Introduction

The aim of this protocol suite is to implement the features needed to **drive** and to **monitor** motor control applications embedded in STM32 MCUs.

It makes it possible, for instance, to send commands for starting or stopping the motor(s) controlled by an application, to set the target speed of the motors, and to tune relevant control variables (such as speed, torque or flux PI regulator coefficients, ...) in real-time. It also provides features to monitor motor control related quantities, like the speed of a motor or the currents flowing through its phases for instance.

The Motor Control Protocol Suite is made of several parts, one of which is the Motor Control Protocol itself. The purpose of this document is to describe, in details, the Motor Control Protocol Suite and all its parts.

In the rest of the document, the *Motor Control Protocol Suite* is also referred to as *MCPS* and the *Motor Control Protocol* is also referred to as *MCP*.

Overview

The *Motor Control Protocol Suite* specifies the communication between two devices. The first one is based on an STM32 MCU and runs a Motor Control application. The second one can be any kind of a device (possibly powered by an STM32) running an application that aims at controlling and monitoring the former.

The two devices play different roles and behave differently on both sides of the communication. To handle this difference, the Motor Control Protocol Suite defines two different roles: the **Controller** role and the **Performer** role.

The **Performer** is the device that runs the Motor Control application. The **Controller** is the one that controls and monitors the **Performer**. This is highlighted in the figure above: The Controlling and Monitoring Application running on the **Controller** side uses the Motor Control Protocol Suite to control and monitor an STM32 Motor Control Application that runs on the **Performer** side. These roles are set at firmware (or software) built time.

The *Motor Control Protocol Suite* is made of a stack of communication protocols that sit on top of one another and that use one another services.

The *Motor Control Protocol* itself is at the top and interfaces directly with the application, whether it runs on the **Controller** or on the **Performer** side. It provides the services needed by a Controller application to properly monitor and control the motors driven by a Performer application.

To transport the data it has to handle, the *Motor Control Protocol* uses lower level protocols that are adapted to the physical link that connect the devices. ASPEP (used across serial links) and STLEP(used across STLink) are two such protocols. ASPEP is specified in the current version of the present document. STLEP is still draft and not fully supported yet. It will be described in a further version of the present document

Motor Control Protocol

Protocol Overview

The Motor Control Protocol (MCP) works by exchanging messages between the **Controller** and the **Performer**. These messages are submitted to the lower level protocol being used to communicate between the Controller and the Performer. This lower level protocol transports the messages and delivers them to the remote.

MCP requires that the lower level protocol offer two independent communication channels: messages sequences exchanged on one of the channel are independent from the ones exchanged on the other. The lower level protocol must provide interfaces that let the MCP specifies on which channel it submits the messages it sends. Also, when receiving a message, the lower level protocol shall notify the MCP about both the message and the channel on which it is received.

MCP does not provide any message segmentation mechanism: an MCP message must be small enough to fully fit into a service data unit of the lower level protocol.

Data are sent in little-endian order: least significant bits of bytes are transmitted first.

The Motor Control Protocol is designed to minimize the amount of computing power needed on the STM32 Performer side to operate it.

This section specifies **version 1** of the Motor Control Protocol.

Services overview

The Motor Control Protocol features are organized around four services: the Command, Registry, Datalog and Notification services.

The Command service deals with the sending, by the **Controller**, of commands that are executed by the **Performer**. After execution, the Performer returns a status possibly preceded by additional data. An example of a command is the `START_MOTOR` command that instructs the Motor Control application on the Performer side to set a motor under control.

The Registry service formalizes the access, by the Controller, to internal variables and states of the embedded motor control application on the Performer. Registers are used to let the Controller read measurements made by the embedded motor control application, write run time application parameters, or even trigger actions. Examples of registers include the `I_A` register that references the current measured on phase A, the `CONTROL_MODE` register used to get and set the control mode of the application or the `SPEED_RAMP` register that allows the controller to program a speed ramp for a motor, just to name a few.

The Datalog service lets the Controller monitor the changing values of registers in a controlled way. The registers to monitor and the value sampling rate are configurable. This service allows for plotting registers values like an oscilloscope would do with physical signals.

The Notification service provides the Controller with the possibility to be notified when the values of a set of registers change. For instance, it can be used to be notified whenever the `STATUS` register, that represents the state of the motor control state machine, changes.

The Command service

The Motor Control Protocol defines three core commands that must be available in all MCP implementations. Some other commands are defined, that are applicative ones and which availability depends on the target application. Some of these applicative commands, targeting the most common Motor Control applications are defined in section Motor Control and other Commands.

In addition, the protocol also provisions for one user defined command that is free for embedded applications designers to implement.

Commands are sent by the **Controller** (and only the **Controller**). They are identified by a **Command ID** and can contain additional data. Upon reception, they are immediately executed by the **Performer** that returns a status, possibly preceded by additional data.

At most one command can be pending at any time: the **Controller** must not send a new command before either the preceding one has been answered by the **Performer** or the **Controller** has received an indication stating that it has been lost and will never be answered.

The Command service uses a synchronous channel of the lower level protocol: both commands and response messages are transferred on this channel.

All commands share the following message format:

Figure : MCP Command format

The 13-bit `Command ID` field uniquely identifies the command. Known command IDs are listed in section 4 below.

The 3-bit `Motor #` field identifies the motor targeted by the command. A value of 0 usually means that the command does not target a specific motor. It can also mean that it targets all motors. This latter behavior of the Motor # 0 value is indicated in the specification of the commands to which it applies.

The `Command Payload` field is optional. It is used to transport the parameters of the command, for commands that have some. The content and meaning of the payload is defined independently for each command.

Response messages sent by the Performer to answer commands conform to the following format:

Figure : MCP Command response format

The last 8 bits of a response message, make the Status Code field. It provides the status of the execution of the command.

MCP Commands

MCP Response Codes

Command Status Codes are found in Response messages. They refer to the last received command.

Command status codes can be defined independently for each command. However, the codes listed in the Table below are defined and reused across commands where applicable.

The first response code, `CMD_OK` indicates the successful execution of the preceding command. All other codes indicate an error.

ID	Name	Description
0x00	`CMD_OK`	Execution of the command was successful
0x01	`CMD_NOK`	Execution of the command failed
0x02	`CMD_UNKNOWN`	Command is unknown
0x03	*Unused*	
0x04	`RO_REG`	Target register is Read Only. Its value cannot be written
0x05	`UNKNOWN_REG`	Target register is unknown
0x06	`STRING_FORMAT`	The format of a text string in the command payload is wrong

0x07	`BAD_DATA_TYPE`	The type of a register in the command payload is wrong
0x08	`NO_TXSYNC_SPACE`	The size of the response to the command exceeds the maximum payload size
0x09	`NO_TXASYNC_SPACE`	The number of signals requested for the Datalog exceeds the maximum supported by the Performer
0x0A	`WRONG_STRUCT_FORMAT`	The reported size of a structure transmitted in the command does not match its actual size
0x0B	`WO_REG`	Target register is Write Only. Its value cannot be read
0x0C	*Unused*	
0x0D	`USER_CMD_NOT_IMPL`	Command is a non implemented user command.

Table 1: Common MCP command status codes

<tt>GET_MCP_VERSION</tt> command

Purpose:

The `GET_MCP_VERSION` command requests the version of the Motor Control Protocol that the Performer supports.

Command ID: `0x0000`.

Command Payload:

This command does not have a payload.

Response:

The payload of the response to this command has a four bytes payload that contains an unsigned integer value set to the version of the Motor Control Protocol supported by the Performer. For the version of the MCP described in this document, this value is `0x00000001`.

The status code to this command is always `CMD_OK`.

<tt>SET_REGISTER</tt> command

Purpose:

The `SET_REGISTER` command requests to set the values of one or more registers of the STM32 embedded application.

Command ID: `0x0001`.

Command Payload:

The payload of this command is the concatenation of the identifiers of the registers which values are requested followed by their new values. The figure below depicts this.

In the payload of the command, each register ID is followed by the new value to set for it.

For all details about registers and their identifier, refer to the Registry Service section.

Response:

If the command requests to set the value of only one register, the response has no payload and consists solely in a status code, either `CMD_OK` if the register could be set or an error code otherwise.

If more than one register is to be set by the command, the response consists in either of the following:

- The `CMD_OK` status code alone, with no payload, if all the requested registers could be set successfully;
- The `CMD_NOK` status code preceded by a payload made of as many status codes as there are registers to set in the command. In that case, the **Performer** tried to set all the registers listed in the command and failed for some of them. The status codes of the register set operations are listed in the order of the registers in the command. The status codes of registers which value could be set correctly is `CMD_NOK`; that of registers which value that could not be set correctly reflect the reason of the issue;

- The `CMD_NOK` status code preceded by a payload made of less status codes than there are registers to set in the command. In that case, the **Performer** tried to set the registers listed in the command, in order, and failed for some of them. At some point, the processing of the command failed because of one of the following reason:

    - The type of a register is not known. Since the **Performer** does not know the type of the register to set from its identifier, it cannot know the length of its value and thus it cannot decode the remainder of the command message. This issue is reported with the `BAD_DATA_TYPE` status code;
    - The length of a Raw Structure parameter does not fit into the remainder of the payload of the command message. Since the **Performer** cannot know where the structure actually stops, it cannot decode the remainder of the command message. This issue is reported with the `WRONG_STRUCT_FORMAT` status code.

    In any of these cases, the decoding of the command stops at the first occurrence of such an error. Any register that would appear in the command after the one that caused the error is not assigned the value set in the command.

Refer to the status codes table above for a list of possible status codes. Also, for all details about registers and their identifier, refer to the Registry section.

<tt>GET_REGISTER</tt> command

Purpose:

The `GET_REGISTER` command requests to get the values of one or more registers from the STM32 embedded application.

Command ID: `0x0002`.

Command Payload:

The payload of this command is the concatenation of the identifiers of the registers which values are requested. The figure below depicts this.

As registers identifiers are 16 bits in length, the size of the payload of the command is N × 2 bytes with N being the number of registers (N ≥ 1).

It is the responsibility of the application on the **Controller** side to make sure that the cumulated size of the values of the requested registers fit into the maximum payload size supported by the lower level protocol.

For all details about registers and their identifier, refer to the Registry section.

Response:

The response consists in the concatenation of the values of the registers listed in the command, in the order where they appear in the command, followed by a response code.

If the values of all the requested registers could be read, the response code is `CMD_OK`.

If one of the registers cannot be read, the processing of the command stops at this point. The response then consists in the concatenation of the values of the registers that could be read up to the point where the error occurred, followed by a response code. The response code indicates the cause of the error. Refer to the status codes table above for possible codes.

As an example, consider a `GET_REGISTER` command that requests the reading of 5 registers. If the third register of the command cannot be read because tis value is too long to fit in the response message, the response will consists in the values of the first two registers followed by the `NO_TXSYNC_SPACE` status code.

For all details about registers and their identifier, refer to the Registry section.

User Command

Purpose:

The `USER_COMMAND` command is reserved for the needs of the application. Its command ID is not reserved for another purpose. An application on the Performer side is free to implement it or not for its own purposes.

Command ID: `0x0100`.

Command Payload:

The Payload of this command depends on the embedded application.

Response:

The payload of the response depends on the embedded application.

If the embedded application does not implement the command the response only contains the `USER_CMD_NOT_IMPL` status code and the payload is empty.

The embedded application can define its own status codes for this command provided that they are different from the one defined in the status codes table above.

The Registry service

In MCP, registers are a way to provide the application on the **Controller** side a handle on variables, states and more generally data that belong to the embedded application on the **Performer** side.

This handle, also named **register identifier** in this document is structured around four items, as outlined in the figure below:

- `Motor #`: number of the motor the register refers to. 0 means that the register does target a motor or that it targets all motors;
- `Type`: Information on the size of the value of the Data Element (8, 16, 32, ...). The indication of the type of registers either gives the size of their values or provides a means to get it. See below;
- `Identifier`: *Uniquely* identifies a register of a given type;
- `Semantics`: *Uniquely* identifies the purpose of a register independently of the Motor The combination of the `Identifier` and `Type` fields define the `Semantics` of a register.

The following table defines the different possible values for the `Type` field and their meaning.

Type ID	Purpose	Description

0	*RESERVED*	Unused type value
1	8-bit data	
2	16-bit data	
3	32-bit data	
4	Text string	
5	Raw Structure	
6	*RESERVED*	Unused type value
7	*RESERVED*	Unused type value

Register identifiers are used in the `GET_REGISTER` and `SET_REGISTER` commands to identify the registers these command deal with. They can also be used in other circumstances.

Types 0 and 7 are reserved for a future use and must not be used in the current version of the MCP.

The value of type 1, 2 and 3 registers are 8-bit, 16-bit and 32-bit long respectively.

The values of type 4 registers are NULL terminated text strings. The MCP does not specifies anything about the encoding of the text. The only requirement is that a NULL character indicates the end of the string and the end of the value.

The values of type 5 registers are buffers which sizes are indicated in the first two bytes of the values. This type of often used to store structures. The size of a Raw Structure register buffer is stated in bytes and stored in little endian. See the figure below.

Note that registers may be readable and/or writable through MCP, thanks to the `GET_REGISTER` and `SET_REGISTER` commands. Some registers may event be neither writable nor readable. Such registers are usable with the Datalog service specified below.

The full list of registers depends of the application and can change in time: new registers are added periodically to go with the evolution of the MCSDK. See the Registers section below for a list of some of them.

The Datalog service

The Datalog service offers the periodic recording by the Performer of the values of a number of registers that are then transmitted to the Controller. The Controller can configure the number of registers to record, their identifiers and their sampling frequency, within the limits allowed by the Performer. Recorded values are accumulated and sent by the Performer to the Controller in Datalog messages. This feature allows for sampling registers values in a controlled and coherent way.

The Datalog service uses an asynchronous channel of the lower level protocol. The presence of this service is optional in MCP.

The registers that can be monitored through the Datalog service are a subset of the whole register catalog that depend on the Controller application. These registers are classified into two categories, depending on their update frequency: **High Frequency** registers are updated on the execution periodicity of the High Frequency Task of the MCSDK; **Medium Frequency** registers are updated on the execution periodicity of the Medium Frequency Task of the MCSDK. The Datalog service allows for setting a sampling rate for the High Frequency registers and another sampling rate for the Medium Frequency registers.

The values of the High Frequency registers are recorded as 16-bit values whatever their real type. A configurable shift can be applied on the values of 32-bit registers.

The values of the Medium Frequency registers are recorded as 32-bit values whatever their real type.

The figure below depicts the format of a Datalog messages.

The **Timestamp** field is a 32-bit counter that is incremented on each High Frequency task execution. It can be used to detect if data have been lost between two Datalog messages. Data can be lost either because a Datalog messages is lost or because the Datalog buffers are filled faster than they are transmitted. The value of this field corresponds to the value of the counter at the time the registers of the first block in the packet (see below) are recorded.

The timestamp field is followed by one or more blocks of registers values. A block of registers values is made of an optional sequence of High Frequency registers values followed by an optional sequence of Medium Frequency registers values. The presence of the High Frequency registers sequence in a block depends on the High Frequency registers sampling rate (`HF_SAMPLE_RATE`). That of the Medium Frequency registers sequence depends on the Medium Frequency registers sampling rate (`MF_SAMPLE_RATE`). These two parameters are 8-bit unsigned values. They work with two counters: `HF_SAMPLE_COUNT` and `MF_SAMPLE_COUNT` .

When the Datalog service is configured and activated, both these counters are set to 0. On each execution of the High Frequency task, `HF_SAMPLE_COUNT` is compared to `HF_SAMPLE_RATE`. If they are equal, `HF_SAMPLE_COUNT` is set to 0 and the values of the High Frequency registers are recorded in a Datalog message. Otherwise `HF_SAMPLE_COUNT` is incremented by 1.

Similarly, On each execution of the High Frequency task, `MF_SAMPLE_COUNT` is compared to `MF_SAMPLE_RATE`. If they are equal, `MF_SAMPLE_COUNT` is set to 0 and the values of the Medium Frequency registers are recorded in a Datalog message. Otherwise `MF_SAMPLE_COUNT` is incremented by 1.

This procedure is repeated until the Datalog message is filled up to a level where no further block can fit in it. The Datalog message is then sent and a new one is being prepared.

There are two special values for the `MF_SAMPLE_RATE` parameter: 255 and 254. If `MF_SAMPLE_RATE` is set to 255, then Medium Frequency registers are not recorded and Datalog messages only contain High Frequency registers values. If `MF_SAMPLE_RATE` is set to 254, then the Medium Frequency registers are only recorded once per Datalog message, with the last register block. The following figure illustrates this.

The **Async Datalog ID** field is set to **0**. It identifies the message as a Datalog message: messages sent from the Performer to the Controller on the asynchronous channel of the lower level protocol used to transport them that have an **Async Datalog ID** field (that is: the last but one byte of their content) set to **0** are Datalog messages.

The **Mark** field identifies the configuration of the datalog service that corresponds to the content of the Datalog message. Its value is set by the controller at configuration time. It is an 8-bit unsigned value that cannot be 0. This value is useful when changing from one Datalog configuration to another. When the Controller changes the Datalog service configuration (because it adds a register to record for instance), it should also change the Mark value. It will then know when this new configuration is indeed active when it starts receiving Datalog messages with the **Mark** field set to the new mark value. A Performer must apply a newly received Datalog configuration when starting to fill the next Datalog message. Any Datalog message being in construction at the time the new configuration is received by the Performer must first be filled up as explained above and sent before the new configuration is applied. As a consequence, the Controller may receive up to two Datalog messages with the previous configuration after it has sent a new Datalog configuration.

The MCP provisions for the possibility to configure several instances of the Datalog service, one per active transport link between the Controller and the Performer. However, how this possibility is implemented not specified yet and thus cannot be used in this version of the protocol.

The Datalog is configured thanks to a set of registers `DATALOG_UARTA`, `SHIFT_UARTA`, `DATALOG_UARTB`, `SHIFT_UARTA`, `DATALOG_STLINK` and `SHIFT_STLINK`. Which registers need to be used depend on the physical link (and on the lower level protocol) used to communicate between the Controller and the Performer. Configuring the Datalog service

on a physical link requires the setting of one or two registers a `DATALOG_*` register and the corresponding `SHIFT_`register. With these registers, all the parameters of the Datalog service can be set. Refer to the Registers section for a complete description of these registers.

The Notification service

The Notification service is provisioned, but not fully specified in this version of the protocol, so it is not available in the current version of the protocol.

The Notification service messages use an asynchronous channel of the lower level protocol. The presence of this service is optional in MCP.

Notification messages share part of their format with Datalog messages: they have an **Async Datalog ID** field, located at the same place as the in the Datalog message (last but one byte of the message). The **Async Datalog ID** field of a notification message is set to 1.

Multiple Transport links

The Motor Control Protocol provisions for supporting several connections with the lower level protocols at the same type to one **Performer** device. For instance, a **Controller** can be connected to a **Performer** through a serial port and through the STLink at the same time. A mechanism is to be designed to identify the transport link used to exchange messages.

However, this feature of the MCP is not available in the current version of the protocol.

Transport Layers

The transport layers are protocols on which MCP relies to transport its messages between the Controller and the Performer. Their task is to adapt MCP communication to the physical link being used to transport it.

The protocols of the Transport Layers are connected protocols: For MCP communication to occur using them, a connection between the Controller and the Performer need to be established at their level.

The protocols of the Transport Layers provide at least two channels that are multiplexed over the physical link: a Synchronous channel and an asynchronous channel. The Synchronous channel is used by MCP to transport the messages of the command service. The Asynchronous channel is used by MCP to transport the messages of the Datalog and Notification services. The Synchronous Asynchronous channels are independent: a Datalog message may be sent on the Asynchronous channel between a Command and its Response on the Synchronous channel.

Asymmetric Serial Packet Exchange Protocol

ASPEP Overview

The Asymmetric Serial Packet Exchange Protocol – *ASPEP* in short – is a light-weight protocol designed for controlling STM32 embedded applications that produce real time data, across a serial port.

It is a point-to-point protocol where both sides play different roles and have different traffic shapes. It works in **connected** mode: no payload data can be exchanged between the two sides before a formal connection has been established.

Communication with ASPEP involves two **Hosts**, a **Controller** and a **Performer**, that exchange **Packets**.

The **Controller** is the host that initiates the connection. It is also the one that can send requests to the **Performer** while the latter can only answer them. The **Performer** role is meant for the controlled STM32 application while the **Controller** role is that of the controller application.

A data **Packet** is made of a 4-byte **Header** optionally followed by a **Payload**. When a packet has a payload, the size of this payload is indicated in the header. Different packet types are defined to handle all the possible communication schemes and situations that ASPEP has to face.

To start communicating, the **Controller** initiates a connection that the **Performer** accepts.

The purpose of the connection establishment procedure is to negotiate the values of a set of parameters that are needed for the connection to work. These parameters mainly deal with maximum packet sizes and the support of some protocol optional features.

Once a connection is successfully established, **Controller** and **Performer** can start exchanging useful data.

An ASPEP connection provides three communication channels: The **Synchronous** channel, the **Asynchronous** channel, and the **Control** channel. Channels are identified by the types of packets that are exchanged between the **Controller** and the **Performer**.

On the **Synchronous** channel, the Controller sends requests to which the Performer responds: The Controller starts a transaction by sending a `DATA` packet. The Performer ends the transaction by sending a `RESPONSE` packet. Only the **Controller** can start a transaction and the **Performer** cannot send `RESPONSE` packets outside of a transaction. At most one transaction can be pending at any given time. Both `DATA` and `RESPONSE` packets can carry a **Payload**. The main purpose of the **Synchronous** channel is for sending commands from the Controller to the Performer. The maximum sizes of `DATA` and `RESPONSE` packets are negotiated at connection establishment time, as part of the connection procedure.

The **Asynchronous** channel is unidirectional. It allows the Performer for sending `ASYNC` packets from the Performer to the Controller. `ASYNC` packets carry a payload. The presence of this channel on an ASPEP connection is optional and depends on the parameters negotiated at connection establishment time. The main purpose of the Asynchronous channel is to transfer the real time data produced by the embedded STM32 application (the Performer) towards the Controller in the scope of the Datalog service for instance. The maximum size of an `ASYNC` packet is negotiated at connection establishment time.

The **Control** channel is used to establish and manage the connection. Three packet types are exchanged on this channel: `BEACON`, `PING`, and `ERROR` packets. They are used to implement three important features of ASPEP:

- The connection establishment procedure that involves `BEACON` and `PING` packets.
- The Keep-alive feature, used to check whether the embedded application is responsive. This feature uses `PING` packets.
- The Recovery procedure that handles cases where transmission issues may endanger the maintenance of the connection. Its goals it to allow for recovering the synchronization and thus the communication between the **Controller** and the **Performer**. This procedure may be triggered by the Controller, on its reception of faulty or unexpected packets from the Performer or by the Performer on its sending of an `ERROR` packet. The Performer sends such packets when receiving a faulty or unexpected packet from the Controller. Like the connection establishment procedure, the recovery procedure involves `BEACON` and `PING` packets.

These features are described in full details later in this document.

The three channels are multiplexed over the serial link with different priorities: The **Synchronous** channel has priority over the **Control** channel that has priority over the **Asynchronous** one. This means that if several packets are ready for transmission on either side of a connection, when the serial link is available (not busy), the

packet belonging to the channel with the highest priority will be sent first.

Packets are not interrupted nor split during transmission: a packet which transmission has started will be fully transmitted, even if a higher priority packet is submitted (after the start of the transmission).

The payload of ASPEP packets can be protected by a 16-bit CRC that allows for detecting errors on it. This is an optional feature that is negotiated at connection establishment time.

The parameters negotiated at connection establishment time allow for finely tuning the balance between the features provided by the protocol and its impact on the Flash, the RAM and MIPS consumption of the STM32.

ASPEP hardware requirements

There are currently 2 ways to support ASPEP from hardware point of view. The first one uses a USART coupled to two DMA channels. This configuration is common to every supported STM32 MCU except for STM32C0 in single shunt topology.

One DMA channel is configured for data reception, the other one is configured for data transmission. DMA is used to save time by organising communication in parallel with motor control computations. This allows to reach high baudrates and to plot numerous signals using Datalog.

However, for STM32C0 in single shunt topology, such configuration in not available. The ASPEP communication is thus done using only the USART and byte per byte communication is handled by FW. Data reception and transmission are performed at higher priorities than current sensing, which may hinder motor control computations at high baudrates. However, most of the communication time is spent during data processing, which is performed before the Medium Frequency Task whose priority is the lowest. This implementation was proven functional with a baudrate of 115200 bd/s and 2 simultaneous Datalog signals.

Packet structure and transmission

All ASPEP packets share the same structure, depicted in the following figure.

Figure: ASPEP general packet structure

An ASPEP packet is made of a 4-byte header optionally followed by a Payload that is, in turn, optionally followed by a CRC.

Packet Headers

The content of the **Header** of a packet depends on the type of this packet. However, all packet Headers share a common structure, depicted in the following figure.

Figure: ASPEP packet header structure

Type is a 4-bit field. Its value indicates the type of the packet.

The 4-bit CRCH field contains the result of a cyclic redundancy check computation performed on the first 28 bits of the header. The generator polynomial used compute it is that of the CCITT-G.704 standard:

$$x^{4} + x + 1$$

The 28 first bits of the Header are split into 7 nibbles that are processed from the least significant to the most significant one as follows:

- The initial CRC value is set to 0.
- the least significant (4-bit) nibble is processed first as if it were the most significant part of the dividend.

- the order of bits in each nibble is unchanged for processing which would leads to the following bit processing sequence: 3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12, 19, 18, 17, 16, 23, 22, 21, 20, 27, 26, 25, 24.

The structure of the 24 bits of the ***Header Content*** part is defined with each different packet types.

Packet payload and CRC

The Payload of a packet is a sequence of bytes submitted by the user for transmission across an ASPEP connection. Whether a packet accepts a Payload depends on its type.

Packets that accept a Payload can optionally feature a 16-bit CRC field containing the result of a cyclic redundancy check computation performed on the whole payload.

Support for this CRC field is negotiated at connection establishment time.

The generator polynomial used to compute if is that of the CRC-16 from CCITT-X.25 standard:

$$x^{16} + x^{12} + x^{5} + 1$$

The initial value of the CRC, at start of computation, is set to 0. Bytes of the payload are processed in submission order.

Packet transmission

The transmission of an ASPEP Packet starts with the Header. Least significant bits are transmitted first.

If the packet has a payload, an **Intra Packet Pause** delay is inserted after the last bit of the Header has been transmitted and before the first bit of the Payload is sent. The duration of this pause, T_{IPP}, is not specified here. It would typically be **1 ms** in the **Controller** to **Performer** direction and **0 ms** in the **Performer** to **Controller** direction. This pause is sized so that the Performer has the time to properly set the hardware configurations it needs in order to receive the payload with a minimum CPU load expenditure. As an example, this pause should let enough time for the Performer to process the Header, to determine whether a payload is to be expected and, if it is the case, to configure the UART and its attached DMA properly to receive that payload. See

The payload is transmitted byte per byte, least significant bit first for each byte.

If the payload has a CRC, it is transmitted immediately after the end of the payload, least significant bits first also.

Packet Types

ASPEP defines six packet types:

- BEACON
- PING
- ERROR
- REQUEST
- RESPONSE
- ASYNC

Packets of type `BEACON`, `PING` and `ERROR` constitute the **Control** channel. `BEACON` and `PING` packets can be sent by both the **Controller** and the **Performer**. They are involved in the Connection procedure and the Recovery procedure. The `PING` packet is also used in the Keep Alive procedure. The `ERROR` packet is only sent by the **Performer** to signal its receiving an unexpected or erroneous packet to the **Controller**. `BEACON`, `PING` and `ERROR` packets do not

have a payload. They made of the **Header** alone.

Packets of type REQUEST and RESPONSE make the **Synchronous** channel. Both packet types accept a payload. They are involved in Synchronous transactions. REQUEST packets are sent by the **Controller**, RESPONSE packets are sent by the **Performer**.

Packets of type ASYNC constitute the **Asynchronous** channel. They carry a payload and are only sent by the **Performer**.

BEACON packet type

BEACON packets are used by both **Controller** and **Performer** to exchange and negotiate the parameters of the connection – see the Connection procedure. They are also used in the Recovery procedure. BEACON packets do not have a payload.

The figure below describes the structure of a BEACON packet.

Figure : BEACON packet structure

The Type field of BEACON packets is set to **5**.

The value of the **Version** field is the version number of the ASPEP protocol to which the sender of the message conforms. The current version of the protocol has number **0**.

The other fields of BEACON packets describe the connection parameters negotiated by the sender of the message:

- CRC: Set to **1** to indicate supports for computing a CRC on the payload of ASPEP packets. Set to **0** otherwise.
- RXS Max: Represents the maximum payload size of a REQUEST packet allowed on the connection. The maximum payload size in bytes is derived from the value of the field with the following formula:

$$Max\ REQUEST\ payload\ size = P_{REQMax} = (RXS\ Max + 1) \times 32$$

- TXS Max: Represents the maximum payload size of a REPONSE packet allowed in the connection. The maximum payload size in bytes is derived from the value of the field with the following formula:

$$Max\ RESPONSE\ payload\ size = \ P_{RESPMax} = (TXS\ Max + 1) \times 32$$

- TXA Max: Represents the maximum payload size of an ASYN packet allowed in the connection. The maximum payload size in bytes is derived from the value of the field with the following formula:

$$Max\ ASYNC\ payload\ size = \ P_{ASYNCMax} = (TXA\ Max)\times64$$

PING packet type

PING packets are used by both Controller and Performer in the Connection and Keep-Alive procedures. They allow the Performer to report state information and physical link identification to the Controller. PING packets do not have a payload.

The Figure below describes the structure of a PING packet.

Figure: PING packet structure

The Type field of PING packets is set to **6**.

The Controller does not use the C, N and LIID fields. In PING packets sent by a Controller, these fields are reserved and should be set to **0**.

The C field is set to **1** by the Performer to indicate that it is already configured and to **0** otherwise (A Performer is configured when it is not in the IDLE state; see the Connection procedure section below). This information can be used by the Controller to detect a reset of the Performer.

The N field is set, by the Performer to the last bit of the number of the next expected REQUEST packet from the Performer. This number is set to **0** at ASPEP reset time and incremented on each valid reception of a REQUEST packet from the Controller. See the Synchronous Transactions section below. This information can be used by the Controller to detect the loss of a REQUEST or a RESPONSE packet in some situations.

The C and N field are written twice in the packet, as shown in the figure above.

The LIID field identifies the ASPEP Layer instance (and the physical serial interface) on which the packet is sent. **LIID** stands for **Layer Instance IDentifier**. An embedded STM32 application can accept several parallel connections to the same Controller. This field allows for identifying which ASPEP instance / serial port pair sent the packet. Specification of the values of the LIID field is outside of the scope of the specification of ASPEP.

The Packet Number field is used in the Keep-Alive procedure. It is set by the Controller to to a value that is incremented on each PING packet it sends. When sending a PING packet, the Performer sets it to the value of the last Packet Number received from the Controller.

ERROR packet type

ERROR packets are used by the Performer to notify protocol errors to the Controller. An ERROR packet is sent on the reception of an incorrect ASPEP packet from the Controller. ERROR packets do not have a payload.

The Type field of ERROR packets is set to **15**.

The Figure below describes the structure of an ERROR packet.

Figure : ERROR packet structure

The 8-bit **Error Code** field is duplicated in the packet. The table below lists possible error code and their meaning.

Code	Description
1	*Bad Packet Type*: the type of the packet sent by the Controller is either unknown or not allowed.
2	*Bad Packet Size*: The payload of the last REQUEST packet sent by the Controller is too large: the value of its Payload Length field is greater than the RXS Max parameter negotiated on the connection.
4	*Bad Header*: Verification of the CRCH field of the last packet received from the Controller failed.
5	*Bad Payload CRC*: Verification of the CRC field of the last packet received from the Controller failed.

Table : MCP Commands status codes

REQUEST and ASYNC packet type

REQUEST and ASYNC packets have the same Type field value and both accept a Payload. REQUEST packets are sent by the Controller only while ASYNC packets are only sent by the Performer.

The Figure below describes the structure of REQUEST and ASYNC packets.

Figure: REQUEST / ASYNC packet structure

The **Type** field of REQUEST and ASYNC packets is set to **9**.

The value of the **Payload Length** field is set to the number of bytes of the Payload part of the packet. The CRC

field is not part of the payload and is only present if negotiated at connection establishment time. In addition, if the Payload size is **0**, no CRC field is transmitted even if negotiated on the connection. The total length of a `REQUEST` or an `ASYNC` packet, in bytes, is:

- 4 + **Payload Length** + 2, if a CRC field is present.
- 4 + **Payload Length**, if no CRC is present.

`REQUEST` and `ASYNC` packets are used to transmit data buffers submitted by users of the protocol. These data buffers cannot be empty, so the **Payload Length** field of these packets should not be **0**. However, the Controller can send `REQUEST` packets with no payload (and thus a **Payload Length** field set to **0**) in order to request the retransmission of the last `RESPONSE` packet sent by the Performer. See the Synchronous Transactions section for more details.

REPONSE packet type

`REPONSE` packets are only sent by the Performer and they have a payload. They are involved in synchronous Transactions. See the Synchronous Transactions section for more details.

The Figure below describes the structure of a `REPONSE` packet.

Figure : `RESPONSE` packet structure

The **Type** field of `RESPONSE` packets is set to **10**.

The value of the **Payload Length** field is set to the number of bytes of the Payload part of the packet. The CRC field is not part of the payload and is only present if negotiated at connection establishment time. The total length of a `RESPONSE` packet, in bytes, is:

- 4 + **Payload Length** + 2, if a CRC field is present.
- 4 + **Payload Length**, if no CRC is present.

`RESPONSE` packets are used to transmit data buffers submitted by users of the protocol. These data buffers cannot be empty, so the Payload length field of these packets must not be **0**.

Connection Procedure

Prior to exchanging data across ASPEP, two hosts must connect. The connection procedure describes how such a connection is established.

The need for establishing a connection comes from the fact that both hosts can have different capabilities. For example, one of the hosts can be a PC that has plenty of RAM and computing power while the other is an STM32 device that is much more limited in both respects. And different STM32 devices have very different capabilities. So, for the communication to be possible, hosts need to know the capabilities of one another, and they need to agree on a usage of these capabilities that both can handle. The main purpose of the ASPEP connection procedure is to let both hosts negotiate this agreement. In addition, the connection procedure also allows the Controller to get some state variables from the Performer that it needs to manage the connection.

Connection Procedure overview

The connection is initiated by the **Controller**. It is triggered by a user submitting a *Connection Request*, while the **Controller** is in the `IDLE` state. If the **Controller** is not in the `IDLE` state, the connection procedure cannot start, and the connection request is aborted.

The connection procedure is made of two phases.

In the first phase, the parameters of the connection are negotiated. This is done by exchanging pairs of BEACON packets: the Controller sends a BEACON packet with values it can support for the parameters and the Performer answers with parameters that both can support. The negotiation ends when a pair of identical BEACON packets is exchanged: The Controller sends a BEACON packet and the Performer responds with the very same BEACON. A Typical parameters negotiation phase then requires the exchange of one or two pairs of BEACON packet: one if the capabilities of the Controller are compatible with that of the Performer, two otherwise. In some cases, it may require more exchanges, for instance when the Controller and the Performer disagree on the version of ASPEP to use.

The purpose of the second phase is to let the Controller identify the link and get some status information from the Performer that are needed for the remainder of the connection.

This procedure is outlined in the figure below, it is described in more details in the following sections.

Connection procedure details

Figure: Connection Procedure

The connection procedure is made of 6 steps that are highlighted in the Figure above:

Step 1: The procedure starts with the Controller sending a BEACON packet. The Controller sets the fields of this BEACON packet according to its capabilities:

- **CRC** is set to **1** if the Controller supports computing, inserting, and checking the **CRC** field of packets with payload. It is set to **0** otherwise.
- **RXS Max** is set according to the Controller's P_{REQMax} parameter reset value
- **TXS Max** is set according to the Controller's $P_{RESPMax}$ parameter reset value
- **TXA Max** is set according to the Controller's $P_{ASYNCMax}$ parameter reset value

The Controller also sets the Version field of the BEACON to the number of the latest version of ASPEP it supports.

The computation of the **RXS Max**, **TXS Max** and **TXA Max** fields is done according to the rules stated in sections Maximum REQUEST payload size, Maximum RESPONSE payload size and Maximum ASYNC payload size.

On sending the BEACON packet, the Controller switches to the CONFIGURING state.

Step 2: On reception of the BEACON packet, the Performer first transitions to the IDLE state (unless it is already in it). In this state, the Performer can only send BEACON and PING packets and it can only do so in order to respond to similar packets sent by the Controller. If a synchronous transaction is pending in the Performer, it is aborted, so the Performer will not send a RESPONSE packet to close it.

The Performer then checks whether the capabilities proposed by the Controller are acceptable. To that end, it first checks the Version field.

If its value is not that of an ASPEP version number it supports, the Performer sends a BEACON packet with the highest ASPEP version number it can handle and sets the other fields according to its own reset values of the P_{CRC}, P_{REQMax}, $P_{RESPMax}$ and $P_{ASYNCMax}$ parameters. It remains in the IDLE state.

If the value of the Controller's Version field is that of an ASPEP version it supports, the Performer checks the compatibility of the other parameters sent by the Controller according to the following criteria:

- P_{REQMax}: Controller's P_{REQMax} ≤ Performer's P_{REQMax}
- $P_{RESPMax}$: Controller's $P_{RESPMax}$ = Performer's $P_{RESPMax}$
- $P_{ASYNCMax}$: Controller's $P_{ASYNCMax}$ = Performer's $P_{ASYNCMax}$
- P_{CRC} : Controller's = Performer's

Parameters sent by the Controller that are compatible with the Performer are kept. If a parameter is incompatible a new, compatible, value is computed for it according to the following indications:

- P_{REQMax}: minimum of Controller's P_{REQMax} and Performer's P_{REQMax}
- $P_{RESPMax}$: minimum of Controller's $P_{RESPMax}$ and Performer's $P_{RESPMax}$
- $P_{ASYNCMax}$: minimum of Controller's $P_{ASYNCMax}$ and Performer's $P_{ASYNCMax}$
- P_{CRC}: set to false if Controller's and Performer's P_{CRC} differ

The Performer then builds and sends a new BEACON packet with the **RXS Max**, **TXS Max** and **TXA Max** parameters set according to compatible values (the ones kept from the Controller and the newly computed ones) of the parameters and the same Version field as in the Controller's BEACON.

If the BEACON sent by the Performer is identical to the one received from the Controller, the Performer switches to the CONNECTED state. The Performer is then ready to exchange data with the Controller. From that point on, it can send ASYNC packets and respond to REQUEST packets.

Step 3: When the Controller receives the BEACON packet returned by the Performer, it first checks if this BEACON is identical to the one it sent in step 1. If that is the case, the Controller switches directly to the CONNECTING state and the procedure carries on from step 5.

If that is not the case, two possibilities are at stake: either the Version field returned by the Performer differs from that sent by the Controller or it is identical.

If the Version fields differ, the connection procedure restarts from the beginning (step 1) with the Version value sent by the Performer, provided that the Controller supports it. If the Controller does not support the protocol version proposed by the Performer, either the connection aborts or the Controller can try again with a lower version number, that it supports, if any exists.

If the Version fields are the same, the Performer has sent its own proposal for the parameters and their values should be acceptable to the Controller. The Controller checks if these values are indeed compatible with its own. If they are not, it computes compatible values and sends a new BEACON packet and the procedure carries on from step 2. If they are compatible, the Controller send a new BEACON packet identical to the one received by the Performer.

Step 4: On reception of the BEACON packet from the Controller, the Performer acts as in Step 2: It sends the BEACON packet back to the Controller with compatible parameters. If this BEACON is identical to the one received from the Controller, the Performer switches to the CONNECTED state.

Step 5: In this step, the Controller expects to receive a BEACON packet that is identical to the one it previously sent. If this is not the case, the connection procedure carries on from step 3.

On reception of an identical BEACON packet, the Controller switches to the CONNECTING state and sends a PING packet. The purpose is to request the status of the Performer and the identification of the link.

Step 6: On reception of the PING packet from the Controller, the Performer answers with a PING packet with the following field values:

- **C**: set to **1** unless the Save device has been reset between the end of step 4 and the reception of the PING packet from the Controller
- **N**: set to the bit **0** of the N_{NREQ} counter of the Performer. **N** is **0** if N_{NREQ} is even and 1 if N_{NREQ} is odd.
- **LIID**: set to the Layer Instance Identifier attached to the link on the Performer. For more details on this parameter, see section Layer Instance IDentifier

When the Controller receives this PING packet, it switches to the CONNECTED state, unless the C field is 0, and

records the values of the fields:

- **N** is used to initialize the value of the N_{NREQ} counter of the Controller. See the Next Request Packet Number section for more details
- **LIID** is saved to be made available to the user of this ASPEP instance so that it can identify the link it is working on.

If the **C** field of the received PING is set to **0**, it is up to the Controller to either abort the *Connection Request* or retry it.

BEACON and PING repetitions

The Controller is free to initiate the connection procedure when it wants. Then, the Performer may be unavailable at the time the first BEACON packets are sent (it may be Off, or halted on a breakpoint, or the link quality may be temporarily low…). In that case, it will not receive this packet, and the connection cannot be established.

To solve this issue, the Controller keeps on sending BEACON packets periodically until it gets a BEACON answer from the Performer. This BEACON transmission periodicity is a Controller side parameter named T_{BEACON}. Its value is set by and on the Controller side only and is not negotiated with the Performer. It should be set at a value that is reasonably higher than the expected round trip time between the Controller and the Performer. See the section BEACON packets transmission period below for more information on this parameter.

This BEACON retransmission scheme is used whenever the Controller sends a BEACON packet, whether it is the first BEACON filled with Controller's capabilities or a later BEACON packet sent with merged capabilities.

Figure : Connection procedure with BEACON and PING repetition

Similarly, the Performer may not fail receiving PING packets that are sent in the second phase of the connection procedure. To overcome this issue, the Controller keeps on sending PING packets periodically until it gets a PING packet as answer from the Performer. The PING transmission periodicity is a Controller side parameter named T_{PING}. Just like for the T_{BEACON} parameter, its value is set by and on the Controller side only and is not negotiated with the Performer. It should be set at a value that is reasonably higher than the expected round trip time between the Controller and the Performer. See section Keep-Alive packets transmission period below for more information on this parameter.

These BEACON and PING retransmission schemes are not specific to the connection procedure. They are applied each time the Controller sends a BEACON or a PING packet. Note that the Performer only sends BEACON and PING packets to answer BEACON and PING packets sent by the Controller, respectively.

Error handling

Packets can be badly received, by the Controller or the Performer. During the connection procedure, this can be either because the **CRCH** field of the received packet is wrong or because the type of the packet is unknown or unexpected at that stage.

When the **Performer** receives such a packet, it answers with an ERROR packet which **Error Code** field is either *Bad Header* (4) if the **CRCH** field is wrong or *Bad Packet Type* (1) if the **CRCH** field is correct, but the packet type is unknown or unexpected. Upon reception of an ERROR packet during the connection procedure, the Controller simply resends the last packet it sent whether it was a BEACON or a PING.

Figure: Errors in the connection procedure

When the Controller receives a bad packet, during the connection procedure, it restarts the Connection

Procedure from the beginning which is equivalent to initiating a Recovery Procedure (see below).

Recovery Procedure

The goal of the recovery procedure is is to resynchronize the the communication between the Controller and the Performer. It is initiated by the Controller after its receiving any of the following:

- A packet with an incorrect **CRCH** field (Bad Header)
- A packet with a wrong **Type** field (Bad Type)
- A packet with a wrong **Payload Length** field (Bad Length)
- An `ERROR` packet from the Performer indicating either a Bad Header, a Bad Type or a Bad Length error.

In any of these conditions, either the Controller or the Performer is unable to determine the starting bit of the next packet it will receive. Indeed a malformed packet Header has been received that can be followed by a payload which size is unknown. So, such a packet header can be followed by an unknown number of bytes before a new packet it sent and the receiving side cannot know where to look for the start of the next packet. The Recovery procedure is meant to recover from this issue.

The following figure illustrates the Recovery Procedure.

The Controller starts sending `BEACON` packets with the T_{BEACON} periodicity. The capabilities proposed in these packets are the ones previously negotiated with the Performer. The Controller keeps on sending these until it receives an identical `BEACON` packet in return.

When the Controller sends the first `BEACON` packet of the recovery procedure, the Performer may still be transmitting. In that case, the `BEACON` sending periodicity can be relaxed. It is possible (even though the probability is low) that the data that the Performer is sending at that time contains a sequence of byte that is identical to the `BEACON` packet expected by the Controller. For this reason, once the Controller has received the expected BEACON packet from the Performer, it waits for a period, $T_{SILENCE}$, during which the Performer must not send anything. Then, the Controller sends another `BEACON` packet, identical to the first one. Once the Performer has answered it, the recovery procedure ends and the two devices can resume their communication.

If the Performer sends anything during the $T_{SILENCE}$ period, the Controller restarts the Recovery Procedure from the beginning.

The $T_{SILENCE}$ period should be chosen so that it lets enough time for the Performer to prepare **and** send its biggest possible packet. See

On receiving a `BEACON` packet while it is already connected, the Performer must do the following:

1. If it is transmitting a packet, it carries on transmitting it to the end. If this packet is a `RESPONSE` packet, the synchronous transaction is complete.
2. answer the `BEACON` packet with an identical `BEACON` packet
3. If a synchronous transaction is pending (its `RESPONSE` packet has not been sent), it is aborted and the `RESPONSE` packet will not be sent.
4. Any packet that would have been submitted for transmission by the upper level protocol is discarded and will not be sent

1. The upper level protocol using ASPEP is notified that a recovery procedure is on-going and that any service that uses the Asynchronous channel must be reset. The Controller needs to reconfigure any such service before `ASYNC` packets can be sent again. on the connection.

On initiating a Recovery Procedure, the Controller notifies the upper level protocol using ASPEP of the fact. It also notifies it when the the Recovery Procedure has completed so that any service that uses the Asynchronous channel can be reconfigured.

The Controller may decide to close the connection after a number of restarts of the Recovery Procedure. This number is not specified and depends on the implementation.

Keep-Alive procedure

The Keep-Alive procedure assesses the availability of the connection and provides statuses from the Performer. It is triggered by the Controller on its own initiative and on criteria that are not specified by this document and an implementation choice. But the intention with the Keep-Alive procedure is to use it to check that the Performer is still online and connected. So, it would rather be used if and when no packets have been exchanged between the Controller and the Performer for a while.

The figure below illustrates the Keep-Alive procedure in its typical course:

The Controller is sending a `PING` message to start the Keep-Alive procedure. The Performer shall answer it with a `PING` message, which terminates the procedure.

If the Performer does not answer the `PING` message from the Controller, the controller would keep on sending `PING` packets with a T_{PING} periodicity until either the Performer answers or the Controller decides that the Performer is not active anymore and that the connection is closed. This latter decision is an implementation choice on the Controller side and it is not specified in this document. The following figure depicts this situation.

The Keep-Alive procedure can also be used outside of a connection. It is then used to check the presence of a Performer and query its status.

The Keep-Alive procedure can be terminated by the Performer sending an `ERROR` message. This occurs when the Performer fails to properly decode a `PING` packet sent by the Controller. Also, it can be terminated if the Controller fails to properly decode a `PING` packet received from the Performer. If either of these situation occur while a connection is active, a Recovery Procedure is initiated by the Controller. This document does not specify what should happen if any of these situations occur outside of a connection.

Synchronous Transactions

A Synchronous transaction consists in the sending by the Controller of a `REQUEST` packet that is answered by the Performer with a `RESPONSE` packet. Packets of type `REQUEST` and `RESPONSE` constitute the Synchronous channel. Synchronous Transactions can only occur once a connection is established. Both `REQUEST` and `RESPONSE` packet transport a payload. Synchronous transaction are a way for the Controller to send commands to the Performer and for the Performer to send data as a reply to these commands.

The figure below illustrates a Synchronous transaction.

A Synchronous transaction is always initiated by the Controller. It does this by sending a `REQUEST` packet. The Performer shall send exactly one `RESPONSE` packet as an answer to the `REQUEST` packet. The sending (and the reception) of this `RESPONSE` packet terminates the transaction.

At most one transaction can be on-going on an ASPEP connection. So, the Controller needs to wait the end of an on-going transaction before it can start another one.

The Performer shall not send unsolicited `RESPONSE` packets.

A synchronous transaction can also be terminated by the Performer sending an `ERROR` message with the Bad Header, Bad Packet Type or Bad Packet Size error code. This occurs when the Performer fails to properly decode the Header of a `REQUEST` packet sent by the Controller. It can also be terminated if the Controller fails to properly decode the Header of a `RESPONSE` packet received from the Performer (Bad Header, Bad Packet Type or Bad Packet Size errors). If either of these situation occur, a Recovery Procedure is initiated by the Controller.

If the CRC verification on the Payload of the `REQUEST` packet fails, the Performer sends an `ERROR` packet with the *Bad Payload CRC* error code. The Master shall then retransmit the `REQUEST` packet.

If the CRC verification on the Payload of the `RESPONSE` packet fails, the Controller can send an empty `REQUEST` packet (payload size set to 0) to request the retransmission, by the Performer of this `RESPONSE` packet.

A CRC verification on the Payload is mandatory on either side of the connection if the P_{CRC} parameter is negotiated to `true` at connection establishment time.

Asynchronous data transfers

The Performer can send unsolicited data to the Controller by sending `ASYNC` packets. Packets of type `ASYNC` can only be sent by the Performer, and when a connection is established. They constitute the Asynchronous channel.

There is no way for the Controller to request the retransmission of an `ASYNC` packet in case the verification on the Payload of one of these packets would fail. A CRC verification on the Payload is mandatory on the Controller if the P_{CRC} parameter is negotiated to `true` at connection establishment time.

`ASYNC` packets are a way for the Performer to send large amount of real time data to the Controller.

Errors handling on the Performer

The processing of errors is done differently on the Controller and on the Performer. As the Controller has the initiative, it can trigger a recovery procedure if need be or take any other action needed to keep the connection active and working.

The Performer, however, cannot initiate any procedure. The only thing it can do when it comes across an erroneous situation is sending an `ERROR` packet to the Controller. On any error detected on the reception of a packet from the Controller (see the ERROR packet type), and at any time, the Performer can send an `ERROR` packet.

When this occurs during the normal operations of a connection, the handling of `ERROR` packets by the Controller is defined in the Keep-Alive procedure, Synchronous Transactions and Asynchronous data transfers sections above.

When an `ERROR` packet is sent in the course of a Connection Procdecure or in that of a Recovery Procedure, the whole procedure in which this occurs is restarted.

ASPEP Parameters Details

As described above, the configuration of an ASPEP connection depends on a set of parameters. These parameters are negotiated at the very start of the connection, as part of the Connection Procedure. Each host has its own value – the *reset* value – for each of them, that indicate its level of support for the related feature. When the negotiation of a connection procedure completes successfully both hosts share a common, mutually acceptable, value for each of these parameters – the *connection* value.

Some other parameters are not negotiated and their values are defined on a per-application basis. It is then expected that sensible values are chosen by the application. In these cases, a hint is provided in this document to explain how to choose a value.

Payload CRC Support

In the context of an established connection, the *Payload CRC support* parameter, P_{CRC}, indicates whether a CRC is computed on the payload of `REQUEST`, `RESPONSE` and `ASYNC` packets and transmitted with them.

If P_{CRC} is true, each packet with a payload is sent with a CRC field computed as described in section Packet payload and CRC above. Hosts shall then check the validity of the CRC field of each received payload.

If P_{CRC} is false, no CRC field is sent with the payloads and thus, no payload validity check can be performed.

Maximum REQUEST payload size

In the context of an established connection, the *Maximum REQUEST payload size* parameter, P_{REQMax}, is the maximum size of the payload of a `REQUEST` packet that can be sent on this connection. Put another way, it is the maximum size of a payload that a Performer will receive (and that a Controller will send) on the Synchronous channel of that connection.

Outside of an established connection context, P_{REQMax} is the maximum size of the payload of a `REQUEST` packet that a Host can handle. It is typically the size of buffers allocated to sending or receiving the payload of `REQUEST` packets in a Controller or a Performer respectively.

P_{REQMax} allowed values are multiples of 32 bytes, ranging from 32 to 2048 inclusive.

The value of the P_{REQMax} connection parameter is negotiated with the RXS Max field of `BEACON` packets. The relationship between P_{REQMax} and RXS Max is:

$$P_{REQMax} = (RXS\ Max + 1) \times 32$$

the RXS Max field is 6 bits wide allowing for a maximum value of 63. An RXS Max of 0 specifies a P_{REQMax} of 32 bytes and an RXS Max of 63 specifies a P_{REQMax} of 2048 bytes.

Maximum RESPONSE payload size

In the context of an established connection, the *Maximum RESPONSE payload size* parameter, $P_{RESPMax}$, is the maximum size of the payload of a `RESPONSE` packet that can be sent on this connection. Put another way, it is the maximum size of a payload that a Performer will send (and that a Controller will receive) on the Synchronous channel of that connection.

Outside of an established connection context, $P_{RESPMax}$ is the maximum size of the payload of a `RESPONSE` packet that a Host can handle. It is typically the size of buffers allocated to sending or receiving the payload of `RESPONSE` packets in a Performer or a Controller respectively.

$P_{RESPMax}$ allowed values are multiples of 32 bytes, ranging from 32 to 4096 inclusive.

The value of the $P_{RESPMax}$ connection parameter is negotiated with the TXS Max field of `BEACON` packets. The relationship between $P_{RESPMax}$ and TXS Max is:

$$P_{RESPMax} = (TXS\ Max + 1)\times 32$$

the TXS Max field is 7 bits wide allowing for a maximum value of 127. A TXS Max of 0 specifies a $P_{RESPMax}$ of 32 bytes and a TXS Max of 127 specifies a $P_{RESPMax}$ of 4096 bytes.

Maximum ASYNC payload size

In the context of an established connection, the *Maximum ASYNC payload size* parameter, $P_{ASYNCMax}$, is the maximum size of the payload of an `ASYNC` packet that can be sent on this connection. Put another way, it is the maximum size of a payload that a Performer will send on the Asynchronous channel of that connection.

Outside of an established connection context, $P_{ASYNCMax}$ is the maximum size of the payload of an ASYNC

packet that a Host can handle. It is typically the size of buffers allocated to sending or receiving the payload of ASYNC packets in a Performer or a Controller respectively.

$P_{ASYNCMax}$ allowed values are multiples of 64 bytes, ranging from 0 to 8128 inclusive.

The value of the $P_{ASYNCMax}$ connection parameter is negotiated with the TXA Max field of `BEACON` packets. The relationship between $P_{ASYNCMax}$ and TXA Max is:

$$P_{ASYNCMax} = (TXA\ Max)\times 64$$

the TXA Max field is 7 bits wide allowing for a maximum value of 127. A TXA Max of 0 specifies a $P_{ASYNCMax}$ of 0 byte and a TXA Max of 127 specifies a $P_{ASYNCMax}$ of 8128 bytes. In other words, a value of 0 means that the Asynchronous channel is either not negotiated in the current connection or not supported by the Host depending on the context in which the $P_{ASYNCMax}$ is evaluated. If $P_{ASYNCMax}$ is negotiated to 0 in a connection, the Performer is not allowed to send ASYNC packets in that connection and thus, all services relying on `ASYNC` packets are disabled for the duration of the connection.

Intra Packet Pause

T_{IPP}, the Intra Packet Pause is a time interval inserted between the end of the transmission of the Header and the start of the transmission of the payload of an ASPEP packet.

This pause is meant to relax real time constraints that the implementation of ASPEP faces on the Performer side. In this context, a good value for this parameter is greater than the firing period of SysTick interrupt. As an example, in the Motor Control SDK, the ASPEP protocol management routines are executed on that interrupt on the Performer. Then, having a pause greater than this period between the end of the Header and the beginning of the Payload of packets sent to the Performer ensures that the implementation has the opportunity to decode the Header and configure the DMA channel linked to the serial port for the reception of the payload. The use of a DMA channel to read the data received on the serial port is an easy enhancement over the legacy version of the Motor Control Protocol.; It is also one that significantly reduces the CPU power consumption needed to handle the communication while allowing for a much higher data throughput in the Controller to Performer direction.

A T_{IPP} value of 1 ms, in the Controller to Performer direction seems to be a good value. It is the one that is expected by default in the firmware of the MCSDK. In the Performer to Controller direction, the default value used by the firmware is 0 ms. The firmware considers that the Controller is hosted on a processor that has sufficient CPU power to allow for this and it does not insert the Intra Packet Pause.

In the current version of the protocol, this parameter is **not** negotiated between the Controller and the Performer.

BEACON packets transmission period

The T_{BEACON} parameter is the time interval between the sending of two `BEACON` packets by the Controller. It is used both for the Connection Procedure and for the Recovery Procedure.

In the current version of the protocol, this parameter is not negotiated between the Controller and the Performer.

This parameter should not be seen as a strict periodicity. It's value should rather be a compromise between workload and reactivity. A sensible value for this parameter is one that is reasonably higher than the expected round trip time between the Controller and the Performer. Such a value needs to consider the throughput of the serial link.

Keep-Alive packets transmission period

The T_{PING} parameter is the time interval between the sending of two `PING` packets by the Controller. It is used both for the Connection Procedure and for the Keep-Alive procedure.

This parameter is not negotiated between the Controller and the Performer.

Like the T_{BEACON} parameter, it should not be seen as a strict periodicity but more as a compromise. A sensible value for this parameter is one that is reasonably higher than the expected round trip time between the Controller and the Performer, in the case of the Connection Procedure. Such a value needs to consider the throughput of the serial link.

Recovery procedure Silence period

The Recovery procedure silence period, $T_{SILENCE}$ aims at making as sure as possible that the Controller and the Performer are synchronized. So, the silence period should be significantly longer than the time it takes to transmit the biggest possible packet from the Performer (usually the $P_{ASYNCMax}$, but it may also be $P_{RESPMax}$) plus the time it would take for the Performer to fill such a packet. In the case of the usage of `ASYNC` packets by MCP's Datalog service, this can easily be done.

The fact the Performer does not send any data during this period followed by its responding to a `BEACON` from the Controller is considered a reliable indication that both devices are synchronized. if $T_{SILENCE}$ is too short, the indication is not reliable. If it is too long, the connection performance penalty suffers (and the indication reliability may also be impacted if the Performer may enter a sleep state for instance).

This parameter is not negotiated between the Controller and the Performer.

Layer Instance Identifier

The **LIID** field of `PING` identifies the transport layer on which the connection takes place. ASPEP provisions for two concurrent serial port connections taking place at the same time between a Controller and an Performer, named `UART_A` and `UART_B`. The LIID field is set to 0 when the `PING` packet is transported across the `UART_A` serial port and to 1 when the `PING` packet is transported across the `UART_B` serial port.

ASPEP implementations offer an API for the MCP implementation that sits on top of them to know which Transport Layer is connected. This is useful when configuring the DataLog service for instance.

Next Request packet Number

The N_{NREQ} counter (Next Request Packet Number) is maintained on both sides of the connection. It contains the number of REQUEST packet sent (on the Controller side) or received (on the Performer side). The Controller increments this counter when it has successfully sent a REQUEST packet. The Performer increments this counter when it has successfully received a REQUEST packet. For the Controller, checking the value of the **N** field of a PING packet received from the Performer can help understanding the status of the last transaction before a Recovery Procedure.

ASPEP Versions

The ASPEP version as transported in the **VERSION** field of `BEACON` packets is a monotonous number that is incremented on each update of the ASPEP protocol. An implementation of the ASPEP protocol on the Controller side should support all the versions of the protocol up to (and including) the one it advertises in the first `BEACON` packet it sends for establishing a connection. On the Performer side, an implementation of the ASPEP protocol shall support at least one version of the protocol. It can optionally support earlier versions.

The first byte of a BEACON packet is guaranteed to be stable across versions of the protocol. This first byte contains the Type and the VERSION fields. Stable means that the Type field will always be 5 and the version will

always be three bits. If later versions of the protocol need to change the header structure of the packets, these two fields will anyway keep their original meaning.

STLink Exchange Protocol

STLEP Overview

The STLink Exchange Protocol – in short STLEP – is a buffer exchange protocol designed on top of the STLink USB protocol. The STLink feature provides external devices with read and write access to the whole the memory of a target STM32 MCU.

STLEP uses this capability to exchange data buffers between two Hosts, one of which is an STM32 MCU. For more information on STLink, see https://wiki.st.com/stm32mpu/wiki/ST-LINK.

STLink is a half-duplex protocol in which the target STM32 MCU plays a totally passive role. It cannot trig any action from the controller. In this context, the RAM of the STM32 should be seen as an exchange area.

The STLEP protocol is not available in the version of the Motor Control Protocol Suite described in this document.

Registers and Applicative MCP Commands

Motor Control and other Commands

The START_MOTOR command

Purpose:

The `START_MOTOR` command sets the target motor under control. The motor may start rotating as a result of the execution of this command.

Command ID: `0x0003`.

Command Payload:

This command does not have a payload. The id of the target motor is indicated by the **Motor #** field of the command.

Response:

The Payload of the response is empty. If the command can be executed, the `CMD_OK` response code is returned. Otherwise, the `CMD_NOK` response code is returned.

Note that the Response is sent as soon as the command is scheduled for execution. So, returning the `CMD_OK` recode does not mean that the motor is indeed under control. It only means that the start up procedure is about to be executed.

The STOP_MOTOR command

Purpose:

The `STOP_RAMP` command sets the target motor out of control. The motor may stop rotating as a result of the execution of this command.

Command ID: `0x0004`.

Command Payload:

This command does not have a payload. The id of the target motor is indicated by the **Motor #** field of the command.

Response:

The Payload of the response is empty. If the command can be executed, the `CMD_OK` response code is returned. Otherwise, the `CMD_NOK` response code is returned.

Note that the Response is sent as soon as the command is scheduled for execution. So, returning the `CMD_OK` recode does not mean that the motor is indeed out of control. It only means that the stop procedure is about to be executed.

The STOP_RAMP command

Purpose:

The `STOP_RAMP` command command stops any ramp under execution on the target motor, whether a speed or a torque ramp, if any is under execution at the time of sending.

Command ID: `0x0005`.

Command Payload:

This command does not have a payload. The id of the target motor is indicated by the **Motor #** field of the command.

Response:

The Payload of the response is empty. The command always returns `CMD_OK`.

The START_STOP command

Purpose:

The `START_STOP` command either starts or stops the target motor. If the motor is idle, the command behaves like the `START_MOTOR` command, otherwise it behaves like the `STOP_MOTOR` command.

Command ID: `0x0006`.

Command Payload:

This command does not have a payload. The id of the target motor is indicated by the **Motor #** field of the command.

Response:

The Payload of the response is empty. If the command can be executed, the `CMD_OK` response code is returned. Otherwise, the `CMD_NOK` response code is returned.

See the `START_MOTOR` and `STOP_MOTOR` commands above for more details.

The FAULT_ACK command

Purpose:

The `FAULT_ACK` command acknowledges any past but not yet acknowledged motor control subsystem fault. If such fault condition exists, it is cleared and if no fault condition is currently active, the Motor Control subsystem goes back to the `IDLE` state and a `MOTOR_START` command can be sent to set the motor under control again.

If no past but not yet acknowledged fault exists, nothing is done.

Command ID: `0x0007`.

Command Payload:

This command does not have a payload. The id of the target motor is indicated by the **Motor #** field of the command.

Response:

The command always returns `CMD_OK`.

The USER_CMD command

Purpose:

The `USER_CMD` commands are a set of command codes reserved for application use that will never be allocated to MCP commands. So, users are free to use these code for their own needs. In the ST MCSDK firmware implementation of the MCP, a call-back mechanism is provided for the application to handle them if it needs. Up to 32 such commands can be implemented. See the documentation of ST MCSDK firmware for more details on how to use them in the context of the firmware.

Command ID: `0x0020` to `0x003F`.

Command Payload:

This command have an arbitrary payload that depends on the user's implementation.

Response:

The Payload of the response is arbitrary and depends on the user's implementation. The same is true for possible result codes.

Registers

The registers listed here are the ones that are present on all MCPS implementations and that are needed to set the Datalog service up. Other registers exist that are defined separately.

Registers have a type as specified in The registry service section. This type provides the information needed for transmitting the value of registers but it does not allow for properly interpreting it. To that end, two additional information are provided with each register definition: the format and the unit. The unit is the physical unit of the value a register such as Amperes, Volts or Hertz for example.

The format can be one of the following

- S8: signed 8-bit integer
- U8: unsigned 8-bit integer
- U8ENUM: 8-bit enumeration
- U8FLAGS: 8-bit bitfield

- S16: signed 16-bit integer
- U16: unsigned 16-bit integer
- U16FLAGS: 16-bit bitfield
- S32: signed 32-bit integer
- U32: unsigned 32-bit integer
- U32FLAGS: 32-bit bitfield
- F32: 32-bit floating point number
- Q32: 32-bit Q-format number

Registers as described below have a scope, either Global or Motor. A Global scope means that the register does not target any specific motor. The **Motor #** field of the register identifier of such registers is 0. A Motor scope register is a register that targets a motor. There can be several instances of such a register in an application, differentiated by the **Motor #** field of the register identifier, which is non 0.

The GLOBAL_CONFIG register

Identifier	Type	Scope	Identification Token	Access
0	Raw Structure	Global	`0x28`	Read Only, Constant

This register is a Raw Structure that provides high level, non motor dependent, information about the embedded Motor Control FW application. This allows a remote tool to start discovering the configuration of the embedded application.

The following table describes the content of the structure.

Size	Name	Description	Unit	Format
32	MCSDKVersion	Motor Control SDK Version number.	N/A	U32
8	MotorNumber	Number of motors driven by the application	N/A	U8
8	MCP Flag	MCP feature flag - bit 0 : MCP Over ST Link - bit 1 : MCP Over UART A - bit 2 : MCP Over UART B	N/A	U8 Flags
8	MCPA UARTA LOG	Number of simultaneous streams supported by the Datalog service on ASPEP with UART A LIID.	N/A	U8
8	MCPA UARTB LOG	Number of simultaneous streams supported by the Datalog service on ASPEP with UART B LIID	N/A	U8
8	MCPA STLNK LOG	Number of simultaneous streams supported by the Datalog service over ST Link	N/A	U8

The **MCSDKVersion** field is structured as follows:

- bits 0-5: Pre-release number (only used if Release type is not public)
- bits 6-7: Release type:
  - 0: Alpha
  - 1: Beta
  - 2: RC
  - 3: Public (official)
- bits 8-15: Patch release number
- bits 16-23: Minor release number
- bits 24-31: Major release number

The **MCP Flag** field is a bitfield with the following meaning:

- bit 0 : MCP Over ST Link
- bit 1 : MCP Over UART A
- bit 2 : MCP Over UART B
- bit 3-7: reserved

The MOTOR_CONFIG register

Identifier	Type	Scope	Identification Token	Access
1	Raw Structure	Motor	`0x69` to `0x6F`	Read Only, Constant

These registers are Raw Structure that contain the parameters of the motors driven by the embedded Motor Control FW application.

The following table describes the content of the structure.

Size	Name	Description	Unit	Format
32	PolePairs	Number of pole pairs of the motor.	N/A	F32
32	RatedFlux	Rated magnetic flux of the motor	V/Hz	F32
32	Rs	Rated resistance of the motor	Ohm	F32
32	RsSkinFactor	Coils skin factor	N/A	F32
32	Ls	Inductance of the Motor (In the Q direction in case of of an anisotropic motor)	H	F32
32	Ld	Inductance of the Motor in the D direction	H	F32
32	MaxCurrent	Maximum rated motor current	A	F32
32	CopperMass	Amount of copper used in the windings of the stator of the motor.	kg	F32
32	CoolingTau	Thermal time constant of copper & direct environment of motor	s	F32
24*8	Name	Name of the motor. A human friendly way to identify it.	N/A	U8

The APPLICATION_CONFIG register

Identifier	Type	Scope	Identification Token	Access
2	Raw Structure	Motor	`0xA9` to `0xAF`	Read Only, Constant

Motor applicative configuration. This register is useful for a remote tool that need to know the operative boundaries of a motor driven by the FW application.

The following table describes the content of the structure.

Size	Name	Description	Unit	Format
32	MaxMechSpeed	Absolute value of the maximum mechnical speed.	RPM	U32
32	MaxReadableCurrent	Absolute value of the maximum current readable by the board. Basically (Vref/(2*Rshunt*OpampGain))	A	F32
16	nominalCurrent	Min of the board current and motor nominal current	S16A	U16
16	nominalVoltage	VBus operating point	V	U16
8	DriveType	Motor Driving technology	N/A	U8ENUM
8	Padding	Padding field to align structure length on 16-bit boundaries	N/A	U8

The drive type enum is defined as follows:

Value	name	description

0	FOC	Field Oriented Control
1	SIX_STEP	Six Step

The FOC_FIRMWARE_CFG register

Identifier	Type	Scope	Identification Token	Access
3	Raw Structure	Motor	`0xE9` to `0xEF`	Read Only, Constant

FOC specific motor drive configuration. This register is present only if the **DriveType** enumeration field of the `APPLICATION_CONFIG` register for the same motor is set to **FOC**. Note that this register has the same identifier as the `SIXSTEP_FIRMWARE_CFG` register that is only present for a Six Step drive.

The following table describes the content of the structure.

Size	Name	Description	Unit	Format
8	PrimSpeedPosSensingType	Primary speed and position sensing type	N/A	U8ENUM
8	AuxSpeedPosSensingType	Auxilliary speed and position sensing type	N/A	U8ENUM
8	CurrentSensingTopology	Current Sensing Topology	N/A	U8ENUM
8	FOCRate	Rate of the FOC loop execution in PWM periods. 1 means FOC loop is run every PWM period; 2 means it is run every other PWM periods…	N/A	U8
32	PWMFrequency	PWM frequency	Hz	U32
16	MediumFrequencyTaskFreq	Medium Frequency Task execution frequency	Hz	U16
16	ConfigurationFlags1	Configuration flags	N/A	U16FLAGS
16	ConfigurationFlags2	Configuration flags	N/A	U16FLAGS

The **PrimSpeedPosSensingType** and **AuxSpeedPosSensingType** enumerations are defined as follows:

Value	name	description
0	NONE	No type used
1	SENSORLESS_STO_PLL	Sensorless, State Observer + PLL
2	SENSORLESS_STO_CORDIC	Sensorless, State Observer + Cordic (Software flavor)
3	QUADRATURE_ENCODER	Quadrature Encoder
4	HALL_SENSORS	Hall effect sensors

The **CurrentSensingTopology** enumeration is defined as follows:

Value	name	description
0	THREE_SHUNT	Three Shunt current sensing topology
1	SINGLE_SHUNT_ACTIVE_WIN	Single Shunt current sensing topology with active window injection (ST patented)
2	SINGLE_SHUNT_PHASE_SHIFT	Single Shunt current sensing topology with phase shifting
3	TWO_ICS	Two Insulated Current Sensors

The **ConfigurationFlags1** is defined as follows:

Bit	name	description
0	FLUX_WEAKENING	The Flux Weakening feature is present
1	FEED_FORWARD	The Feed Forward feature is present

2	MTPA	The Maximum Torque Per Ampere feature is present
3	PFC	The Power Form factor Correction feature is present
4	ICL	The Inrush Current Limiter feature is present
5	RESISTIVE_BREAK	The Resistive Break feature is present
6	OCP_DISABLE	The Over Current Protection Disabling feature is present
7	STGAP	STGAP1S intelligent gate drivers are used
8	POSITION_CTRL	The Position Control feature is present
9	VBUS_SENSING	The VBUS sensing feature is present
10	TEMP_SENSING	The Temperature sensing feature is present
11	VOLTAGE_SENSING	The Voltage sensing feature is present
12	FLASH_CONFIG	
13	DAC_CH1	The DAC, channel 1 debug feature is present
14	DAC_CH2	The DAC, channel 2 debug feature is present
15	ON_THE_FLY_STARTUP	The On the Fly Startup feature is present

The **ConfigurationFlags2** is defined as follows:

Bit	name	description
0	OVERMODULATION	The Overmudulation feature is present
1	DISCONTINUOUS_PWM	The Discontinuous PWM feature is present
2	RESERVED_1	
3	RESERVED_2	
4	RESERVED_3	
5	RESERVED_4	
6	RESERVED_5	
7	RESERVED_6	
8	RESERVED_7	
9	RESERVED_8	
10	RESERVED_9	
11	RESERVED_10	
12	RESERVED_11	
13	PROFILER	The profiler is enabled in the FW
14	DBG_MCU_LOAD_MEASURE	CPU load measurement feature is present (only posssible on Cortex M3/M4)
15	DBG_OPEN_LOOP	Open Loop Voltage and Open Loop Current modes are supported

The SIXSTEP_FIRMWARE_CFG register

Identifier	Type	Scope	Identification Token	Access
3	Raw Structure	Motor	`0xE9` to `0xEF`	Read Only, Constant

Six Step specific motor drive configuration. This register is present only if the **DriveType** enumeration field of the `APPLICATION_CONFIG` register for the same motor is set to **SIX_STEP**. Note that this register has the same identifier as the `FOC_FIRMWARE_CFG` register that is only present for an FOC drive.

The following table describes the content of the structure.

Size	Name	Description	Unit	Format
8	PrimSpeedPosSensingType	Primary speed and position sensing type	N/A	U8ENUM
8	CurrentSensingTopology	Current Sensing Topology	N/A	U8ENUM
32	PWMFrequency	PWM frequency	Hz	U32
16	MediumFrequencyTaskFreq	Medium Frequency Task execution frequency	Hz	U16
16	ConfigurationFlags1	Configuration flags	N/A	U16FLAGS

The **PrimSpeedPosSensingType** enumeration is defined as the **PrimSpeedPosSensingType** enumeration of the `FOC_FIRMWARE_CFG` register.

The **CurrentSensingTopology** enumeration is defined as the **CurrentSensingTopology** enumeration of the `FOC_FIRMWARE_CFG` register.

The **ConfigurationFlags1** is defined as the **ConfigurationFlags1** of the `FOC_FIRMWARE_CFG` register.

The DATALOG_UARTA, DATALOG_UARTB and DATALOG_STLINK registers

Name	Identifier	Type	Scope	Identification Token	Access
`DATALOG_UARTA`	20	Raw Structure	Global	`0x528`	Read/Write
`DATALOG_UARTB`	21	Raw Structure	Global	`0x568`	Read/Write
`DATALOG_STLINK`	22	Raw Structure	Global	`0x5A8`	Read/Write

These registers configure the mix of high and low frequency register to record for the Datalog Service available on the UARTA, the UARTB and the STLINK interfaces respectively.

The following table describes the content of the structures of these registers:

Size	Name	Description	Unit	Format
8	HFRate	High Frequency registers sampling rate	N/A	U8
8	HFNum	Number of High Frequency registers to record	N/A	U8
8	MFRate	Medium Frequency registers sampling rate	N/A	U8
8	MFNum	Number of Medium Frequency registers to record	N/A	U8
HFNum × 16	HFID	Regsiters identifiers of the High Frequency registers to record	N/A	U16ARRAY
MFNum × 16	MFID	Regsiters identifiers of the Merdium Frequency registers to record	N/A	U16ARRAY
8	Mark	Datalog configuration Identifier	N/A	U8

Setting the Mark field to 0 stops the corresponding Datalog service. For all the details on the fields of these registers, refer to the Datalog service section.

The SHIFT_UARTA, SHIFT_UARTB and SHIFT_STLINK registers

Name	Identifier	Type	Scope	Identification Token	Access
`SHIFT_UARTA`	25	Raw Structure	Global	`0x668`	Read/Write
`SHIFT_UARTB`	26	Raw Structure	Global	`0x6A8`	Read/Write
`SHIFT_STLINK`	27	Raw Structure	Global	`0x6E8`	Read/Write

These registers configure the shift to apply to the value of 32-bit High Frequency registers before recording them in the Datalog service on the UARTA, the UARTB and the STLINK interfaces. These registers are only present when 32-bit HF registers exist which is not the case currently in ST MCSDK.

The following table describes the content of the structures of these registers:

Size	Name	Description	Unit	Format
n × 8	HFRegShift	High Frequency registers shift value	N/A	U8ARRAY

n is the number of HF register configured for the related Datalog service.

Document history

Version	Date	Comment
1	2022-12-13	Initial version

© 2024, ST Microelectronics