ECom

Release 1.1.1

Aster

Feb 12, 2023

CONTENTS:

CHAPTER

ONE

COMMUNICATION DATABASE

1.1 Base types

Types are used throughout the communication database. They convey how the data can be serialized to binary data and how that binary can be parsed back to its original value. They also define the size of the data in the binary format.

The database allows to define complex combined and nested types, but they must all define a base typ. These are the base type and their attributes currently supported:

Name	Binary Size [Byte]	Min Value	Max Value	C Type	Python Type	Description
int8	1	-128	127	int8_t	int	This is the mapping of the C type for small numbers.
uint8	1	0	255	uint8_t	int	This is the mapping of the C type for small unsigned numbers.
bool	1			bool	bool	This is a type, that has only two values: True and False.
int16	2	-32768	32767	int16_t	int	This is the mapping of the C type for small numbers.
uint16	2	0	65535	uint16_t	int	This is the mapping of the C type for small unsigned numbers.
int32	4	-2147483648	2147483647	int32_t	int	This is the mapping of the C type for numbers.
uint32	4	0	4294967295	uint32_t	int	This is the mapping of the C type for unsigned numbers.
int64	8	-9223372036854775808	9223372036854775807	int64_t	int	This is the mapping of the C type for large numbers.
uint64	8	0	18446744073709551615	uint64_t	int	This is the mapping of the C type for large unsigned numbers.
float	4	-3.4e+38	3.4e+38	float	float	A type for small floating point values. **WARNING**: This type does not have a standard size in C. If the size is not 4 bytes for a platform, the generated code will not compile.
double	8	-1.7e+308	1.7e+308	double	float	A type for larger floating point values with higher precision. **WARNING**: This type does not have a standard size in C. If the size is not 8 bytes for a platform, the generated code will not compile.
char	1			char	bytes	This type represents a character. It is usually used as an array to represent a string.
bytes	1			uint8_t	bytes	This type is similar to the char type, but it is used to represent a byte.

1.2 Telecommand arguments

Telecommands can optionally have arguments. They are declared by providing a file with the same name as the command and the ending `.csv` in the `commandArguments` subdirectory of the communication database (see example).

Each file consists of a csv table with four columns and one header row containing the name of each column. Each other row declares one telecommand argument:

Column name	Column description
Name	The name of the telecommand argument.
Type	The type of the data that can be sent via this argument. It can be a *unit* or *shared* type.
Default	An optional default value for this argument. This is used in the serializer, if the value was not provided.
Description	An optional human readable description of this argument.

The following example defines an argument for a `CALIBRATE_IMU` command and defines one argument, `imu` which is an unsigned 8-bit integer, without a default value.

Filename: `CALIBRATE_IMU.csv`

```
Name,Type,Default,Description
imu,uint8,,The IMU to calibrate.
```

Using all defined arguments, the database will automatically create a constant named `MAX_TELECOMMAND_DATA_SIZE`, which will contain the maximum size in bytes needed to store the serialized version of their arguments for all commands. It is the maximum amount of bytes needed for any command to transmit its arguments.

1.3 Telecommands

Telecommands are packages of data that can be sent from the base to the secondary device. They can have zero, one or multiple arguments, which are additional data that is sent alongside the command. The telecommand arguments are further detailed in the *telecommand arguments documentation*.

The telecommands can also optionally have a return value, in which case the secondary device is expected to respond with a `RESPONSE` *telemetry package* which contains the return value. If no return value is specified, the secondary device is expected to respond with an `AKNOWLEDGE` *telemetry package*.

1.3.1 Definition

Telecommands are defined in the `commands.csv` file. The file consists of a csv table with six columns and one header row containing the name of each column. Each other row declares one telecommand:

Column name	Column description
Name	The name of this telecommand.
Debug	Either `true` or `false`, indicates whether this command is only available if debugging commands are allowed (*details*).
Description	An optional human readable description of the telecommand.
Response name	A name of the return value, if this command has one. Otherwise it can be left empty.
Response type	The type of the return value, if this command has one. Otherwise it can be left empty. It can be a *unit* or *shared* type.
Response description	An optional human readable description of the return value.

The following example defines two telecommands, one telecommand to enable a motor, and one telecommand to query the enabled state of the motor. The last command can only be used, if the debug telecommands are enabled.

```
Name,Debug,Description,Response name, Response type,Response description
ENABLE_MOTOR,false,Enable the motor.,,,
GET_MOTOR_ENABLED,true,Get whether the motor is enabled.,enabled,bool,The enabled state␣
↪of the motor.
```

Using the declared telecommands, the database automatically generates a `TelecommandType` enum type which contains an identifier value for all telecommands.

1.3.2 Structure

All telecommands are serialized into the format of the special `TelecommandMessageHeader` *shared data type*. If the command has any arguments, they are attached to the end of the header and are also included in the generated checksum.

1.3.3 C Code generation

The C code generator will generate a parser which can parse serialized telecommands and calls the correct handler function for it. The parser declares the following function to feed received data into the parser:

```
bool parseMessage(BufferAccessor* buffer, MessageHandler* handler);
```

This function will call the appropriate command handler that is given via the `MessageHandler` object. It is a collection of function pointers, defining a command handler function for each telecommand (*config commands* are the only exception). The implementation of these handlers must be provided by the secondary devices code.

Required functions

The generated C code expects the following functions to be present:

- areDebugCommandsEnabled: This function is called when a telecommand is received whose debug attribute is `true`. The handler for the command is only executed if this function returns `true`.

```
/**
 * @note This function is only called when a debug command is received.
 * @note This function has to be provided by the ECom user.
 * @return Whether executing debug commands is currently enabled.
```

(continues on next page)

(continued from previous page)

```
 */
bool areDebugCommandsEnabled(void);
```

- printDebug: Used to print a formatted debugging message.

```
/**
 * Print the formatted debug message.
 *
 * @note This function has to be provided by the ECom user.
 * @param format The format string.
 * @param ... Format values.
 */
void printDebug(const char* format, ...);
```

- printDebug: Used to print a formatted error message.

  Note: This function is only required when a *config command* is defined.

```
/**
 * Print the formatted error message.
 *
 * @note This function has to be provided by the ECom user.
 * @param format The format string.
 * @param ... Format values.
 */
void printError(const char* format, ...);
```

1.4 Configuration

The configuration system allows to define named values with defaults that can be changed and queried while the secondary device is running with a telecommand. The system has two components.

1.4.1 Configuration definition

The configurations are defined in the `configuration.csv` file. It consists of a csv table with four columns and one header row, which contains the name of the columns. Each row represents one configuration item:

Column name	Column description
Name	The name of the configuration item.
Type	The type that is used to represent this configuration item. Can be a *unit* or *shared data* type.
Default Value	The default value of this configuration item.
Description	An optional human readable description of this configuration item.

The following example defines two configuration items: A boolean `imu enabled`, which is enabled by default, and a `sensor check interval`, which is `5000` milliseconds by default, using a `ms` *unit*.

```
Name,Type,Default Value,Description
imu enabled,bool,true,Wheter the IMU is enabled.
sensor check interval,ms,5000,How much time to wait between checking the sensor.
```

Using the configuration items specified in the `configuration.csv` file, the database will automatically add the following constants:

- `NUM_CONFIGURATIONS`: The number of provided configuration items. In the example above, it would be the number `2`.
- `DEFAULT_CONFIGURATION`: A struct that contains the default configuration of the declared config items. For the example above, represented as a json object, it would be

```
{
  "imu enabled": true,
  "sensor check interval": 5000
}
```

- `MAX_CONFIG_VALUE_SIZE`: The maximum size in bytes of any configuration value. In the example above, it would be the maximum between `1` (the size of `bool`) and the size of the `ms` unit.

Additionally, the following types will be defined:

- `ConfigurationId`: An enum type containing an enum value for each configuration item.
- `Configuration`: A struct type that contains a child for each configuration item. This is the type used in the `DEFAULT_CONFIGURATION` constant, and it can be used to hold a configuration state.

1.4.2 Configuration telecommands

If a *command* is defined with one argument of type `ConfigurationId` and a return value of the special type `config?`, the database will mark this command as a configuration getter telecommand. An automatic handler for this telecommand will be generated in the C generator, which responds to this telecommand with the value of the configuration indicated by the `ConfigurationId` argument. A `ResponseParser` is able to automatically convert the response to the correct data type. The automatically generated C code requires the following function to be defined somewhere and to provide access to the current configuration:

```
const Configuration* getConfig();
```

Also, if a *command* is defined with no return types and two arguments, one of type `ConfigurationId` and another one of the special type `config?`, the database will mark this command as a configuration setter telecommand. An automatic handler for this telecommand will be generated in the C generator, which correctly parses the configuration value to its correct data type and updates the configuration with the new value. The automatically generated C code requires two functions to be defined somewhere:

A function to get a mutable configuration object and indicate that a change will be made.

```
Configuration* beginConfigUpdate();
```

And a function to commit the changes made to the mutable configuration object retrieved with `beginConfigUpdate`.

```
void commitConfigUpdate();
```

1.5 Shared constants

The communication database allows to specify constants that are the same for the receiver and sender. This is intended to share constants which are used in the payload definition. These constants can be used as default values and array size specifiers in the payload descriptions.

The shares constants are defined in the `sharedConstants.csv` file. It consists of a csv table with 4 columns and one header row, which contains the names of the columns:

Column name	Column description
Name	The name of the shared constant.
Value	The value of the shared constant. The value is interpreted based on the type.
Type	The type of the shared constant. A *unit type* can be used here.
Description	A human readable description of the shared constant.

1.6 Shared Data Types

The communication database allows to define data types which can be used in the receiver and the sender. They allow to define a way to pack multiple individual datapoint into a structure and to define enumeration types and thus restrict possible values for a type. They allow to build message objects that transmit complex and nested data structures, and they even allow to define the base message structure (see *Special types*).

1.6.1 Definition

They are defined in the `sharedDataTypes.json` file. The file consists of a json object, whose attributes are the shared data types. The value of these attributes describes information about the type of the shared data type.

In the following example, there are three data types defined, `a`, `b` and `c`. The attributes of these data types are empty for this example, in a real example the `{}` wouldn't be valid:

```
{
  "a": {},
  "b": {},
  "c": {}
}
```

Type description

The value for each attribute of the top level json object describes the type of this shared data type. It can be either a string or an object.

Simple

In the simple case, the type description is a string. In this case, the string is the name of the *base type*. The base type can also be a *unit type*. Additionally, the base type can also be a shared data type that was declared before the new type.

In the following example, the `a` type is an 8-bit unsigned integer, the `b` type is a `meter` unit and the `c` type is synonym for the `a` type and therefore also an 8-bit unsigned integer:

```
{
  "a": "uint8",
  "b": "meter",
  "c": "a"
}
```

Arrays

The simple data type description can have an opening bracket, then a number or a name of a *shared constant* which resolves to a positive integer number, and then a closing bracket. This indicates that the type represents an array, or a list.

in the following example, two types are declared: The type `a`, which represents a list of `5` unsigned 8-bit integers, and the type `b`, which also represents a list of unsigned 8-bit integers whose size depends on the value of the shared constant `B_LENGTH`.

```
{
  "a": "uint8[5]",
  "b": "uint8[B_LENGTH]"
}
```

Advanced

A type can be enhanced with additional metadata. To add metadata to `a` type from the simple case example, the value must become an object and the string name of the base type moves into the `__type__` attribute of the new object. This allows to represent the following metadata about the type:

Attribute Name	Description
`__type__`	The base type of the shared type, has the same restrictions as in the simple case.
`__doc__`	An optional human readable description of this type.
`__value__`	An optional default value for this type. This is used for example in telecommands when the value is not provided.

The following example shows the `a` type from the simple example with a description and a default value:

```
{
  "a": {
    "__doc__": "A description of this type.",
    "__type__": "uint8",
    "__value__": 1
  }
}
```

Enumerations

If a `__values__` attribute is given (note the `s` at the end), the type is interpreted as an enumeration type. These types have a restricted set of named values called enumeration members, that the type can represent.

The enumeration members are automatically assigned integer values based on the C standard for enumeration default values: If no explicit value is defined for the first member, it is assigned the value `0`. Fore each other member, if the no explicit value is specified, it's value becomes the value of the previous member incremented by `1`. The value of an enumeration member can be specified via its `__value__` attribute. If two enumeration members have the same value, they are the second is considered an alias of the first. If serialized, both values will be deserialized to the first member.

If a `__type__` attribute is given, it represents the underlying base type of the enumeration type. If it is omitted, the smallest possible base type is automatically chosen based on the largest enumeration member value.

These named values are either provided in a list, in which case the list item must be a string representing the name of that enumeration member, or as an object where each value is represented as an attribute of that object. The name of each attribute is the name of that enumeration member, and the value must be an object with additional metadata about this enumeration value:

Attribute Name	Description
`__doc__`	An optional description of this enumeration value.
`__value__`	An optional explicit value of this enumeration value.

In the following example, two enumerations are defined: `a` has two enumeration members, both of which have a documentation. The first member has the value `a.A_STATE_1 = 0`, the second member has the value `a.A_STATE_2 = 42`. `b` has also two enumeration members, but none of them have a documentation. The first member has the value `b.B_STATE_1 = 0`, the second member has the value `b.B_STATE_2 = 1`.

```
{
  "a": {
    "__doc__": "A description of the a type.",
    "__type__": "uint8",
    "__values__": {
      "A_STATE_1": {
        "__doc__": "Description for the A_STATE_1 value."
      },
      "A_STATE_2": {
        "__doc__": "Description for the A_STATE_2 value.",
        "__value__": 42
      }
    }
  },
  "b": {
    "__doc__": "A description of the b type.",
    "__type__": "uint8",
    "__values__": [
      "B_STATE_1",
      "B_STATE_2"
    ]
  }
}
```

Structure types

Structural datatypes allow for grouping of and nesting of multiple datatypes. A structural type can include members of any other data type that was defined before, or of a new substructure.

Structural types can't have a base type, therefore they don't allow to specify a `__type__` attribute. They do allow to specify a `__doc__` documentation description. All other attributes of the metadata are interpreted as children of the structural type. The name of the attributes are the names of the children. If a new type is defined in the child, it's going to have the same name as the attribute. The value of each child attribute is a new *type description*. Circular types are not possible, because the type can't reference itself.

The following example defines two structural types: The first one `a` has two children, an unsigned 8-bit integer `x` and a character `y`. The second type `b` has a documentation and three children: A boolean `element1` and an element of the previously defined type `a`, which has the two mentioned children. Additionally, `b` has a third child named `element3`, which is a structural type whose name is also `element3` with two children, a 32-bit signed integer `subElement1` and an array of four bytes named `subElement2`.

```
{
  "a": {
    "x": "uint8",
    "y": "char"
  },
  "b": {
    "__doc__": "A description of this type.",
    "element1": "bool",
    "element2": "a",
    "element3": {
      "subElement1": "int32",
      "subElement2": "bytes[4]"
    }
  }
}
```

1.6.2 Special types

There are some special types that are automatically defined by the database and some that are expected to be defined, for the serializer and parser to function properly. The later are:

- `TelemetryMessageHeader`: The parser expects a structural type with this name (can be configured when constructing the parser). The type must define an integer `sync byte 1`, `sync byte 2`, as well as an `type` child of the automatically generated `TelemetryType` enum type.
- `TelecommandMessageHeader`: The serializer expects a structural type with this name (can also be configured when constructing the serializer). The type must have an integer `sync byte 1`, `sync byte 2`, as well as a `type` member which is of the automatically generated enum type `TelecommandType`.

1.7 Telemetry

Telemetry is data that the secondary device can send to the base. It encapsulates housekeeping data and responses to *commands* (see *Special Telemetry*).

1.7.1 Definition

The types of telemetry are defined in the `telemetry.csv` file. It contains a csv table with two columns and one heading row that contains the name of the columns. Every other row defines a new telemetry type. The file does not contain the contents of the telemetry, this is covered in the *telemetry arguments documentation*.

Column name	Column description
Name	The name of the telemetry type.
Description	An optional human readable description of the telemetry type.

The following example defines an `ACKNOWLEDGE` and `RESPONSE` telemetry type with their documentation.

```
Name,Description
RESPONSE,A response to a telecommand with the response data.
ACKNOWLEDGE,Acknowledge receiving a command.
```

Special Telemetry

The `ResponseTelemetryParser` expects two telemetry types to be defined:

- `RESPONSE`: A telemetry that will be sent back as a response to a telecommand, which contains the result of that telecommand.
- `ACKNOWLEDGE`: A telemetry that will be sent back as a response to a telecommand, if the telecommand doesn't have a return value. It is used to acknowledge the retrieval of that telecommand.

1.7.2 Structure

All telecommands are serialized into the format of the special `TelemetryMessageHeader` *shared data type* shared data type. The messages are of variable sizes, dependent on the telemetry type. The serialized data is attached to the end of the header and is also included in the generated checksum.

1.7.3 C Code generation

The C code generator will generate a serializer which can serialize data into telemetry packages. The parser declares a `sendTelemetryName` function where `Name` is replaced by the name of each telemetry type. Each function accepts the data defined for that telemetry type in the *telemetry arguments documentation*. The serializer will call a `writeTelemetry` function, which is expected to be defined somewhere:

```
bool writeTelemetry(const void* data, size_t dataLength);
```

1.8 Telemetry arguments

Each *telemetry type* is expected to have a file in the `telemetryArguments` subdirectory of the communication database (see example), which describes the data send for the specific telemetry type. The file must be named like the telemetry type with a `.csv` ending. It consists of a csv table with three columns. The first row is a header which contains the names of each column. Every other row defines a data point which is sent for this telemetry type:

Column name	Column description
Name	The name of this telemetry data point.
Type	The type of data this telemetry data point represents. It can be a *unit* or *shared* type.
Description	An optional human readable description of the telemetry data point.

The following example shows the telemetry arguments for a `RESPONSE` telemetry type. It specifies that when sending this telemetry it will include an unsigned 8-bit number representing a `command number` and an array of bytes as `value`, whose size depends on the shared variable `MAX_TELECOMMAND_RESPONSE_SIZE`.

Filename: `RESPONSE.csv`

```
Name,Type,Description
command number,uint8,The counter number of the command that generated this response.
value,bytes[MAX_TELECOMMAND_RESPONSE_SIZE],The response value of the command.
```

1.9 Units

The communication database allows to specify units to be used in communication payloads. These units are named wrappers around a *base type*. They are declared in the `units.csv` file. The file consists of a csv table with 3 columns and a heading row, containing the names of each column:

Colum name	Column description
Name	The name of the unit type. This name can be used where a type is required.
Type	The underlying *base type* of this unit. Only raw types, no arrays are accepted.
Description	A human readable description of this unit.

1.9.1 Variants

Units can have variants, which are datatypes representing the same unit but with a different underlying data type. In the communication database, these don't have to be a defined explicitly in the `units.csv` file. Whenever it is possible to use a unit, one can instead give a base type with one of the defined units in parentheses.

As an example, given the following `units.csv` file, which defines a `unit1` unit:

```
Name,Type,Description
unit1,double,A demonstration unit.
```

One can define and use a variant, for example as a telemetry argument:

```
Name,Type,Description
datapoint1,float (unit1),Some datapoint for the demonstration.
```

These variants show up in the `units` dictionary of the communication database as additional units with different base types in the list after the base unit.

```
>>> from ecom.database import CommunicationDatabase
>>> database = CommunicationDatabase(...)
>>> database.units['volt']
[
    Unit(type=<class 'float'>, name='volt', baseTypeName='double', description='Voltage.
↪', default=None),
    Unit(type=<class 'float'>, name='volt', baseTypeName='float', description='Voltage.',
↪ default=None),
]
```

On the embedded side, the two units are two different types:

```
volt_t baseUnitValue = volt_t(0);
voltFloat_t variantUnitValue = voltFloat_t(0);
```

The communication database specifies the communication between to devices. It is spread out over multiple files and folders:

- *commands.csv* (Example): Contains a list of all telecommands and describes an optional return value.
- *commandArguments* (Example): This folder contains the arguments for the telecommands. If a telecommand has arguments, they must be in a file with the same name as the telecommand.
- *telemetry.csv* (Example): A list of possible types of telemetry messages.
- *telemetryArguments* (Example): This folder contains the definition of the data that is sent with each telemetry type. The definition must be in a file with the same name as the telemetry type.
- *sharedConstants.csv* (Example): Constant values that are used in the communication.
- *sharedDataTypes.json* (Example): A description of data types that are used in the communication.
- *configuration.csv* (Example): A description of configuration parameters that can be changed with the configuration telecommand.
- *units.csv* (Example): A description of units and their base datatypes used in the communication.

CHAPTER

TWO

API REFERENCE

This page contains auto-generated API reference documentation[1].

2.1 ecom

Provides parser, serializer and code generator to use a shared communication database.

2.1.1 Submodules

ecom.checksum

Module Contents

Classes

ChecksumVerifier	A checksum verifier that can add and verify a CRC16 checksum.

Functions

calculateChecksum(→ int)	Calculate a CRC16 checksum from the data.

ecom.checksum.**calculateChecksum**(*data: bytes*, *polynomial: int = 34833*) → int

Calculate a CRC16 checksum from the data.

Parameters

- **data** – The data to calculate the checksum from.
- **polynomial** – The polynomial to use for the crc calculation in nominal form.

Returns

The calculated checksum.

[1] Created with sphinx-autoapi

class ecom.checksum.**ChecksumVerifier**(*database:* ecom.database.CommunicationDatabase, *checksumDatapointName='checksum'*)

Bases: *ecom.verification.MessageVerifier*

A checksum verifier that can add and verify a CRC16 checksum.

verify(*data:* *bytes*, *message:* ecom.message.MessageType, *header:* ecom.datatypes.TypeInfo*[*ecom.datatypes.StructType*]*, *messageSize:* *int*)

Verify the integrity of the message.

Raises

VerificationError – if the message fails the verification check and the original data can't be recovered.

Parameters

- **data** – The raw data of the message. May also include other trailing data.
- **message** – The type of message.
- **header** – The header type information of the message.
- **messageSize** – The size of the message, as calculated with the unverified data.

Returns

None if the message was verified successfully or the bytes of the error corrected message and any bytes from the given data that were not part of the message.

addVerificationData(*data:* *bytes*, *message:* ecom.message.MessageType, *header:* ecom.datatypes.TypeInfo*[*ecom.datatypes.StructType*]*) → bytes

Add extra verification data to a message.

Parameters

- **data** – The message data.
- **message** – The type of message.
- **header** – The header type information of the message.

Returns

The message with verification data.

addPlaceholderVerificationData(*data: Dict[str, Any]*)

When constructing a message, this allows to add a placeholder value to the header that will be filled later by *addVerificationData*.

Parameters

data – A dictionary to store the placeholder verification data to.

ecom.database

Module Contents

Classes

Unit	A representation of a physical unit.
Configuration	A configuration item of the secondary device.
ConfigurationValueDatapoint	A datapoint that is the value of a configuration.
ConfigurationValueResponseType	A response type that is the value of a configuration.
Constant	A constant in the communication database.
CommunicationDatabase	The shared communication database. Contains all information about the communication.

Functions

main()	Verifies the communication database.

class ecom.database.**Unit**

Bases: *ecom.datatypes.TypeInfo*

A representation of a physical unit.

classmethod fromTypeInfo(*name: str*, *typeInfo:* ecom.datatypes.TypeInfo, *description: Optional[str] = None*) → *Unit*

Create a new unit type from a given base type.

Parameters

- **name** – The name of the unit.
- **typeInfo** – The base type.
- **description** – The description of the unit.

Returns
The new unit.

class ecom.database.**Configuration**

Bases: Generic[*ecom.datatypes.V*]

A configuration item of the secondary device.

id: *ecom.datatypes.EnumType*

The id of the configuration item.

name: *str*

The name of the configuration item.

type: *ecom.datatypes.TypeInfo*[ecom.datatypes.V]

The type of the configuration.

defaultValue: ecom.datatypes.V

The default value of the configuration.

description: Optional[*str*]

A description of the configuration.

exception ecom.database.**CommunicationDatabaseError**

Bases: RuntimeError

Indicates an error in the communication database.

exception ecom.database.**UnknownTypeError**(*typ: str*)

Bases: *CommunicationDatabaseError*

Information about an unknown shared datatype was requested.

exception ecom.database.**UnknownConstantError**(*constant: str*)

Bases: ValueError

Information about an unknown shared constant was requested.

exception ecom.database.**UnknownDatapointError**(*datapointName: str*)

Bases: *CommunicationDatabaseError*

Information about an unknown datapoint was requested.

property datapointName

Returns
The name of the unknown datapoint.

class ecom.database.**ConfigurationValueDatapoint**

Bases: *ecom.message.DependantTelecommandDatapointType*, _DbContainer

A datapoint that is the value of a configuration.

configureWith(*providerValue*)

Create an instance of this dependant telecommand datapoint for the value of the provider.

Parameters
providerValue – The value of the provider that this datapoint depends on.

class ecom.database.**ConfigurationValueResponseType**(*database: CommunicationDatabase*, **args*, ***kwargs*)

Bases: *ecom.message.DependantTelecommandResponseType*

A response type that is the value of a configuration.

configureWith(*providerValue*)

Create an instance of the dependant response for the value of the provider.

Parameters
providerValue – The value of the provider that this response depends on.

Returns
The configured TelecommandResponseType instance.

class ecom.database.**Constant**

Bases: Generic[*ecom.datatypes.V*]

A constant in the communication database.

name: str

The name of the constant.

value: ecom.datatypes.V

The value of the constant.

type: *ecom.datatypes.TypeInfo*[ecom.datatypes.V]

Information about the type of the constant.

description: str

A description of the constant.

class ecom.database.**CommunicationDatabase**(*dataDirectory: str*)

The shared communication database. Contains all information about the communication.

property **constants:** Dict[str, *Constant*]

Returns
A name to value, description and type information mapping for all shared constants.

property **telecommandTypes:** List[*ecom.message.TelecommandType*]

Returns
All telecommand types.

property **telecommandTypeEnum:** Optional[Type[*ecom.datatypes.EnumType*]]

Returns
The enum class that is used for the telecommand types or None if no telecommand types exist.

property **telemetryTypes:** List[*ecom.message.TelemetryType*]

Returns
All telemetry types.

property **telemetryTypeEnum:** Optional[Type[*ecom.datatypes.EnumType*]]

Returns
The enum class that is used for the telemetry types or None if no telemetry types exist.

property **dataTypes:** Dict[str, *ecom.datatypes.TypeInfo*]

Returns
All shared data types.

property **units:** Dict[str, List[*Unit*]]

Returns
All mapping of names to all known units and their variants.

property **configurations:** List[*Configuration*]

Returns
All configuration items of the secondary device.

property **configurationEnum:** Optional[Type[*ecom.datatypes.EnumType*]]

Returns
The enum class that is used for the configurations or None if no configurations exist.

__eq__(*o: object*) → bool

Return self==value.

__repr__() → str

Return repr(self).

getTypeInfo(*typeName: str*) → *ecom.datatypes.TypeInfo*

Get the type info for a known datatype by its name.

Parameters
typeName – The name of the type.

Returns
The type info for the data type.

getTelecommandByName(*name: str*) → *ecom.message.TelecommandType*

Parameters
name – The name of a telecommand type.

Returns
The telecommand type with the specified name.

getTelecommand(*telecommandId:* ecom.datatypes.EnumType) → *ecom.message.TelecommandType*

Parameters
telecommandId – The id enum value of a telecommand type.

Returns
The telecommand type with the specified id.

getTelemetryByName(*name: str*) → *ecom.message.TelemetryType*

Parameters
name – The name of a telemetry type.

Returns
The telemetry type with the specified name.

getTelemetry(*telemetryId:* ecom.datatypes.EnumType) → *ecom.message.TelemetryType*

Parameters
telemetryId – The id enum value of a telemetry type.

Returns
The telemetry type with the specified id.

parseKnownTypeInfo(*typeStr: str*, *name: Optional[str] = None*, *documentation: Optional[str] = None*)

Get the type info for a known datatype. Supports parsing type strings declaring an array of a known datatype.

Parameters

- **typeStr** – The type string to parse.
- **name** – The name of the array data type, if the type is an array. If not provided, the typeStr is used.
- **documentation** – The documentation of the data type, if the type is an array.

Returns
The type information for the parsed type.

replaceType(*typ: Type[ecom.datatypes.V]*, *name: Optional[str] = None*)

Replace a type registered in the database with another Python class. This allows to define own types like enums for types defined in the communication database, make sure that the two types match and keep up to date with each other, and use the types on the Python side with IDE autocompletion support.

Parameters

- **typ** – The new type to replace the old type.

- **name** – The name of the type to replace, if omitted the class name of the new type is used.

registerChangeListener(*listener: Callable[[], None]*)

Register a listener that will be called every time when the communication database has been changed.

Parameters
listener – The callable that will be called when a change occurs.

ecom.database.**main**()

Verifies the communication database.

Returns
Whether the verification was successful.

ecom.datatypes

Module Contents

Classes

StrEnum	str(object='') -> str
CommunicationDatabaseAccessor	A generic base for classes that want to access the communication database.
StructTypeMeta	Metaclass for the StructType.
StructType	Represents a C struct. Can be iterated over to get the struct members.
ArrayTypeMeta	Metaclass for the ArrayType.
ArrayType	Represents a C array. Provides the type and constant size of the array.
EnumTypeMeta	Metaclass for Enums that are compatible with C style enums and can be compared.
EnumType	Represents a C style enum.
DefaultValueInfo	A default value for a type.
TypeInfo	Describes a data type.

Functions

dataclass_transform()	
loadTypedValue(→ Any)	Load a value with the given type.
structField(→ dataclasses.Field)	Declare a new struct dataclass field. This is a wrapper around the `dataclasses.field()` function.
isStructField(→ bool)	Check whether the given field is a struct field created with *ecom.datatypes.structField()*.
getStructFieldName(→ str)	Get the name of the element in the struct of a struct field. This can be different from the dataclass field name.
getStructFieldType(→ Optional[TypeInfo])	Get the type of the struct field or None, if the struct field does not define its type.
structDataclass(database[, replaceType])	A decorator for a class that inherits from StructType that allows it to act as a dataclass.

Attributes

T

V

class ecom.datatypes.**StrEnum**

Bases: str, enum.Enum

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

ecom.datatypes.**dataclass_transform**()

class ecom.datatypes.**CommunicationDatabaseAccessor**(*database:* database.CommunicationDatabase)

A generic base for classes that want to access the communication database.

ecom.datatypes.**loadTypedValue**(*value: Any*, *typ: Type*) → Any

Load a value with the given type.

Parameters

- **value** – A value.
- **typ** – The type of the parsed value.

Returns

The value parsed with the given type.

class ecom.datatypes.**StructTypeMeta**

Bases: type

Metaclass for the StructType.

__iter__() → Iterator[Tuple[str, *TypeInfo*]]

Iterate over all children of the struct.

Returns

A list of child names and child type infos.

__getitem__(*item*)

Get the type info for a child of the struct.

Parameters

item – The name of the child.

Returns

The type info for that child.

Raises

AttributeError – If the struct does not have a child with that name.

__contains__(*childName: str*) → bool

Whether this struct contains a child element with the given name.

Parameters
childName – The name of a child element.

Returns
Whether the struct contains a child element with that name.

__call__(*value: Union[str, Dict, Iterable[Iterable], None] = None*, *childrenTypes: Optional[Dict[str, TypeInfo]] = None*, *documentation: Optional[str] = None*, ***kwargs*)

Create a new struct type.

Parameters

- **value** – The name of the struct typ.
- **childrenTypes** – A mapping of children names and their types for the struct type.
- **documentation** – Documentation for the new type.

Returns
The new struct type.

__eq__(*o: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

offsetOf(*database:* database.CommunicationDatabase, *name: str*) → int

Calculate the offset of the child element with the given name.

Parameters

- **database** – A communication database.
- **name** – The name of the child element whose offset should be calculated.

Returns
The offset of the child element in bytes.

class ecom.datatypes.**StructType**

Bases: dict

Represents a C struct. Can be iterated over to get the struct members.

ecom.datatypes.**T**

ecom.datatypes.**structField**(*typ: Union[str, TypeInfo, Callable[[database.CommunicationDatabase], TypeInfo], None] = None*, *name: Optional[str] = None*, ***kwargs*) → dataclasses.Field

Declare a new struct dataclass field. This is a wrapper around the `dataclasses.field()` function.

See
`dataclasses.field()`

See
ecom.datatypes.structDataclass()

Parameters

- **typ** – The type of this field or a callable that given a communication database resolves the type of this field. This is required unless the field replaces a field of another structure, in which case the type is inferred from the other field.
- **name** – The name of this field. If omitted, the name of dataclass field is used.
- **kwargs** – Additional arguments to the `dataclasses.field()` function.

Returns
A struct dataclass field.

`ecom.datatypes.isStructField(childField: dataclasses.Field) → bool`

Check whether the given field is a struct field created with *ecom.datatypes.structField()*.

Parameters
childField – The field to check.

Returns
Whether the field is a struct field.

`ecom.datatypes.getStructFieldName(childField: dataclasses.field) → str`

Get the name of the element in the struct of a struct field. This can be different from the dataclass field name.

Raises
KeyError – if the field is not a struct field.

Parameters
childField – The struct field to get the name of.

Returns
The field name of the struct field.

`ecom.datatypes.getStructFieldType(childField: dataclasses.field, database: database.CommunicationDatabase) → Optional[TypeInfo]`

Get the type of the struct field or None, if the struct field does not define its type.

Raises
KeyError – if the field is not a struct field.

Parameters

- **childField** – The struct field to get the type of.
- **database** – A communication database to lookup type info.

Returns
The type information about the struct field type or None.

`ecom.datatypes.structDataclass(database: database.CommunicationDatabase, replaceType: Union[str, bool] = False)`

A decorator for a class that inherits from StructType that allows it to act as a dataclass. Members can be defined as structFields, to declare their type. The resulting class is a dataclass.

The field values can be given as a dict, just like a regular StructType:

```
floatTypeInfo = TypeInfo.lookupBaseType(TypeInfo.BaseType.FLOAT)

@structDataclass(database)
class Quaternion(StructType):
    x: float = structField(typ=floatTypeInfo)
    y: float = structField(typ=floatTypeInfo)
```

(continues on next page)

(continued from previous page)

```
    z: float = structField(typ=floatTypeInfo)
    w: float = structField(typ=floatTypeInfo)

value = Quaternion({'x': 1, 'y': 2, 'z': 3, 'w': 4})
print(value.x)  # -> 1
```

The field values can also be given as keyword arguments:

```
Quaternion(x=1, y=2, z=3, w=4)
```

Alternatively, the field values can be given as a mixture of both methods:

```
@structDataclass(database)
class Quaternion(StructType):
    x: float = structField(typ=floatTypeInfo)
    y: float = structField(typ=floatTypeInfo)
    z: float = structField(typ=floatTypeInfo)
    w: float = structField(typ=floatTypeInfo)
    other = 5.0

Quaternion({'x': 1, 'y': 2, 'z': 3, 'w': 4}, other=6.0)
```

A structDataclass can also replace an existing struct type in the database. If *replaceType* is set to `True`, the class will replace a struct with the same name as the class. If *replaceType* is a string, the class will replace a struct by that name. In both cases, the struct have the same fields as the replacing struct. The fields don't need to be explicitly typed, they will inherit their type from their respective field in the replacing struct:

```
@structDataclass(database, replace=True)
class Quaternion(StructType):
    x: float
    y: float
    z: float
    w: float
```

Parameters

- **database** – A communication database.
- **replaceType** – If `True`, replace a type in the database with the same class name. If a string, replace a type with that name.

Returns

The struct dataclass.

exception ecom.datatypes.**DynamicSizeError**(*sizeMember: str*)

Bases: `RuntimeError`

An operation was requested that required a known size, but the size must be read dynamically from the parent struct.

property sizeMember: `str`

Returns

The name of the member that must be read to get the size.

class ecom.datatypes.**ArrayTypeMeta**

Bases: type

Metaclass for the ArrayType.

__len__() → int

Returns
The length of the array.

Raises
DynamicSizeError – If the length must be dynamically read from a member of the parent struct.

__eq__(*o: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

getElementTypeInfo() → *TypeInfo*

Returns
The type info for all elements.

__call__(*value: Union[str, Iterable]*, *typ: Optional[*TypeInfo*] = None*, *size: Optional[Union[int, str]] = None*, *documentation: Optional[str] = None*)

Create a new array type.

Parameters

- **value** – The name of the array type.
- **typ** – The type info for the elements of the array.
- **size** – The size of the array or the name of the member in the parent struct from which the size can be read.
- **documentation** – Documentation of the new type.

Returns
The new array type.

class ecom.datatypes.**ArrayType**

Bases: list

Represents a C array. Provides the type and constant size of the array.

class ecom.datatypes.**EnumTypeMeta**

Bases: enum.EnumMeta

Metaclass for Enums that are compatible with C style enums and can be compared.

__eq__(*o: object*) → bool

Return self==value.

__hash__() → int

Return hash(self).

class ecom.datatypes.**EnumType**

Bases: int, enum.Enum

Represents a C style enum.

__eq__(*o: object*) → bool
Return self==value.

__hash__() → Any
Return hash(self).

ecom.datatypes.**V**

class ecom.datatypes.**DefaultValueInfo**
Bases: Generic[*V*]

A default value for a type.

value: V
The default value.

constantName: Optional[str]
The name of the default value as a shared constant.

class ecom.datatypes.**TypeInfo**
Bases: Generic[*V*]

Describes a data type.

class BaseType
Bases: enum.StrEnum

A base type that can be used in the communication database.

INT8 = 'int8'
This is the mapping of the C type for small numbers.

UINT8 = 'uint8'
This is the mapping of the C type for small unsigned numbers.

BOOL = 'bool'
This is a type, that has only two values: *True* and *False*.

INT16 = 'int16'
This is the mapping of the C type for small numbers.

UINT16 = 'uint16'
This is the mapping of the C type for small unsigned numbers.

INT32 = 'int32'
This is the mapping of the C type for numbers.

UINT32 = 'uint32'
This is the mapping of the C type for unsigned numbers.

INT64 = 'int64'
This is the mapping of the C type for large numbers.

UINT64 = 'uint64'
This is the mapping of the C type for large unsigned numbers.

FLOAT = 'float'
A type for small floating point values.
WARNING: This type does not have a standard size in C.
If the size is not *4* bytes for a platform, the generated code will not compile.

DOUBLE = 'double'

A type for larger floating point values with higher precision.
WARNING: This type does not have a standard size in C.
If the size is not *8* bytes for a platform, the generated code will not compile.

CHAR = 'char'

This type represents a character. It is usually used as an array to represent a string.

BYTES = 'bytes'

This type is similar to the *char* type, but it is used to represent a byte.

type: Type[V]

The Python type representing the type.

name: str

The name of the type.

baseTypeName: Optional[str]

The name of the base type.

description: Optional[str]

A description of the type.

default: Optional[*DefaultValueInfo*[V]]

The default value of the type.

getBaseType(*database:* database.CommunicationDatabase) → *BaseType*

Parameters
database – A database of all known types.

Returns
The base type name of this type.

getSize(*database:* database.CommunicationDatabase) → int

Parameters
database – A database of all known types.

Returns
The size of this type in bytes.

getFormat(*database:* database.CommunicationDatabase, *values: Optional[Dict[str, Any]] = None*) → str

Parameters

- **database** – A database of all known types.
- **values** – An optional dictionary of other values on which the format might depend on. For example, this could include a size member for a dynamically sized list.

Returns
The format string of this type.

getFormats(*database:* database.CommunicationDatabase) → List[str]

Get the format strings for this type. A type that consists of multiple subtypes will yield a concatenated format string. If a type is dynamically sized, a new format string is started that can be formatted with the parsed values from the previous format string to fill the required size information.

Parameters
database – The communication database.

Returns
The format strings.

`getMinNumericValue`(*database:* database.CommunicationDatabase) → Union[int, float]

Get the minimum number that this type can represent.

Raises
`TypeError` – if the type is not numeric.

Parameters
`database` – A communication database.

Returns
The minimum value of this type.

`getMaxNumericValue`(*database:* database.CommunicationDatabase) → Union[int, float]

Get the maximum number that this type can represent.

Raises
`TypeError` – if the type is not numeric.

Parameters
`database` – A communication database.

Returns
The maximum value of this type.

`classmethod lookupBaseType`(*name:* BaseType) → *TypeInfo*

Find a base type by its name.

Parameters
`name` – The name of the base type.

Returns
The TypeInfo of the base type.

`copyWithType`(*typ: Type[V]*) → *TypeInfo*[V]

Copy this TypeInfo but replace the type with the given new type.

Parameters
`typ` – The new type that this type info should contain.

Returns
The copied type info with the new type.

ecom.message

Module Contents

Classes

MessageDatapointType	A base for all message datapoint types.
MessageType	A base for all message types.
TelecommandResponseType	A type of telecommand response.
DependantTelecommandResponseType	A telecommand response that depends on the value of a datapoint of the telecommand.
TelemetryDatapointType	A telemetry response data type.
TelemetryType	A type of telemetry message.
Message	A generic message.
Telemetry	A telemetry message. This is a message that has been sent to the base.
Telecommand	A telecommand message. This is a message that has been sent from the base.
TelecommandDatapointType	A datapoint of a telecommand.
DependantTelecommandDatapointType	A datapoint that depends on the value of another datapoint.
TelecommandType	A telecommand message.

Functions

iterateRequiredDatapoints(...)	Iterate over the datapoints of the given telecommand that are required to serialize the telecommand.

class ecom.message.**MessageDatapointType**

A base for all message datapoint types.

name: str

The name of this datapoint.

type: *ecom.datatypes.TypeInfo*

The type information of this datapoint.

description: Optional[str]

The description of this datapoint.

class ecom.message.**MessageType**

A base for all message types.

id: *ecom.datatypes.EnumType*

An enum value representing this message type.

data: List[*MessageDatapointType*]

A list of datapoints that will be transmitted with this message.

class ecom.message.**TelecommandResponseType**

A type of telecommand response.

name: str

The name of the response value.

typeInfo: *ecom.datatypes.TypeInfo*

The type of the response value.

description: Optional[str]

A description of the response value.

class ecom.message.**DependantTelecommandResponseType**

Bases: *TelecommandResponseType*, abc.ABC

A telecommand response that depends on the value of a datapoint of the telecommand.

provider: *TelecommandDatapointType*

The datapoint that this response is dependent on.

abstract configureWith(*providerValue: Any*) → *TelecommandResponseType*

Create an instance of the dependant response for the value of the provider.

Parameters
providerValue – The value of the provider that this response depends on.

Returns
The configured TelecommandResponseType instance.

class ecom.message.**TelemetryDatapointType**

Bases: *MessageDatapointType*

A telemetry response data type.

class ecom.message.**TelemetryType**

Bases: *MessageType*

A type of telemetry message.

id: *ecom.datatypes.EnumType*

An enum value representing this telemetry type. See CommunicationDatabase.telemetryTypeEnum.

data: List[*TelemetryDatapointType*]

A list of datapoints that will be transmitted with the telemetry.

class ecom.message.**Message**

A generic message.

type: *ecom.datatypes.EnumType*

The type of the message. See CommunicationDatabase.telemetryTypeEnum and CommunicationDatabase.telecommandTypeEnum.

data: Dict[str, Any]

The data that was transmitted with this message.

header: Dict[str, Any]

The data that was transmitted with this message.

class ecom.message.**Telemetry**

Bases: *Message*

A telemetry message. This is a message that has been sent to the base.

class ecom.message.**Telecommand**

Bases: *Message*

A telecommand message. This is a message that has been sent from the base.

class ecom.message.**TelecommandDatapointType**

Bases: *MessageDatapointType*

A datapoint of a telecommand.

default: Optional

The default value for the datapoint.

class ecom.message.**DependantTelecommandDatapointType**

Bases: *TelecommandDatapointType*, abc.ABC

A datapoint that depends on the value of another datapoint.

provider: *TelecommandDatapointType*

The argument that this datapoint is dependent on.

abstract configureWith(*providerValue: Any*) → *TelecommandDatapointType*

Create an instance of this dependant telecommand datapoint for the value of the provider.

Parameters
providerValue – The value of the provider that this datapoint depends on.

class ecom.message.**TelecommandType**

Bases: *MessageType*

A telecommand message.

id: *ecom.datatypes.EnumType*

An enum value representing this telecommand type. See CommunicationDatabase.telecommandTypeEnum.

data: List[*TelecommandDatapointType*]

The datapoints of the telecommand.

response: Optional[*TelecommandResponseType*]

The type information of the return value of the telecommand.

description: Optional[str]

A description of the telecommand.

isDebug: bool

Whether the telecommand is a debugging command.

ecom.message.**iterateRequiredDatapoints**(*telecommand:* TelecommandType) → Iterator[*TelecommandDatapointType*]

Iterate over the datapoints of the given telecommand that are required to serialize the telecommand. Parameters not included in the resulting iterator can be deduced from other parameters.

Parameters
telecommand – The telecommand type whose required datapoints should be iterated.

Returns
An iterator over the required datapoints of the telecommand type.

ecom.parser

Module Contents

Classes

Parser	A parser for messages.
TelemetryParser	A parser for telemetry messages.
TelecommandParser	A parser for telecommand messages.

Attributes

M

exception ecom.parser.**ParserError**(*message: str*, *buffer: bytes*)

Bases: RuntimeError

An error occurred during parsing.

property buffer: bytes

Returns
The parser buffer at the time the error was generated.

ecom.parser.**M**

class ecom.parser.**Parser**(*database:* ecom.database.CommunicationDatabase, *headerType: str*, *messageTypes: List[*ecom.message.MessageType*]*, *verifier: Optional[*ecom.verification.MessageVerifier*] = None*, *maxDynamicMemberSize: int = DEFAULT_MAX_DYNAMIC_MEMBER_SIZE*)

Bases: *ecom.datatypes.CommunicationDatabaseAccessor*, abc.ABC, Generic[*M*]

A parser for messages.

property numParsedBytes: int

Returns
The number of bytes of all successfully parsed messages.

property numInvalidBytes: int

Returns
The number of bytes that could not be parsed into a message.

DEFAULT_MAX_DYNAMIC_MEMBER_SIZE = 512

The default maximum size of a dynamically sized member that the parser will allow during parsing.

parse(*message: bytes*, *errorHandler: Optional[Callable[[*ParserError*], None]] = None*, *ignoredBytesHandler: Optional[Callable[[bytes, int], None]] = None*) → Iterator[M]

Parse messages from the data. Multiple messages might be parsed from the data. If the data contains an incomplete message, it will be added to an internal buffer and parsed once the rest of the data is available.

Parameters

- **message** – The raw data to parse.
- **errorHandler** – An optional error handler for handling parser errors.
- **ignoredBytesHandler** – An optional handler for bytes that will get ignored during parsing. Takes the parsr buffer and the number of bytes that will be ignored.

Returns
An iterator over all parsed messages.

Raises
ParserError – If the data contains a message with invalid data and no error handler is specified.

class ecom.parser.**TelemetryParser**(*database:* ecom.database.CommunicationDatabase, ***kwargs*)

Bases: *Parser*[*ecom.message.Telemetry*]

A parser for telemetry messages.

class ecom.parser.**TelecommandParser**(*database:* ecom.database.CommunicationDatabase, ***kwargs*)

Bases: *Parser*[*ecom.message.Telecommand*]

A parser for telecommand messages.

ecom.response

Module Contents

Classes

ResponseTelemetryParser	A parser and serializer for telemetry and telecommand messages.
ResponseTelemetrySerializer	A serializer that can serialize a response to a received telecommand.
FixedSizeResponseTelemetrySerializer	A serializer that supports sending responses in a data-point that is a list of bytes with a constant size.
VariableSizedResponseTelemetrySerializer	A serializer that supports sending responses in a variable sized response package.

class ecom.response.**ResponseTelemetryParser**(**args*, ***kwargs*)

Bases: *ecom.parser.TelemetryParser*, *ecom.serializer.TelecommandSerializer*

A parser and serializer for telemetry and telecommand messages. It adds extra functionality for response and acknowledge messages:

- Both gain a 'command' data point with the telecommand instance that initiated the response.
- The 'value' data point of the response message is parsed into it's Python representation.

serialize(*telecommand:* ecom.message.TelecommandType, ***kwargs*) → bytes

Serialize the telecommand and its data into bytes.

Parameters

- **telecommand** – The telecommand.
- **kwargs** – Data for the telecommand.

Returns
The serialized telecommand.

parse(*message: bytes*, *errorHandler: Optional[Callable[[ecom.parser.ParserError], None]] = None*, *ignoredBytesHandler: Optional[Callable[[bytes, int], None]] = None*) → Iterator[*ecom.message.Message*]

Parse messages from the data. Multiple messages might be parsed from the data. If the data contains an incomplete message, it will be added to an internal buffer and parsed once the rest of the data is available.

Parameters

- **message** – The raw data to parse.
- **errorHandler** – An optional error handler for handling parser errors.
- **ignoredBytesHandler** – An optional handler for bytes that will get ignored during parsing. Takes the parsr buffer and the number of bytes that will be ignored.

Returns
An iterator over all parsed messages.

Raises
ParserError – If the data contains a message with invalid data and no error handler is specified.

class ecom.response.**ResponseTelemetrySerializer**(**args*, *responseTelemetryName: str = 'RESPONSE'*, *acknowledgeTelemetryName: str = 'ACKNOWLEDGE'*, *responseValueDatapointName: str = 'value'*, ***kwargs*)

Bases: *ecom.serializer.TelemetrySerializer*, abc.ABC

A serializer that can serialize a response to a received telecommand. This handles logic for responses whose content depend on a received telecommand message.

serializeTelecommandResponse(*telecommand:* ecom.message.Telecommand, *value: Any*, *additionalData: Optional[Dict[str, Any]] = None*) → bytes

Serialize a response for the given telecommand with the given value.

Parameters

- **telecommand** – The telecommand message that this response is responding to.
- **value** – The response value to the telecommand.
- **additionalData** – Optional additional data that is required by the response telemetry package.

Returns
The serialized response.

serializeTelecommandAcknowledge(*telecommand:* ecom.message.Telecommand, *additionalData: Optional[Dict[str, Any]] = None*) → bytes

Serialize an acknowledgement for the given telecommand.

Parameters

- **telecommand** – The telecommand message that this response is acknowledging.
- **additionalData** – Optional additional data that is required by the acknowledgement telemetry package.

Returns
The serialized acknowledgement.

class ecom.response.**FixedSizeResponseTelemetrySerializer**(**args*, *maxResponseSize: Optional[int] = None*, *maxResponseSizeConstant: str = 'MAX_TELECOMMAND_RESPONSE_SIZE'*, ***kwargs*)

Bases: *ResponseTelemetrySerializer*

A serializer that supports sending responses in a datapoint that is a list of bytes with a constant size.

class ecom.response.**VariableSizedResponseTelemetrySerializer**(**args*, *sizeDatapointName: str = 'size'*, ***kwargs*)

Bases: *ResponseTelemetrySerializer*

A serializer that supports sending responses in a variable sized response package.

ecom.serializer

Module Contents

Classes

Serializer	A serializer of messages.
TelecommandSerializer	A serializer of telecommands.
TelemetrySerializer	A serializer of telemetry messages.

class ecom.serializer.**Serializer**(*database:* ecom.database.CommunicationDatabase, *headerType: str*, *verifier: Optional[*ecom.verification.MessageVerifier*] = None*)

Bases: *ecom.datatypes.CommunicationDatabaseAccessor*, abc.ABC

A serializer of messages.

class ecom.serializer.**TelecommandSerializer**(*database:* ecom.database.CommunicationDatabase, *telecommandHeaderType='TelecommandMessageHeader'*, *counterMemberName='counter'*, ***kwargs*)

Bases: *Serializer*

A serializer of telecommands.

property **nextTelecommandCounter**: int

Returns
The next telecommand counter that will be used when serializing a telecommand.

serialize(*telecommand:* ecom.message.TelecommandType, ***kwargs*) → bytes

Serialize the telecommand and its data into bytes.

Parameters

- **telecommand** – The telecommand.
- **kwargs** – Data for the telecommand.

Returns
The serialized telecommand.

class ecom.serializer.**TelemetrySerializer**(*database:* ecom.database.CommunicationDatabase, *telemetryHeaderType='TelemetryMessageHeader'*, ***kwargs*)

Bases: *Serializer*

A serializer of telemetry messages.

serialize(*telemetry:* ecom.message.TelemetryType, ***kwargs*) → bytes

Serialize the telemetry and its data into bytes.

Parameters

- **telemetry** – The telemetry.
- **kwargs** – Data for the telemetry.

Returns
The serialized telemetry.

ecom.verification

Module Contents

Classes

MessageVerifier	A helper that can add data to verify the integrity of a message and use that data to verify a message.

exception ecom.verification.**VerificationError**

Bases: RuntimeError

An error indicating that a message failed its verification check.

exception ecom.verification.**MissingDataForVerificationError**

Bases: *VerificationError*

An error indicating that a message failed its verification check because some it is incomplete.

class ecom.verification.**MessageVerifier**

Bases: abc.ABC

A helper that can add data to verify the integrity of a message and use that data to verify a message.

abstract verify(*data: bytes*, *message:* ecom.message.MessageType, *header:* ecom.datatypes.TypeInfo*[*ecom.datatypes.StructType*]*, *messageSize: int*) → Optional[bytes]

Verify the integrity of the message.

Raises
VerificationError – if the message fails the verification check and the original data can't be recovered.

Parameters

- **data** – The raw data of the message. May also include other trailing data.
- **message** – The type of message.
- **header** – The header type information of the message.

- **messageSize** – The size of the message, as calculated with the unverified data.

Returns
None if the message was verified successfully or the bytes of the error corrected message and any bytes from the given data that were not part of the message.

abstract addVerificationData(*data: bytes*, *message:* ecom.message.MessageType, *header:* ecom.datatypes.TypeInfo[ecom.datatypes.StructType]) → bytes

Add extra verification data to a message.

Parameters

- **data** – The message data.
- **message** – The type of message.
- **header** – The header type information of the message.

Returns
The message with verification data.

abstract addPlaceholderVerificationData(*data: Dict[str, Any]*)

When constructing a message, this allows to add a placeholder value to the header that will be filled later by *addVerificationData*.

Parameters
data – A dictionary to store the placeholder verification data to.

2.1.2 Package Contents

ecom.**__version__** = '1.1.1'

CHAPTER

THREE

ECOM (EFFICIENT COMMUNICATION LIBRARY)

A library that provides a way to define a communication between devices. Both high level platforms (via a Python module) and a low-level embedded platforms, like an Arduino (via generated C code) are supported. The communication is as efficient as possible with a focus on minimizing the size of the transmitted payload, and minimizing memory usage and copy operations on the embedded device. It supports variable sized messages, where the size is included as part of the message.

Communication between tow high level platforms (Python to Python), one high level and one low level platform (Python to embedded device) and between two low level platforms (embedded to embedded) are all supported.

The library was originally developed to be used in the ASTER project.

Online documentation is available here.

3.1 Terminology

The communication happens between two sides. One of these sides is considered the `base` and the other side is considered the `secondary device`. The base is usually controlling the secondary device to some extent. In the typical groundstation and satellite case, the groundstation is the base and the satellite represents the secondary device.

Within the ECom system, messages from the base to the secondary device are called `Telecommands`. Messages from the secondary device to the base are called `Telemetry`.

3.2 Examples

Some examples demonstrating the usage of the library can be found in the `examples` directory. This includes an example communication database with further documentation on the database.

3.3 Library

The `ecom` Python module provides access to a communication database (see example). It allows to parse and serialize messages defined in the communication protocol. It also includes a code generator named `ecomUpdate` to generate a serializer and parser in C.

3.3.1 Installation

The `ecom` library can be installed with pip, this requires git to be installed:

```
pip install git+https://gitlab.com/team-aster/software/ecom
```

Alternatively, if the repository is already downloaded locally, it can be installed with the following command:

```
pip install .
```

This will cause the `ecomUpdate` script to be added to the `PATH`, so it can be called from everywhere.

3.3.2 Usage

The library provides a parser and serializer. In case the data is both parsed and serialized, a `ResponseParser` should be used, as it can handle deserialization of telecommand responses.

Parsing

```
from ecom.database import CommunicationDatabase
from ecom.parser import TelemetryParser

# Initialization
database = CommunicationDatabase('/path/to/database')
parser = TelemetryParser(database)
connection = ...

def handleParseError(error):
    print(f'Parsing error: {error}')

# Read loop
for data in connection.read():
    for telemetry in parser.parse(data, errorHandler=handleParseError):
        print(telemetry)
```

Note: The parser provides an `errorHandler` argument, which should always be provided. Otherwise, the parser will throw an exception when it encounters a parsing error, which causes it to ignore all data after the parsing error that was given to that call of `parse`.

Serializing

```
from ecom.database import CommunicationDatabase
from ecom.serializer import TelecommandSerializer

# Initialization
database = CommunicationDatabase('/path/to/database')
serializer = TelecommandSerializer(database)
connection = ...

# Send a telecommand
motorEnabledCommand = database.getTelecommandByName('MOTOR_ENABLED')
```

(continues on next page)

(continued from previous page)

```
telecommandBytes = serializer.serialize(motorEnabledCommand, enabled=True)
connection.write(telecommandBytes)
```

Database verification

The following command will attempt to load the database and can be used to verify the database.

```
python ecom/database.py --dataDir /path/to/database
```

Code generation

The code generation scripts expect a path to a directory which should contain the source code and a path to a directory which should contain the header files. It will generate a `generated` directory in both and write the generated code into them. Invoking the following command will generate the C code in the `path/to/include/directory/generated` and `path/to/source/directory/generated` directory.

```
ecomUpdate --dataDir /path/to/database path/to/source/directory path/to/include/directory
```

Unit tests

The unit tests for the module can be run as follows:

```
python -m unittest discover tests
```

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

U

V