Motor Control Protocol

1 Introdution

Applications on the market, that require an electrical motor to be driven, usually have the electronics split into two parts: application board and motor drive board. To drive the system correctly, the application board requires a method to send a command to the motor drive board and get a feedback from it. This is usually performed using a serial communication.

To target this kind of application, a dedicated serial communication protocol has been developed for real-time data exchange. The aim of this protocol is to implement the features requested by motor control related applications. The implemented protocol is called Motor Control Protocol – aka MCP.

MCP makes it possible to send commands such as start/stop motor and set the target speed to the STM32 FOC motor control firmware, and also to tune relevant control variables such as PI coefficients in real-time. It is also possible to monitor motor control related quantities, such as the speed of the motor or the voltage of the bus present on the board and feeding the controlled system.

2 Protocol Overview

Communication with the Motor Control Protocol is organized around a master-slave scheme in which the motor control firmware, running on an STM32 microcontroller, is the slave while the master is usually a PC or another microcontroller. The master and the slave are physically connected through a plain serial link. The data they exchange are placed into messages that are packed into frames. A frame contains exactly one message.

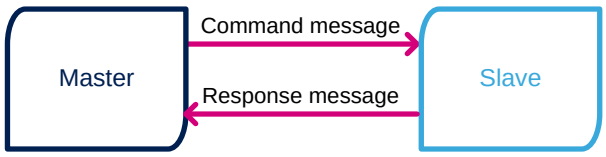

Figure 1: Master-slave communication scheme

Communication between the master and the slave occur during message exchanges. A message exchange consists in exactly two messages:

- A command message sent by the Master, on its initiative;
- A response message, sent by the Slave, as an answer to or an acknowledgment of the master's previous message.

There can be only one message exchange at any given time. In other words, the master initiates a message exchange by sending a command message then, it shall wait for the reception of a response message from the slave prior to sending another command message. For exceptions to this rule, protocol error processing and more details, see section 4.1 below.

3 Motor Control Protocol Frames

Motor Control Protocol messages are encapsulated into frames. A frame is a sequence of bytes that conforms to the format depicted in Figure 2: Frame structure.

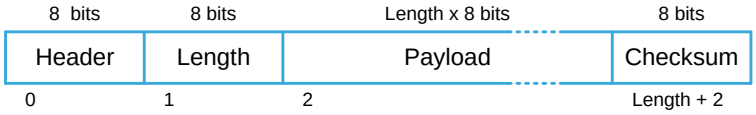

Figure 2: Frame structure

A Motor Control Protocol frame is made of the following fields, in the following order:

- The Header field is an 8-bit data that identifies what the target of the frame is.
  It is split into two parts, as outlined in Figure 3: Motor Control Protocol Header structure:

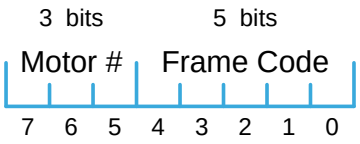

Figure 3: Motor Control Protocol Header structure

  - The 5 least significant bits of the field are a Frame Code that state the nature of the message and what it is expected to perform on the motor control subsystem if the message is sent by the Master. For a list of known Frame Codes, see section 3.1 below.
  - The 3 most significant bits of this field indicate the motor that the command targets. For single drive motor control subsystems, this field is not relevant. For dual or multi drive systems, however, it states the numeric identifier of the motor the command targets. Section 3.2 below specifies how this field is interpreted by a Motor Control Application.
- The Length field is an 8-bit unsigned integer which value is the number of bytes found in the payload of the frame. Allowed values range from 0 to 128 bytes. The structure of the payload depends on the Frame Code. Sections from 4.4 onwards specify the payload structure for each Frame Code.
- The Payload field contains the actual message transported by the frame. Its size is stated in the Length field and depends on the nature of the message, indicated in the Header field.
- The Checksum field is an 8-bit data used to check the validity of the content of the frame. It is computed and used as described in section 3.3.

3.1 Frame Codes

The following table lists existing frame codes.

Code	Name	Description
0x01	SET_REG	*Set register* The frame contains a command message that requests the firmware to set the value of an internal motor control variable. See section 4.4 below for more details.

0x02	GET_REG	*Get register* Requests the firmware to return the current value of an internal motor control variable. See section 4.5 on page 13 for more details.
0x03	EXECUTE_CMD	*Execute command* Requests the execution of a motor control command by the firmware. See section 4.6 on page 13 for more details.
0x06	GET_BOARD_INFO	*Get board information* The frame contains a command message that requests the firmware to return its version number in a form suitable for the MC Monitor. See section 4.7 on page 15 for more details.
0x07	SET_RAMP	*Set ramp* The frame contains a command message that requests the firmware to program and eventually execute a speed or torque ramp. See section 4.8 on page 15 for more details..
0x08	GET_REVUP_DATA	*Get revup data* The frame contains a command message that requests the firmware to return current revup information. See section 4.9 on page 16 for more details.
0x09	SET_REVUP_DATA	*Set revup data* The frame contains a command message that requests the firmware to set the revup data to new values. See section 4.10 on page 17 for more details.
0x0A	SET_CURRENT_REF	*Set current reference* The frame contains a command message that requests the firmware to change $I_q$and I_d current references. See section 4.11 on page 17 for more details.
0x0C	GET_FW_VERSION	*Get firmware version* The frame contains a command message that requests the firmware to return its full version string. See section 4.12 on page 18 for more details.
0x0D	GET_DATA_ELEMENT	*Gets the content of a data segment* This frame contains a command message that requests the firmware to return a Data Segment. See section **Error! Reference s ource not found.** on page **Error! Bookmark not defined.** for more details.
0x0E	CONFIG_DYN_STRUCT	*Configures a dynamic data structure* This frame contains a command message that configures the

		content of a Data Segment. See section **Error! Reference source not found.** on page **Error! Bookmark not defined.** for more details.
0x0F0	ACK	*Positive Acknowledge* The frame contains the response message to the last command message sent by the master. The ACK frame code informs the master that its previous frame was well received. See section 4.2 on page 6 for more details.
0x0FF	NACK	*Negative Acknowledge* The frame contains the response message to the last command message sent by the master. The NACK frame code informs the master that its previous frame was not received. The payload of the frame contains an error code indicating what went wrong. See section 4.3 on page 6 for more details.

Table 1: Existing Frame Codes

Frames with the frame code set to ACK or the NACK can only be sent by the Slave, as the answer to the last message received from the Master. Frames with any of the other frame codes can only be sent by the Master.

3.2 Motor Number code

The 3 most significant bits of the Header field of a frame are only meaningful for a command frame. They make up the Motor Number code which indicates the motor that the message contained in the frame targets. These three bits have no meaning in a response frame and should then always be set to 0.

In a motor control subsystem, motors are numbered, from 1 to N, where N is the number of motors it drives.

A motor Number code set to 0 indicates that the command message contained in the frame deals with the current motor. At startup (after a power-on sequence or after a reset), the current motor is motor 1.

A Motor Number code set to a value between 1 and N indicates that the command message contained in the frame deals with the motor set in the Motor Number code and that this motor becomes the current motor.

A Motor Number code set to a value strictly higher than N is an error. The command message contained in the frame is not processed and an error response message is sent back in a NACK frame, with the BAD_MOTOR_SELECTED error code. See section 4.3 for more details on error response messages.

3.3 Frame Checksum

A checksum is computed for every frame whether it contains a command or a response message. The following algorithm is used to compute it:

1. A 16-bit, unsigned accumulator is set to 0
2. The Header and the Length bytes considered as 8-bit unsigned numbers are added to the accumulator

3. Each of the bytes in the payload, considered an 8-bit unsigned integer, is added to the accumulator.
4. The 8 upper and 8 lower bits of the accumulator are turned into 8-bit unsigned integers and summed together. The 8 lower bits of the result are the checksum.

The following equation summarizes this algorithm.

$$Accumulator = Header + Length + \sum_{n=0}^{Length-1} Payload[n]$$
$$Checksum = \text{LowByte}(\text{HighByte}(Accumulator) + \text{LowByte}(Accumulator))$$

4 Motor Control Protocol Messages

4.1 Message Control Protocol details

A message exchange begins when the Master sends a command message. For this message to be a valid command message, it must be encapsulated in a frame which Frame Code can be any of the codes listed in Table 1 except ACK and NACK.

On the receiving side, the slave first receives the Header byte and the Length byte. Then, it waits for receiving Length bytes and finally the Checksum byte.

On the sending side, once the master has sent the whole command message up to the Checksum byte, a one second timer is started. If this timer expires before the answer from the salve has been received, an error is raised and the communication with the slave is closed.

When all this data has been received by the slave, the checksum is verified by computing it on the received data and comparing the result with the received Checksum field. If the verification fails, the slave sends a Negative Acknowledge message with the BAD_CRC error code as answer which terminates the message exchange. See section 4.3 for all information on NACK messages and error codes.

If the full message is not received by the slave within 25 ms, the slave sends a Negative Acknowledge message with the TIMEOUT error code as answer which terminates the message exchange.

Once the slave has fully received the command message it checks the Motor# field of the Header byte. The value of this field defines the motor targeted by the message. It may also set the current motor.

If Motor# is 0, the message targets the current motor. Otherwise, the message targets the motor indicated by Motor#: Motor 1 if Motor# is 1, Motor 2 if Motor# is 2, etc…

If the value of Motor# refers to a motor that does not exist in the application, the slave sends a Negative Acknowledge message with the BAD_MOTOR_SELECTED error code as answer which terminates the message exchange. This happens, for instance if Motor# is set to 2 on a single drive application.

If the value of Motor# differs from 0 and is valid, the current motor is set to this value. At startup, the current motor is Motor 1.

If the Motor# check phase is successful, the slave considers the Frame Code field of the message. If this field does not match with any known frame code, the slave sends a Negative Acknowledge message with the BAD_FRAME_ID error code as answer which terminates the message exchange.

Otherwise, the slave processes the message and its payload according to the semantic of the Frame Code. If this processing is successful, the slave sends a Positive Acknowledge frame, otherwise it sends a Negative Acknowledge one. In either case, this sending terminates the exchange and another one can be initiated by the master. The content of the payload Positive Acknowledge messages as well as the error codes of Negative Acknowledge messages depends on the command message sent by the master at the start of the exchange. This is described in the following sections.

Note that all numeric data field present in Motor Control Protocol messages are transferred and stored in little endian format.

4.2 Positive Acknowledge Response Message

A Positive Acknowledge message is encapsulated in a frame with a frame code set to ACK (`0xF0`). This message is sent by the slave as answer to a command message received from the master.

It indicates that the command message was processed successfully and contains the response payload if any is required by the command message.

The structure of a Positive Acknowledge message – the payload of the frame – then depends on the command message it answers. See the sections related to each command message to get a complete information on the content of Positive Acknowledge messages.

4.3 Negative Acknowledge Response Message

A Negative Acknowledge message is encapsulated in a frame with a frame code set to ACK (`0xFF`). This message is sent by the slave as answer to a command message received from the master.

It indicates that the command message received from the master was not processed successfully.

The structure of a Negative Acknowledge message – the payload of the frame – is illustrated in the following figure.

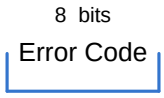

Figure 4: Negative Acknowledge message structure

It consists in a single Error Code byte which value indicate the reason why the command message was not processed successfully. Some errors occur before the command message is even processed; these are described in section 4.1 above . Other errors occur during command message processing and are described in the next sections. Existing error code are listed in Table 2 below.

Table 2: Negative Acknowledge frames Error Code values

Error ID	Name	Description
0x00	NONE	Not an error. This code is never found in a NACK message
0x01	BAD_FRAME_ID	The Frame Code field is unknown.
0x02	SET_READ_ONLY	Attempt to write into a Read Only or an unknown register
0x03	GET_WRITE_ONLY	Attempt to read a Write Only or an unknown register
0x05	WRONG_SET	The value used in the frame is invalid.

0x07	WRONG_CMD	The Command Identifier passed to the last EXECUTE_CMD message is unknown. The command was not executed.
0x08	OVERRUN	Transmission speed is too high, the frame was not received correctly
0x09	TIMEOUT	The reception of the current frame did not complete in time. The frame reception may be corrupted.
0x0A	BAD_CRC	Computed checksum is not equal to received checksum
0x0B	BAD_MOTOR_SELECTED	The value of the Motor# field of the frame is invalid.
0x0C	MP_NOT_ENABLED	Motor Profiler commands are not supported

4.4 Set Register Message

A Set Register message is encapsulated in a frame with a frame code set to SET_REG (0x01). This message is sent by the master. It requests that the slave writes a value in a motor control variable. The structure of the Set Register message is illustrated in the following figure.

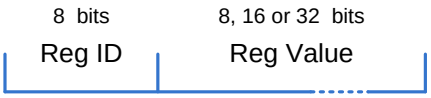

Figure 5: Structure of a Set Register message

The Reg ID field identifies the motor control variable to write into. The Reg Value field contains the value to write into the variable. The size of this field depends on the Reg ID code. It can be either 8, 16 or 32 bits. Table 3 below lists all possible Reg ID codes and details the nature, size, unit and access type of the related variables.

On reception of a Set Register message, the slave first checks if the variable identified by Reg ID can be written. If that is the case, this variable is set to the value of Reg Value and an empty Positive Acknowledge message is sent to the master as answer.

> **Beware!** The slave does not check the size of the data transmitted in the Reg Value field against that of the variable identified by Reg ID. As long as the variable exists and can be written, it will be. If the size of the Reg Value field is bigger than that of the target variable, only the lower bytes of Reg Value will be written into the variable and the higher bytes that cannot fit into the variable will be discarded. If the size of Reg Value is smaller than that of the target variable, then the value that gets written is partially undefined. In both of these two cases, the slave sends a Positive Acknowledge message as answer.

If the variable cannot be written or if it does not exist, the slave sends a Negative Acknowledge frame with the SET_READ_ONLY error code as answer.

Note that the existence of some variables depend on the configuration of the motor control application. For instance, the OBSERVER_CR_C1 variable only exists if a State Observer with CORDIC is used as speed and position sensing mechanism in the application.

Table 3: Motor Control Registers

Reg ID	Name	Description	Type	Unit	Access
0x0	TARGET_MOTOR	Target motor	u8	-	RW
0x1	FLAGS	Fault Flags	u32	-	R
0x2	STATUS	Motor control subsystem State	u8	-	R
0x3	CONTROL_MODE	Control mode	u8	-	RW
0x4	SPEED_REF	Speed reference	s32	RPM	R
0x5	SPEED_KP	Speed PID regulator K_P parameter (Numerator)	s16	s16A/"Unit"	RW
0x6	SPEED_KI	Speed PID regulator K_I parameter (Numerator)	s16	s16A/"Unit"	RW
0x7	SPEED_KD	Speed PID regulator K_D parameter (Numerator)	s16	s16A/"Unit"	RW
0x8	TORQUE_REF	Torque reference (I_q)	s16	s16A	RW
0x9	TORQUE_KP	Torque (I_q) PID regulator K_P parameter (Numerator)	s16	s16V/s16A	RW
0xA	TORQUE_KI	Torque (I_q) PID regulator K_I parameter (Numerator)	s16	s16V/s16A	RW
0xB	TORQUE_KD	Torque (I_q) PID regulator K_D parameter (Numerator)	s16	s16V/s16A	RW
0xC	FLUX_REF	Flux reference (I_d)	s16	s16A	RW
0xD	FLUX_KP	Flux (I_d) PID regulator K_P parameter (Numerator)	s16	s16V/s16A	RW
0xE	FLUX_KI	Flux (I_d) PID regulator K_I parameter (Numerator)	s16	s16V/s16A	RW
0xF	FLUX_KD	Flux (I_d) PID regulator K_D parameter (Numerator)	s16	s16V/s16A	RW
0x10	OBSERVER_C1	Parameter C_1 of the Luenberger Observer + PLL (only available if State Observer + PLL is used whether as main or auxiliary)	s16	N/A	RW
0x11	OBSERVER_C2	parameter C_2 of the Luenberger Observer + PLL (only available if State Observer + PLL is used whether as main or auxiliary)	s16	N/A	RW

0x12	OBSERVER_CR_C1	parameter C_1 of the Luenberger Observer + CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	N/A	RW
0x13	OBSERVER_CR_C2	parameter C_2 of the Luenberger Observer + CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	N/A	RW
0x14	PLL_KI	PLL PID regulator K_I parameter (Numerator). Only available if State Observer + PLL is used whether as main or auxiliary	s16	dpp/s16V	RW
0x15	PLL_KP	PLL PID regulator K_P parameter (Numerator). Only available if State Observer + PLL is used whether as main or auxiliary	s16	dpp/s16V	RW
0x16	FLUXWK_KP	Flux weakening PID regulator K_P parameter.	s16	s16A/(s16V^2)	RW
0x17	FLUXWK_KI	Flux weakening PID regulator K_I parameter.	s16	s16A/(s16V^2)	RW
0x18	FLUXWK_BUS	Flux weakening BUS Voltage allowed percentage reference	u16	%	RW
0x19	BUS_VOLTAGE	Bus Voltage	u16	V	R
0x1A	HEATS_TEMP	Heatsink temperature	u16	°C	R
0x1B	MOTOR_POWER	Motor power	u16	W	R
0x1C	DAC_OUT1	DAC Out 1	u8	N/A	RW
0x1D	DAC_OUT2	DAC Out 2	u8	N/A	RW
0x1E	SPEED_MEAS	Average Mechanical Speed measure	s32	RPM	R
0x1F	TORQUE_MEAS	Torque measure (I_q)	s16	s16A	R
0x20	FLUX_MEAS	Flux measure (I_d)	s16	s16A	R
0x21	FLUXWK_BUS_MEAS	Flux weakening BUS Voltage allowed percentage measured	u16	%	R
0x22	RUC_STAGE_NBR	Revup stage numbers	u8	-	R
0x23	I_A	I_a phase current, measured	s16	s16A	D
0x24	I_B	I_b phase current, measured	s16	s16A	D
0x25	I_ALPHA	I_α current	s16	s16A	D

0x26	I_BETA	I_β current	s16	s16A	D
0x27	I_Q	I_q current, measured (synonymous to Torque measure)	s16	s16A	D
0x28	I_D	I_d current, measured (synonymous to Flux measure)	s16	s16A	D
0x29	I_Q_REF	I_q current reference (synonymous to Torque reference)	s16	s16A	D
0x2A	I_D_REF	I_d current reference (synonymous to Flux reference)	s16	s16A	D
0x2B	V_Q	V_q voltage	s16	s16V	D
0x2C	V_D	V_d voltage	s16	s16V	D
0x2D	V_ALPHA	V_α voltage	s16	s16V	D
0x2E	V_BETA	V_β voltage	s16	s16V	D
0x2F	MEAS_EL_ANGLE	Measured electrical angle (only available if either Hall sensors or an Encoder are used whether as main or auxiliary)	s16	s16degree	D
0x30	MEAS_ROT_SPEED	Measured mechanical rotation speed (only available if either Hall sensors or an Encoder are used whether as main or auxiliary)	s16	s16speed	D
0x31	OBS_EL_ANGLE	Observed electrical angle (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16degree	D
0x32	OBS_ROT_SPEED	Observed mechanical rotation speed (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16speed	D
0x33	OBS_I_ALPHA	Estimated I_α current (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16A	D
0x34	OBS_I_BETA	Estimated I_β current (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16A	D
0x35	OBS_BEMF_ALPHA	Observed Back EMF, Alpha component (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16V	D

0x36	OBS_BEMF_BETA	Observed Back EMF, Beta component (only available if State Observer + PLL is used whether as main or auxiliary)	s16	s16V	D
0x37	OBS_CR_EL_ANGLE	Observed electrical angle, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16degree	D
0x38	OBS_CR_ROT_SPEED	Observed mechanical rotation speed, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16speed	D
0x39	OBS_CR_I_ALPHA	Estimated I_α current, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16A	D
0x3A	OBS_CR_I_BETA	Estimated I_β current, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16A	D
0x3B	OBS_CR_BEMF_ALPHA	Observed Back EMF, Alpha component, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16V	D
0x3C	OBS_CR_BEMF_BETA	Observed Back EMF, Beta component, CORDIC (only available if State Observer + CORDIC is used whether as main or auxiliary)	s16	s16V	D
0x3D	DAC_USER1	DAC User value, channel 1	s16	N/A	D
0x3E	DAC_USER2	DAC User value, channel 2	s16	N/A	D
0x3F	MAX_APP_SPEED	Maximum application speed	u32	RPM	R
0x40	MIN_APP_SPEED	Minimum application speed	u32	RPM	R
0x41	IQ_SPEEDMODE	I_q reference in speed mode (synonym to Flux reference)	s16	s16A	W
0x42	EST_BEMF_LEVEL	Expected BEMF level (PLL)	s16	$s16V^2$	R
0x43	OBS_BEMF_LEVEL	Observed BEMF level (PLL)	s16	$s16V^2$	R
0x44	EST_CR_BEMF_LEVEL	Expected BEMF level (CORDIC)	s16	$s16V^2$	R
0x45	OBS_CR_BEMF_LEVEL	Observed BEMF level (CORDIC)	s16	$s16V^2$	R
0x46	FF_1Q	Feedforward constant C_{1q} for Q axis	s32	N/A	RW

0x47	FF_1D	Feedforward constant C_{1d} for D axis	s32	N/A	RW
0x48	FF_2	Feedforward constant C_2	s32	N/A	RW
0x49	FF_VQ	Feedforward contribution to V_q	s16	s16V	R
0x4A	FF_VD	Feedforward contribution to V_d	s16	s16V	R
0x4B	FF_VQ_PIOUT	Feedforward Current PI regulator output, q axis (V_q)	s16	s16V/s16A	R
0x4C	FF_VD_PIOUT	Feedforward Current PI regulator output, d axis (V_d)	s16	s16V/s16A	R
0x4D	PFC_STATUS	PFC State	s8	-	R
0x4E	PFC_FAULTS	PFC Faults	u32	-	R
0x4F	PFC_DCBUS_REF	PFC DC Bus Reference	s16	V	RW
0x50	PFC_DCBUS_MEAS	PFC DC Bus Measure	s16	V	R
0x51	PFC_ACBUS_FREQ	PFC AC Bus frequency Measure	s16	01Hz	R
0x52	PFC_ACBUS_RMS	PFC AC Bus Voltage 0-to-Peak Measure	s16	V	R
0x53	PFC_I_KP	PFC Current PID regulator $K_P$parameter	s16	s16V/s16A	RW
0x54	PFC_I_KI	PFC Current PID regulator $K_I$parameter	s16	s16V/s16A	RW
0x55	PFC_I_KD	PFC Current PID regulator $K_D$parameter	s16	s16V/s16A	RW
0x56	PFC_V_KP	PFC Voltage PID regulator $K_P$parameter	s16	s16A/V	RW
0x57	PFC_V_KI	PFC Voltage PID regulator $K_I$parameter	s16	s16A/V	RW
0x58	PFC_V_KD	PFC Voltage PID regulator $K_D$parameter	s16	s16A/V	RW
0x59	PFC_STARTUP_DURATION	PFC Startup sequence duration	s16	ms	RW
0x5A	PFC_ENABLED	PFC Enabled Status	s8	-	R
0x5B	RAMP_FINAL_SPEED	Ramp final speed	s32	RPM	RW
0x6E	SPEED_KP_DIV	Speed PID regulator K_PDivisor	s16	N/A	R
0x6F	SPEED_KI_DIV	Speed PID regulator K_IDivisor	s16	N/A	R
0x72	CTRBDID	Control Board UID	u32	-	R

0x73	PWBDID	Power Board ID	u32	-	R
0x81	PWBDID2	Power Board ID, Motor 2	u32	-	R

4.5 Get Register Message

A Get Register message is encapsulated in a frame with a frame code set to `GET_REG` (`0x01`). This message is sent by the master. It requests that the slave reads a value from a motor control variable. The structure of the Get Register message is illustrated in the following figure.

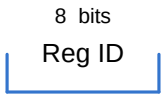

Figure 6: Structure of a Get Register message

The Reg ID field identifies the motor control variable to read from. Table 3 above lists all possible Reg ID codes and details the nature, size, unit and access type of the related variables.

On reception of a Get Register message, the slave first checks if the variable identified by Reg ID can be read. If that is the case, this variable is read and a Positive Acknowledge message containing the value of the variable is sent to the master as answer. Figure 7 below Illustrates the structure of such a message.

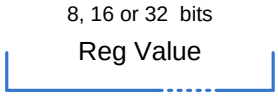

Figure 7: Structure of the Positive Acknowledge answer to a Get Reg command message

The Reg Value field contains the value read from the variable. Its size depends on the Reg ID code of the command message. It can be either 8, 16 or 32 bits.

If the variable cannot be read or if it does not exist, the slave sends a Negative Acknowledge frame with the `GET_WRITE_ONLY` error code as answer.

4.6 Execute Command Message

An Execute Command message is encapsulated in a frame with a frame code set to `GET_REG` (`0x01`). This message is sent by the master. It requests the slave to execute a motor control command. The structure of the Get Register message is illustrated in the following figure.

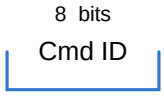

Figure 8: Structure of an Execute Command message

The Cmd ID field identifies the command to be executed by the motor control subsystem. Table FFF lists all possible commands.

On reception of an Execute Command message, the slave first checks if the Cmd ID field matches an existing command. If that is the case, this command is executed, otherwise a Negative Acknowledge frame with the WRONG_CMD error code is sent as answer.

If the execution of the command completes successfully, an empty Positive Acknowledge message is sent to the master as answer. If the execution fails, a Negative Acknowledge frame with the WRONG_CMD error code is sent as answer.

Note that these commands accept no parameters.

Table 4: Possible values for the Cmd ID field of the Execute Command Message

Cmd ID	Name	Description
0x01	START_MOTOR	Starts the current motor. If the motor has already been started, nothing happens. If the motor's state machine is in an error state, nothing happens. This command never fails.
0x02	STOP_MOTOR	Stops the current motor. If the motor has already been started, nothing happens. This command never fails.
0x03	STOP_RAMP	Stops the current ramp (either torque or speed) on the current motor. If no ramp is on-going on this motor, nothing happens. This command never fails.
0x06	START_STOP	Starts the current motor if it is idle and stops it if it is spinning. This command does nothing if the motor's state machine is in an error state. This command never fails.
0x07	FAULT_ACK	Acknowledges all pending past faults detected by the motor control subsystem, if any and if no fault condition is still true. Otherwise, does nothing. This command never fails.
0x08	ENCODER_ALIGN	Starts the encoder alignment procedure. If the motor's state machine is not a state allowing for encoder alignment, nothing happens. This command never fails.
0x09	IQDREF_CLEAR	Resets the I_{qdref} parameter to its default value. This command never fails.
0x0A	PFC_ENABLE	Enables the Power Factor Correction feature if it is configured for the current motor and is not started yet. Otherwise, the command does nothing. This command succeeds unless the PFC feature has not been configured for the current motor.
0x0B	PFC_DISABLE	Disables the Power Factor Correction feature if it is configured for the current motor and is not started yet. Otherwise, the command does nothing. This command succeeds unless the PFC feature has not been configured for the current motor.

0x0C	PFC_FAULT_ACK	Acknowledges all pending past PFC faults detected by the current motor control subsystem, if any and if no fault condition is still true. Otherwise, does nothing. This command succeeds unless the PFC feature has not been configured for the current motor.

4.7 Get Board Info Message

A Get Board Info message is encapsulated in a frame with a frame code set to GET_BOARD_INFO (0x06). This message is sent by the master. It requests the slave to send the version of the Motor Control SDK used to build the application. This message does not accept any parameter; its payload is empty.

The processing of this message never fails and the slave sends a Positive Acknowledge frame with a 32 bytes' payload as answer. The structure of this payload is illustrated in Figure 9 below.

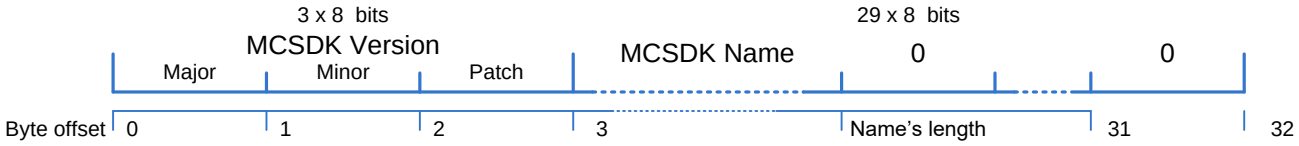

Figure 9: Get Board Info response message payload structure

The first three bytes are unsigned 8-bit integers representing respectively the major, minor and patch version numbers. The rest of the payload is a null-terminated characters string containing the name of the SDK, typically: "ST MC SDK".

Note that the possible Release Candidate, Alpha of Beta releases status is not published with this command message. For a more complete information on the version of the Motor Control firmware, refer to command message Get Firmware Version described in section 4.12, page 18.

4.8 Set Ramp Message

A Set Ramp message is encapsulated in a frame with a frame code set to SET_RAMP (0x07). This message is sent by the master. It requests the slave to program a speed ramp with the speed and duration provided in the message for the current motor. If the motor is already spinning in steady state, the ramp is executed immediately. Otherwise, its execution is deferred until the motor reaches that state.

If the control mode of the current motor is torque mode, it is changed to speed mode before the command is executed. This mode can be changed back with a Set Register command message – see section 4.4 on page 7.

The structure of the Set Ramp message is illustrated in Figure 10 below.

Figure 10: Structure of a Set Ramp message.

The Final Speed field is the speed reference target to be reached at the end of the ramp. It is expressed in RPM and transferred as a signed 32-bit integer. The Duration field specifies the time the ramp will take to go from the current rotation speed to the target one, in milliseconds. It is transferred as an unsigned 16-bit integer.

The processing of this message never fails and the slave sends an empty Positive Acknowledge frame as answer.

4.9 Get Revup Data Message

A Get Revup Data message is encapsulated in a frame with a frame code set to `GET_REVIUP_DATA` (`0x08`). This message is sent by the master. It requests that the slave returns the parameters of the revup phase passed as parameter with the message.

The structure of the Get Revup Data message is illustrated in the following figure.

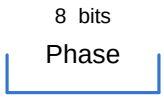

Figure 11: Structure of the Get Revup Data message

The Phase field indicates the phase which data are to be send to the master. These data consist in the Target Speed, Target Torque and Duration of the phase:

- The Target Speed is a 32-bit signed integer and specifies the mechanical rotation speed of the motor in RPM to reach at end of the phase.
- The Target Torque is a 16-bit signed integer that specifies the motor's torque to reach at end of the phase. It is actually the I_q current target and is expressed in the s16A unit.
- Duration is an unsigned 16-bit integer and specifies the duration of the phase, in milliseconds.

The processing of this message never fails and the slave sends a Positive Acknowledge frame as answer with these three pieces of data laid out as illustrated in Figure 12 below.

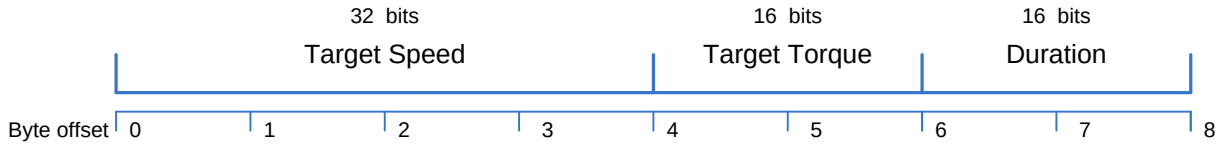

Figure 12: Get Revup Data response message payload structure

Note that, though this command message always returns Revup data, these are meaningful for sensor-less configurations only.

4.10 Set Revup Data Message

A Set Revup Data message is encapsulated in a frame with a frame code set to SET_REVUP_DATA (0x09). This message is sent by the master. It requests the slave to set the data for a revup phase to the values provided in the message. The structure of the Set Revup Data message is illustrated in the following figure:

Figure 13: Structure of the Set Revup Data structure

- The Phase field specifies the phase of the revup procedure to which the provided data apply. It is an 8-bit unsigned integer that ranges from 0 to 4.
- The Target Speed field is a 32-bit signed integer and specifies the mechanical rotation speed of the motor in RPM to reach at end of the phase.
- The Target Torque field is a 16-bit signed integer that specifies the motor's torque to reach at end of the phase. It is actually the I_q current target and is expressed in the s16A unit.
- The Duration field is an unsigned 16-bit integer and specifies the duration of the phase, in milliseconds.

The processing of this message never fails and the slave sends an empty Positive Acknowledge frame as answer.

Note that, this command message is meaningful for sensor-less configurations only.

4.11 Set Current Reference

A Set Current Reference message is encapsulated in a frame with a frame code set to SET_CURRENT_REF (0x09). This message is sent by the master. It requests the slave to set the $I_q$and $I_d$references. If the motor is already spinning in steady state, these references are applied immediately. Otherwise, their application is deferred until the motor reaches that state.

The structure of the Set Revup Data message is illustrated in the following figure.

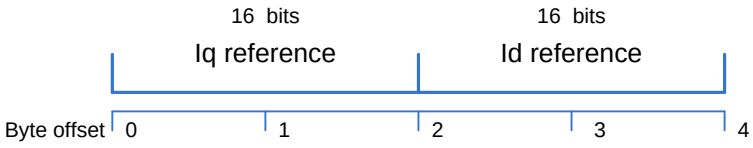

Figure 14Structure of a Set Current Reference message

It consists in two 16-bit signed integer fields: Iq reference and Id reference, representing respectively the $I_q$and $I_d$references to set.

The processing of this message never fails and the slave sends an empty Positive Acknowledge frame as answer.

4.12 Get Firmware Version

A Get Firmware Version message is encapsulated in a frame with a frame code set to GET_FW_VERSION (0x0C). This message is sent by the master. It requests the slave to send the version of the Motor Control Firmware used to build the application. This message does not accept any parameter; its payload is empty.

The processing of this message never fails, and the slave sends a Positive Acknowledge frame with a 32 bytes' payload as answer. This payload contains a null terminated string with the full version name of the firmware. This version name conforms to the following scheme:

```
Name\tVer.<MAJOR>.<MINOR>.<PATCH>[-<TYPE><INCREMENT>]
```

Where:

- Name is the name of the firmware, followed by a tabulation character. Typically set to ST MC SDK.
- MAJOR is the major version number. Its minimal value is 1. It gets incremented on a major change.
- MINOR is the minor version number. Its minimal value is 0. MINOR is incremented on each new feature changing release of the firmware. MINOR is reset to 0 on a MAJOR number increment.
- PATCH is the patch number. Its minimal value is 0. It differs from 0 for patch releases only. PATCH is set to 1 for the first patch release of a Minor release and gets incremented on each new patch based on the same Minor Release.
- TYPE is an optional identifier of the release type of the firmware. TYPE admits one of the following values:
    - A for Alpha releases
    - B for Beta releases
    - RC for release candidates
- INCREMENT is the number of the version for the given TYPE. Its minimal value is 1.