deltarobot Message Format and Parsing Library

Specification v1.0

Table of Contents

Lucas Pires Camargo
EMB5613 – Projeto Integrado de Sistemas Embarcados
Prof. Diego Santos Greff

Introduction

This document defines the message formats used by deltarobot. They are to be used in the implementation of both the firmware and the accompanying software interface.

For more information on the nature of the project and the robot itself, please refer to the full report.

Purpose

The project aims for making the platform fully operational, which means giving the user means of controlling the platform position. This could be accomplished with an interface on the controller device itself, but we figure it is better (and more simply) done by use of a personal computer connected to the platform. This makes possible the use of an user interface with interactive 3D graphics, the creation of a software library for integration into other applications, path drawing, etc.

However, to accomplish this, there must be a standardized way for the controller to talk to a personal computer. Therefore, it was decided that a simple protocol would have to be defined.

Definitions

In this specification, “Controller” means the device that operates the motors of the parallel actuators, reads sensor inputs, and executes the control loop. “Interface”, “Interface software”, or “Host” refers to the computer that is communicating with the Controller, monitoring its status, and defining its goal positions.

An “Opcode” is be understood as “Operation code”. Different Opcodes describe the various functionalities of the protocol in compartmentalized commands.

About this specification

This document is organized as follows:

- Section “Connection and Transmission” gives an overview on the physical link layer.
- Section “Message Frame Structure” defines the basic structure of a command.
- Section “Opcodes” defines commands, their requirements and outcomes.
- Section “Implementation Notes” gives further details in regards to implementation and noteworthy behaviors.

Connection and Transmission

The Controller is to be connected to the Host by means of a serial port device. This device has to be configured as follows:

- Baud rate: 115200 baud
- Data bits: 8 bits
- Stop bits: 1 bit
- Parity bits: (none)
- Hardware flow control: (disabled)

Furthermore, a “serial port”, this can mean a physical, standard RS-232 connection, or a connection emulated in software or in hardware. A valid implementation is a USB connection through which a serial data stream is sent, and that the Host can recognize as such. This has to translate into a COM port or TTY device file (“/dev/ttyS?”, “/dev/ttyUSB?”) in the operating system of the Host, to which the Interface Software connects.

Communication is full-duplex, that is, there can be a transmission occurring from both sides simultaneously. Every valid piece of communication is define in terms of “message frames”. A message frame is a group of bytes with a defined start and end, means for verification, and a specific purpose. The next section goes into detail about the structure of a message frame and related definitions.

Message Frame Structure

Commands, status information, and control information are organized into frames. A frame is a validated sequence of bytes structured as follows:

Starter	Opcode	Data Length	Data	Checksum	Terminator
'$' (ASCII)	"What the frame is"	How many data bytes	Opcode-dependant	(See below)	':' (ASCII)
1 byte	1 byte	1 byte	(length – 5) bytes	1 byte	1 byte

- **Starter:** this marker (ASCII character '$') denotes the beginning of a transmission frame. Both the Controller and the Host scan the incoming stream of bytes, expecting to find this marker. When it is found, decoding of a frame starts.
- **Opcode:** "operation-code", a single byte from a predefined list. Individual opcodes are discussed in greater detail in a further section.
- **Data Length:** how many bytes of data the message contains. The value is usually tied to the opcode, but may vary (up to 255 bytes).
- **Data:** the data bytes of the message. Again, their meaning is opcode-specific.
- **Checksum:** a code that is calculated from the contents of the message. It is used to ascertain that the message has been received correctly. How the checksum is calculated is presented on the next section.
- **Terminator:** a single terminator byte (ASCII character ':') that indicates the end of a message.

Checksum Calculation

Checksums are calculated as a modular sum of all the bytes of the message, including starter and terminator. A modular sum is defined, in this context, as adding the bytes contents to an accumulator (initialized with zero), one after the other. Following every sum, the accumulator is set to its own value modulo 256.

This can be implemented by repeatedly summing the byte values to an unsigned byte variable.

Notes

- There is no message resend requests.
- Incomplete or corrupted packets are lost.
- If an ACK times out, the operation request has to be repeated, depending on the opcode.

Opcodes

This section describes each opcode in detail. However, for the purpose of an overview, a complete list is given beforehand. The opcodes are divided in three categories: “common” opcodes that can be sent in both directions, “down” opcodes that only make sense from Host to Controller, and “up” opcodes that only apply in the Controller to Host direction.

Following the lists of opcodes, there is a section explaining each opcode in detail.

List of Common Opcodes

- Acknowledge – 0x41 – ASCII ‘A’
- No-Acknowledge – 0x4E – ASCII ‘N’

List of Down Opcodes

- Homing Request – 0x48 – ASCII ‘H’
- Move Request – 0x4D – ASCII ‘M’
- Status Request – 0x53 – ASCII ‘S’
- Cancel Request – 0x43 – ASCII ‘C’

List of Up Opcodes

- Homing Finished – 0x68 – ASCII ‘h’
- Move Finished – 0x6D – ASCII ‘m’
- Status Report – 0x73 – ASCII ‘s’
- General Error – 0x65 – ASCII ‘e’

Acknowledge – 0x41 – ‘A’

Description: This opcode is a general acknowledgement (ack) message, to be issued in response to a previous request. It depends on the specific request (opcode) if it requires a corresponding ack or not. The message signifies that the request was understood, considered valid, and will be attempted to be fulfilled. If that is not the case, a No-Acknowledge (0x4E) is to be issued instead.

Data length: 1 byte

Data fields:

Data
Original Opcode
1 byte

There is only a single data field: one byte containing the opcode that prompted the Acknowledge.

Expected behavior: Opcode-specific. Generally none.

No-Acknowledge – 0x4E – ‘N’

Description: This opcode is a general non-acknowledgement (nack) message, to be issued in response to a previous request. It depends on the specific request (opcode) if it requires a corresponding ack/nack or not. The message signifies that the request was not understood, considered invalid, or can’t be fulfilled at this current time. It is the negative alternative to the Acknowledge (0x41) opcode.

Data length: 1 byte

Data fields: There is only a single data field: one byte containing the opcode that prompted the NACK.

Data
Original Opcode
1 byte

Homing Request – 0x48 – ‘H’

Description: This opcode is a request for the Controller to home in the actuators to a known default state. The specific behavior depends on the controller implementation, but is is expected to be something similar to retracting the arms until the stops are reached, defining the zero-movement state.

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Move Request – 0x4D – ‘M’

Description: This opcode is a request for the Controller to move the platform to a specified position. The path followed is expected to be linear.

Data length: TBD

Data fields: This opcode has the following data fields:

Data		
DA – float	DB – float	DC – float
4 bytes	4 bytes	4 bytes

- Data field DA – displacement of arm “A” – 4 bytes – IEEE 32-bit float
- Data field DB – displacement of arm “B” – 4 bytes – IEEE 32-bit float
- Data field DC – displacement of arm “C” – 4 bytes – IEEE 32-bit float

Status Request – 0x53 – ‘S’

Description: This opcode queries the Controller to send a status report message to the Host.

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Cancel Request – 0x43 – ‘C’

Description: This opcode requests the Controller to cancel the currently executing operation (Homing or Moving).

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Homing Finished – 0x68 – ASCII ‘h’

Description: This opcode informs the Host that the homing process has finalized.

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Move Finished – 0x6D – ASCII ‘m’

Description: This opcode informs the Host that the Controller moved the platform to the desired position.

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Status Report – 0x73 – ASCII ‘s’

Description: This opcode informs the Host of the current status of the Controller and the platform.

Data length: 13 bytes

Data fields: This opcode has the following data fields:

Data			
STAT	DA – float	DB – float	DC – float
1 byte	4 bytes	4 bytes	4 bytes

- STAT – Status bitmask – 1 byte
  - This byte is a bitmask containing information on the status of the platform:

STAT bits							
7	6	5	4	3	2	1	0
RESERVED	*RESERVED*	*RESERVED*	`STPC`	`STPB`	`STPA`	`ST1`	`ST0`

  - `ST[0..1]`: Status bits
    - Value 00: Platform is Idle
    - Value 01: Platform is Homing

        - Value 10: Platform is Homing
        - Value 11: Platform is at Error
    - STPA: bit indicating the status of the end-of-course switch of arm A.
        - Value 1: pressed
        - Value 0: not pressed
    - STPB: bit indicating the status of the end-of-course switch of arm B (1 means pressed).
    - STPC: bit indicating the status of the end-of-course switch of arm C (1 means pressed).
    - *Bits 5-7 are reserved for use in future revisions.*
- Data field DA – displacement of arm “A” – 4 bytes – IEEE 32-bit float
- Data field DB – displacement of arm “B” – 4 bytes – IEEE 32-bit float
- Data field DC – displacement of arm “C” – 4 bytes – IEEE 32-bit float

Note: If any arm displacement field is a NAN value, the arm position is unknown, and the Controller requires a Homing operation.

General Error – 0x65 – ASCII ‘e’

Description: This opcode informs the Host that the platform has encountered an error and the current operation cannot continue.

Data length: 0 bytes (none)

Data fields: This opcode has no associated data fields.

Parsing Library

The reference implementation of the protocol is a portable “pure-C” function library with no external dependencies, not even on the C standard library. This makes it suitable not only for use on a desktop application, but also for the controller firmware. Using the same implementation on both ends of the communication simplifies implementation on both ends and ensures code consistency.

The library is comprised of two files, a “`.h`” header and a “`.c`” source. Both files can be included in a source file for convenience, provided the definition.

Usage

The library provides facilities for sending, parsing and receiving frames. It is comprised of two main data structures, some definitions, and three functions. The definitions and functions are covered in the following sections.

The first structure is `protocol_data`, a circular buffer used for storing streamed bytes for parsing. The function `dp_process()` function is used to add bytes to the buffer, and the `dp_recv()` function attempts to parse bytes of the buffer into a message.

A message is stored in the `protocol_msg`. The function `dp_recv()` stores a received message in it, and the function

The following section goes into the library interface in greater detail.

Interface

Return Codes

The library defines common integer return codes for the functions. According to C convention, a function returning 0 usually means the operation completed successfully or returned something meaningful.

The return code definitions start with the `DP_RC_` prefix.

Return Code	Description
`DP_RC_SUCCESS`	The operation completed successfully.
`DP_RC_ERROR_GENERIC`	There was an error completing the operation.
`DP_RC_NO_FRAME`	No frame was available in the buffer.
`DP_RC_OVERFLOW`	Some buffer contents were lost because an overflow occurred.
`DP_RC_GARBAGE`	There was garbage in the buffer and no frame could be parsed.

Functions

Return Code	Description
`dp_process()`	Insert incoming bytes into a `protocol_data` structure. **Arguments:**

Return Code	Description
	• `protocol_data *data` → which buffer to insert data in • `int bytes` → how many bytes to insert • `void *buf` → where the data is
`dp_recv()`	Analyzes protocol data to look for a message, possibly returning it. **Arguments:** • `protocol_data *data` → which buffer to parse from • `protocol_msg *msg` → the message to fill out with parsed data
`dp_send()`	Sends a message using the given send function. **Arguments:** • `protocol_msg *msg` → the message to be sent • `send_func_t f` → the function that will be fed with the resulting stream bytes • `void *usr` → opaque pointer that will be forwarded as an argument to the send function

Structure protocol_data

Type	Field	Description
`uint8_t`	`opcode`	Message opcode.
`int`	`data_s`	How many data bytes the message has.
`char[32]`	`data`	Data bytes of the message
`int`	`checksum_ok`	Flag indicating if the checksum was correct when parsed.

Structure protocol_msg

Type	Field
`uint8_t`	`opcode`
`int`	`data_s`
`char[32]`	`data`

Other definitions

Identifier	Description
`send_func_t`	Type definition the the send function used by `dp_send()`. Defined as: `typedef int (*send_func_t)(void *buf, int bytes, void *usr);`
`DP_PROTO_FRAME_START`	Frame start marker (`'$'`)
`DP_PROTO_FRAME_END`	Frame end marker (`':'`)
`DP_PROTO_ALL_OPCODES`	Character string containing all valid opcodes.

`DP_OPCODE_ACK`
`DP_OPCODE_NACK`
`DP_OPCODE_HOMING`
`DP_OPCODE_STATUS`
`DP_OPCODE_MOVE`
`DP_OPCODE_CANCEL`
	Defines for every single opcode, for ease of use of the library.

Identifier	Description
DP_OPCODE_HOMING_REPL	
DP_OPCODE_STATUS_REPL	
DP_OPCODE_MOVE_REPL	
DP_OPCODE_GENERAL_ERR	

Examples

The library includes two example programs, in the `tests/` directory.

- **`"cross_comm.cpp"`** → instantiates several library structures and sends messages back and forth, testing the main functionality in thre process.
- **`"send_opcode.cpp"`** → sends a simple ACKNOWLEDGE message on the serial port, to test basic communication functions.