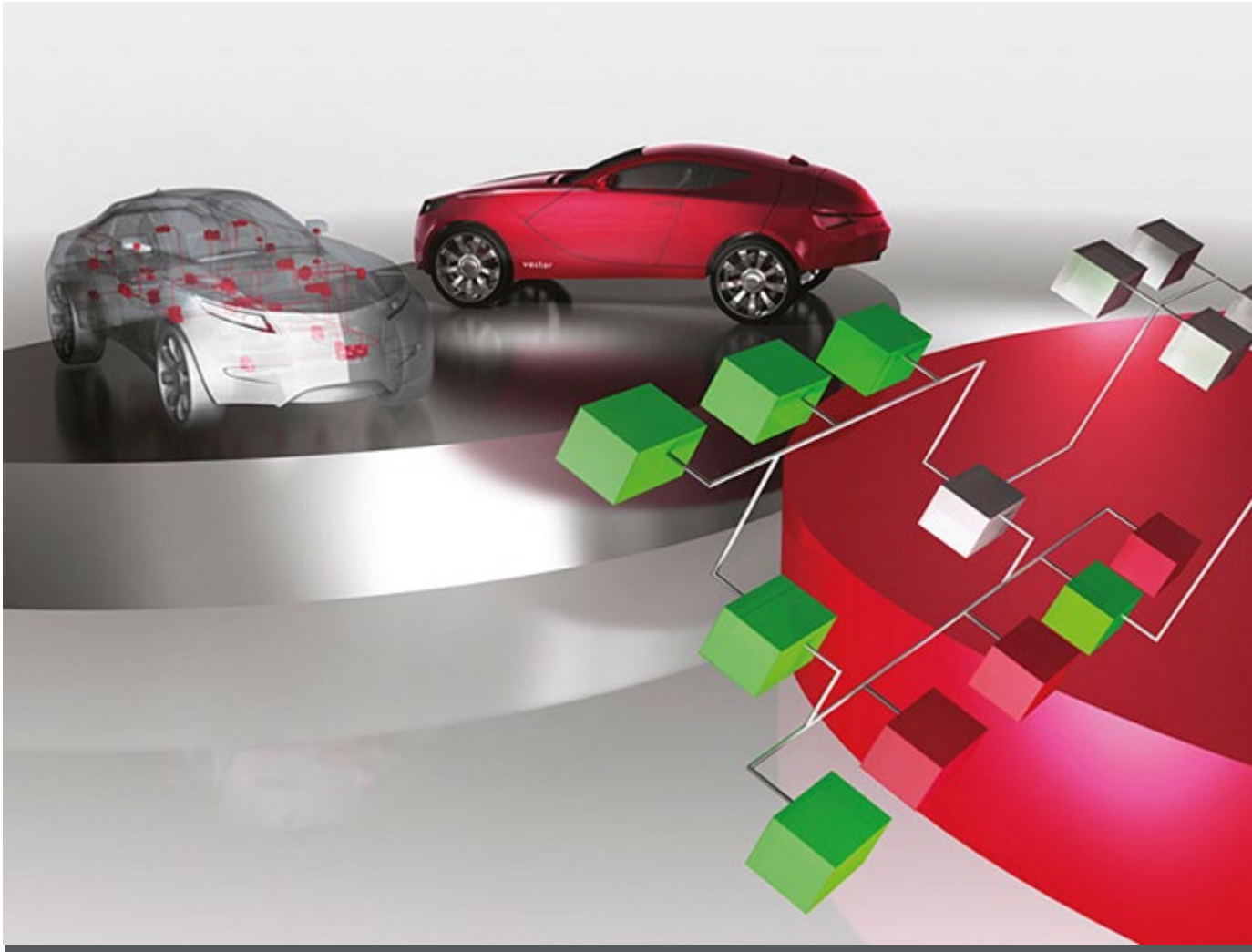
VECTOR



Manual
CANoe FDX Protocol

Version 2.0
English

Imprint

Vector Informatik GmbH
Ingersheimer Str. 24
D-70499 Stuttgart

Table of Contents

1 Introduction

This chapter contains the following information:

1.1 FDX at a Glance

Real-time data exchange

CANoe FDX (**F**ast **D**ata e**X**change) is a protocol for simple, fast, real-time exchange of data between CANoe and other systems via an Ethernet connection. The protocol enables other systems read and write access to CANoe system variables, environment variables and contents of bus messages.

The other system may be, for example, an HIL system on a test bench or a computer used to display CANoe data. The other system is referred to below as the **HIL system**.

1.2 History FDX Protocol

CANoe	Description	Protocol Version
CANoe 7.2 SP5	First release of FDX Feature	1.0
CANoe 7.2 SP6	Status Request Command added	1.0
CANoe 7.6	**kFreeRunningFlag_TransmitAtTrigger** added to FreeRunningRequest Command so that the **FreeRunning** mode can be triggered by a CAPL function.	1.0
CANoe 8.0	> Sequence Number Error Command added > Data types floatarray, doublearrray and int32arrray added	1.1
CANoe 8.0 SP3	Item types frame and PDU added	1.1
CANoe 9.0 SP2	Increment Time Command added	1.2
CANoe 10.0 SP2	Configurable byte order added. For compatibility reasons, CANoe 10.0 SP2 implements FDX protocol version 1.2 and 2.0.	2.0
CANoe 10.0 SP4	> TCP as Transport Layer added > Support for IPv6 added > For compatibility reasons, CANoe 10.0 SP4 implements FDX protocol version 1.2 and 2.1.	2.1

1.3 About this User Manual

1.3.1 Access Helps and Conventions

To find information quickly

The user manual provides you the following access helps::

- at the beginning of each chapter you will find a summary of its contents,
- in the header you see the current chapter and section,
- in the footer you see to which program version the user manual replies.

Conventions

In the two following two tables you will find the conventions used in the user manual regarding utilized spellings and icons.

Style	Utilization
bold	Fields, interface elements, window and dialog names in the software. Accentuation of warnings and notes. **[OK]** Buttons are denoted by square brackets **File** | **Save** Notation for menus and menu commands
CANoe	Legally protected proper names and side notes.
`Source code`	File name and source code.
Hyperlink	Hyperlinks and references.
<Ctrl>+<S>	Notation for key combinations.

Symbol	Utilization
	This icon indicates notes and tips that facilitate your work.
	This icon warns of dangers that could lead to damage.
	This icon indicates more detailed information.
	This icon indicates examples.
	This icon indicates step-by-step instructions.
	This icon indicates text areas where changes of the currently described file are allowed or necessary.
	This icon indicates files you must not change.
	This icon indicates multimedia files like e.g. video clips.
	This icon indicates an introduction into a specific topic.
	This icon indicates text areas containing basic knowledge.

Symbol	Utilization
	This icon indicates text areas containing expert knowledge.
	This icon indicates that something has changed.

1.3.2 Certification

Certified Quality Management System

Vector Informatik GmbH has ISO 9001:2008 certification. The ISO standard is a globally recognized quality standard.

1.3.3 Warranty

Limitation of warranty

We reserve the right to modify the contents of the documentation or the software without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation.

1.3.4 Support

Need support?

You can get through to our hotline by calling

+49 (711) 80670-200

or you can send a problem report to CANoe Support.

1.3.5 Trademarks

Protected trademarks

All brand names in this documentation are either registered or non registered trademarks of their respective owners.

2 General

This chapter contains the following information:

2.1 Protocol Properties

2.1.1 UDP or TCP as Transport Foundation

UDP protocol

The FDX protocol is based on the UDP protocol (IPv4 or IPv6). UDP is a widespread standard protocol, which means there will most likely be an implementation available on the HIL system.

The exchange of data between CANoe and the HIL system is achieved through reciprocal transmission of UDP datagrams. As a matter of principle, the HIL system always sends a datagram to CANoe first and therefore has to know the IP address and port number used by CANoe for the FDX protocol. CANoe evaluates every incoming datagram to determine the sender's IP address and port number and always sends the requested data back to the sender that requested the data.

TCP protocol

For special operation purposes, where a reliable transport of data is more important than realtime issues, you can also choose TCP instead of UPD as transport layer. In this case, CANoe opens a TCP port and listens for incomming connections from the HIL system. After the connection is established, the exchange of data follows the same principle as in the UDP case, apart from that the FDX datagrams are now transfered over the TCP connection.

Transport layer and port number

By default, CANoe uses UDP/IPv4 on port 2809 for the FDX protocol. The transport layer and the port number can be changed in the **Options** dialog of CANoe (**File|Options|Extensions|XIL API & FDX Protocol**).
The exact structure of the datagrams for UDP and TCP is shown in section 2.2 Protocol Description. The datagram structures have been kept purposely simple so that an experienced developer can implement the protocol within one or two days on the HIL system, provided there is already an IP stack available on the HIL system.

2.1.2 Data Groups and Data Types

Data groups

Message contents and variables are predefined in groups for the data exchange and are then always exchanged in groups between CANoe and the HIL system. The groups enable more efficient packing of the data, as it is not necessary to provide an identifier for every single item. The groups are defined in a separate file (see section 2.3 FDX Description File). The actual exchange of data is carried out by means of the **DataExchange** command (see section 2.2.6 DataExchange Command).

Data types

To reduce the bandwidth required for data transfer over the network, the FDX protocol permits the use of different data types for data exchange. To facilitate the data access on the HIL side, these are limited to the data types that are directly available in most C/C++ compilers and most other programming languages.

- int8 (1 Byte signed integer)
- uint8 (1 Byte unsigned integer)
- int16 (2 Byte signed integer)
- uint16 (2 Byte unsigned integer)
- int32 (4 Byte signed integer)
- uint32 (4 Byte unsigned integer)
- int64 (8 Byte signed integer)
- uint64 (8 Byte unsigned integer)
- float: (4 Byte floating point)
- double (8 Byte floating point)
- string: (null terminated ASCII character string)

- bytearray (sequence of individual data bytes)
- floatarray (sequence of 4 byte floating point numbers)
- doublearray (sequence of 8 byte floating point numbers)
- int32array (sequence of 4 byte signed integer numbers)

Byte order

Up to and including protocol version 1.2 the data types are encoded for the x86 architecture (integer values use Little Endian as the byte sequence, signed integer types use the 2's complement, and floating-point values use IEEE format). Protocol version 2.0 introduces a configurable encoding. A bit in the datagram header (see section 2.2.1) determines the byte order, which can be either Little Endian or Big Endian.

Strings are made up of single-byte ASCII characters and are always null-byte terminated. This corresponds to the string format from the C programming language.

Arrays

The **array** data types bytearray, floatarray, doublearray and int32array contain a value (Uint32) in the first 4 bytes indicating the number of data bytes used in the array, followed by the actual data bytes.

Size

When data groups with string, bytearray, floatarray, doublearray and int32array data types are defined it is necessary to specify the maximum length of the string or array in the FDX description file. The length specification for String includes the terminating null bytes and for arrays (bytearray, floatarray, doublearray and int32array) includes the 4 bytes for the number of data bytes.

Offset

For the definition of data groups each signal/variable is assigned a fixed offset (value in bytes). You have to ensure that signals/variables do not overlap in the memory layout.

To enable the use of fixed offsets for the signals/variables within a data group, **strings** and **arrays** always take up the maximum number of bytes (according to the data group definition) in the data transmission. Unused bytes are initialized with a 0 value.

2.1.3 FreeRunning Mode

Cyclical sending of data groups

CANoe sends a data group to the HIL system either in response to an explicit request (see 2.2.5 DataRequest Command) or because **FreeRunning** mode is configured. In **FreeRunning** mode, the data groups are sent cyclically or triggered by a call of the CAPL function **FDXTriggerDataGroup**. Additionally, the data transmission can be carried out just before measurement start and at measurement end.

FreeRunning mode is configured using the **FreeRunningRequest** command (see section 2.2.8 FreeRunningRequest Command). CANoe always sends the data to the UDP or TCP address that issued the request command.

2.2 Protocol Description

Communication via datagrams

The communication between the HIL system and CANoe is implemented through datagrams that are exchanged via UDP or TCP. A datagram consists of the datagram header followed by one or more commands.

Every command begins with the **CommandCode** and the size of the command. By specifying the size of the command it is possible to insert additional fields later without this resulting in incompatibility with the protocol. The **CommandCode** determines what type of command is being used (measurement start, exchange of signals/variables…).

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of this command in bytes
2	2	uint16	commandCode	Command code

The datagram header and the individual commands are described in the following sections.

Datagram length

The maximum length of an FDX datagram depends on the used transport layer. For UDP as transport layer, the FDX datagram must fit into a single UDP datagram. For TCP as transport layer, the limit is the 16 bit length field in FDX datagram header.

Transport Layer	Maximum datagram length
UDP/IPv4	65507 bytes
UDP/IPv6	65527 bytes
TCP/IPv4	65535 bytes
TCP/IPv6	65535 bytes

2.2.1 Datagram Header

The datagram header consists of a signature, a two-digit version number (major and minor version) for the protocol, the number of subsequent commands and a sequence number.

Offset	Size	Type	Field	Description
0	8	uint8[8]	fdxSignature	Signature of FDX protocol. This are always the following eight bytes: 0x43 0x41 0x4E 0x6F 0x65 0x46 0x44 0x58
8	1	uint8	fdxMajorVersion	Protocol version (major part) kFdxMajorVersion = 2
9	1	uint8	fdxMinorVersion	Protocol version (minor part) kFdxMinorVersion = 0
10	2	uint16	numberOfCommands	Number of commands in the datagram.
12	2	uint16	seqNrOrDgramLen	Datagram sequence number (seqNr) of FDX session when

Offset	Size	Type	Field	Description
				using UDP as transport layer or datagram length (dramLen) when using TCP as transport layer.
13	1	uint8	fdxProtocolFlags	Additional protocol flags Bit 0: Byte Order, Little Endian (0) or Big Endian (1) Bits 1-7: unused, should be initialized with 0
14	1	uint8	reserved	1 unused byte for better alignment. This field should be initialized with 0.

Signature

The signature serves as a magic cookie to check whether the datagram was actually intended for CANoe (or whether a program sent the datagram to CANoe by mistake). CANoe ignores all datagrams that have a wrong signature

Version number

The two-digit version number is used to check compatibility. A new protocol version with the same major version number may contain additional information but the datagram can still be processed, except for the new information, by older programs with the same major version number. The protocol remains compatible until the major version number changes.

CANoe 10.0 SP4 implements the FDX protocol version 1.2 and 2.1. If CANoe sends a datagram to the FDX client, CANoe uses protocol version 1.2 if the recently received datagram from this client uses protocol major version 1. CANoe uses protocol version 2.1 if the recently received datagram from the client uses protocol major version 2.

Byte order

The least significant bit of field **fdxProtocolFlags** determines the byte order, that is used by this datagram. Big Endian is used, when the bit is set, Little Endian is used, when the bit is cleared. Protocol version 2.0 or new is required for Big Endian support. If you use protocol version 1.2 or older, this bit must be zero. If CANoe sends a datagram to the FDX client, is always uses the same byte order as used by the recently received datagram from this client. There is no need for the FDX client to implement more than one byte order.

Sequence numbers

When UDP is used as transport layer, then the field **seqNrOrDgramLen** contains the sequence number of the FDX session.

The sequence number enables you to recognize the loss of individual datagrams. The datagrams are numbered sequentially from 0x0001 to 0x7FFF by the sender. The special value 0x0000 is used to start a new counting sequence. To end a counting sequence the current sequence number is ORed with the value 0x8000. The overwrap of sequence numbers goes from 0x7FFF to 0x0001.

Example

Sequence number	Description
0x0000	Special number 0x0000 indicates start of sequence counting.
0x0001	
0x0002	
0x0003	
…	
0x7FFF	Overwrap of sequence number happens from value 0x7FFF to value 0x0001
0x0001	

Example

Sequence number	Description
0x0002	
…	
0x1233	
0x1234	
0x9235	Sequence number 0x1235 is combined with 0x8000 to indicate the end of sequence counting.

Sequence counting

Sequence counting is an optional feature. Use of the special value 0x8000 as sequence number indicates that no sequence counting is being used.

When a datagram is received that indicates the end of sequence counting, CANoe implicitly deletes all remaining open **DataRequests** that are assigned to the sender of the respective datagram and also ends the **FreeRunning** mode.

If CANoe determines any sequence numbering errors in the incoming datagrams, a warning is written to the CANoe Write Window (after the next end of measurement) indicating the number of errors that occurred.

Datagram length

When TCP is used as transport layer, then the field seqNrOrDgramLen contains the length of the FDX datagram in bytes. The FDX datagram lengths includes the 16 bytes of the FDX datagram header and all the bytes of the following FDX commands.

2.2.2 Start Command

Start a measurement

The **Start** command is sent to CANoe by the HIL system in order to start a measurement. If the measurement is already running the **Start** command is ignored.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (4 Bytes)
2	2	uint16	commandCode	kCommandCode_Start = 0x0001

2.2.3 Stop Command

Stop a measurement

The **Stop** command is sent to CANoe by the HIL system in order to stop a measurement. If the measurement is not running the **Stop** command is ignored.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (4 Bytes)
2	2	uint16	commandCode	kCommandCode_Stop = 0x0002

2.2.4 Key Command

Pressing a key

The Key command is sent to CANoe by the HIL system to achieve the effect of pressing a key. In CANoe, the **On Key** handlers in the CAPL programs are then called. Some function blocks in CANoe (e.g. the Trigger block or the Replay Block) can also be configured to respond to keystrokes.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (8 Bytes)
2	2	uint16	commandCode	kCommandCode_Key = 0x0003
4	4	uint32	canoeKeyCode	Key code

2.2.5 DataRequest Command

Query a group of signal/variable values

The **DataRequest** command is sent to CANoe by the HIL system to query a group of signal/variable values a single time. CANoe responds with a **DataExchange** command or with a **DataError** command in case of an error.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (6 Bytes)
2	2	uint16	commandCode	kCommandCode_DataRequest = 0x0006
4	2	uint16	groupID	Identifies the group of signals and variables inside the FDX description file.

2.2.6 DataExchange Command

Exchange a group of signals or variables

The **DataExchange** command is used to exchange a group of signals or variables between CANoe and the HIL system. This command is sent by the HIL system to CANoe in order to set variables and signals in CANoe and is also sent by CANoe to the HIL system to inform the system of the current variable and signal values. In this case, the group has to be defined in an FDX description file (see section 2.3 FDX Description File).

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (8+dataSize)
2	2	uint16	commandCode	kCommandCode_DataExchange = 0x0005
4	2	uint16	groupID	Identifies the group of signals and variables inside the FDX description file.
6	2	uint16	dataSize	Number of bytes of the following array.
8	data Size	uint8[]	dataBytes	Contains the values of signals and variables as defined in the FDX description file.

2.2.7 DataError Command

Data exchange error

The **DataError** command is sent by CANoe to the HIL system as a response to a **DataRequest** command that cannot be processed by CANoe. The **dataErrorCode** field describes the cause of the error:

Data Error Code	Description
1: kDataErrorCode_MeasurmentNotRunning	The measurement is not currently running. Data exchange is only possible when a measurement is running.
2: kDataErrorCode_GroupIdInvalid	The specified **GroupID** does not exist in any FDX description file.
3: kDataErrorCode_DataSizeToLarge	The **DataExchange** command for the specified data group combined with the required datagram header exceeds the maximum length of a FDX datagram. The maximum depends on the used transport layer (see section 2.2 Protocol Description).

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (8 Bytes)
2	2	uint16	commandCode	kCommandCode_Dataerror = 0x0007
4	2	uint16	groupID	Identifies the group of signals and variables inside the FDX description file.
6	2	uint16	dataErrorCode	The reason, why CANoe cannot process the data request command.

2.2.8 FreeRunningRequest Command

Switch on FreeRunning mode

The **FreeRunningRequest** command switches **FreeRunning** mode on for a data group. The command may be sent to CANoe by the HIL system prior to the measurement start.

In **FreeRunning** mode, CANoe sends data to the HIL system independently. The exchange of data can be carried out in the start preparation phase (PreStart), at the end of the measurement, cyclically while the measurement is running or triggered by the call of a CAPL function. One or more of these options can be selected using the flags field. For cyclical transmission, the **firstDuration** field determines when the data group is sent the first time. If **FreeRunning** mode is already configured before the measurement start, then **firstDuration** is the duration from the measurement start to the first transmission of the data group. If the **FreeRunningRequest** command is received while the measurement is running, then the delay of **firstDuration** begins at this point.

At the end of measurement in CANoe all **FreeRunningRequests** are deleted. In addition the **FreeRunningCancel** command can be used to end the **FreeRunning** mode explicitly for a data group while the measurement is running.

If **FreeRunning** mode is already activated for a data group and an additional **FreeRunningRequest** command for the data group is received by CANoe, this request is processed in addition to the al-ready active request.

Free Running Flags	Description
1: kFreeRunningFlag_TransmitAtPreStart	Transmits data group at pre start phase of the CANoe measurement.
2: kFreeRunningFlag_TransmitAtStop	Transmits data group at stop of measurement.
4: kFreeRunningFlag_TransmitCyclic	Transmits the data group cyclically while the measurement is running. The transmit cycle is defined by the fields **cycleTime** and **firstDuration**.
8: kFreeRunningFlag_TransmitAtTrigger	Transmits the data group if the CAPL function **FDXTriggerDataGroup** is called.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (16 Bytes)
2	2	uint16	commandCode	kCommandCode_FreeRunningRequest = 0x0008
4	2	uint16	groupID	Identifies the group of signals and variables inside the FDX description file.
6	2	uint16	flags	The flags describe when the data group will be transmitted.
8	4	uint32	cycleTime	Time period in nanoseconds for the free running mode.
12	4	uint32	firstDuration	Time interval in nanoseconds for the first transmit cycle.

2.2.9 FreeRunningCancel Command

Stop FreeRunning mode

The **FreeRunningCancel** command is sent to CANoe by the HIL system in order to stop **FreeRunning** mode for a data group.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (6 Bytes)
2	2	uint16	commandCode	kCommandCode_FreeRunningCancel = 0x0009
4	2	uint16	groupID	The data group, for which the free running mode should be disabled.

2.2.10 Status Command

Measurement time and status

The **Status** command is sent by CANoe to the HIL system. It contains the current measurement time and status.

The current time has the value 0 if there is no measurement running at the moment.

When CANoe transmits data to the HIL system via **DataExchange** command (in **FreeRunning** mode or in response to a **DataRequest** command), the **Status** command is added to the datagram.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (16 Bytes)
2	2	uint16	commandCode	kCommandCode_Status = 0x0004
4	1	uint8	measurement State	Current state of the measurement.
5	3			3 unused bytes for better alignment
8	8	int64	timestamps	Current time in nanoseconds of a measurement in CANoe.

Sate of Measurement	Description
1: kMeasurementState_NotRunning	Measurement is not running
2: kMeasurementState_PreStart	Start of measurement is prepared
3: kMeasurementState_Running	Measurement is running
4: kMeasurementState_Stop	Measurement is stopping

2.2.11 Status Request Command

Request transmission of status command

The **Status Request** command is sent to CANoe by the HIL system in order to request the transmission of a Status command. The status request command does not require groups to be configured.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (4 Bytes)
2	2	uint16	commandCode	kCommandCode_StatusRequest = 0x000A

2.2.12 Sequence Number Error Command

Sequence Number Error Command

The **Sequence Number Error** command is sent to the HIL system by CANoe, if the sequence number in the header of a received datagram (see section 2.2) is incorrect. This is typically an indication for the loss of one or more previous datagrams.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (8 Bytes)
2	2	uint16	commandCode	kCommandCode_StatusRequest = 0x000B
4	2	uint16	receivedSeqNr	Sequence number of the received datagram
6	2	uint16	expectedSeqNr	Sequence number that was expected by CANoe

2.2.13 Increment Time Command

Increment Time Command

The **Increment Time** command is sent by the the HIL system to CANoe in order to increasse the system time during the measurment in simulated mode. The command is only applicable when the CANoe measurement is running in **slave mode**.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (12 Bytes)
2	2	uint16	commandCode	kCommandCode_IncrementTime = 0x0011
4	4			4 unused bytes for better alignment
8	8	uint64	timestep	Time in nanoseconds for the simulation step

2.2.14 Function Call Command

Function Call Command

Network **Function Call** commands are sent by the HIL system to CANoe in order to invoke a function call on a consumer side function or service method port. The input parameters to the function call are serialized into a byte array according to the definitions in section 2.4 Serialization.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (10 + dataSize)
2	2	uint16	commandCode	kCommandCode_FunctionCall = 0x000C
4	2	uint16	functionID	Identifies the function to be called as defined in the FDX description file.
6	2	uint16	requestID	Identifies the function call instance such that the corresponding response can be matched.
8	2	uint16	dataSize	Number of bytes in the following array
10	dataSize	uint8[]	dataBytes	Contains the serialized input parameters of the function call.

By specifying a unique request identifier in the function call, the response of multiple such calls can be associated with the corresponding call. The response is sent back to the HIL system with the same identifier, while this time the data bytes contain the serialized output parameter and return value (in that order). In general function call parameters are serialized in the order of definition. If the function is marked as a one-way function, then CANoe will not send any response if the function call was successful.

2.2.15 Function Call Error Command

Function Call Error Command

If any Function Call command was not executed successfully, a **Function Call Error** response is sent from CANoe to the HIL system.

Offset	Size	Type	Field	Description
0	2	uint16	commandSize	Size of command (10)
2	2	uint16	commandCode	kCommandCode_FunctionCallError = 0x000D
4	2	uint16	functionID	Identifies the function as defined in the FDX description file.
6	2	uint16	requestID	Identifies the instance of a function call which was rejected.
8	2	uint16	errorCode	The reason why CANoe cannot process the function call command.

Again, the request identifier allows to uniquely match the response to the original function call (assuming the HIL client sends unique identifiers).

The error code returned can have any of the following values:

Error Code	Description
1: kFunctionCallErrorCode_MeasurementNotRunning	The measurement is not currently running. Function calls can only be invoked when a measurement is running.
2: kFunctionCallErrorCode_FunctionIdInvalid	The specified FunctionID does not exist in any active FDX description file.
3: kFunctionCallErrorCode_DataSizeTooLarge	The FunctionResponse command for the specified function combined with the required datagram header will not fit into a single UDP datagram.
4: kFunctionCallErrorCode_ParameterFormat	The deserialization of input parameters to the function call has failed because the data contains an invalid array length or union discriminator.
5: kFunctionCallErrorCode_Timeout	The function call was not answered by the server before it timed out. This error is not sent as an immediate response to a function call invocation, but if the context of a successfully invoked function call is finalized before it got a response.

Error Code	Description
6: kFunctionCallErrorCode_MeasurementStopped	The measurement was stopped before a response for the function call was sent.

2.3 FDX Description File

Description of signals/variables within data groups

The FDX description file is an XML file that describes the message contents and variables within data groups (see section 2.1.2 Data Groups and Data Types). The syntax used to refer to frames, signals, PDUs, system variables and environment variables is based on the syntax for XML test modules. For a sample FDX XML file see section 4.1 FDX Description File.

Note: You can edit these FDX Description (XML) files with the **FDX Editor** provided with CANoe (**Tools** tab). The **FDX Editor** allows FDX descriptions to be graphically created and edited based on XML.

Basic structure

The XML file consists of one or more elements for the description of data groups and has the following basic structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<canoefdxdescription version="1.0">

  <datagroup>
    ...
  </datagroup>

  <datagroup>
    ...
  </datagroup>

  <function>
    ...
  </function>
  ...
</canoefdxdescription>
```

Data group

The data group itself comprises a list of items that describe the frames, signals, PDUs, system variables and environment variables. The **groupID** attribute contains a numeric value that serves as a unique identifier of the data group. The size attribute specifies the number of data bytes used for the transfer of data in the **DataExchange** command (see section 2.2.6 DataExchange Command).

```
<datagroup groupID="..." size="...">
  list of items
</datagroup>
```

Function Calls

The **function** element predefines how a network function should be called. Similar to the value entity specification a port **path** attribute must be given, which in this case must identify a consumer side function or service method port. Further a numerical identifier analogous to the group identifier is specified in attribute **callID**. The number of bytes to reserve in the datagram for the function call and its response is specified by attributes **callSize** and **returnSize** respectively. An optional **representation** attribute allows to specify whether parameter data should be transferred as **Raw, Impl** or **Phys** values. Besides the already known **identifier** sub-element, there are no further children to the **function** element.

```
<function callID="…" callSize="…" returnSize="…" path="…"/>
```

Item

Each item (**item** element) contains a frame, a signal, a PDU, an evironment variable or a system variable. In case of system variables individual members of a struct or generic array can be addressed. The type attribute describes which data type is used to write the value of the signal or variable to the data field of the **DataExchange** command. The offset attribute specifies the position of the value within the data field, and the size attribute describes the required space (in bytes). For the numerical data types, the size is already determined by the data type; for the string and **Bytearray** data types, the size is the maximum length of the string or byte array (see section 2.1.2 Data Groups and Data Types).

```
<item type="..." size="..." offset="…">
  frame, signal, PDU, environment variable or system variable
</item>
```

Value Entity

The **value** element is used to identify a communication object port's value entity or its complex data type member. Its **path** attribute specifies the path to a communication object port. The optional **member** attribute then further identifies a specific value entity of that port and may navigate to complex data type members. The representation attribute allows to transfer the **Raw, Impl** or **Phys** representation of the current value, where **Impl** is the default. Whenever a complex data type is addressed, the item's data type must be "bytearray" and the parameter is serialized into the byte array according to the rules in section 2.4 Serialization.

```
<value path="…" member="…" representation="Raw|Impl|Phys"/>
```

Member Paths

Whenever a value entity shall be addressed, there is possibly more than one value entity associated with the specified port. In addition in case of complex data types, the member path allows to navigate to struct or union submembers *(.Member*) or array elements ([*index*]). The following table gives an overview of possible member paths depending on port type and direction. Any keywords are printed in **bold** typeface while user defined names are printed in *italics*. All enumerated members have defined 16 bit integer values, identical to the C API representation of the same value. This is the case for Service State, Connection State, Subscription State, I/O-Parameter Behavior and VSIM Auto Answer Mode.

Type	Side	Value Entity	Member Spec
Service FEP	Provider	Service State	**.ServiceState**
Service AP	Consumer	Service State	**.ConnectionState**
Service AP	Provider	Connection State	**.ConnectionState**
Signal AP	Sender	Tx Value	[.*Member*]
Signal AP	Receiver	Rx Value	[.*Member*]
PDU AP	Sender	Tx Value	[.*MappedSignal*[.*Member*]]
PDU AP	Provider	Subscription State	**.SubscriptionState**
PDU AP	Receiver	Rx Value	[.*MappedSignal*[.*Member*]]
PDU AP	Consumer	Subscription State	**.SubscriptionState**
Event AP	Provider	Tx Value	[.*Member*]
Event AP	Provider	Subscription State	**.SubscriptionState**
Event AP	Consumer	Rx Value	[.*Member*]
Event AP	Consumer	Subscription State	**.SubscriptionState**
Event FEP	Provider	Model Value	[.*Member*]
Service PDU FEP	Provider	Model Value	[.*Member*]
Function AP	Provider	Latest Call	**.LatestCall**[.*InParam*[.*Member*]]
Function AP	Provider	Latest Return	**.LatestReturn**[.*OutParam*[.*Member*]] **.LatestReturn.Result**[.*Member*]
Function AP	Consumer	Latest Call	**.LatestCall**[.*InParam*[.*Member*]]
Function AP	Consumer	Latest Return	**.LatestReturn**[.*OutParam*[.*Member*]] **.LatestReturn.Result**[.*Member*]
Function AP	Provider	VSIM Defaults	**.ParamDefaults**.*OutParam*[.*Member*] **.ParamDefaults**.*IOParam*[**.Value**][.*Member*] **.ParamDefaults**.*IOParam*.**Behavior** **.DefaultResult**[.*Member*] **.AutoAnswerMode** **.AutoAnswerTimeNS**

Frame

A CAN message can be described with the frame element. A message is identified by its name (**name** attribute). If the name is not unique, additional attributes (database, bus, channel, node) may be used in order to identify the message uniquely.

Only the byte array data type is permissible for a frame element. The length of the byte array must be equal to the length of the message. The 4 bytes containing the number of bytes are not included in the calculation.

The sending of a CAN message to CANoe via FDX is only appropriate if the CANoe interaction layer is used. The content of the message is then updated by CANoe with the received data. The actual sending of the message must be initiated by the interaction layer.

On the other hand, the querying of the content of a CAN message via FDX operates independently of the CANoe interaction layer. The last received content of the message will be transmitted. If the message has not yet been received, a byte array with length 0 is transmitted.

Database

The database name (**database** attribute) only has to be specified if the same message name occurs in two databases.

Bus

The bus (**bus** attribute) or channel (**channel** attribute) only has to be specified if the same database is used for more than one bus or channel.

Sending node

The send node (**node** attribute) only has to be specified if the same message is sent by multiple nodes.

Signal

A signal is described as follows

```
<signal name="..." msg="..." database="..." node="..."
        bus="..." channel="..."
        direction="auto/txrq" value="phys/raw" />
```

A signal is specified by its name (**name** attribute), the name of the message (**msg** attribute) it is sent with and the name of the database (**database** attribute).

Sending node

The sending node (**node** attribute) only has to be specified when the protocol (J1939, NMEA 2000, GM LAN) allows a message to be sent by several nodes.

Bus

The bus (**bus** attribute) or channel (**channel** attribute) only has to be specified when the database is assigned to more than one bus in CANoe. In this case you can specify either the name of the bus in the CANoe Simulation Setup or the channel assigned to the bus (CAN1, CAN2, LIN1,…).

Raw/physical value

The **value** attribute indicates whether the raw value (**value=”raw”**) or the physical value (**value=”phys”**) of the signal is to be written to the data field.

Directions

For every signal, there are two values stored in CANoe. One is the last value transferred successfully on the bus (also known as the Rx value), the other is the next value due to be sent by CANoe on the bus (TxRq value). The Rx value can only be read via the interface described here, whereas the TxRq value can be read or set, although it is typically only set by an external HIL system. The **direction** attribute is used to distinguish which value is read. When set to **direction=”auto”** the Rx value is read with CANoe -> HIL data exchange and the TxRq value is set with HIL -> CANoe data exchange. When set to **direction="txrq"** the TxRq is accessed in both cases.

PDU

A PDU is used to describe a Protocol Data Unit for the FlexRay bus system. A PDU is identified by its name (**name** attribute). If the name is not unique, additional attributes (database, bus, channel, node) may be used in order to identify the PDU uniquely.

Only the byte array data type is permissible for a PDU. The length of the byte array must be equal to the length of the PDU. The 4 bytes containing the number of bytes are not included in the calculation.

If the content of a PDU is queried via FDX, CANoe determines the last received content of the PDU. If the PDU has not yet been received, a byte array with length 0 is transmitted.

```
<pdu name="..." database="..." bus="..." channel="..."
node="..." />
```

Database

The database name (**database** attribute) only has to be specified if the same PDU name occurs in two databases.

Bus

The bus (**bus** attribute) or channel (**channel** attribute) only has to be specified if the same database is used for more than one bus or channel.

Sending node

The send node (**node** attribute) only has to be specified if the same PDU is sent by multiple nodes.

Environment variable

An environment variable is specified using the **envvar** element; its only attribute **name** is the name of the environment variable.

```
<envvar name="..." />
```

System variable

The **sysvar** is used to describe system variables or their struct or array members; in addition to the name (**name** attribute) it also has a **namespace** attribute for specification of the namespace and supports the **value** attribute for physical/raw member access (equivalent to the attribute for signal elements). If system variable structs are used, the item's data type must be **"bytearray"**. The syntax for specifying struct member access is like in the C programming language: **"variable.member"**. For accessing array elements the equivalent square bracket notation from C is used: **"variable[index]"**. Member and array element specification can be mixed and chained (e.g. for multidimensional arrays or an array of structs).

The **value** attribute is optional and defaults to **"raw"**.

```
<sysvar name="..." namespace="..." value="phys/raw"/>
```

Identifier

The **datagroup** and **item** elements can contain an **identifier** element, which assigns a symbolic descriptor to the data group or its items. The identifier is optional and is only used by CANoe for more descriptive error messages; otherwise you can use it however you want (e.g. when generating code for the HIL system).

```
<identifier>CarSpeed</identifier>
```

Element attributes

The following table provides an overview of the available and required attributes of the elements in the XML description.

Element	Attribute	Required	Valid Values	Default Value
canoefdxdescription	version	x	„<x>.<y>“	–
datagroup	groupID	x	0…65535	–
	size	x	0…65535	–
item	offset	x	0…65535	–
	size	(x)[1]	0…65535	According to type specification
	type	x	“int8”, “int16”,… (see types)	–
sysvar	name	x	any string of characters	–
	name-space	x	any string of characters	–
	value		“raw“ , “phys“	“raw“
envvar	name	x	any string of characters	–
signal	name	x	any string of characters	–
	msg	x	any string of characters	–
	database	x	any string of characters	–
	bus	x	any string of characters	–
	channel	x	any string of characters	–
	dir		“auto“, “txrq“	“auto”
	value	x	“raw“ , “phys“	–
frame	name	x	any string of characters	–
	database		any string of characters	–
	bus		any string of characters	–
	channel		any string of characters	–
	node		any string of characters	–
PDU	name	x	any string of characters	–
	database		any string of characters	–

[1] The **size** attribute has to be specified in the **item** element if the type is defined as **string**, **bytearray**, **int32array**, **floatarray** or **doublearray**. Otherwise the specified size has to be at least as much as the default value.

Element	Attribute	Required	Valid Values	Default Value
	bus		any string of characters	–
	channel		any string of characters	–
	node		any string of characters	–
function	path	x	any string of characters	-
	callID	x	0..65535	-
	callSize	x	0..65535	-
	returnSize	x	0..65535	-
	representation		raw, impl, phys	impl
value	path	x	any string of characters	-
	member		any string of characters	“”
	representation		raw, impl, phys	impl

2.4 Serialization

Serialization

The payload of Data Exchange and Function Call commands can contain values with complex data types which may not have a fixed binary layout (e.g. a struct with an optional member or an array with variable length). In that case the value needs to be serialized into the command’s byte array member by member or element by element, retaining the variable information. If the FDX description requests a specific representation, then the values are first converted to that representation before being serialized. In case the selected representation is **Phys**, then any complex data type needs to be serialized member by member again, since the physical representation has no native binary representation which could be copied to the datagram.

Please note that any complex data type value in **Impl** or **Raw** representation is simply copied to the datagram from memory without any further conversion, i.e. it is especially not subject to byte ordering – CANoe expects to receive such values in Little Endian byte order and will also send out values in that byte order.

2.4.1 Primitive Types

Primitive Types

Any primitive data type values are serialized using the memory layout as defined in (see section 2.1.2 Data Groups and Data Types). If the value is a data group item by itself, then its raw, impl or phys value is converted to the data type specified for the item and then written to the datagram. If the primitive value is a whole function call parameter or a member of a complex data type being serialized, then its raw, impl or phys value is written to the datagram directly using its data type defined by the parent complex data type or the function prototype.

2.4.2 Strings

Strings

String values of Value Entities or Function Call parameters as opposed to other string values are UTF-8 encoded. Any such string will be serialized by copying its UTF-8 byte array to the datagram. This is compatible to ASCII strings as long as only characters in the range 0x00 through 0x7F are used.

2.4.3 Unions

Unions

Unions which do not have a fixed binary layout are serialized by first writing a discriminator value. The discriminator is a 16 bit signed integer specifying the index of the currently valid union member (in order of declaration). The value -1 is written if currently no union member is valid. Otherwise the discriminator value is followed by the currently valid union member serialized according to these rules.

2.4.4 Structs

Structs

Structs which do not have a fixed binary layout are serialized member by member in order of declaration. Each member is serialized according to these rules. If a member is optional, then a single byte flag is written to the datagram first, indicating the presence of the optional member's value (1 if it is present, 0 otherwise). In case the value is present it immediately follows the flag.

2.4.5 Generic Arrays

Generic Arrays

Arrays which do not have a fixed binary layout are serialized by first writing the current array length as an unsigned 32 bit integer value to the datagram, followed by the array elements serialized according to these rules in ascending order.

3 Performance

This chapter contains the following information:

3.1 Overview

Bandwidth

Section 3.1.2 Network Load shows that a 100MBit/s Ethernet connection is sufficient for most applications. Today's typical computers as well as Vector's VN8911 device are even equipped with a 1GBit/s Ethernet inter-face. Thus, the Ethernet connection provides more than enough bandwidth.

The performance of CANoe's FDX interface is determined mainly by the available processing power. In addition to being transferred via the Ethernet interface, the data also have to be processed by CANoe. At the same time, CANoe has to process the messages on the connected field bus systems (CAN, LIN, FlexRay…) and also perform a remaining bus simulation. It is the extent of the remaining bus simulation and the processing power of the CPU that determines how much data can be exchanged via the FDX.

3.1.1 VN8911 and VN8914

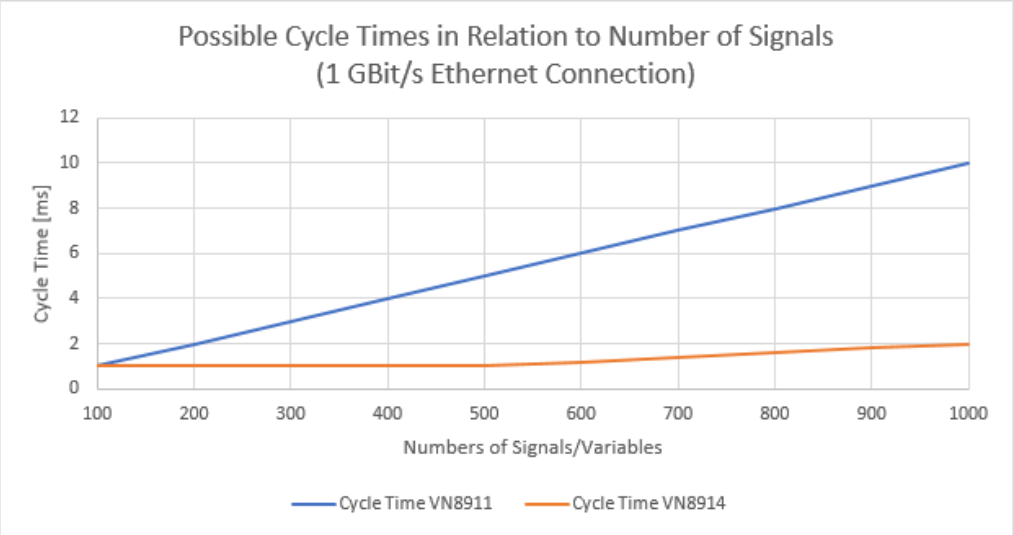
Performance of VN8911

Vector's VN8911 device is equipped with a Intel ATOM E3845 Quad Core processor.

If you exchange 100 signals or variables per millisecond bidirectionally between CANoe and the HIL system this would take about 10-20% of the CPU processing capacity, so there is still sufficient processing power for a remaining bus simulation.

The following diagram shows the cycle times that can be used for bidirectional exchange of a given number of signals/variables between VN8911 and the HIL system.

Given a certain number of signals and variables (data type double) that are to be exchanged bidirectionally between CANoe and HIL system, this diagram displays the cycle time that can be used for the data exchange.



Compared to the VN8911, Vector's VN8914 device (equipped with a Intel Core-i7 6822EQ Quad Core processor) can process about five times as much data.

Note that the cycle time numbers given in the datagram represent average values. Actual inter-arrival times of data sent periodically using FDX data exchange commands are subject to variations. The deviation might be in the low ms range in rare cases depending on, e.g., network traffic and CPU load on both the FDX server (e.g. VN8911) and client computer.

3.1.2 Network Load

The following table shows the network load when data is exchanged between CANoe and the HIL system. This is based on a 100MBit/s Ethernet connection in **Full Duplex** mode and UDP/IPv4 is used as transport layer.
The data is exchanged bidirectionally in a data group at a rate of **n** units of type **int8** (1 byte per unit) or of type **double** (8 bytes per unit); in other words, one UDP datagram is sent in each direction every millisecond on the Ethernet connection.

Number of data units	Network load with data type int8 (1 byte per unit)	Network load with data type double (8 bytes per unit)
1	1%	1%
2	1%	1%
5	1%	1%
10	1%	2%
20	1%	3%
50	1%	7%
100	2%	13%
200	4%	27%
500	9%	66%
1000	17%	99% (full duplex mode required) (approx. 14% with 1GBit/s Ethernet)

3.1.3 Transfer Duration

The transfer duration for a UDP/IPv4 datagram with FDX header, **Status** command and a **DataExchange** command with **n** data units of type **int8** or **double** via Ethernet connection. The value is the calculated period during which the datagram occupies the transfer medium.

Number and type of data units	Payload bytes	Transfer duration	
		1GBit/s Ethernet	100 Mbit/s Ethernet
0	0	6µs [1]	9µs
10 x int8	10	6µs [1]	10µs
10 x double	80	6µs [1]	17µs
100 x int8	100	6µs [1]	19µs
500 x int8	500	6µs	59µs
100 x double	800	9µs	89µs
1000 x int8	1000	11µs	109µs
1000 x double	8000	85µs [2]	854µs [2]

[1] The minimum length of an Ethernet frame for 1GBit/s Ethernet is 520 bytes (64 bytes for 100Mbit/s Ethernet)
[2] The maximum length of an Ethernet frame is 1518 bytes; the UDP datagram is split into 6 Ethernet frames

4 Example

This chapter contains the following information:

4.1 FDX Description File

Defining two data groups

An example for an FDX description file that defines two data groups. Data group **12** contains two signals **AccelerationForce** and **CarSpeed** where each has a numerical value, a system variable **DeviceDescription** of type string and an environment variable **DeviceConfigurationBytes**. The content of data group **13** was omitted.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<canoefdxdescription version="1.0">

  <datagroup groupID="12" size="40">

    <identifier>DataGroup12</identifier>

    <item type="double" size="8" offset="0">
      <identifier>AccelerationForce</identifier>
      <signal name="AccelerationForce" msg="ABSdata"
              database="PowerTrain"
              direction="txrq" value="raw" />
    </item>

    <item type="int16" size="2" offset="8">
      <identifier>CarSpeed</identifier>
      <signal name="CarSpeed" msg="ABSdata"
              database="PowerTrain"
              direction="auto" value="phys" />
    </item>

    <item type="string" size="9" offset="10">
      <identifier>DeviceDescription</identifier>
      <sysvar name="ECU_X" namespace="DeviceDescription" />
    </item>

    <item type="bytearray" size="20" offset="20">
      <identifier>DeviceCfg"</identifier>
      <envvar name="DeviceConfigurationBytes"> </envvar>
    </item>

  </datagroup>

  <datagroup groupID="13" size="1024">
     ...
  </datagroup>

</canoefdxdescription>
```

4.2 FDX Description File Struct/Element Access

Struct/element access

The following is an example for an FDX description file with extended syntax for struct (member) access.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<canoefdxdescription version="1.0">

  <datagroup groupID="1" size="24">
    <identifier>MemberAccess</identifier>

    <item type="int64" size="8" offset="0">
      <identifier>BasicInt</identifier>
      <sysvar name="basic.int" namespace="FDX" />
    </item>

    <item type="double" size="8" offset="8">
      <identifier>BasicFloat</identifier>
      <sysvar name="basic.float" namespace="FDX" />
    </item>

    <item type="double" size="8" offset="16">
      <identifier>BasicPhys</identifier>
      <sysvar name="basic.encoded" namespace="FDX" value="phys"
/>
    </item>

  </datagroup>

  <datagroup groupID="2" size="46">
    <identifier>StructAccess</identifier>

    <item type="bytearray" size="22" offset="0">
      <identifier>InnerStruct</identifier>
      <sysvar name="complex.inner" namespace="FDX" />
    </item>

    <item type="bytearray" size="24" offset="22">
      <identifier>ArrayElement</identifier>
      <sysvar name="array[1]" namespace="FDX" />
    </item>

  </datagroup>

</canoefdxdescription>
```

4.3 UDP Datagram

Datagram format

This example shows a datagram that is sent from the HIL system to CANoe.

The datagram contains the following commands:

> A **DataExchange** command to set the **AccelerationForce** and **CarSpeed** signals and the **DeviceDescription** and **DeviceCfg** variables. The signals are described in the data group with ID **12** (see section 4.1 FDX Description File).

> A **DataRequest** command to request the data group with ID **13**.

Offset	Size	Structure	Field	Signal or Variable with offset from GroupDefinition
0	8	DatagramHeader	fdxSignature	
8	1		fdxMajorVersion	
9	1		fdxMinorVersion	
10	2		numberOf Commands	
12	2		seqNrOrDgramLen	
14	1		fdxProtocolFlags	
15	1		reserved	
16	2	DataExchange Command	commandSize (48)	
18	2		commandCode (5)	
20	2		groupID (12)	
22	2		dataSize (40)	
24	8		dataBytes (40 Bytes)	0: AccelerationForce
32	2			8: CarSpeed
34	9			10: DeviceDescription
43	1			19: unused
44	20			20: DeviceCfg
64	2	DataRequest Command	commandSize (6)	
66	2		commandCode (6)	
68	2		groupID (13)	

CANoe would then typically response with a datagram containing a **Status** command and a **DataExchange** command for the requested data group **13**.

4.4 Byte Array

Bytearrays

This example shows the usage of a **bytearry** in **DataExchange** command.

In the following FDX description file the data group with ID 7 contains a bytearray with maximum 8 data bytes.

Since for arrays the number of the actual used data bytes is transmitted in the datagram, 4 bytes for this number must be added. As a result in the FDX description file a size of 12 bytes must be entered for the bytearry.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<canoefdxdescription version="1.0">
  <datagroup groupID="7" size="12">
    <item type="bytearray" size="12" offset="0">
      <identifier>theArray</identifier>
      <sysvar name="theArray" namespace="ECU_X" />
    </item>
  </datagroup>
</canoefdxdescription>
```

DataExchange command

The **DataExchange** command that transmits the 5 data bytes 0x11, 0x22, 0x33, 0x44 and 0x55 is specified in the following table.

Offset	Size	Data Type	Value	Field
0	2	uint16	20	commandSize
2	2	uint16	0x0008	CommandCode
4	2	uint16	7	groupID
6	2	uint16	12	dataSize
8	4	uint32	5	number of used array elements
12	1	uint8	0x11	data byte 0
13	1	uint8	0x22	data byte 1
14	1	uint8	0x33	data byte 2
15	1	uint8	0x44	data byte 3
16	1	uint8	0x55	data byte 4
17	1	uint8	0x00	data byte 5 (unused)
18	1	uint8	0x00	data byte 6 (unused)
19	1	uint8	0x00	data byte 7 (unused)

More Information

- News
- Products
- Demo Software
- Support
- Training Classes
- Addresses

www.vector.com