the (PAYLOAD) field when the message is a reply-type. The vast majority of commands return data of a small size. Usually in the order of `uint8_t` and `uint32_t`. N can be up to 255 bytes which is enough for almost all commands.

0	1	2	3	4	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$N-2$	$N-1$	N
DATA																	

Figure 4.5: The PAYLOAD section of the PDU when the message type is a reply

5 Appendix

5.1. Known bugs and issues

This section will list all known bugs and issues encountered during the development of the firmware.

Bibliography

[1] Max Kostic. Propagating commands from a ground station through a rover. Bachelor Thesis, August 2021. Lunar Zebro.