FEI COMPANY™
TOOLS FOR NANOTECH

FEI Electron Optics B.V.
Building AAE,
P.O. Box 80066
5600 KA Eindhoven

HEP and ODD Primitives for Microbus

For

Next-Generation Ethernet-Based Communication Infrastructure

DSR-no	:	7322
Authors	:	Kreuwels, Eric; Muyzenberg, Milan van den; Petřek, Jiří
Title	:	HEP and ODD Primitives for Microbus for Next-Generation Ethernet-Based Communication Infrastructure
Project Name	:	NGCOM
Path	:	N:\Documents\NGCOM\HEP and ODD primitives\7322_v1.9.docx
Version (Status)	:	1.10 (Proposal)
Print date	:	2013-05-27
Save date	:	2012-12-10
Category	:	ODD/HEP interface and top-level design specification
Keywords	:	Microbus, NGCOM, object dictionary, HEP, firmware, software
Summary	:	This specifies the object dictionary interface for Microbus compliant devices. It provides the top-level design of HEP primitives.

DOCUMENT HISTORY

Version (Status)	Date	Author	Change Description
0.1 (Initial)	2010-05-06	M. van den Muyzenberg	Start of document, based on the memo provided by Eric Kreuwels and Jiří Petřek.
0.2 (Draft)	2010-05-07	M.van den Muyzenberg	Updated after discussion with Antonín Kubíček
0.3 (Draft)	2010-05-27	M. van den Muyzenberg	Update after F2F in Brno. Only specify types that are relevant for ETEM. Remainder of specification is to be reviewed later.
0.4 (Draft)	2010-06-17	J. Petřek	Update C++ interfaces and callbacks
0.5 (Proposal)	2010-06-25	M. van den Muyzenberg	Updated ADC. Added eTADRegister32. Updated delta-monitor to fit 64 bit.
1.0 (Proposal)	2010-07-13	M. van den Muyzenberg	Updated after formal review
1.5 (Proposal)	2010-11-01	M. van den Muyzenberg	Updated after formal review, updated HEP primitive methods, added `eData` type.
1.6 (Proposal)	2010-02-01	M. van den Muyzenberg	`eData` type: changed subidx 2 access rights from (r) to (r/w)
1.7 (Proposal)	2010-04-07	M. van den Muyzenberg	Updated `eError` with a reference to the section describing error types. Added lifecycle errors for eApplication type.
1.8 (Proposal)	2011-10-26	M. van den Muyzenberg	Added FWBuildNr to the 0x2000 range of the interface of the generic application, Section 4.2
1.9 (Proposal)	2012-11-19	M. van den Muyzenberg	Added InstanceID to generic application, allowing for identification of different instances of the same module type. Specify mandatory application name for generic application. OD entries of generic application: Changed 'Firmware version' to 'FirmwareVersion' to improve naming consistency.
1.10 (Proposal)	2013-05-27	M. van den Muyzenberg	Introduced `eFloat64`. Updated distribution list. Removed approver.

DISTRIBUTION LIST

Name	Department	Location	Roles in the project / Relation to the document
Holt, Arno	Electronics	Acht	Reviewer
Kabel, Martijn	Software	Acht	Reviewer
Kooijman, Kees	Architect	Acht	-
Kreuwels, Eric	Software	Acht	Core team reviewer
Kubíček, Antonín	Electronics	Brno	Core team member
Lapointe, Mike	Electronics	HBO	-
Melville, Ian	Electronics	Acht	-
Muyzenberg, Milan v.d.	Electronics	Acht	Document owner, Core team reviewer
Petřek, Jiří	Software	Brno	Core team reviewer
Prchlík, Martin	Software	Brno	Reviewer
Smits, Arno	Electronics	Acht	Reviewer
Šofr, Pavel	Electronics	Brno	-
Talanda, Michal	Software	Brno	Reviewer
Vlimmeren, Bernard van	Architect	Acht	-

APPROVAL

Name	Department	Function	Date	Signature

TABLE OF CONTENTS:

1 Introduction

1.1 Purpose of the Document

The purpose of this document is to define a set of primitives to be used in the context of the new Ethernet-based communication infrastructure, named Microbus. Compliance with this specification is mandatory for OD-based control of electronics connected to the Ethernet-based microscope control network.

The document specifies:

- standardized object dictionary (OD) primitives,
- a standardized OD
- standardized usage of OD primitives by basic (HAL and Embedded Platform) HEP primitives exposing a C++ interface

This infrastructure is based on programmable electronics that can host multiple applications. Each application has its own OD. The HEP Primitives (software) communicate with/relate to objects in the OD of a particular application on the electronics board (firmware). The first instance of such an electronics board is the so-called Subsystem Control Unit (SCU).

This document will be completed and reviewed incrementally. Relevant items are black. Future items are light grey and must be considered volatile…

1.2 Scope of the System/Function/Component

1.2.1 Contents

This document specifies types, units, and address space for HEP and ODD primitives in Section 0. Section 3 provides a detailed description of the OD interface of all primitives, including their C++ software interface. Section 0 gives an overview of standardized OD entries. Appendices illustrate the application of a number of primitives.

This document uses the terms client and server. By client, we mean the party remotely accessing the object dictionary. Typically, the client is the microscope PC software. By server we mean the party implementing the object dictionary. Typically, that is firmware running on embedded hardware, connected to the microscope PC via Ethernet.

1.2.2 Intended audience

The intended audience of this specification consists of architects, system engineers, software/firmware designers/engineers, and service/PE engineers.

We assume the reader is familiar with [R_1].

1.3 Definitions, Abbreviations & Acronyms

1.3.1 Definitions

Below in Table 1 you will find the definitions used in this document.

Table 1: Definitions

Definition	Description
Application	A grouping of coherent functionality.
Application Lifecycle	The sequence of defined statuses an application traverses.
Asynchronous execution	After initiating execution of some activity (e.g. by means of calling a software function), control immediately returns to the initiating party, while execution of the spawned activity continues. Opposite of synchronous execution.
Cyclic buffer	A data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

Hysteresis	Hysteresis refers to systems that have memory, where the effects of the current input (or stimulus) to the system are experienced with a certain delay in time.
Little-endian	Byte ordering where least significant byte is stored first.
HEP primitive	An FEI standardized object type in the HAL and Embedded Platform layer of FEI microscope server software used for communication via the FEI Ethernet-based communication infrastructure named Microbus.
Object dictionary	An array of variables with a 16-bit index.
ODD primitive	An FEI standardized object dictionary type used for communication via the Ethernet-based communication infrastructure named Microbus.
Synchronous execution	After initiating execution of some activity (e.g. by means of calling a software function), control will return when the spawned activity has finished. Opposite of asynchronous execution.

1.3.2 Abbreviations & Acronyms

Below in Table 2 you will find the abbreviations and acronyms used in this document.

Table 2: Abbreviations & Acronyms

Abbreviation & Acronyms	Description
ADC	Analog to Digital Converter
CAN	Controller Area Network
CANopen	Communication and communication protocol/primitive profile spec on top of CAN
DAC	Digital to Analog Converter
FEICAN	A FEI specific implementation of CAN
HAL	Hardware Abstraction Layer
HEP	HAL and Embedded Platform
OD	Object Dictionary
ODA	Object Dictionary Access
ODD	Object Dictionary Definition
ODDI	Object Dictionary Primitive Index
ODSI	Object Dictionary Sub-Index
SCU	Subsystem Control Unit
SI	Système Internationale d'unités

1.4 References

Below in Table 3 you will find a list of documents to which this document is referring.

Table 3: References

No	Doc. ID	Doc. Date	Version (Status)	Document title	Author(s)	AR*
[R_1]	OCR-09-0047		0.12 (Draft)	CRD for SCU Firmware	Kubíček, Antonín; Muyzenberg, Milan van den	
[R_2]	N/A	N/A	Version number in file must match version of this specification, i.e. v1.9.	CORE_PVOB\hep_ms\HEP\sdk\include\HepCommunicationLibrary\HepLibraryErrors.h	Petřek, Jiří	
[R_3]	TBD	TBD	TBD	CRS / CDD for Microbus SW For FEI Common Ethernet-based Communication Infrastructure	Petřek, Jiří	
[R_4]	E-ISBN 978-0-7381-5752-8	20090829		754-2008 IEEE Standard for Floating-Point Arithmetic	IEEE	

[R_5]	DSR-7087	2011012 4	1.7 (Proposal)	Bootstrapping Protocol for FEI Common Ethernet-Based Communication Infrastructure	Muyzenberg, Milan van den	

***AR:** Authorization required means that this document can be accepted or approved only after the referenced document had been approved.

<This page is intentionally left blank>

2 Overview of HEP Primitives

2.1 Relation to (FEI) CANopen

Conceptually, the way software interfaces to electronics firmware will remain similar to FEICAN-based boards. This is to limit the impact on the client software. The Microbus compliant Object Dictionaries (ODs) are CANopen inspired only. They are not CAN-compliant:

- CANopen targets boards, instead of applications running on a SCU board.
  Decisions:
  - The standardized fields in the 0x1000 range will be redefined.
  - All OD entries are encoded little-endian
- Our communication is Ethernet-based instead of CAN-based.
  Decision:
  - We get rid of the size/type restrictions of CANopen
- We aim for more OD interface abstraction:
  Decision:
  - We standardize primitives on the OD. There is, for example, just one DAC interface for all linear DACs.
  - The firmware abstracts the physical DACs used on the board
  - OD primitives are self-describing
    - The OD specifies all relevant board specific hardware properties of the OD primitives (primitive type, actual ranges, calibrations, resolutions).
    - The client fetches information about primitives from the server at run-time.
- More hardware identification support;
  - The SCU hardware infrastructure allows for identification of several levels of hardware, including secondary hardware (a.k.a. satellite boards) connected to an application board.

2.2 Predefined OD Types:

All OD primitives are identifiable by a primitive type of 8 bits. We specify the following primitive types:

```
typedef enum
{
   eUndefined     = 0x00,     // Reserved
   eVersion3_8    = 0x01,     // Version X.Y.Z; stored as 3 numbers of 8 bits
   eString        = 0x02,     // String
   eData          = 0x03,     // Dynamic-sized Data (e.g. for calibration tables)
   eError         = 0x04,     // 32-bit Error register
   eState         = 0x05,     // 32-bit State register
   eCommand       = 0x06,     // 32-bit Command register
   eDAC_LIN       = 0x07,     // Linear DAC
   eADC_LIN       = 0x08,     // Linear ADC
   eTripMonitor   = 0x09,     // 2x64-bit ADC triggered trip levels
   eDeltaMonitor  = 0x0a,     // 64-bit ADC triggered delta monitor
   eGroupSwitch   = 0x0b,     // 32-bit Group switch controls individual bits (on/off type)
   eNumberSwitch  = 0x0c,     // 16-bit Number switch controls one switch with multiple positions
   eConfiguration = 0x0d,     // 32-bit Unsigned Integer Register for configuration items
   eFloat64       = 0x0e,     // 64-bit Register
   eID8           = 0x0f,     // Array of 8-bit IDs
   eID16          = 0x10,     // Array of 16-bit IDs
   eApplication   = 0x11,     // Type describing application, and allowing for lifetime control
   eNullPrimitive = 0xfe      // Type used to indicate last primitive in OD range
} ePrimitiveTypes_t, *pePrimitiveTypes_t;
```

2.3 Usage of Units

We define the following enumeration type that defines types that are used throughout the Microbus Framework (e.g. in DACs, ADCs).

```
typedef enum eUnits
{
   eUnit_NULL              = 0x00,   // No unit
   eUnit_LENGTH            = 0x01,   // Length, displacement in meters (m)
   eUnit_MASS              = 0x02,   // Mass in kilograms (kg)
   eUnit_TIME              = 0x03,   // Time in seconds (s)
   eUnit_TEMPERATURE       = 0x04,   // Thermodynamic Temperature in Kelvin (K)
   eUnit_AMOUNTSUBSTANCE,  = 0x05,   // Amount of Substance in moles
   eUnit_LUMINOUSINTENSITY = 0x06,   // Luminous Intensity in candela (cd)
```

```
   eUnit_FREQUENCY        = 0x07,   // Frequency (Hertz)
   eUnit_FORCE            = 0x08,   // Force (N = kg m/s2)
   eUnit_PRESSURE         = 0x09,   // Pressure/Stress in Pascal (N/m2)
   eUnit_ENERGY,          = 0x0a,   // Joule (kg m2/s2)
   eUnit_ELECTRICPOTENTIAL= 0x0b,   // Voltage (J/C)
   eUnit_ELECTRICCURRENT  = 0x0c,   // Electric current (ampere)
   eUnit_ANGLE            = 0x0d,   // Radians
   eUnit_CAPACITANCE,     = 0x0e,   // Farad (Charge over Potential: C/V)
   eUnit_CHARGE,          = 0x0f,   // Coulomb (A s)
   eUnit_DENSITY,         = 0x10,   // Amount of mass in every cubed unit length kg / m3
   eUnit_ELECTRICFIELD,   = 0x11,   // Electric potential per meter (V / m)
   eUnit_ELECTRICFLUX,    = 0x12,   // Electric potential times meter (V m)
   eUnit_ELECTRONVOLT,    = 0x13,   // Electron Volt (eV)
   eUnit_ENTROPY,         = 0x14,   // Entropy (J / K)
   eUnit_MAGNETICFIELD,   = 0x15,   // Tesla (Wb/m2)
   eUnit_MAGNETICFLUX,    = 0x16,   // Magnetic flux (kg m2 s2/A)
   eUnit_MOMENTUM,        = 0x17,   // Momentum (kg m/s)
   eUnit_POWER,           = 0x18,   // Amount of work done in any given time (J/s)
   eUnit_REUNITSTANCE,    = 0x19,   // Ohm (V/A)
   eUnit_TORQUE,          = 0x1a,   // Torque (N m)
   eUnit_VELOCITY,        = 0x1b,   // Displacement per unit of time (m / s)
   eUnit_ACCELERATION,    = 0x1c,   // Change in velocity per unit of time (m/s2)
   eUnit_JERK,            = 0x1d,   // Change in acceleration per unit of time (m / s3)
   eUnit_PERCENTAGE       = 0x1e,   // Percentage
   eUnit_RPM              = 0x1f,   // Revolutions Per Minute
   eUnit_Gain             = 0x20,   // Gain factor/amplification
   eUnit_PPM              = 0x21    // Parts Per Million
} eUnit_t, *peUnit_t;
```

2.4 Reserved Address Ranges

General Address range convention:

- Microbus-standardized OD entries start at index 0x1000
- Application-specific OD entries start at Index 0x2000
- TAD-related OD entries start at Index 0x8000

Within each range, primitives are placed adjacent in the OD, i.e. there is no hole in the object index numbering.

Each index range must be terminated with a primitive of type `eNullPrimitive`.

Within each range, at least one primitive must be present[1].

[1] So, if one and only one primitive is present in a range, it must be of type `eNullPrimitive`.

3 Specification of Standardized HEP Primitives

This chapter describes the C++ interfaces of the HEP primitives and their corresponding OD primitives from the client point of view. For a detailed explanation of navigation and usage of HEP primitives, we refer the reader to [R_3]

3.1 The OD Structure

The OD is organized similar to CANopen, where each primitive is addressed by an *index* and a *sub-index.*

Indexing convention:

- OD primitives cover a single index.

Sub-indexing convention:

- Sub-index 0 is reserved for specification of the OD primitive type
- Sub-index 1 is reserved for specification of the OD primitive name
- The interpretation of the remaining sub-indices [2…n] depends on the value of sub-index 0, i.e. the type of the primitive

The binding to a particular primitive in the object dictionary on the server (firmware) from the client (software) is based on the primitive type <u>and</u> the primitive name. This implies that for all primitives of the same type the primitive name needs to be unique[2].

The client must access all primitives via the HEP primitive interfaces. HEP exposes no additional interfaces to directly access the object dictionary via the object index and sub-index.

Note that one cannot assume the same object to be located at the same index given subsequent versions of object dictionaries.

This specification uses the following convention for type specification:
TypeName[(*access rights*), *NrOfBits*]

The following characters define access rights from the client point of view:

- r: read access
- w: write access
- c: read access, value will not change over lifetime of the server

The typename *VisibleString* is used for defining the type of values at particular locations in the primitives. *VisibleString* is a null-terminated string of unsigned characters with admissible values of 0x0 and the range from 0x20 to (and including) 0x7E.

The following table contains the sub-indices that are mandatory for every object:

Object index	**SubIdx[0]**	**SubIdx[1]**
ODDI	PrimitiveType[(c), 8b]	PrimitiveName as *VisibleString*[(c)]

Note 1: ODDI = OD Primitive Index; Primitive Type = Any primitive type (see 2.2)
Note 2: The primitive name at sub-index 1 is null-terminated.

On the client, all HEP primitives derive from:

```
public IHepODPrimitive
{
public:
   // Gets type of the object @ subidx 0
   virtual ePrimitiveType_t GetType() = 0;
   // Gets name of the object @ subidx 1
   virtual string GetName() = 0;
   virtual E_HEPLIB_Result Initialize() = 0;   // Initialize primitive; read values from HW
   virtual E_HEPLIB_Result Uninitialize() = 0; // Un-initialize primitive
};
```

For information about the `IHepTransaction and TTransactionResponse` types, the reader is referred to HEP documentation[3]

[2] To avoid confusion, it is good practice to have unique names over an entire object dictionary, not only in the namespace of a particular primitive type.
[3] At the moment of writing this specification, a document ID was not available.

All OD structures in this chapter extended the OD basic structure described above, i.e. the first two sub-indices are used for primitive type and primitive name

It is mandatory to provide values for all elements of an object.

3.2 OD Interaction

This section describes the interaction between client (typically microscope PC software) and server (typically embedded firmware).

Interaction between client and server is asynchronous with respect to manipulation of primitives on the server. So, upon successful return of a Set…() call on a primitive, one only knows that communication to the server was successful, that the object/sub-index exist on the server, and that no read/write access violations occurred. Successful return of a Set…() call does not mean that the primitive actually has the new value set. Figure 1 depicts this asynchronous behavior.

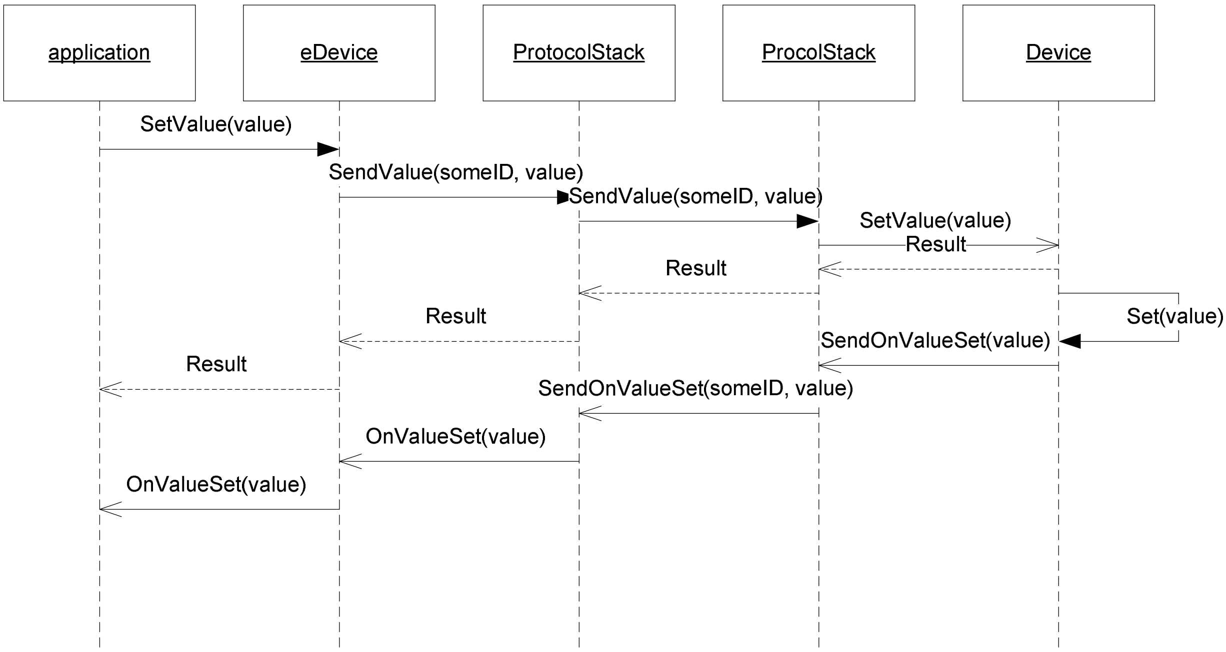


Figure 1 - Asynchronous Manipulation of a Primitive

To allow for implementation of Set…() calls with blocking semantics, objects offer callback functions. One needs to implement these functions with blocking semantics on top of the asynchronous calls and the callback functions that the primitives provide. Figure 2 shows how to implement synchronous setters.

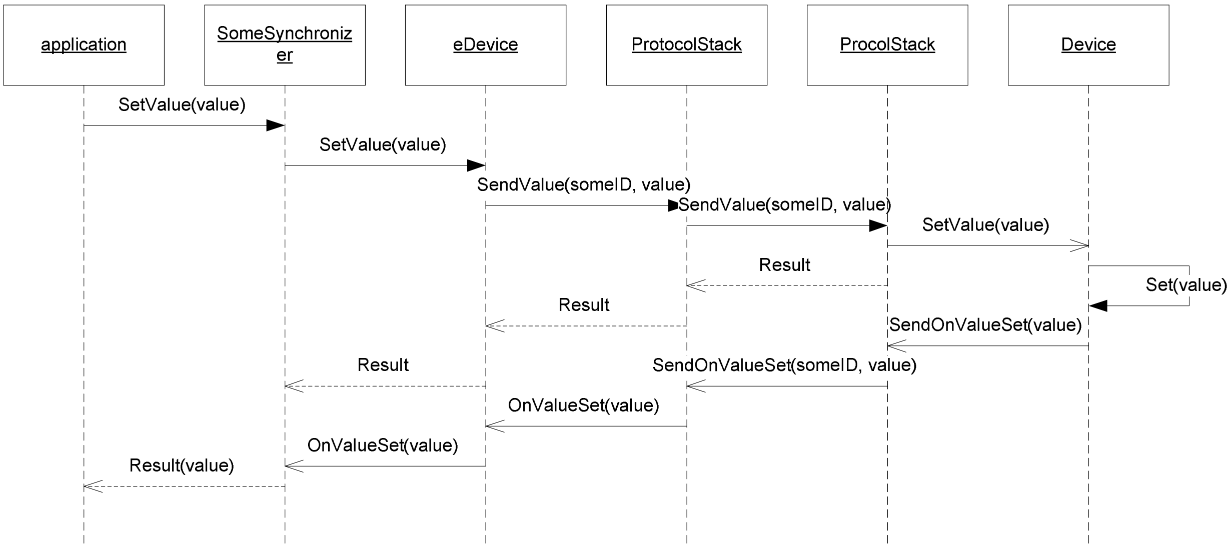


Figure 2 – SomeSynchronizer Is Used to Make Manupulation Synchronous

Interaction between client and server is synchronous with respect to the retrieval of data from primitives on the server. So, upon successful return of a `Get…()` call on a primitive, one knows that communication to the server was successful, that the object/sub-index exist on the server, and that no read/write access violations occurred. Moreover, the output parameter of the `Get…()` call will contain the data that was retrieved from the server.

3.3 Escalation of Errors

Two types of errors can be distinguished:

1. Non-application-specific errors. Typically, these relate to usage of out-of-bound values, to (network) errors in the communication, or to violation of access rights on primitives.
2. Application specific errors. Examples: issuing of a command to an application that is not allowed in that application state, or the occurrence of an error while reading a value from an ADC.

Errors of the first type are expressed by the return values of C++ calls on primitives. Errors of the second type are communicated by using application specific primitives of type `eError`, see Section 0.

3.4 Error Types

Primitives of type `eError` and `eApplication` expose error values on their interface. Error values are 32-bit values. The semantics of these values are as follows:

- The top byte specifies the error type (thus, the error type mask equals 0xFF000000)
- The semantics of the remaining bytes are determined by the error type.

Error Type	Byte 0 (type)	Byte 1	Byte 2	Byte 3
`eErrorWithReference`	0x0	object idx [1]	object idx[0]	error value
`eErrorWide`	0x1	error value[2]	error value[1]	error value[0]

Errors of type `eErrorWithReference` contain a reference to the object in the object dictionary that raised the particular error.

3.5 Alignment

All values in the OD are right-aligned.

3.6 Types

If not mentioned otherwise, values in OD fields are of unsigned type.

64-bit numbers are either doubles (consistent with IEEE754-2008 [R_4]), or integers. The actual format is indicated where 64-bit values are used.

Values of boolean type are defined as follows:

- false ≡ 0
- true ≡ ¬false

3.7 C++ Interfaces

3.7.1 Navigation and Usage

For detailed explanation of navigation and usage of C++ interface, we refer the reader to [R_3].

3.7.2 Semantics and Return Values of Methods

[R_2] specifies return values and more detailed semantics of C++ interface functions. Note that this file must refer to the proper version of this specification.

3.8 eVersion3_8

3.8.1 Behavior

Versions of type `eVersion3_8` are formatted as three unsigned 8-bit numbers that represent a version as X.Y.Z. This version number may be used for any entity, such as hardware, software, firmware, etc.

For firmware/software, the following semantics apply:

X: Major version. SW/FW with a higher numbers may expect different/additional hardware.
Y: Release number. SW/FW with a higher number may include new functions and extend the OD interface, but is backwards compatible with lower release numbers within the same major version.
Z: Bug fix releases/new builds. SW/FW with a higher number implements no functional changes, and is fully downwards compatible.

3.8.2 OD structure

Object index	**SubIdx[2]**	**SubIdx [3]**	**SubIdx [4]**
ODDI	num ‘X’ [(c), 8b]	num ‘Y’ [(c), 8b]	num ‘Z’ [(c), 8b]

Note 1: ODDI = OD Primitive Index; Primitive Type = `eVersion3_8` (see 2.2)

3.8.3 C++ interface

```
class IHepVersion3_8: public IHepODPrimitive
{
public:

   virtual E_HEPLIB_Result Get(UINT8& X, UINT8& Y, UINT8& Z) = 0; // gets version digits
   virtual E_HEPLIB_Result Get(string& Version) = 0;              // gets version string: X.Y.Z
};
```

3.9 eString

3.9.1 Behavior

Typically, strings are defined at startup of the server, and therefore do not change at run-time.

Writing text to firmware is considered obscure, but if needed, the `eData` primitive type allows writing of any data, including text, to the server.

3.9.2 OD structure

Object index	**SubIdx [2]**
ODDI	*VisibleStringn*[(r), *n* bytes]

Note 1: ODDI = OD Primitive Index; Primitive Type = `eString` (see 2.2)
Note 2: *n* equals the size of the buffer required to store the string (so including a terminating zero).

3.9.3 C++ interface

```
class IHepString: public IHepODPrimitive
```

```
{
public:
   virtual E_HEPLIB_Result Get(string& hepString) = 0; // gets the string
};
```

3.10 eData

3.10.1 Behavior

This data type is used for transfer of data of arbitrary type. Examples of data transfer:

- Extended logging to the PC
- Read/write (calibration) tables from/to the server

3.10.2 OD structure

Object index	**SubIdx [2]**	**SubIdx[3]**	**SubIdx[4]**	**SubIdx[5]**
ODDI	Actual Size in bytes [(r/w), 16b]	Max Size in bytes [(r), 16b]	Data [(r/w/c), *n* bytes]	DataChanged [(r), bool]

Note 1: ODDI = OD Primitive Index; Primitive Type = *eData* (see 2.2)

The element at sub-index 5 is used for event notification only. Reading of this value by the client is discouraged.

3.10.3 C++ interface

```
class IHepData: public IHepODPrimitive
{
public:
    virtual E_HEPLIB_Result Read(UINT8*& pData, UINT16& size) = 0; // gets pointer to the data
    virtual E_HEPLIB_Result Write(const UINT8* pData, UINT16 size) = 0; // writes a buffer
    virtual E_HEPLIB_Result GetDataSize(UINT16& dataSize) = 0; // Actual data size in bytes
    virtual E_HEPLIB_Result GetMaxDataSize(UINT16& maxDataSize) = 0; // Maximum data size
                            in bytes
    //callback
    boost::function<void(void)> onChange; // hook for monitoring data changes
};
```

3.11 eError

3.11.1 Behavior

See Section 3.4 for a description of the available error types.

The semantics of the error value are application-dependent:

- Each bit represents the fact that a particular error occurred. Zero means that there are no errors. This type of usage allows for a maximum of 24 possible errors in case the error is of type `eErrorWide` or 8 possible errors in case the error is of type `eErrorWithReference`.
- The error value register represents the error value of the error that occurred. This type of usage allows for a maximum of 2^{24} possible errors in case the error is of type `eErrorWide` or 2^{8} possible errors in case the error is of type `eErrorWithReference`

Errors are reported to the client, and are added to the history when they occur. An error is cleared when it is resolved. If the history size exceeds the maximum size, the oldest errors are overwritten (cyclic buffer behavior).

To avoid race conditions, reading of sub-index 3 and 4 must be done atomically, i.e. reading of values of these sub-indices must be done in a single transaction.

3.11.2 OD Structure

Object index	**SubIdx [2]**	**SubIdx[3]**	**SubIdx[4]**	**SubIdx[5]**
ODDI	Current Error	Error History [(r),	Oldest Error Index	History Size [(r), 8b]

	[(r), 32b]	array of 32b dwords]	[(r), 8b]	

Note 1: ODDI = OD Primitive Index; Primitive Type = `eError` (see 2.2)
Note 2: Oldest Error Index points to the index in the error history containing the oldest error.
Note 3: History Size describes the number of entries in the error history.

3.11.3 C++ interface

The following error types are defined:

```
typedef enum eErrorType
{
   eErrorWithReference = 0x0,
   eErrorWide = 0x1
};

class IHepError: public IHepODPrimitive
{
public:
   virtual E_HEPLIB_Result GetError(UINT32& retValue) = 0; // gets error bits
   virtual E_HEPLIB_Result GetErrorHistory(vector<UINT32>& errorArray) = 0; // gets error history
                           sorted
   virtual E_HEPLIB_Result ErrorType(const UINT32& errCode, eErrorType& errorType) = 0; // get
                           type of error
   virtual E_HEPLIB_Result TranslateErrorWithReference(const UINT32& baseErrCode, wstring&
                           primitiveName, UINT8& errCode) = 0;     // translate error code
   virtual E_HEPLIB_Result TranslateErrorWide(const UINT32& baseErrCode, UINT32& errCode) = 0;
   boost::function<void(const UINT32& val)> onChange; // hook for monitoring error changes
};
```

3.12 eState

3.12.1 Behavior

The semantics of the state register are application-dependent:

- Each bit represents a particular state (mode) is active. The modes are either on or off (Boolean).
- The state register represents the actual state of one state machine that has multiple states.

3.12.2 OD structure

Object index	**SubIdx [2]**
ODDI	State [(r), 32b]

Note: ODDI = OD Primitive Index; Primitive Type = `eState` (see 2.2)

3.12.3 C++ interface

```
class IHepState: public IHepODPrimitive
{
public:
    virtual E_HEPLIB_Result Get(UINT32& retValue) = 0;              // gets status value
    virtual E_HEPLIB_Result PrepareGet(IHepTransaction& transaction) = 0;
    virtual E_HEPLIB_Result ParseGet(TTransactionResponse& response, UINT32& retValue) = 0;

    virtual boost::function<void(const UINT32& state)> onChange;     // hook for monitoring
};
```

3.13 eCommand

3.13.1 Behavior

Commands can be written to the command register

1. This register will contain the last command written to it, until it has been executed. At completion the register will be set to the reserved command `NoCommand`, which has value 0xFE1CFE1C.
2. Command value zero is reserved for `Cancel`. It is optional for the server to support cancelation of commands.

3. As long as execution of a command is in progress, new commands will be refused, except for the Cancel command. In case of a command being refused, the `Set()` function will return an error indicating this.
4. After completion of a command, it will be put in the 'Previous Command' register, also in case it is a `Cancel` command.

Commands must be set sequentially. If a new command is set before the previous command has finished, an error will be raised.

3.13.2 OD structure

Object index	SubIdx[2]	SubIdx[3]
ODDI	Command[(r/w), 32b]	Previous Command[(r), 32b]

Note: ODDI = OD Primitive Index; Primitive Type = `eCommand` (see 2.2).

3.13.3 C++ interface

```
class IHepCommand: public IHepODPrimitive
{
public:
   virtual E_HEPLIB_Result Get(UINT32& retValue) = 0;          // gets command value
   virtual E_HEPLIB_Result Set(const UINT32& value) = 0;       // sets command value
   virtual E_HEPLIB_Result GetPrevious(UINT32& retValue) = 0;  // gets previous command value

   virtual E_HEPLIB_Result PrepareSet(IHepTransaction& transaction, const UINT32& value) = 0;
   virtual E_HEPLIB_Result ParseSet(TTransactionResponse& response) = 0;
   virtual E_HEPLIB_Result PrepareGet(IHepTransaction& transaction) = 0;
   virtual E_HEPLIB_Result ParseGet(TTransactionResponse& response, UINT32& retValue) = 0;
   // Hook for monitoring
   virtual boost::function<void(const UINT32& commandComplete)> onComplete;
};
```

3.14 eDAC_LIN

3.14.1 Behavior

Figure 3 illustrates the behavior of a linear DAC. The DAC input (horizontal axis) is linearly mapped to the board output (vertical axis). Note that the DAC value represents the signal on the output connector of the 'board' (e.g. lens current or vacuum level), rather than the output of the DAC itself (typically a voltage level).

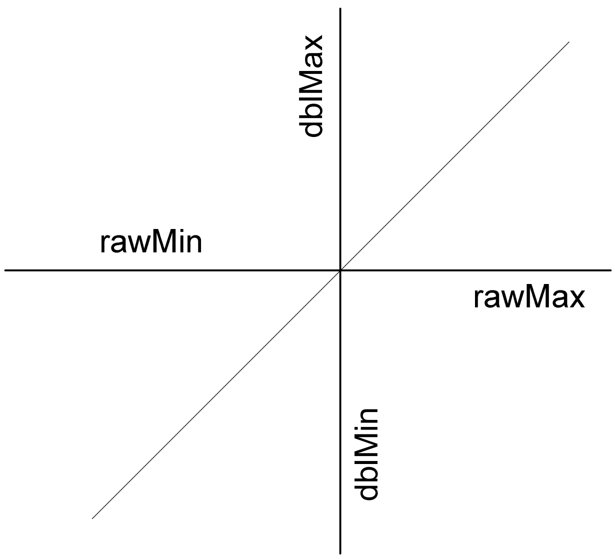


Figure 3 - Linear DAC

Note that the value used to set the DAC from the client may differ from the value that is actually set on the server. This is because the DAC has a resolution of at 32 bits at most, so the value to set is rounded to the nearest value corresponding to the DAC resolution. Therefore, use `Get()` to get the value that is actually set after calling `Set()` on the primitive. See Appendix B for a more detailed description.

3.14.2 OD structure

Object index	
ODDI, SubIdx[2]	Board Input [(r/w), 32b]
ODDI, SubIdx[3]	Unit[[(c), 8b]
ODDI, SubIdx[4]	Resolution[(c), 8b] *(note 2)*
ODDI, SubIdx[5]	dblMin[(r), 64b double]
ODDI, SubIdx[6]	dblMax[(r), 64b double]
ODDI, SubIdx[7]	rawMin[(r), 32b]
ODDI, SubIdx[8]	rawMax[(r), 32b]

Note 1: ODDI = OD Primitive Index; Primitive type = `eDAC_LIN` (see 2.2)
Note 2: Resolution equals the number of significant bits on the board input, i.e. for a 16-bit DAC the value of Resolution equals 16.

3.14.3 C++ interface

```
class IHepDacLinear: public IHepODPrimitive
{
public:
   virtual E_HEPLIB_Result Set(const double& value) = 0;                     // sets value
   virtual E_HEPLIB_Result Get(double& value) = 0;                           // gets value
   virtual E_HEPLIB_Result PrepareSet(IHepTransaction& transaction, const double& value) = 0;
   virtual E_HEPLIB_Result ParseSet(TTransactionResponse& response) = 0;
   virtual E_HEPLIB_Result PrepareGet(IHepTransaction& transaction) = 0;
   virtual E_HEPLIB_Result ParseGet(TTransactionResponse& response, double& retValue) = 0;

   virtual E_HEPLIB_Result Increment(double& value, int incr_count = 1) = 0;  // increments value
by incr_count least significant bits (LSB) of the DAC
   virtual E_HEPLIB_Result Decrement(double& value, int decr_count = 1) = 0;  // decrements value
by decr_count least significant bits (LSB) of the DAC
   virtual eUnit_t Units() const = 0;                                  // units definition
   virtual double Resolution() const = 0;                              // resolution in units per bit
   virtual double Min() const = 0;                                     // defined minimal value
   virtual double Max() const = 0;                                     // defined maximal value

   boost::function<void(double value)> onChange;  // update in case of other client change
};
```

3.15 eADC_LIN

3.15.1 Behavior

The board input is linearly mapped to the ADC value.

3.15.2 OD Structure

Object index	**Data**
ODDI, SubIdx[2]	Board Input [(r), 64b integer]
ODDI, SubIdx[3]	Unit[(c), 8b]
ODDI, SubIdx[4]	Resolution[(c), 8b] *(note 2)*
ODDI, SubIdx[5]	dblMin[(r), 64b double]
ODDI, SubIdx[6]	dblMax[(r), 64b double]
ODDI, SubIdx[7]	rawMin[(r), 64b integer]
ODDI, SubIdx[8]	rawMax[(r), 64b integer]

Note 1: ODDI = OD Primitive Index; Primitive Type = `eADC_LIN` (see 2.2)
Note 2: Resolution equals the number of significant bits on the board input, i.e. for a 16-bit ADC the value of Resolution equals 16.

3.15.3 C++ interface

```
class IHepAdcLinear: public IHepODPrimitive
{
```

```
public:
    virtual E_HEPLIB_Result Get(double& retValue) = 0;      // gets value
    virtual E_HEPLIB_Result PrepareGet(IHepTransaction& transaction) = 0;
    virtual E_HEPLIB_Result ParseGet(TTransactionResponse& response, double& retValue) = 0;

    virtual eUnit_t Units() const = 0;                      // units definition
    virtual double Resolution() const = 0;                  // bit resolution
    virtual double Min() const = 0;                         // defined minimal value
    virtual double Max() const = 0;                         // defined maximal value
};
```

3.16 eTripMonitor

3.16.1 Behavior

The trip monitor raises an event when a pre-defined ADC primitive trips certain levels. Two levels model hysteresis to limit trip events due to noise.

The server generates an event in these cases:

- When the initial ADC value is below the lower trip level, an “above upper” event is raised when both the lower and the upper level are tripped (in that order)
- When the initial ADC value is above the upper trip level, an “below lower” event is raised when both the upper and lower level are tripped (in that order)
- When the initial ADC value is in between the upper and lower level, an event is raised when either the upper or the lower level is tripped.

When a trip level changes, the trip monitor will behave as if it were disabled and enabled again.

Multiple trip monitors may be bound to the same ADC.

Setting of trip levels fails in case:

1. Lower trip level > Upper trip level
2. Upper trip level and lower trip level are not in the range applicable to the associated ADC

, and this failure will be indicated by the return value of the `SetLevels()` method.

When trip level values of a trip monitor are written, these values may or may not be stored persistently. Whether or not storage is persistent, is application dependent and must be specified in the interface (object dictionary) of the application.

Whether a trip monitor is enabled by default is application dependent.

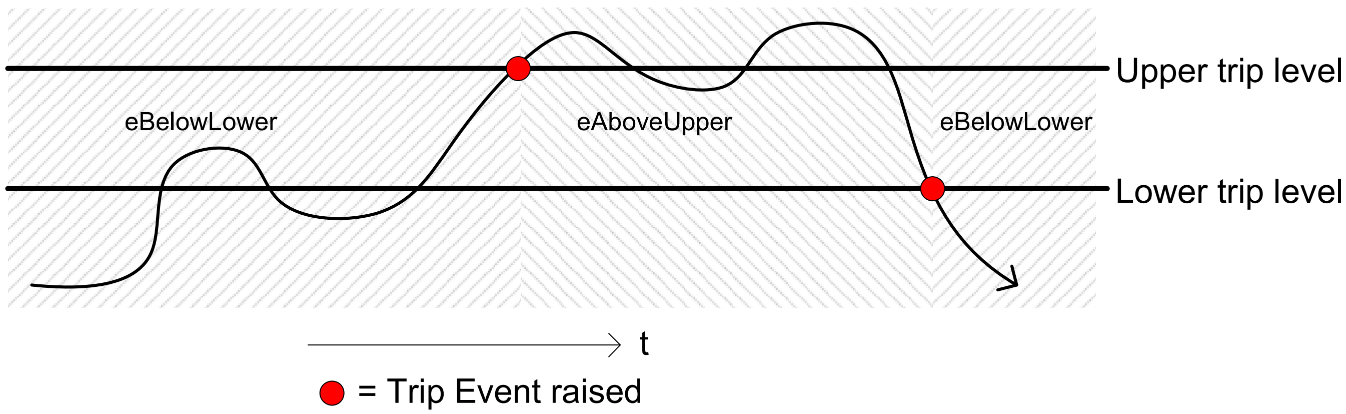


Figure 4 - Trip Monitor

3.16.2 OD Structure

Object index	Data
ODDI, SubIdx[2]	LowerTripLevel [(r/w), 64b integer]
ODDI, SubIdx[3]	UpperTripLevel [(r/w), 64b integer]

ODDI, SubIdx[4]	Enabled [(r/w), 8b, bool]
ODDI, SubIdx[5]	ADC index [(c), 16b]
ODDI, SubIdx[6]	ADC tripped[(r), 8b]

Note: ODDI = OD Primitive Index, Primitive type = `eTripMonitor` (see 2.2)

The element at sub-index 6 is used for event notification only. The value that is written to this element by the server is of type `eTripInfo` (see below). Reading of this value by the client is discouraged.

3.16.3 C++ interface

```
class IHepTripMonitor: public virtual IHepODPrimitive
{
public:

    enum eTripInfo
    {
        eBelowLower = 0,
        eAboveUpper = 1
    };

   virtual E_HEPLIB_Result GetLevels(double &lowerLevel, double& upperLevel) = 0;
   virtual E_HEPLIB_Result SetLevels(const double& lowerLevel, const double& upperLevel) = 0;
   virtual E_HEPLIB_Result Enable() = 0;
   virtual E_HEPLIB_Result Disable() = 0;
   virtual E_HEPLIB_Result GetEnabled(bool& bEnabled) = 0;
   virtual E_HEPLIB_Result GetADCName(wstring& adcName) = 0;

   boost::function<void(const eTripInfo& tripInfo)> onTrip;
   boost::function<void(const bool& bEnabled)> onEnabledChange;
   boost::function<void(const double& lowerLevel)> onLowerLevelChange;
   boost::function<void(const double& upperLevel)> onUpperLevelChange;
};
```

3.17 eDeltaMonitor

3.17.1 Behavior

The delta monitor raises an event when a pre-defined ADC primitive (hard-coded in server) trips. A delta is defined to model hysteresis to limit the amount of delta events due to noise: the server only generates an event when the value changes more than a defined delta compared to the last change event that was raised by the delta monitor. The unit of the delta monitor is equal to the unit of the related ADC.

The primitive supports absolute or relative deltas:

- The choice between absolute or relative is fixed by design and cannot be changed once defined.
- Absolute is suitable for monitoring linear signals (voltages, amps, temperatures). It is important to choose a delta that is significantly larger than the noise.
- Relative deltas are suitable for monitoring of non-linear (exponential/logarithm) signals.
- In case of a relative delta, the delta value is specified in terms of a percentage of the ADC value.

When values of a delta monitor are written, these values may or may not be stored persistently. Whether or not storage is persistent, is application dependent and must be specified in the interface (object dictionary) of the application.

The delta monitor will set the reference value to the current value of the ADC when the specified delta value changes. The delta monitor will set the reference value to the current value of the ADC when it trips.

Whether a delta monitor is enabled by default is application dependent.

3.17.2 OD structure

Object index	**Data**
ODDI, SubIdx[2]	Delta [(r/w), 64b integer or double (Note 2)]
ODDI, SubIdx[3]	Enabled [(r/w), 8b]

ODDI, SubIdx[4]	IsAbsolute[(c), 8b]
ODDI, SubIdx[5]	ADC index [(c), 16b]
ODDI, SubIdx[6]	ADC changed[(r), 64b integer]

Note: ODDI = OD Primitive Index, Primitive type = `eDeltaMonitor` (see 2.2)
Note 2: In case the delta monitor is absolute, the type of this element is 64-bit integer. In case the delta monitor is relative, the type of this element is 64-bit double, where a value of 1.0f is equivalent to 100%.

The 64-bit element at sub-index 6 is used for event notification only.

3.17.3 C++ interface

```
class IHepDeltaMonitor: public IHepODPrimitive
{
 public:
   virtual E_HEPLIB_Result GetDelta(double &delta) = 0;
   virtual E_HEPLIB_Result SetDelta(const double& delta) = 0;
   virtual E_HEPLIB_Result Enable() = 0;
   virtual E_HEPLIB_Result Disable() = 0 ;
   virtual E_HEPLIB_Result GetEnabled(bool& bEnabled) = 0;
   virtual E_HEPLIB_Result GetIsAbsolute(bool& bIsAbsolute) = 0;
   virtual E_HEPLIB_Result GetReferenceValue(double& refValue) = 0;
   virtual E_HEPLIB_Result GetADCName (string& adcName) = 0;
   // Hook to observe changes, delta is not propagated but the concrete value
   boost::function<void(const double& value)> onChange;
   boost::function<void(const bool& bEnabled) onEnabledChange;
   boost::function<void(const double& value)> onDeltaChange;
};
```

3.18 eGroupSwitch

3.18.1 Behavior

The C++ call for setting a value Switch will return when the value is either accepted or refused (`E_HEPLIB_Result`). The group switch represents multiple independent on/off switches (like a group of check boxes in a UI). The mask property indicates which of the bits are used (used=1, don't care=0)

3.18.2 OD structure

Object index	Data
ODDI, SubIdx[2]	Switch state [(r/w), 32b]
ODDI, SubIdx[3]	Mask [(r), 32b]

Note: ODDI = OD Primitive Index; Primitive Type = eGroupSwitch (see 2.2).

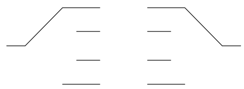
3.18.3 C++ interface

```

class IHepGroupSwitch: public IHepODPrimitive
{
public:
    // Optional Initialize method. Default everything is read from hardware!
    virtual E_HEPLIB_Result Initialize(const UINT32 BitMask) = 0;

    virtual E_HEPLIB_Result GetMask (UINT32& value); // get the bit mask (1 = in use)
    virtual E_HEPLIB_Result GetALL (UINT32& retValue);    // get total switch value
    virtual E_HEPLIB_Result SetALL (const UINT32& value); // set total switch value
    virtual E_HEPLIB_Result GetBit (const int bitNR, bool& retValue);    // true if bit is set
    virtual E_HEPLIB_Result SetBit (const int bitNR, bool Value); // set one bit
};
```

3.19 eNumber Switch

3.19.1 Behavior

The number switch represents a single switch with multiple positions. The switch value is zero-based. Max Number represents the maximum number of possible switch states. Position values must be continuous, including the maximum number.



3.19.2 OD structure

Object index	Data
ODDI, SubIdx[2]	Switch value [(r/w), 16b]
ODDI, SubIdx[3]	Max Number [(r), 16b]

Note: ODDI = OD Primitive Index; Primitive Type = `eNumberSwitch` (see 2.2)

3.19.3 C++ interface

```
class IHepNumberSwitch: public IHepODPrimitive
{
public:
    virtual E_HEPLIB_Result Get(UINT16& retValue);    // get switch value
    virtual E_HEPLIB_Result Set(const UINT16& value); // set switch value
   // Hook to observe changes
   boost::function<void(const UINT16& value)> onChange;
};
```

3.20 eConfiguration

3.20.1 Behavior

Data that is written to this primitive may not induce state changes, and it may not induce execution of commands. Typically, it is used for representing values of options, parameters, or settings. Since it is a relatively weakly typed primitive, its use is discouraged.

3.20.2 OD structure

Object index	Data
ODDI	Parameter[(r/w), 32b] *(note 2)*

Note 1: ODDI = OD Primitive Index; Primitive Type = `eConfiguration` (see 2.2)
Note 2: In ODs applying this type, instances may be read-only. Writing a read-only register will raise an error.

3.20.3 C++ interface

```
class IHEPConfiguration: public IHepODPrimitive
{
public:
    virtual E_HEPLIB_Result Read(UINT32& retValue) = 0;
    virtual E_HEPLIB_Result Write(const UINT32& value) = 0; // return error if read-only

    boost::function<void(const UINT32& value)> onChange; // Hook to observe changes
};
```

3.21 eFloat64

3.21.1 Behavior

Primitives of type *eFloat64* are read-only by default; writing is allowed, but not specified on the OD. Write() will return an error if the register is read-only. This register type is intended to access limits, thresholds, counters, timeouts, etc, that don't induce state changes or induce execution of commands.

Floats follow the IEEE753 standard for floating-point arithmetic.

3.21.2 OD structure

Object index	Data
ODDI	Parameter[(r[/w]), 64b] *(note 2)*

Note 1: ODDI = OD Primitive Index; Primitive Type = *eFloat64* (see 2.2)

Note 2: Writeable is optional (this is not visible on the OD, writing a read-only register will raise an error)

3.21.3 C++ interface

```
class IHepRegister64: public IHepODPrimitive
{
public:
    virtual E_HEPLIB_Result Read(double& retValue) = 0;
    virtual E_HEPLIB_Result Write(double value) = 0; // return error if read-only
};
```

3.22 eApplication

3.22.1 Behavior

Primitives of type `eApplication` contain information identifying the application(s) present on the server. Instances of this type are only present in the application-specific object region of the so-called generic application. This generic application must always be present on the server. It exposes all available applications to the client. Moreover, it allows for driving the lifecycle state of these applications. Objects of type `eApplication` are located in the 0x2000 range of the generic application (see [R_3]).

`eApplication` provides information about status and errors related to the application lifecycle. As previously stated, this primitive exposes a command interface to drive the application state machine. This command interface behaves as follows:

1. The command register will contain the last command written to it, until it has been executed. At completion the register will be set to the reserved command `NoCommand`, which has value 0xFE.
2. As long as execution of a command is in progress, new commands will be refused. In case of a command being refused, the `SetCommand()` function will return an error indicating this.

Application-specific errors are communicated via one or more primitives of type `eError` in the application OD.

The object provides information on the protocols it supports. It does so by providing a comma-separated, null-terminated string of supported protocols. This is the list of protocols that this version of the specification supports:

- FPIP : FEI Powerlink Inspired Protocol [R_1]
- FEICANoE : FEICAN over Ethernet (no specification known to the authors…)

3.22.2 Application Lifecycle State-machine

Figure 5 depicts the lifecycle state-machine of applications running on the server.

Along the edges of the transitions, the commands to drive the application lifecycle state-machine are depicted. Note that the `INITIALIZING` and `SHUTDOWN` states are unstable. This means that transitions to `ACTIVE`, `SHUTDOWN` or `CRITICALERROR` state, respectively to the final state occur without a trigger from the client. All other states are stable, i.e. explicit client triggers drive state transitions.

Typically, applications need to be explicitly created, started and shut down. The exception to this rule is the generic application. This application must start automatically, since without it, there is no way for clients to know what applications the server supports. The generic application can be shut down, but only if there are no applications in `CREATED`, `INITIALIZING`, `CRITICALERROR`, `ACTIVE` or `SHUTDOWN` state[4]. After successful shutdown of the generic application, the server will reset itself.

[4] Note that this implies that applications must 'eventually' transit to final state when they are commanded to shut down. If this requirement is not fulfilled, there is no way to drive the generic application to its final state and, consequently, there is no way to reset the server.

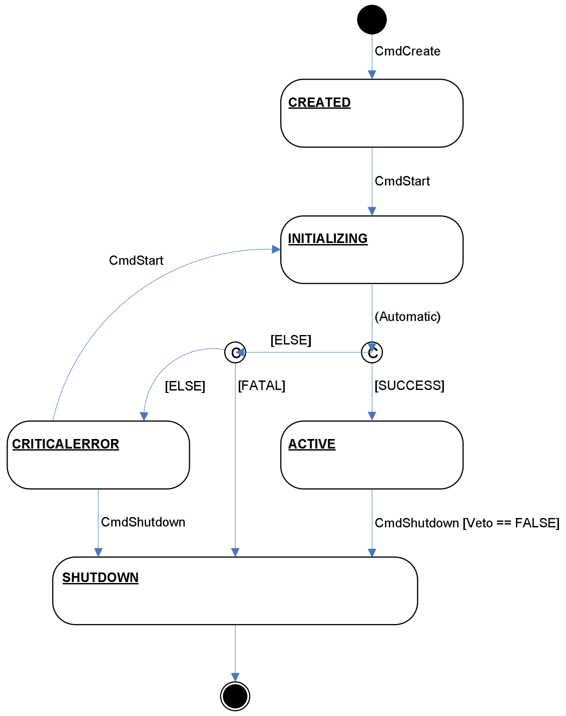


Figure 5 - Application Lifecycle State-machine

3.22.3 Application Lifecycle Errors

In commanding an application to make a transition to another state, errors may occur. The following type describes these errors:

```
typedef enum
{
  eLCOK = 0,              // No lifecycle error
  eLCCreateFailed,        // Creation of application failed
  eLCActivateFailed,      // Activation of application failed
  eLCShutdownFailed,      // Shutdown of application failed
  eLCCommandNotAllowed, // In the current state, the command issued is not allowed
  eLCUnknownCommand       // Unknown command
} eLifecycleError_t, *peLifecycleError_t;
```

The lifecycle error object propagates these errors to the client.

3.22.4 OD Structure

Object index	**SubIdx[2]**	**SubIdx[3]**	**SubIdx [4]**	**SubIdx [5]**	**SubIdx[6]**	**SubIdx[7]**	**SubIdx[8]**	**SubIdx[9]**
ODDI	Application ID [(c), 8b]	Supported protocols *VisibleString*[(c)]	Lifecycle Command [(r/w), 8b]	Lifecycle Status [(r), 8b]	Lifecycle Error [(r), 8b]	Version Major ‘X’ [(c), 8b]	Version Minor ‘Y’ [(c), 8b]	Version Build ‘Z’ [(c), 8b]

Note 1: ODDI = OD Primitive Index; Primitive Type = `eApplication` (see 2.2)
Note 2: The sub-index named ‘Supported protocols’ contains a null-terminated, comma-separated string.

3.22.5 C++ Interface

```
typedef enum
{
  eLifecycleStateNone = 0,
  eLifecycleStateCreated,
  eLifecycleStateInitializing,
  eLifecycleStateActive,
  eLifecycleStateCriticalError,
  eLifecycleStateShutdown
} eLifecycleState_t, *peLifecycleState_t;
```

```
typedef enum
{
  eLifecycleCmdCreate = 0,
  eLifeCycleCmdStart,
  eLifeCycleCmdShutdown
} eLifecycleCommand_t, *peLifecycleCommand_t;

class IHepApplication: public IHepODPrimitive
{
public:
   // Optional Initialize method. Default everything is read from hardware!
   virtual E_HEPLIB_Result GetID(UINT8& ID) = 0;
   virtual E_HEPLIB_Result SetCommand(const eLifecycleCommand_t& Command) =0; // set command
   virtual E_HEPLIB_Result GetCommand(eLifecycleCommand_t & Command) =0;      // get command
   virtual E_HEPLIB_Result GetState(eLifecycleState_t& State) =0;        // get application state
   virtual E_HEPLIB_Result GetError(UINT8& Error) =0;                    // get application error
   virtual E_HEPLIB_Result GetSupportedProtocols(vector<string>& Protocols) = 0; // get protocols
   virtual E_HEPLIB_Result GetVersion(UINT8& X, UINT8& Y, UINT8& Z) = 0;   // get application
version

   boost::function<void(const UINT8& appID, const UINT8& state)> onStateChange;
   boost::function<void(const UINT8& appID, const UINT8& error)> onError;
};
```

<This page is intentionally left blank>

4 Standardized OD Entries

4.1 Base OD Entries for all Firmware Applications (0x1000 Range)

The Base OD entries in Table 4 are mandatory for all OD-based applications, including the Generic Application (see 4.2). These mandatory OD entries start at object index 0x1000. The names of the objects in this object dictionary must comply with the names as specified in Table 4.

The command object in the base OD of other applications than the Generic Application may not be used for issuing of lifecycle commands. Lifecycle commands may only be issued via objects of type `eApplication` in the 0x2000 range of the Generic Application (see Section 4.2).

BaseODVersion refers to the version of this standard. This allows for future extension of this section. Example: an OD that complies with version 1.9 of this specification will have value 1 at sub-index 2, value 9 at sub-index 3, and value 0 at sub-index 4.

SubIdx[0] Primitive type (see 2.2)	SubIdx[1]	SubIdx[2]
`eVersion3_8`	BaseODVersion	*(note 1), (note 2)*
`eVersion3_8`	AppVersion	*(note 1)*
`eError`	AppError	*(note 1)*
`eState`	AppState	*(note 1)*
`eCommand`	AppCommand	*(note 1)*
`eString`	AppName	*(note 1), (note 3)*
`eNullPrimitive`	MandatoryRangeEnd	<n/a>

Table 4 - Base OD Entries for Firmware Applications

Note 1: Remaining sub-indexes are omitted here; refer to the corresponding primitive type.
Note 2: Refers to the version number of this specification to which the object dictionary complies
Note 3: The mandatory application name for the generic application is “Generic Application”

4.2 The OD Entries of the Generic Application

The OD entries for the generic application in Table 5 start at object index 0x2000. The objects are all typed as described in Section 3.

The application version number and application name in each object of type `eApplication` in the generic application are identical to the application version number and application name in the 0x1000 range of the corresponding application.

Multiple instances of a module may be present at system level that all provide the same hardware IDs. To be able to distinct these instances, the generic application exposes an object named InstanceID of type `eConfiguration`. It is up to the designer/implementer of the module to ensure that every instance of a module exposes a unique instance ID. This is a mandatory item. Value 0xFFFFFFFF is reserved (no instance ID). It is up to the designer of the system running the firmware how the instance ID is obtained. It may be obtained from hardware that is part of the system, such as a connector equipped with the ID, but it can also be pushed from the client software accessing the firmware (and consequently stored in some non-volatile memory).

The Generic Application itself is not represented by an object of type `eApplication` in its 0x2000 range. The lifecycle of the Generic Application is driven and observed by objects in its base OD.

SubIdx[0] Primitive type (see 2.2)	SubIdx[1] Primitive name (see 2.2)	SubIdx[2]
`eVersion3_8`	FirmwareVersion	*(note1), (note2)*
`eConfiguration`	FWBuildNr	*(note2)*
`eString`	FWLogicalName	*(note2), (note 3)*
`eData`	HWIDs	*(note 2), (note4)*
`eApplication`	*Application Name 1*	*(note 2)*
`eApplication`	*Application Name 2*	*(note 2)*
`eApplication`	…	*(note 2)*

eApplication	*Application Name n*	*(note 2)*
eConfiguration	InstanceID	*(note 5)*
eNullPrimitive	MandatoryRangeEnd	<n/a>

Table 5 - Base OD Entries for the Generic Application

Note 1: The firmware version is extracted from the metadata of the firmware binary as defined in [R_5].
Note 2: Remaining sub-indexes are omitted here; refer to the corresponding primitive type.
Note 3: The firmware logical name is extracted from the metadata of the firmware binary as defined in [R_5].
Note 4: Data at sub-index 2 describes the hardware that is present on the subsystem.
Note 5: This is a mandatory item. Value 0xFFFFFFFF is reserved (no instance ID).

4.2.1 Hardware IDs

Hardware IDs of hardware that is present on the server are exposed via the OD of the generic application. The available information about hardware present may grow over time. Hence, one cannot assume this information is complete, <u>unless</u> this is enforced by some (application-specific) protocol on a higher level.

```
// ID (Identification) Record
//
// |-------------------------------------------------------------------------------|
// | 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
// |---------------------------------------|---------------------------------------|
// |          Interface Bus Address        |        Interface Bus Type ID          |
// |---------------------------------------|---------------------------------------|
// |           >>Reserved<< (0)            |   Next "Interface Bus Type" Offset    |--
// |---------------------------------------|---------------------------------------|  |
// | >>Reserved<< (0)  |             Board Type ID                                 |  |
// |-------------------|-----------------------------------------------------------|  |
// | >>Reserved<< (0)  |   FW Version ID   | >>Reserved<< (0)  |  HW Version ID    |  |
// |-------------------|-------------------|-------------------|-------------------|  |
// |                         Connected Equipment ID (optional)                     |  |
// |-------------------------------------------------------------------------------|  |
// |                                     . . .                                     |  |
// |-------------------------------------------------------------------------------|  |
// |                         Connected Equipment ID (optional)                     |  |
// |-------------------------------------------------------------------------------|  |
//                                                                                 <--
// Interface Bus Type ID = {SCU, SPI-LVDS, STA, RS232#1, RS232#2, RS485#1, RS485#2 etc}
// Interface Bus Address = 8-bit backplane ID for the SCU
//                       = 4-bit Slot Address for the STA bus
//                       = 5-bit Slot Address + 3-bit Board Address for the SPI-LVDS bus
//                            Board Address 0   => Application Board
//                            Board Address 1-6 => Satellite Board
//                       = 0 (not applicable) for the RS232 peer-to-peer connection
//                       = 8-bit network address for RS485 network
// Board Type ID         = Unique ID of a board on "Interface Bus Address"
// FW Version ID         = FW version ID of a board on "Interface Bus Address"
// HW Version ID         = HW version ID of a board on "Interface Bus Address"
// Connected Equipment ID= Equipment ID of an equipment connected to Application or
//                         Satellite board on "Interface Bus Address".
//                         The equipment has no HW communication interface (coil,
//                         detector, probe etc) but the electrical connection to the
//                         equipment must assure the unique identification of any
//                         equipment for given application (using pin-coding,
//                         resistor-coding, 1-Wire etc).
//                         The application-specific HW and FW shall be capable of
//                         identifying of all possible equipments.
```

Comments:

1) The Identification Records are of variable length, the next record is determined using an 8-bit offset field
2) The ending (dummy) record is defined as “Interface Bus Type ID”=0xFF and “Interface Bus Address”=0xFF
3) Currently, the Board Type ID is 12-bit number (for SPI/LVDS) but the appropriate record item reserves 16-bit for future extension

Appendix A Example: ADC

This appendix illustrates application of the ADC HEP/ODD primitive. The usage of this primitive for linear ADCs, where the ADC raw input linearly maps to the (digital) output is considered obvious. Therefore, this appendix discusses two applications that are less trivial:

1. Pressure gauge: the relation between measuring signal and pressure is logarithmic
2. Electric Current Measurement: …

Example 1. Pressure Gauge

Figure 6 depicts the conversion curves for some pressure gauge. The relation between measuring signal and pressure is given by the following conversion formulae:

$$p = 10^{1.667U-d} \Leftrightarrow U = c + 0.6log_{10}p$$

, where

p pressure
U measuring signal
c, d constant (pressure unit dependent)

Even though the relationship between measuring signal and pressure obviously is non-linear, this gauge can be modeled as a linear ADC on the OD interface of firmware that controls and abstracts this gauge.

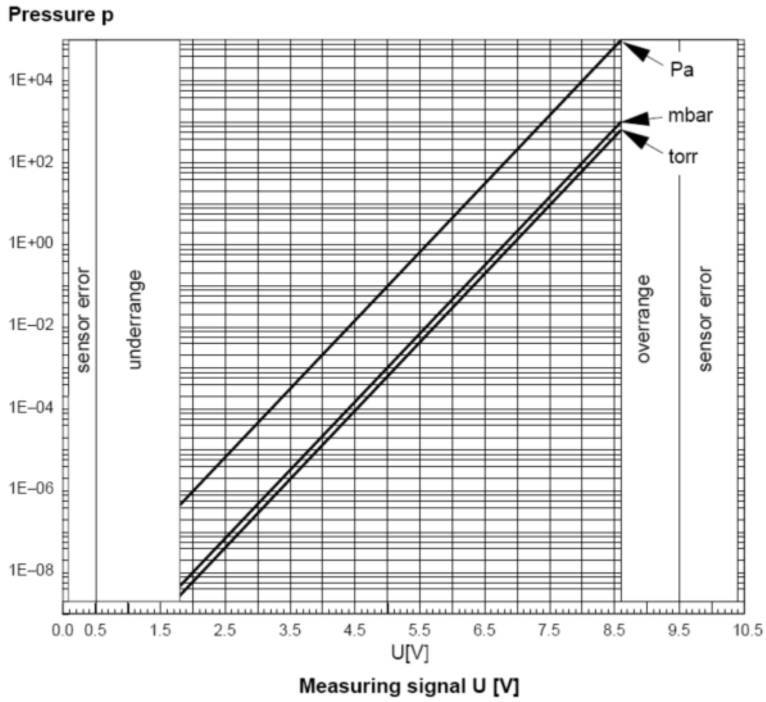


Figure 6 - Pressure Gauge Conversion Curves

The pressure gauge applied in this example supports an input range of 5.0×10^{-7} to 1.0×10^{5} [Pa]. The corresponding measuring signal values are 1.82 and 8.6 [V], respectively, as can be seen in Figure 6.

Two examples of modeling the gauge as an ADC in the object dictionary will be considered:

1. Use the desired step size to calculate the ADC object resolution
2. Use the gauge resolution to calculate the ADC object resolution

Ad 1. Desired Step Size Resolution

In this case, suppose application requirements dictate that the desired step size is 1×10^{-4} [Pa], which equals 1×10^{9} steps over the range. This can be encoded in 32 bits, so we set the resolution of the ADC to 32 bits.

The corresponding object in the object dictionary can be modeled as follows:

Object index	Data Type	Data Value
ODDI, SubIdx[2]	Board Input [(r), 64b]	<actual value, see below>
ODDI, SubIdx[3]	Unit[(c), 8b]	eUnit_PRESSURE
ODDI, SubIdx[4]	Resolution[(c), 8b]	32
ODDI, SubIdx[5]	dblMin[(r), 64b]	1.0E-4
ODDI, SubIdx[6]	dblMax[(r), 64b]	1.0E5
ODDI, SubIdx[7]	rawMin[(r), 64b]	1
ODDI, SubIdx[8]	rawMax[(r), 64b]	1E9

Table 6 – OD Object Representing Pressure Gauge

Ad 2. Gauge Resolution

The input range of the gauge is of order 1×10^{-7} [Pa], which is equivalent to 1×10^{12} steps over the range. This can be encoded in 40 bits, so the resolution of the ADC is set to 40 bits.

The corresponding object in the object dictionary can be modeled as follows:

Object index	Data Type	Data Value
ODDI, SubIdx[2]	Board Input [(r), 64b]	<actual value, see below>
ODDI, SubIdx[3]	Unit[(c), 8b]	eUnit_PRESSURE
ODDI, SubIdx[4]	Resolution[(c), 8b]	40
ODDI, SubIdx[5]	dblMin[(r), 64b]	5.0E-7
ODDI, SubIdx[6]	dblMax[(r), 64b]	1.0E5
ODDI, SubIdx[7]	rawMin[(r), 64b]	5
ODDI, SubIdx[8]	rawMax[(r), 64b]	1E12

Table 7 – OD Object Representing Pressure Gauge

Exception Handling

Note that for notification of exceptional behavior (e.g. sensor errors, under-range, or over-range), a separate object must be present in the object dictionary. Several types of objects can be used for such event notification, e.g. `eError` or `eState`.

Example 2. Current Measurement

Current measurement is implemented as follows:

- 4 current measurement channels (A, B, C, and E/F) are present, each having two ranges
- Channel A has an extra current measurement with a special range

Channel	Low Range	High Range	Special Range
A	[-5.12[nA] … +5.12[nA]]	[-2.56[uA] … +2.56[uA]]	[-5uA … +5uA]
B, C, E/F	[-5.12[nA] … +5.12[nA]]	[-512nA] … +512[nA]]	N/A

Table 8 - Current Measurement Inputs and Corresponding Ranges

Conversion formulae:

$I_{measured} = \frac{2048 \times I_{ADCValue}}{100 \times R_{range} \times 2^{25}}\ [A]$, where

R_{range} = 1E7 in case range is [-512nA] … +512[nA]] or [-5uA … +5uA]
R_{range} = 1E9 in case range is [-5.12[nA] … +5.12[nA]]
R_{range} = 2E6 in case range is [-2.56[uA] … +2.56[uA]]

The requirement is to represent these channels and ranges with a single ADC primitive. The required bit resolution is 1[fA].

In this case, the physical ADC (ADS1256) has a resolution of 24 bits and is signed.

The corresponding object in the object dictionary can be modeled as follows:

Object index	Data Type	Data Value
ODDI, SubIdx[2]	Board Input [(r), 64b]	<actual value, see below>
ODDI, SubIdx[3]	Unit[(c), 8b]	eUnit_ELECTRICCURRENT
ODDI, SubIdx[4]	Resolution[(c), 8b]	36
ODDI, SubIdx[5]	dblMin[(r), 64b]	-5E-6
ODDI, SubIdx[6]	dblMax[(r), 64b]	5E-6
ODDI, SubIdx[7]	rawMin[(r), 64b]	0
ODDI, SubIdx[8]	rawMax[(r), 64b]	1E10

Table 9 – OD Object Representing Pressure Gauge

Notes

Both examples demonstrate the use of an ADC object for some abstraction, i.e. a gauge and a current measurement solution. In case some process or device is abstracted by an ADC object, freedom exists as to how determine values for elements. This is best demonstrated by the pressure gauge solution. Here, the first solution (Ad 1.) uses some requirement in abstracting the gauge. The second solution (Ad 2.) uses properties of the physical gauge to define the element values.

In all examples, the value for Resolution ‘only’ defines the number of bits required to represent the Board Input value. The value cannot be used to deduct/calculate the smallest step-size in terms of the physical unit.

<This page is intentionally left blank>

Appendix B Example: DAC

This appendix illustrates application of the DAC HEP/ODD primitive. It discusses an application for which it may not be obvious to apply a DAC, i.e. volume control of some audio device.

Example: Volume Control

The control has an output range of 0 – 100 [%]. Example object definitions of two different implementations are provided:

1. Control via a stepper motor
2. Control via a DAC

Ad 1. Stepper Motor Implementation

The stepper motor has 40.000 steps over the volume range.

The corresponding object in the object dictionary can be modeled as follows:

Object index	Data Type	Data Value
ODDI, SubIdx[2]	Board Input [(r), 64b]	<value>
ODDI, SubIdx[3]	Unit[(c), 8b]	eUnit_PERCENTAGE
ODDI, SubIdx[4]	Resolution[(c), 8b]	16
ODDI, SubIdx[5]	dblMin[(r), 64b]	0.0
ODDI, SubIdx[6]	dblMax[(r), 64b]	100.0
ODDI, SubIdx[7]	rawMin[(r), 64b]	0
ODDI, SubIdx[8]	rawMax[(r), 64b]	40000

Table 10 – OD Object Representing Stepper Motor Based Volume Control

Obviously, the (almost continuous) board input needs to be rounded to the nearest feasible output value, which, in this case, is limited by the 40.000 increments of the stepper motor. Therefore, after setting the DAC, one must read out the input value to obtain the value that was actually set.

Ad 2. DAC Implementation

The DAC is 8-bit, and it has an output range of 0 – 5 [V].

Object index	Data Type	Data Value
ODDI, SubIdx[2]	Board Input [(r), 64b]	<value>
ODDI, SubIdx[3]	Unit[(c), 8b]	eUnit_PERCENTAGE
ODDI, SubIdx[4]	Resolution[(c), 8b]	8
ODDI, SubIdx[5]	dblMin[(r), 64b]	0.0
ODDI, SubIdx[6]	dblMax[(r), 64b]	100.0
ODDI, SubIdx[7]	rawMin[(r), 64b]	0
ODDI, SubIdx[8]	rawMax[(r), 64b]	255

Table 11 – OD Object Representing DAC-Based Volume Control

In this case, the board input needs to be rounded to the nearest feasible output value as well. The feasible set of output values is limited by the 8-bit resolution of the DAC. Therefore, after setting the DAC, one must read out the input value to obtain the value that was actually set.

Notes

This example demonstrates the use of a DAC object for some abstraction, i.e. volume control. As is shown, depending on the implementation, different values for elements are chosen. Most of the times, the client using the object will only be interested in the board input, unit, dblMin, and dblMax values. Other elements can be seen as implementation-specific and will typically not be used that often.

<This page is intentionally left blank>