Embever Serial Protocol Documentation

Embever GmbH

Mar 05, 2021

CONTENTS:

CHAPTER

ONE

HARDWARE SPECIFICATION

1.1 Hardware architecture

The hardware achitecture is shown in Fig. 1.1. The Embever microcontroller (EM) controls two groups of devices. The EM has exclusive access to a group of private devices, whereas the group of public devices is connected via an I2C bus, from which the EM is the master by default. An application microcontroller (AM) interacts exclusively with the EM via the Embever Serial Protocol (ESP). Optionally, the AM can also request to be the exclusive master of the I2C bus to directly interact with any public device.

Note: At the moment, a multi-master I2C architecture is not supported. If the AM wants to act as I2C master, it needs to request the EM (via ESP) to stop acting as a I2C master to avoid any problems.

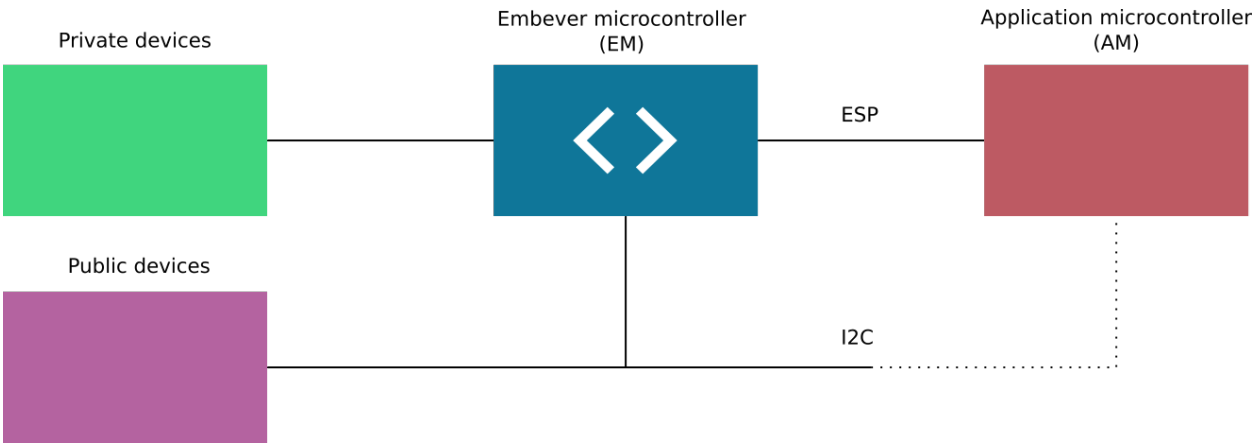

Fig. 1.1: Hardware architecture

1.2 ESP hardware interface

The interface of the ESP is shown in Fig. 1.2.

It consist of a I2C bus and signaling lines. The AM is the I2C bus master, and the EM is a slave. The EM will respond to the AM on I2C address *0x35*. For completeness, besides the *ESP_** lines, the RESET and WAKE lines are included in this specification. These two lines are standard features provided by the manufacturer of the EM. The AM need these lines to be able to interact with the EM.

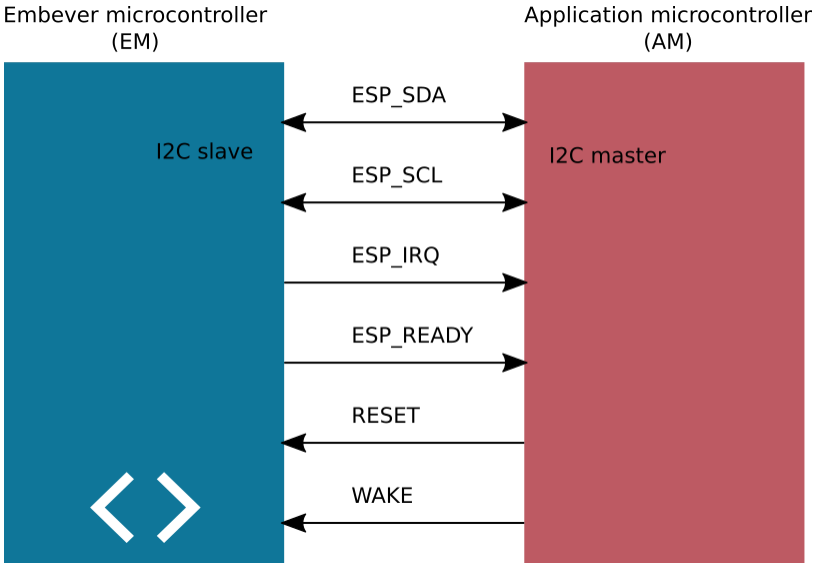

Fig. 1.2: Hardware interface

1.2.1 ESP_SDA line

This is the I2C data line.

1.2.2 ESP_SCL line

This is the I2C clock line. The AM I2C master clock rate can be as high as 1 MHz. The EM may extend the I2C clock to delay the I2C master if it needs more time to perform an operation.

1.2.3 ESP_IRQ line

This is an output pin of the EM used for indicating to the AM there is data waiting to be retrieved. This line is asserted using a negative logic: a low state on this line indicates to the AM that the EM needs servicing.

1.2.4 ESP_READY line

This is an output pin of the EM used for indicating to the AM that the ESP interface is fully functional. This line is asserted using a negative logic: The EM set this line to low as soon as the ESP is ready to process commands. Otherwise, if the ESP is not ready, this line set to high.

1.2.5 RESET line

This is the EM's default reset line. If the ESP becomes unresponsive, the AM can perform a hardware reset using this line.

1.2.6 WAKE line

This is the EM's default wake up line. The AM can wake up the EM from deep powerdown mode using this line.

1.3 Mode transaction protocol

1.3.1 Deep power down mode

The EM is typically in deep power down mode. Depending on the microcontroller, when in deep power down mode, the ESP_READY and ESP_IRQ lines may be floating. It is recommended to force these lines to be high in such cases (e.g. using a 100 kOhm pull-up resistor).

EM goes to sleep according to its own rythm. It wakes up automatically at configurable intervals. The AM can wake up the EM using the WAKE line, but not viceversa. The EM keeps its own internal timer, which is reset if woken up by AM. Otherwise, if the timer expires, EM tries to connect to the cloud on its own behalf. This is a safety mechanism, to avoid a bricked AM (e.g. bad program) to break the whole module.

Note: In case of using an AM that supports NXP's ISP interface: The EM must be able to reset the AM to perform an AM firmware update via ISP. During normal operation the EM should not reset the AM.

1.3.2 Ready mode

After waking up, the EM puts the ESP_READY line to LOW as soon as the interface ESP is fully configured (i.e. it is ready to process commands received from the AM). Before going back to deep power down mode, the ESP_READY line is set back to high. This process is shown in Fig. 1.3.

AM can check for the health status using ESP_CMD_CHECK_OK command. If no response is received, AM should reset the EM if the ESP_READY line is LOW, or wake up the EM otherwise. A typical transaction starts with the AM waking up the EM from deep power down mode. After the ESP_READY goes low, the AM can start sending command packets.

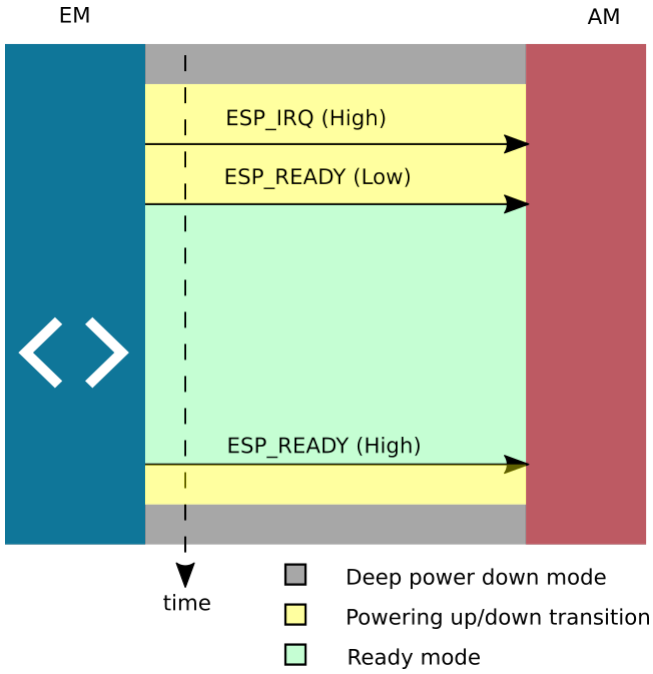

Fig. 1.3: Transition between modes

1.3.3 Commands Data flow

When using I2C, a command is given to the EM by an I2C write transaction. The entire command packet is accepted by the EM and then processing starts. There are two types of commands: commands with *immediate* response, and commands with *delayed* response.

The data flow for both cases is as follows (refer to Fig. 1.4):

- commands with immediate response: After receiving a *command* packet, the EM immediately sends a *response* packet. the ESQ_IRQ line is not asserted (it stays high).
- commands with delayed response: an immediate *acknowledge* packet is returned by the EM after receiving a *command* packet. When the actual *response* packet is ready, it is signaled by setting the ESP_IRQ low. At this point, the AM can get the response packet by sending a *ESP_CMD_READ_DELAYED_RESP* command (which

is itself a commmand with immediate response). The EM will hold the ESP_IRQ pin asserted (low) until the AM receives the entire contents of the response packet.

Certain commands return a response packet with a length unknown to the AM, but fixed and known to the EM. Those response packets are structured in two clearly defined parts:

- The first part of the response packet has a known length and structure. This part contains, among other things, information about the length of the second part of the response packet.
- The second part consists of a fixed amount of bytes.

The AM can read the this type of response packet at once (a single read transaction get all bytes), or as two separate read transactions e.g. first read just enough bytes to get the first part and determine the length of the second part, and then on a second read get the second part using the (now known) length.

Although in practice the AM would usually read the response using one or two read transactions, in theory the AM can read any response in as many separate read transactions as needed. For all responses, the EM must be able to send the whole message in chunks, in as many read transactions as the AM requires. The ESQ_IRQ line is only deasserted, when *all* bytes in the current response have been sent to the AM.

For a detailed explanation of available commands (including *ESP_CMD_READ_DELAYED_RESP*) see *Embever Serial Protocol Commands*.

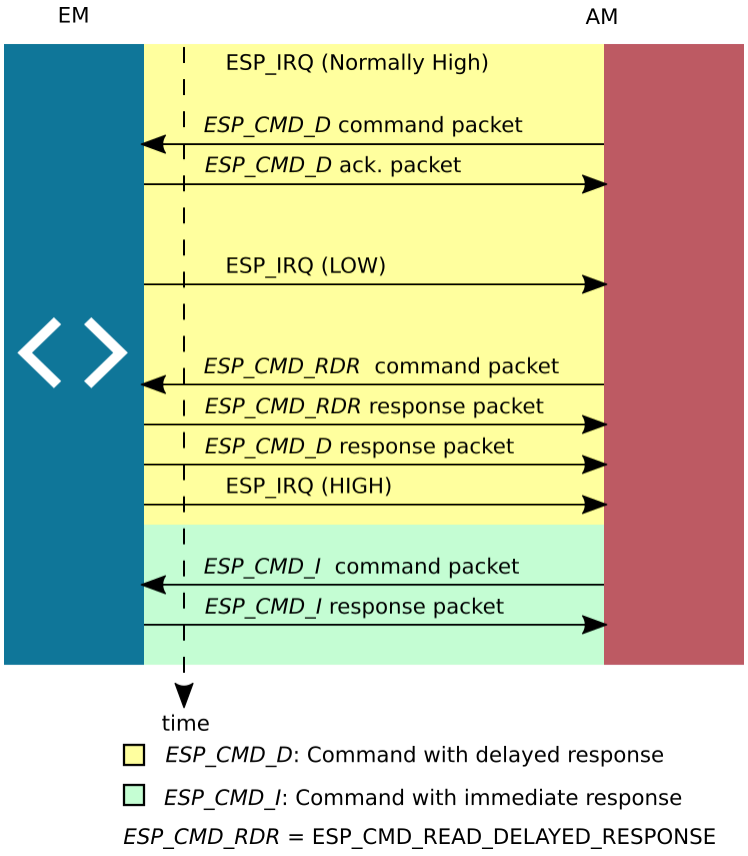

Fig. 1.4: Basic data flow

It is possible to execute other commands while a command with delayed response is in progress. This is exemplified in Fig. 1.5. In Ready mode, if there are no responses waiting to be read, the EM keeps the ESP_IRQ line high. First, the EM receives a command with a delayed response. Before the response packet for this command is ready, the EM receives a command with an immediate response. This has no effect in the ESP_IRQ line. When the delayed response for the original command is ready (*ESP_CMD_D* in Fig. 1.5), the EM asserts the ESP_IRQ line. Afterwards, the

EM receives another command with an immediate response. This again has no effect in the ESP_IRQ line (it stays low). Eventually, the AM decides to read the delayed response, and sends a *ESP_CMD_READ_DELAYED_RESP* command, which gets an immediate response, follow by the response packet for *ESP_CMD_D*. It is up to this point when the ESP_IQR line is deasserted.

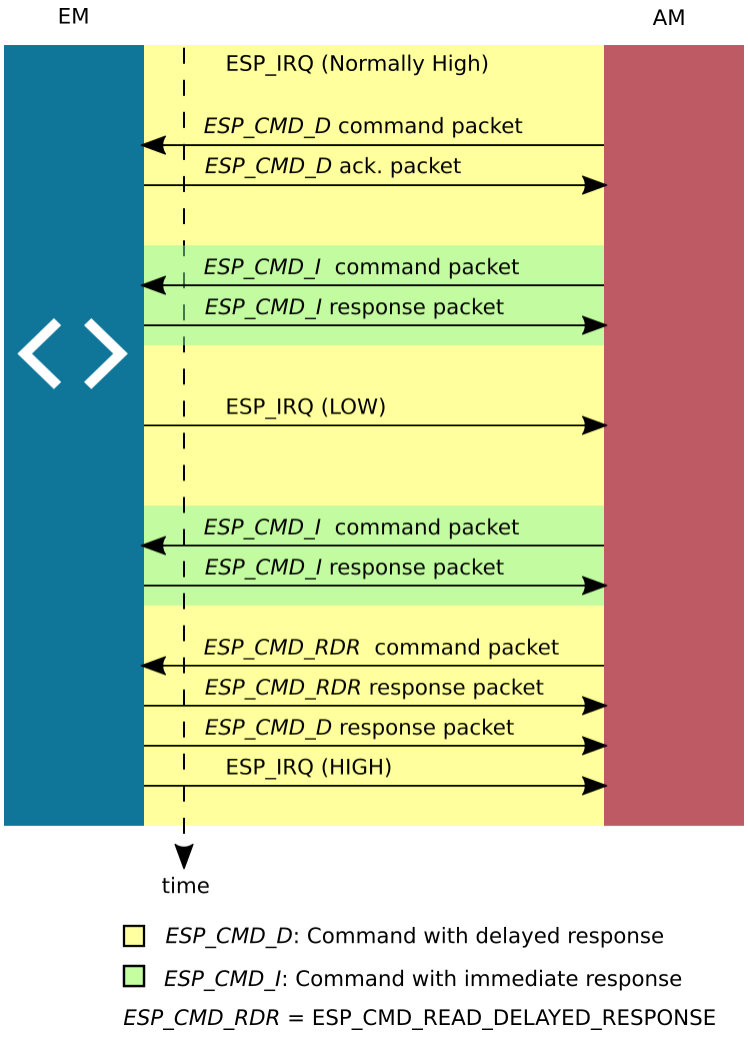

Fig. 1.5: Data flow with nested commands

Note: In theory, it is also possible to have several nesting levels involving many commands with delayed responses being processes simultaneously. However, this probably requires a multithreading operating system. We are not there yet, maybe in the future. ;)

Warning: How far the support for command nesting goes is implementation dependent. Please make sure, as developer of the ESP slave (the EM), to document the limitations in this regard of your implementation.

1.3.4 Interaction with AM

AM goes to sleep according to its own rythm. It wakes up automatically at configurable intervals. Both, AM and EM should each have its own watchdog. AM can also act as watchdog for EM, but not viceversa.

1.4 Specification for LPC845

For more information on this microcontroller see [dslpc84x], and [umlpc84x].

1.4.1 ESP interface

Use I2C0, which supports up to 1 MHz SCL frequency.

Todo: Add figure showing this connections

ESP_SDA = PIO0_11/I2C0_SDA

ESP_SCL = PIO0_10/I2C0_SCL

ESP_IRQ = PIO0_27

ESP_READY = PIO0_26

1.4.2 Application update interface

If the application microcontroller is also an LPC845, the following applies. Use I2C0 interface to connect EM and AM (this is already accomplished by using the I2C interface). EM is in this case the I2C master and AM the I2C slave. The ESP_IRQ line has the same use as before, but the direction is reversed, i.e. it is an input in EM, and an output in the AM. We need to connect AM's PIO0_5/RESET line and PIO0_12 (ISP entry pin) to EM PIO0_25 and PIO0_24 (pins may change).

Todo: For future reference: the EM should have a way to know when it is (not) okay to perform an AM FW update. For example: the AM could notify when is it okay (or not okay) to perform a firmware update (e.g. through a pin). Or the EM should figure it out by itself?

1.4.3 Example using two LPC845

Specify the connections in case of both microcontroller (the AM and the EM) being a LPC845.

CHAPTER

TWO

EMBEVER SERIAL PROTOCOL COMMANDS

Note: This chapter is a work in progress

This chapter assumes some familiarity with the Embever IoT Protocol.

2.1 ESP mode commands, data, and responses

All of the supported commands, associated structures and data formats for those commands, and responses are explained in this section. For an explanation of how this commands are executed see *Mode transaction protocol*.

For all tables, the *Size* column is given in number of bytes.

The *length* field informs how many more bytes remain till the end of the packet (not including the *length* field itself).

For the *data* field (in command or response packets) the following applies:

data is encoded in MessagePack format. It must be at least 1 byte long after encoding. That is, on success, the *data* field must be at least one byte long, which together with the CRC32 field makes for *length>4*.

For example, *data* can consist of an empty list *[]* (*0x90* in MessagePack encoding), or an an empty object *{}* (*0x80* in MessagePack encoding).

In this section, the *GET* keyword refer to commands that retrieve (download) data from the cloud application, whereas the *PUT* keyword refer to commands that send (upload) data to the cloud application.

The fields *length*, *error* and *checksum* use little-endian format.

Todo: Define the memory limitation of commands with variable data length.

2.1.1 The Flags field

Some commands have a field name *flags*. Each bit (or a combination of bits) have additional information about how the command should be executed.

Table 2.1: Command Flags

Field	Offset (bits)	Size (bits)	Description
ignoreCRC	0x00	0x01	**0: do CRC check,** 1: ignore CRC field (set to checksum=0)
QoS	0x01	0x02	**Quality of services levels:** 0: do not confirm this command, 1: send a confirmation packet after execution of this command, 2: Reserved 3: Reserved
Reserved	0x03	0x05	Reserved bits

QoS: if the master sends a command that has the *flags* field, the slave must send a confirmation packet (or not). If the master receives a response with the *flags* field in it, then the master must send a confirmation packet to the slave. Use the command *ESP_CMD_REQUEST_PERFORMED* to confirm that the request in a command with QoS=1 was succesfully performed.

Todo: Explain the QoS levels a bit better

2.1.2 ESP_CMD_READ_DELAYED_RESP

Command with immediate response.

Used to get the response of a command with delayed response. Please refer to *Mode transaction protocol*, in particular see Fig. 1.4.

If ESP_IRQ=HIGH, this can be used to check if the EM is still alive (not hang, as a watchdog function).

Table 2.2: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA1	Read Delayed Response

Table 2.3: Response packet (success, no responses)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA1	Process command identifier
length	0x02	0x02	0x0000	Set to 0 if no response is waiting

Table 2.4: Response packet (success, responses)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA1	Process command identifier
length	0x02	0x02	respsz	Length of response
response	0x04	respsz		The delayed response packet

2.1.3 ESP_CMD_GET_ACTIONS

Command with delayed response.

Table 2.5: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA2	Get Actions

Table 2.6: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xA2	Acknowledge reception of command

Table 2.7: Response packet (success)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA2	Process command identifier
length	0x02	0x02	datasz+5	The combined length of data, checksum and flags
data	0x04	datasz		Data corresponding to actions
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field
flags	datasz+5	0x01		Command flags

Table 2.8: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA2	Process command identifier
length	0x03	0x02	0x0004	On error, this field is set to 4
error	0x05	0x04		Error code

data contains a list of *n* actions, which are application specific and not part of the ESP. Each action is defined by an object (key/value pair) containing the action_type, and its corresponding action_data. action_type is a string (the key), and action_data can be object.

Any action_type must be unique in the list/object.

Todo: Clarify who guarantees this, and what happens if a key appears more than once.

For example, a valid *data* field could look like (show in json format for clarity):

```
[["getFile", 1234, {"file": "2.0/8234765234765/bjL4K"}]]
```

2.1.4 ESP_CMD_PUT_RESULTS

Command with delayed response.

Table 2.9: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA3	Put results
flags	0x01	0x01		Command flags
length	0x02	0x02	datasz+4	The combined length of data and checksum
data	0x04	datasz		Data corresponding to results
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field

Table 2.10: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xA3	Acknowledge reception of command

Table 2.11: Response packet (success)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA3	Process command identifier
length	0x02	0x02	0x0000	On success this field is set to 0

Table 2.12: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA3	Process command identifier
length	0x02	0x02	0x04	On error, this field is set to 4
error	0x04	0x04		Error code

2.1.5 ESP_CMD_PUT_EVENTS

Command with delayed response.

Table 2.13: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA4	Put events
flags	0x01	0x01		Command flags
length	0x02	0x02	datasz+4	The combined length of data and checksum
data	0x04	datasz		Data corresponding to events
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field

Table 2.14: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xA4	Acknowledge reception of command

Table 2.15: Response packet (success)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA4	Process command identifier
length	0x02	0x02	0x0000	On success this field is set to 0

Table 2.16: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA4	Process command identifier
length	0x02	0x02	0x04	On error, this field is set to 4
error	0x04	0x04		Error code

2.1.6 ESP_CMD_PERFORM_ACTIONS

Command with delayed response.

This command is issued by the AM to request the EM to perform certain actions on the AM behalf. There are only a limited number of actions types that are supported by this command. The supported action types are implementation dependent, and are not part of the ESP.

The EM will reply with a response packet containing the results of the actions requested.

Issue this command after receiving any supported action type on the *data* field of the `ESP_CMD_GET_ACTIONS` response packet. The *data* field in the command packet contains a list with the received *action_list*'s whose type is any of the action types supported by this command. (see Embever IoT Protocol for further details).

If the AM requests to perform an action not supported by the EM, the EM returns `ESP_ERR_INVALID_ACTION_TYPE` in its delayed response.

Table 2.17: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA6	Perform Actions
flags	0x01	0x01		Command flags
length	0x02	0x02	datasz+4	The combined length of data and checksum
data	0x04	datasz		Actions data object
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field

Table 2.18: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xA6	Acknowledge reception of command

Table 2.19: Response packet (success)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA6	Process command identifier
length	0x02	0x02	datasz+5	The combined length of data, checksum and flags
data	0x04	datasz		Data corresponding to results
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field
flags	datasz+5	0x01		Command flags

Table 2.20: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA6	Process command identifier
length	0x02	0x02	0x0004	On error, this field is set to 4
error	0x04	0x04		Error code

For example, if the response of a `ESP_CMD_GET_ACTIONS` request contains the following list of actions list (in messagePack format, but for clarity we show it in json format):

```
[["dummy", 1233, {}], ["getFile", 1234, {"file": "2.0/8234765234765/bjL4K"}]]
```

If only the action type `"getFile"` is supported (and `"dummy"` is not), the *data* field in the `ESP_CMD_PERFORM_ACTIONS` command packet would contain the following list of actions list:

```
[["getFile", 1234, {"file": "2.0/8234765234765/bjL4K"}]]
```

If the AM includes the following instead (i.e. an action type not supported by the EM is requested)

```
[["dummy", 1233, {}]]
```

The EM application code should return an error `ESP_ERR_INVALID_ACTION_TYPE` in the response packet.

2.1.7 ESP_CMD_READ_LOCAL_FILE

Command with delayed response.

Read a file stored by the EM.

Table 2.21: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xA7	Read local file
flags	0x01	0x01		Command flags
length	0x02	0x02	datasz+4	The combined length of data and checksum
data	0x04	datasz		getFile data object
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field

Table 2.22: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xA7	Acknowledge reception of command

Table 2.23: Response packet (success)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA7	Process command identifier
length	0x02	0x02	datasz+5	The combined length of data, checksum and flags
data	0x04	datasz		Data read from file
checksum	datasz+4	0x04	CRC32	CRC32 of data excluding this field
flags	datasz+5	0x01		Command flags

Table 2.24: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01	0xA7	Process command identifier
length	0x02	0x02	0x0004	On error, this field is set to 4
error	0x04	0x04		Error code

To read a file, the following ordered list (in messagePack format) is used as the *data* field in the *ESP_CMD_READ_LOCAL_FILE* command packet.

Table 2.25: List for data field of command packet

Field	Position	Type	Description
offset	0	Integer	The offset in bytes from beginning of file
size	1	Integer	The number of bytes to be read starting from offset
path	2	String	Optional: The path to the file to be read

Note: Offset is a 32-bit integer number. This allows to read files larger than 64kB.

The file to be read is identified by its *path* (an optional element on the list). If path is not present in the list, the EM uses the default file path (implementation dependent, and not part of the ESP). Files smaller than the EM ESP data buffer can be read on one single call to this command. For larger files, several calls are required. This commands expects an *offset* and *size* fields. The *offset* indicates from which position the file should be read. The beginning of a file is indicated by *offset=0*. The field *size* indicates how many bytes are to be read, starting from *offset*. The maximum value for *size* is the size ESP data buffer (link to definition?).

If *size* is larger than the available data in the file, the length of *data* in the response packet will be that of the available data (i.e. no error is generated). This happens when *offset* + *size* is beyond the end of the file. If there is no data to read (e.g. *offset* is beyond the end of the file, or *size=0*, or the given *path* does not exist) this command returns the `ESP_ERR_INVALID_FILE_REQUEST` in the error packet.

Todo: Future: Read a full file, with automatic data flow control. The AM specifies how many bytes (automatically increasing the file pointer).

2.1.8 ESP_CMD_REQUEST_GNSS_LOCATION

Todo: find a better name for this function (use other verb instead of REQUEST, like UPDATE, but not GET)

GNSS

Todo: Define input parameters (like min. accuracy), and the output format.

Warning: This command is supported only if GNSS hardware is available.

Todo: set an error code if no GNSS hardware available.

2.1.9 ESP_CMD_REQUEST_PERFORMED

Table 2.26: Command packet

Field	Offset	Size	Value	Description
command	0x00	0x01	0xAF	Confirmation that the request was executed

Table 2.27: Acknowledge packet

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x56	Start of acknowledge packet identifier
ack	0x01	0x01	0xAF	Acknowledge reception of command

2.2 To be implemented later

2.2.1 ESP_CMD_DEL_LOCAL_FILE

2.2.2 ESP_CMD_ACK_ACTIONS

Acknowledge executions of received actions. (useful for QoS Level 2)

2.2.3 ESP_CMD_GET_CONNECTION_STATUS

2.2.4 ESP_CMD_RESET

2.2.5 ESP_CMD_PWROFF

2.2.6 ESP_CMD_REQUEST_I2CMASTER

The AM requests to act as exclusive I2C master.

2.2.7 ESP_CMD_RELEASE_I2CMASTER

The AM notifies to act as exclusive I2C master.

2.3 General error handling

If a command packet is received with an invalid identifier in the *command* field, the EM should immediatly send a error response packet. In the *command* field of the response packet, the same value received in the *command* field of the command packet should be written (i.e. an invalid value).

Table 2.28: Response packet (error)

Field	Offset	Size	Value	Description
sop	0x00	0x01	0x55	Start of response packet identifier
command	0x01	0x01		Received invalid command identifier
length	0x02	0x02	0x0004	On error, this field is set to 4
error	0x04	0x04	ESP_ERR_INVALID_CMD	Error code

2.3.1 Error codes

All *full error codes* are 4 bytes in length and follow the format *0xXXYYB3A5*, i.e, the two least significant bytes are always, *0xB3A5*, and the two most significant bytes, *0xXXYY*, represent the actual *error code*.

The following table summarizes the error codes (shown in big-endian format). Thus, if an error code is e.g. *0x3412*, the full error code in big endian is *0x3412B3A5*, which in little-endian would be *0xA5B31234*.

Thus, first byte sent over I2C is always *0xA5*, followed by *0xB3*. This is done so as a simple error detection measure, given that when an error is returned, the *length* field is always *0x0004* (it has a lot of consecutive zeros).

Todo: Define the most common error codes.

Table 2.29: Response packet (error)

Code	Name	Description
0x0101	ESP_ERR_INVALID_CMD_ID	Invalid command ID received in command packet
0x0102	ESP_ERR_INVALID_CMD_DATA	The received data in the command packet is invalid
0x0103	ESP_ERR_INVALID_ACTION_TYPE	The list of actions to perform contains an invalid action type (only for `ESP_CMD_PERFORM_ACTIONS`)
0x0104	ESP_ERR_INVALID_FILE_REQUEST	The requested file information is invalid (offset, size or path in `ESP_CMD_READ_LOCAL_FILE` data field are wrong)
0x01CD	ESP_ERR_UNKNOWN	A general purpose error code

CHAPTER

THREE

REFERENCES

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[dslpc84x] LPC84x 32-bit Arm Cortex-M0+ MCU - Data Sheet, Rev. 1.7 — 27 February 2018. PDF: LPC84x

[umlpc84x] LPC84x User manual, Rev. 1.6 — 8 December 2017. PDF: UM11029