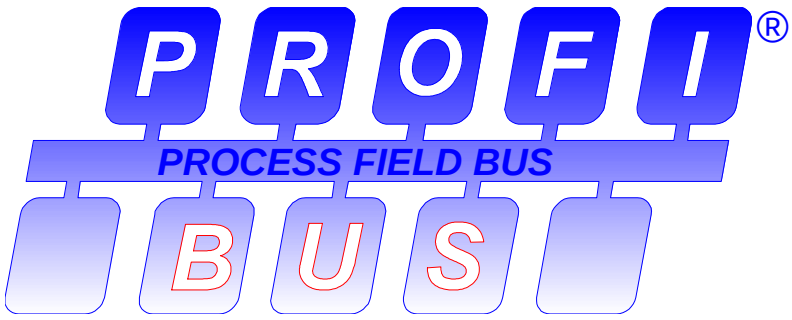
PROFIBUS



PROFIBUS Guideline

Specification for PROFIBUS Device Description and Device Integration

Volume 1: GSD V 3.1

Volume 2: EDDL V 1.1

Volume 3: FDT V 1.1

January 2001

PROFIBUS Guideline – Order No. 2.152

PROFIBUS Guideline, Order No. 2.152

Specification for PROFIBUS Device Description and Device Integration

Volume 1: GSD V 3.1

Volume 2: EDDL V 1.1

Volume 3: FDT V 1.1

January 2001

Prepared by the PROFIBUS Working Groups „GSD Specification“, „Device Description Language“ and „Engineering“ in the Technical Committee „System Integration“.

Publisher:
PROFIBUS Nutzerorganisation e.V.
Haid-und-Neu-Str. 7
D-76131 Karlsruhe

Phone: ++ 721 / 96 58 590
Fax: ++ 721 / 96 58 589
PROFIBUS_International@compuserve.com
www.profibus.com

Preface

Synopsis:

This paper comprises the specifications for GSD (Basic Profibus Device Description), EDDL (Electronic Device Description Language) and FDT (Field Device Tool Interface). They are artefacts of working groups within Technical Commitee 4 of the PROFIBUS Trade Organization.

Trademarks:

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

Abstract:

GSD, EDDL and FDT are representing the means to configure network devices and to parametrize and/or manipulate their operational modes. While GSD and EDD are based on human readable descriptions, FDT defines a set of interface to integrate device specific software components into engineering tools or other frameworks. GSD and EDDL are using device description languages and FDT defines a client/server relationship.

In order to meet the market requirements and the customer's needs this set of specifications is covering all the different aspects of complexity and usage, thus protecting the members' investments and providing scalable and compatible solutions.

Motivation

In process and manufacturing automation, a control system often comprises more than 10,000 binary and analog input/output signals. When a fieldbus is used, these signals are transmitted via the bus. To this end, the field devices are connected directly to the bus or measured via remote I/O. More than 100 different field device types from various device manufacturers are frequently in use.

The devices are configured and parameterized for each task. The device-specific properties and settings must be taken into consideration when configuring the fieldbus coupler and the bus communication, and the devices must be made known to the control system. Input and output signals provided by devices must be created and integrated into the function planning of the control system.

The large number of different device types and suppliers within a control system project makes the configuration task difficult and time-consuming today. Different tools must be mastered and data must be exchanged between these tools and hosting system environments. The electronic data exchange format is now standardized and the interfaces between those tools are defined.

Approach

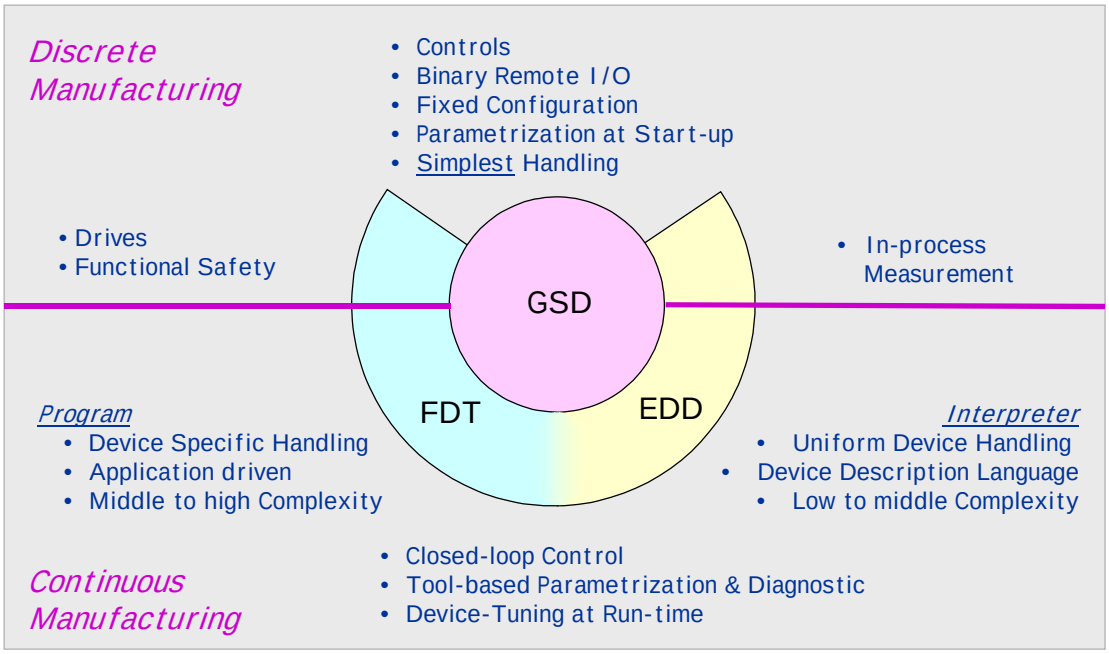


Fig. 1 GSD, EDD, FDT: A Scalable Solution for a Wide Set of Applications

GSD

PROFIBUS devices may have different behavior and performance characteristics. Features differ in regard to available functionality (i.e., number of I/O signals and diagnostic messages) or possible bus parameters such as baud rate and time monitoring. These parameters may vary individually for each device type and vendor and are usually documented in the technical manual. In order to achieve a simple Plug and Play configuration for PROFIBUS, electronic device data sheets (GSD files) are defined for the communication features of the devices. These GSD files allow easy configuration of PROFIBUS networks with devices from different manufacturers.
GSD is a human readable ASCII text file. Keywords are specified as mandatory or optional with the corresponding data type and their border values to support the configuration of PROFIBUS devices. Based on the defined file format it is possible to realize vendor independent configuration tools for PROFIBUS systems. The configuration tool uses GSD files for testing the data. These were entered regarding limits and validity related to the performance of the individual device. New developments of PROFIBUS products will extend the functional range.

The manufacturer of a device is responsible for the functionality and the quality of its GSD file. The device certification procedure is requesting either a standard GSD file based on a PROFIBUS profile or a device specific GSD file.

EDDL

Up to now most of the devices have been configured by its own configuration tool. As a consequence the customer had to deal with as many tools as he was using device types. The Electronic Device Description Language has been designed to implement a vendor independent data set called EDD describing device configuration, maintenance and functionality. The EDDL defines the syntax (form) and the semantics (meaning) of the data and the behavior of a PROFIBUS device or component and the structure of the corresponding user interface. In its most basic form, the EDD source is human readable text written by device developers. The device manufacturer is responsible for completeness and correctness of his EDD source.
The EDD source can be easily incorporated into configuration tools just by reading it into an EDD interpreter (EDDI).
Configuration tool developers no longer need to be responsible for validation testing of all devices supported by their products. Device Description technology is state of the art for describing devices not only in the PROFIBUS arena but also in the environment of other fieldbus systems. Device Description Languages guarantee a uniform handling of all devices independent of the supplier and the type of the device. This means a user handles a temperature transmitter from a supplier A and a remote I/O system from a supplier B in the same way.
The EDDL specification provides a detailed description of the Electronic Device Description Language required for the development of an Electronic Device Description source file. The architecture of the EDD application and its usage during design and operational phases of a device are defined.

FDT

With the integration of fieldbusses into control systems, there are some more tasks that have to be performed. This applies to fieldbusses in general. Up to now there was no unified way to integrate device specific tools into engineering environments, console applications and diagnostic software. Especially within extensive and heterogeneous control systems, the unambiguous definition of interfaces with ease of use is getting growing importance.
As simple as a new printer is added to a PC just by installing a driver, as simple should be the integration of a new device into an automation environment.

With the help of the FDT specification and its interface technology the user will be able to handle devices and their integration into engineering tools and other frameworks in a consistent manner. Due to the well-defined independance of system and device manufacturers the latter are enabled to support any innovative feature of their device without limitations.

This is done via a device-specific software component, called DTM (Device Type Manager). The device manufacturer is responsible for the functionality and the quality of a DTM. The DTM is integrated into engineering tools or other "frame applications" like stand-alone commissioning tools or web browsers that are providing the FDT interfaces. Even EDDI-Tools with the appropriate interfaces may be integrated this way. The approach to integration is in general open for all kind of fieldbusses (different protocols) and thus meets the requirements for integrating different kinds of devices into heterogeneous control systems.

An additional style guide is available for the development of DTMs in order to counteract the risk of proliferation of user interfaces.

Scalability via GSD, EDD and FDT

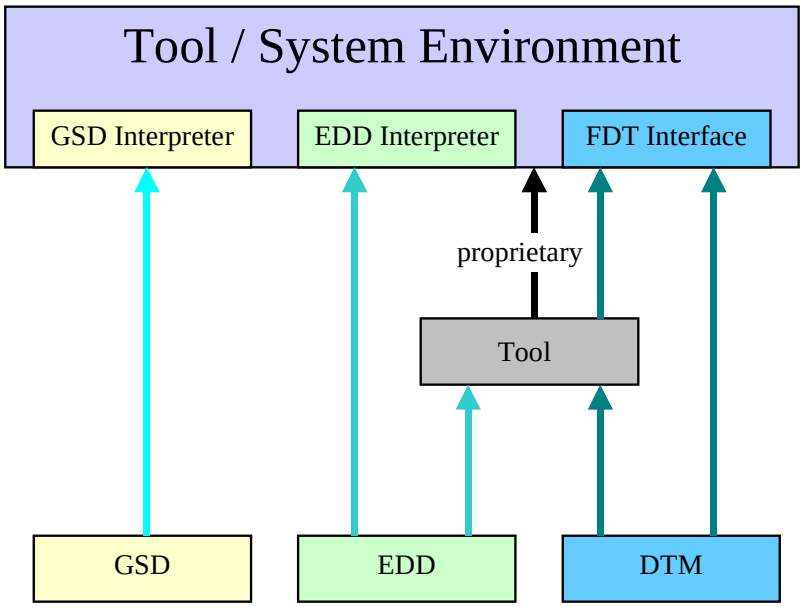


Fig. 2 Potential Integration Structures

Reflecting the current situation, there are a lot of different field devices ranging from simple I/O sensors to complex, modular Remote-I/Os or drives. According to this complexity, the devices can be divided into four categories:

A: Simple devices that communicate only cyclically, for example a light barrier

B: Adjustable devices with fixed hardware and software, for example a pressure transducer

C: Adjustable devices with modular hardware but fixed software blocks, for example a remote I/O or with fixed hardware but modular software blocks, e. g. a radar sensor

D: Adjustable devices with modular hardware and programmable software blocks, for example a complex servo-drive

GSD, EDD and FDT are supporting all ranges of device complexity and integration levels into system environments

1 Preface

Synopsis

This paper comprises the specifications for GSD (Basic Profibus Device Description), EDDL (Electronic Device Description Language) and FDT (Field Device Tool Interface). They are artefacts of working groups within Technical Commitee 4 of the PROFIBUS Trade Organization.

Trademarks

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

Abstract

GSD, EDDL and FDT are representing the means to configure network devices and to parametrize and/or manipulate their operational modes. While GSD and EDD are based on human readable descriptions, FDT defines a set of interface to integrate device specific software components into engineering tools or other frameworks. GSD and EDDL are using device description languages and FDT defines a client/server relationship.

In order to meet the market requirements and the customer's needs this set of specifications is covering all the different aspects of complexity and usage, thus protecting the members' investments and providing scalable and compatible solutions.

2 Introduction

2.1 Scope

The scope of this document is to provide the methodology for the electronic and computable description of device parameters for automation system components. For this description the so called Electronic Device Description (EDDL) is specified.
The Electronic Device Description is used for the configuration and the operational behaviour of a device. It may also be used generally for the description of product properties in other domains. The EDD methodology covers the following aspects:

- Description of the device parameters
- Support of parameter dependencies
- Logical grouping of the device parameters
- Selection and execution of supported device functions
- Description of the device parameter access method

Up to now most of the devices have been configured by its own configuration tool. As a consequence as many devices come up as many configuration tools occur. Each configuration related device change needs also a change in the configuration tool, which results in high software maintenance costs for the configuration tools on vendor side, and the user has to manage every new software version. In addition the configuration tools for the different devices very often come from different sources with different quality. This may result in stability problems of the configuration system. Software bugs in the large number of different configuration tool products may even impact the quality of the complete runtime system of an engineering console after any update of a software product and may potentially result in a system crash. These problems occurs independently which operating system has been installed.

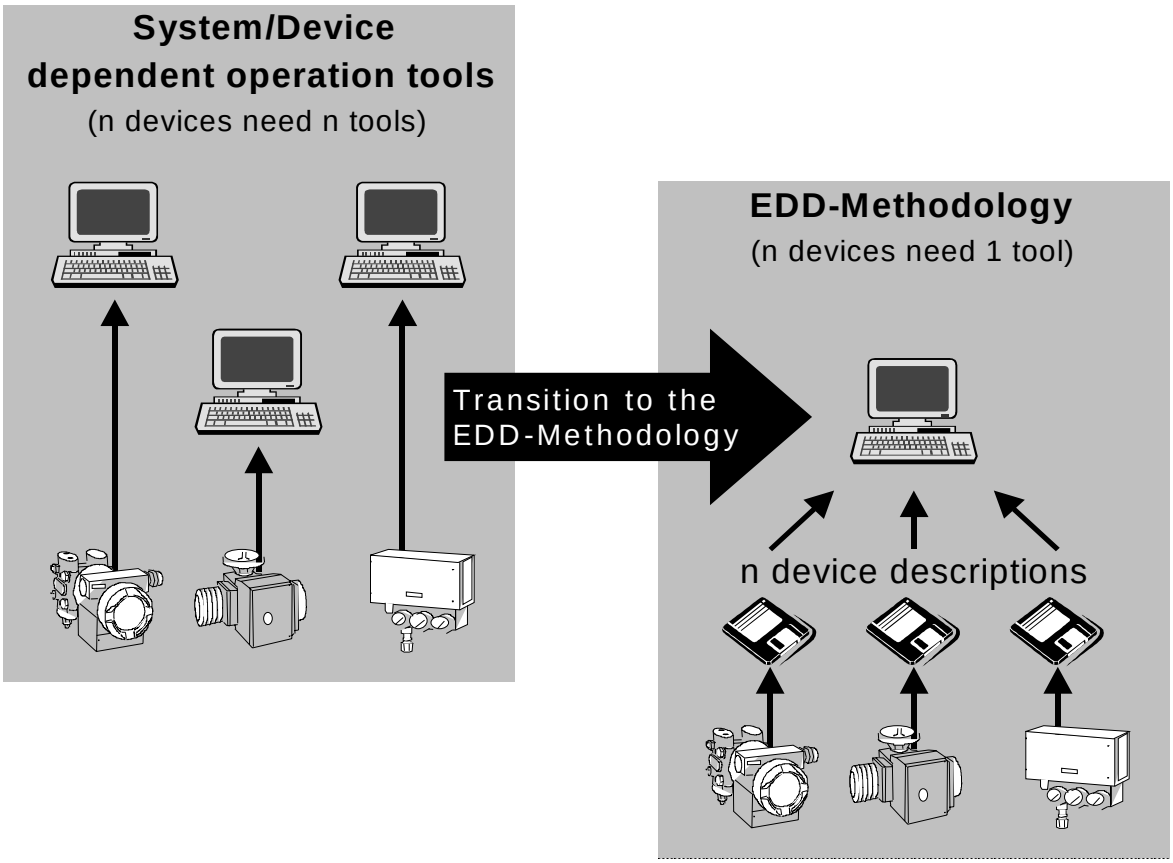


Figure 1: Transition to the EDD-Methodology reduces costs for development and support

An efficient way to avoid such problems is to reduce the number of configuration software packages. This can be achieved by moving device properties from the runtime code of a configuration tool into a data set called "Electronic Device Description". This device properties specified by the "Electronic Device Description Language" has to be delivered with each device and is interpreted by an EDD interpreter of a configuration tool. The EDDI generates the input/output screens by interpreting the EDD data set and allows setting single parameter values, starting sequences of parameters, settings and computing values. Figure 1 shows the transition from a software tool for each devices to their device descriptions handled by an EDDL-Tool.
The "C-based" EDDL is not a programming language. It describes product data in a declarative way. In the case of this document it describes the device properties related to the configuration process. This comprises the identification of the device, the setting of single parameters, sequences of parameters and computation of values.
In this context the term "configuration" comprises the parameter settings of a field device (scaling factors, upper and lower limits, etc.) and the determine of functionality supported by the field device (diagnosis, calibration, etc.).
A prerequisite for setting of parameters is the communication interface of the device. In this context the communication system is not subject of this document. It is assumed to be existent. The EDD is part of the device application and has nothing to do with the communication system. The EDD complemented by the GSD can be considered to be a configuration related electronic data sheet for PROFIBUS devices. It can be delivered either on disc bundled with the device or via internet. It can even reside in every device. The EDD can be accessed either from the configuration tool repository representing the collection of EDDs or directly from the device, if the EDD resides in this device. Thus the consistency of the device version and its associated EDD can easily be checked.
The advantages of this methodology are:

- Only one configuration tool for all devices in the engineering system is needed instead of a bundle of different configuration tools.
- The configuration behaviour is stored in the EDD data set instead of binary software code.
- The EDD can easily be specified in EDDL by the device manufacturer; for the configuration software the manufacturer of the Engineering System is in charge.
- After configuration related device changes, only the EDD update is necessary.
- Only one configuration software edition per operating system is needed.
- The device manufacturer develops the EDD only, the system manufacturer provides the integration of the EDD tools in the engineering system.
- The EDD tools can be easily updated in the engineering system.
- Due to the ASCII format of the EDD, it is suitable for long-term archivation.
- The EDD may be used to derive other information such as HTML pages etc.

The EDD generation process is shown in Figure 2.

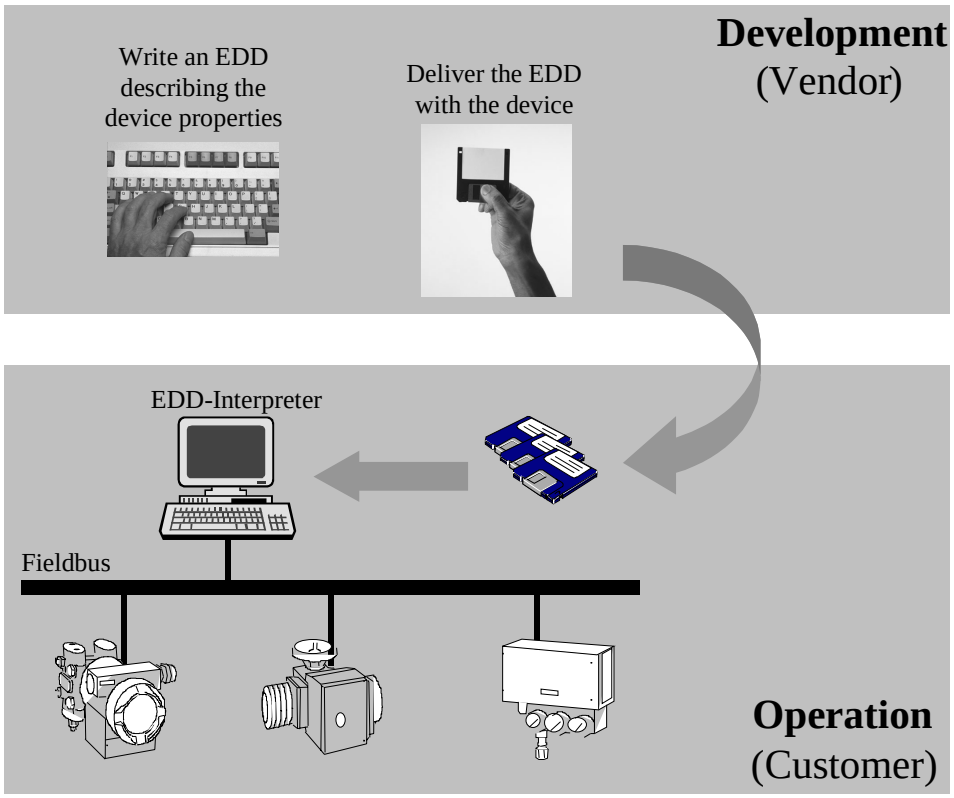


Figure 2: The EDD generation process

2.2 References

- PROFIBUS Specification (FMS, DP, PA) All normative Parts of the PROFIBUS Specification according to European Standard EN 50 170 Vol. 2. (version 1.0)
- GSD Specification for PROFIBUS-FMS Definition of the GSD-File formats for FMS (version 1.0)
- GSD Specification for PROFIBUS-DP Definition of the GSD-File formats for DP (version 3.0)
- Profile for Communication between Controllers FMS-Communication profile, specification of required services (version 1.4)
- Profile for Process Control Devices PA-Branch profile for Process Control devices (version 3.0)
- Profile for NC/RC Controllers DP profile for NC/RC Controllers (version 1.0)
- Profile for Encoders DP profile for rotary, angle and linear encoders (version 1.1)
- Profile for Variable Speed Drives FMS-/DP-Profile for electric drive technique (version 2.0)
- Profile for HMI Devices (Draft) DP-Profile for Human Machine Interface devices (version 1.0)
- Profile for Failsafe with PROFIBUS (Draft) DP-Profile for Safety Applications (version 1.0)
- KERNIGHAN, BRIAN W. AND DENNIS M. RITCHIE [1978]. The C Programming Language, Prentice Hall
  Inc., Englewood Cliffs, N.J.
- Unified Modelling Language Version 1.1

2.3 Abbreviations

ADU	Analog Digital Unit
DAU	Digital Analog Unit
EDD	Electronic Device Description
EDDI	Electronic Device Description Interpreter
EDDL	Electronic Device Description Language
GSD	Gerätestammdatendatei
HMI	Human Machine Interface
HTML	Hypertext Markup Language
PLC	Programmable Logic Controller
UML	Unified Modelling Language

Table 1: Abbreviations

2.4 Definitions

The following naming conventions of EDD components are defined:

Electronic Device Description Technology names the all over technology which starts with the development of EDD sources and ends with the necessary tool chain.

Electronic Device Description (EDD) names a data set describing the configuration behaviour of a device.

EDD source (no abbreviation of source) names the ASCII representation of the device description using the EDD language.

EDD language (EDDL) is the PROFIBUS device descriptive language.

Device Descriptive Language is a language to describe the device objects including their dependencies and their representation.
There exist several device descriptive languages. In general the differences are only found in the communication part of the description.

EDD server provides the device information via a specified interface to an application on a specific software platform. The server is able to load one or more EDD sources.

EDD editor is a software tool supporting the development of the EDD sources.

EDD checker is a test tool which checks the syntax and partly the semantic of EDD sources to guarantee compliance of EDD sources with the EDD language.

EDD compiler translates the EDD source into an EDD server internal format which can be used by the EDD interpreter.

EDD interpreter uses the EDD source to provide the EDD information to the EDD server interface.

2.5 Conventions

2.5.1 UML-Notations

In the appendix of this document, all important EDD language constructs are illustrated using class diagrams. The UML class diagram shows the classes and their relationships.
These illustrations are informal and do not have normative character. The class diagrams are taken from an UML specification, containing both, an abstract device model and the EDD language specification.
Figure 3 shows the UML constructs which are used in the EDDL diagrams.

Aggregation describes the whole-part relation. This relation is asymmetric, this means "A is a part of B" but not "B is a part of A".

Navigation presents an action which is executed by A and concerns B. The description of this action is specified by the text near to the arrow.

Class describes a set of objects with similar behaviour, attributes and relations to other objects.

2.5.2 Explanation of the Syntax- and Built-in-Description

The explanation of the syntax follows always the same scheme:

- name of the EDDL construct
- purpose of the EDDL construct
- syntax of the EDDL construct

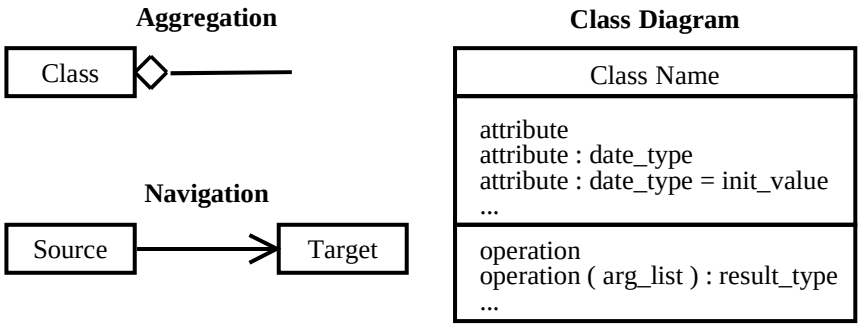


Figure 3: UML notation used for the description of the EDDL syntax

In the chapter "EDDL Method Built-ins Library" all built-ins are presented by the scheme:

- syntax
- description

Language fragments are used to demonstrate the syntax. The syntax uses the following notation conventions:

- Text in

  `typewriter`

  are language fragments described elsewhere in this document. All other text is literal.
- The dots in brackets (...) are a replacement for EDDL- or C-Code

2.6 EDD Background

The model of the EDD language is derived directly from the structure of smart field devices. Historically the smart devices are coming from 4–20 mA devices. The example in Figure 4 of a transmitter shows the way from an analog 4–20 mA device to a smart fieldbus device. Other types of field devices were or are going a similar way.

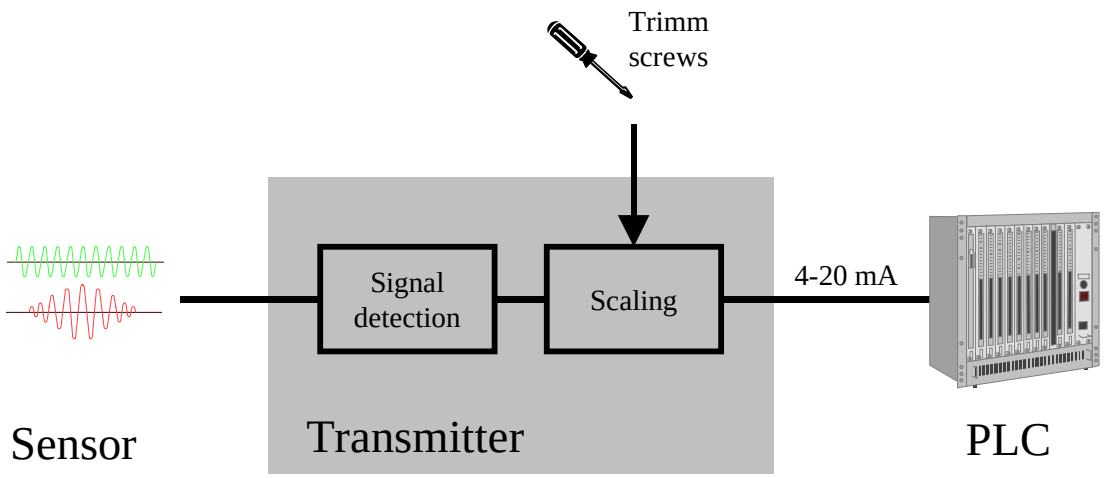


Figure 4: Structure of analog transmitter

The transmitter in Figure 4 is composed of electronics, which detects the specific measurement value (e.g. mV, mA) and which transforms the detected signal into the standardised 4–20 mA. The adjustment to the specific sensor and wiring is done by trim resistors set by a screw driver and checked by a multimeter. Each transmitter is connected to the Programmable Logic Controller (PLC) by its own wires.
Digital signal computation provides a higher accuracy. Therefore, the signal processing is carried out by micro processors (Figure 5). An analog/digital and a digital/analog unit transform the signals two times. The signal processing may be influenced by several variables and parameters, which make the transmitter more flexible. These parameters have to be accessed by the operator. The manufacturer provides a local operator panel with the transmitter consisting of a display and very few buttons. PC tools provide more ergonomic solutions for the commissioning of field devices, if those are more complex. The user gets a higher accuracy and reliability of the field device, but has to deal with many different tools from different manufacturers. The commissioning of the devices turns from the mechanical and electrical adjustment by screw drivers and multimeters to a parameterisation of digital data sets with the according user interface.

In principle, fieldbus devices replace the analog 4–20 mA converter by fieldbus controllers. That increases the accuracy of the devices again. These devices according to Figure 6 need additional communication parametrisation. The commissioning tools interact with the field devices via the fieldbus. The commissioning tool have to replace the local display and keyboard and have to provide all adjustment parameterisation of the device features. The used EDD language have to offer language elements to describe all mentioned device components, i.e. (Figure 7):

- Communication configuration parameters
- Device variables and functions
- Visualisation of device variables and user guidance for commissioning, diagnosis and maintenance

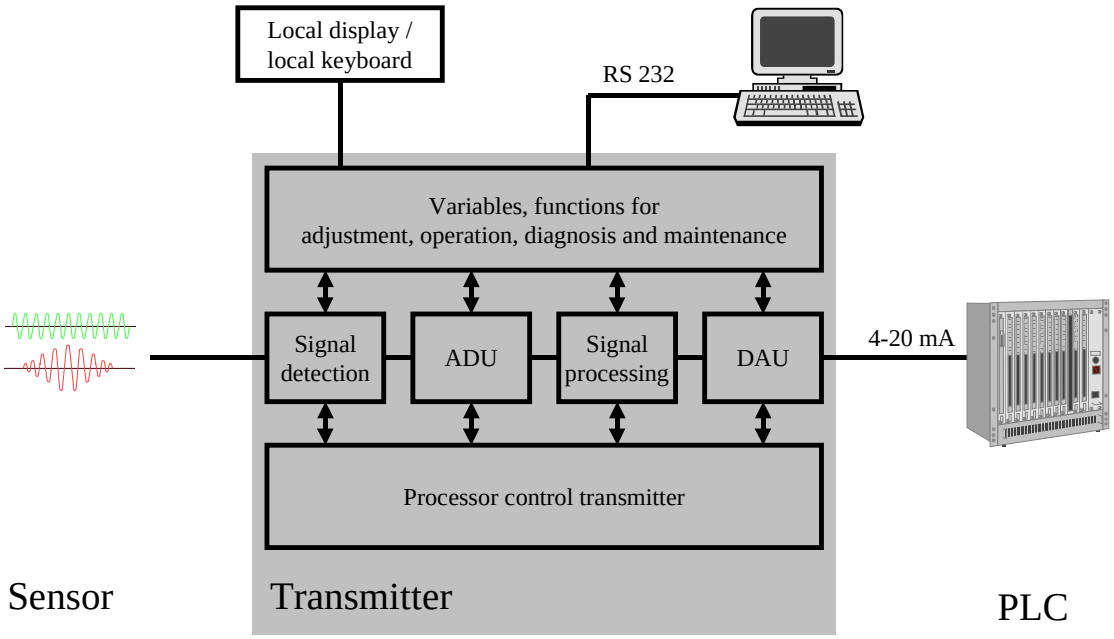


Figure 5: Structure of smart 4–20 mA transmitters

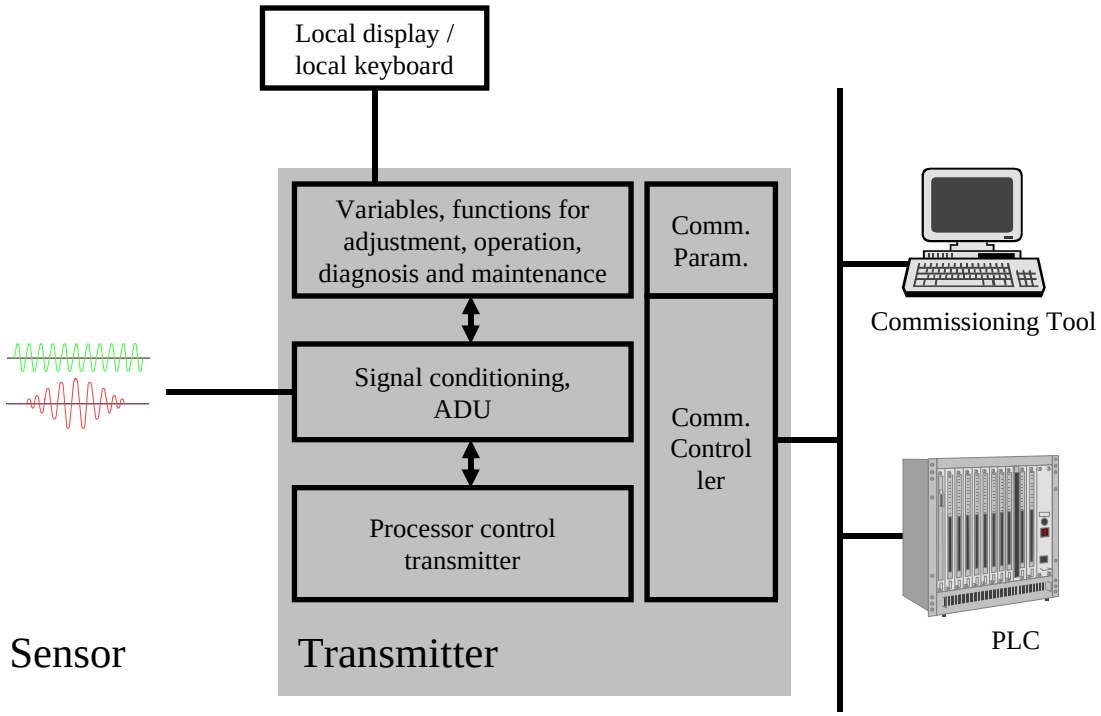


Figure 6: Structure of smart fieldbus transmitter

The EDDL is a language used to describe the information and procedures available through the fieldbus interface in a general and extensible way. It is a human readable structured text language designed to express how a field device can interact with a host device and other field devices. The basic constructs of the language are:

- Arrays
- Blocks
- Collections
- Commands
- Connections
- Domains
- Item Arrays
- Menues
- Methods
- Programs
- Records
- Refresh Relations
- Response Codes
- Unit Relations
- Variable Lists
- Variables
- Write As One Relations

Each of these constructs have a set of attributes associated with them. These attributes are used to define each construct. For example, a menu has three attributes: items, label and style. A specific menu is defined by defining each of these attributes. Attributes can also have sub-attributes, which refine the definition of the attribute and hence the definition of the construct itself. An Electronic Device Description (EDD) source file is developed using the EDDL syntax.

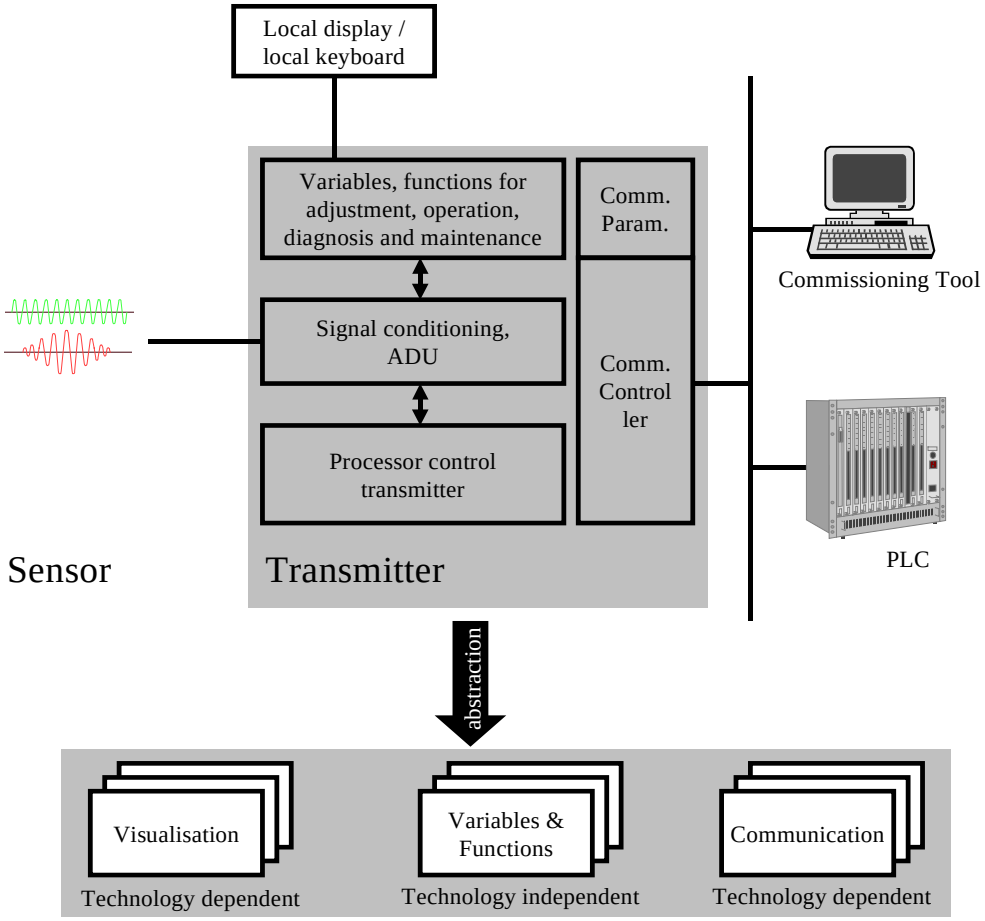


Figure 7: The components of an EDD

Examples of the information specified in a EDD source file are:

Parameter definitions These definitions identify all available device variables used for communication with other devices. For instance, a process variable and operation mode variable can be described.

Relationships Relationships among parameters are common in devices described by EDDL include refresh relationships, write-as-one relationships and unit relationships.

Human interface support Menues, help text, and display formats are available as hints to the interface developer. These facilities provide the device developer with a degree of control over the presentation of the device to the end user. Enough information is provided to implement a menu driven interface for a small display device, or a simple full screen display.

Variable Lists These constructs describe messages that contain groups of parameters for transmission and reception, along with application specific response codes and help. Variable lists allow write-as-one behavior as well as providing communication efficiencies.

Blocks These describe the parameters, parameter lists, and associated menues, relations etc. of a block.

Programs These constructs contain groups of variables that may be transmitted to a program object in a field device. The returned value, along with application specific response codes are available from the program object.

3 EDD Concept

3.1 Overview

The Electronic Device Description Language has been designed to implement a data vendor independent set called EDD describing device configuration, maintenance and functionality.
The EDDL describes the meaning or semantics, of the data sets. In its most basic form, the EDD source, it is human readable text written by device developers to specify all the information to configure the device using the communication interface.
The tightly coupled relationship that currently exists between the release of new field devices and the host device configuration tool will not existing any longer. Field device development schedules are not tied to host development or revision schedules. Field device developers will no longer be involved in verifying the operation of the configuration tool. They will only have to verify their EDD source file. The EDD source can be easily incorporated into a configuration tool just by reading by the EDD interpreter (EDDI).
Configuration tool developers no longer need to be responsible for validation testing of all devices supported in their products. They just have to ensure that they interpret the Electronic Device Descriptions correctly.
This document give a detailed description of the Electronic Device Description Language used to develop an Electronic Device Description source file. The other sections in this chapter briefly describe the architecture of the EDD application and its usage during both the design and operational phases of a device.

3.2 EDD Architecture

The EDD system architecture consists of a collection of specifications of EDDs together with a set of tools which are implemented following these specifications. Specifically the EDD Architecture consists of the following components:

- **Specifications**
  - A specification of a structured text language, called EDDL, used to specify the meaning and relationships between device properties available via the fieldbus. This specifies the syntax of the language used to create EDDL source files.
- **Tools**
  - A tool for converting the EDD source into a binary format. This tool, referred to as the compiler/interpreter, also validates for proper syntax and conformance to interoperability rules. Not all EDDL-constructs may be available in the tool because for specific application only a subset is needed. Therefore refer to the respective tool manual.
  - A tool for extracting information from the binary source and providing the information at an interface when needed by the applications, referred to as the Device Description Server.

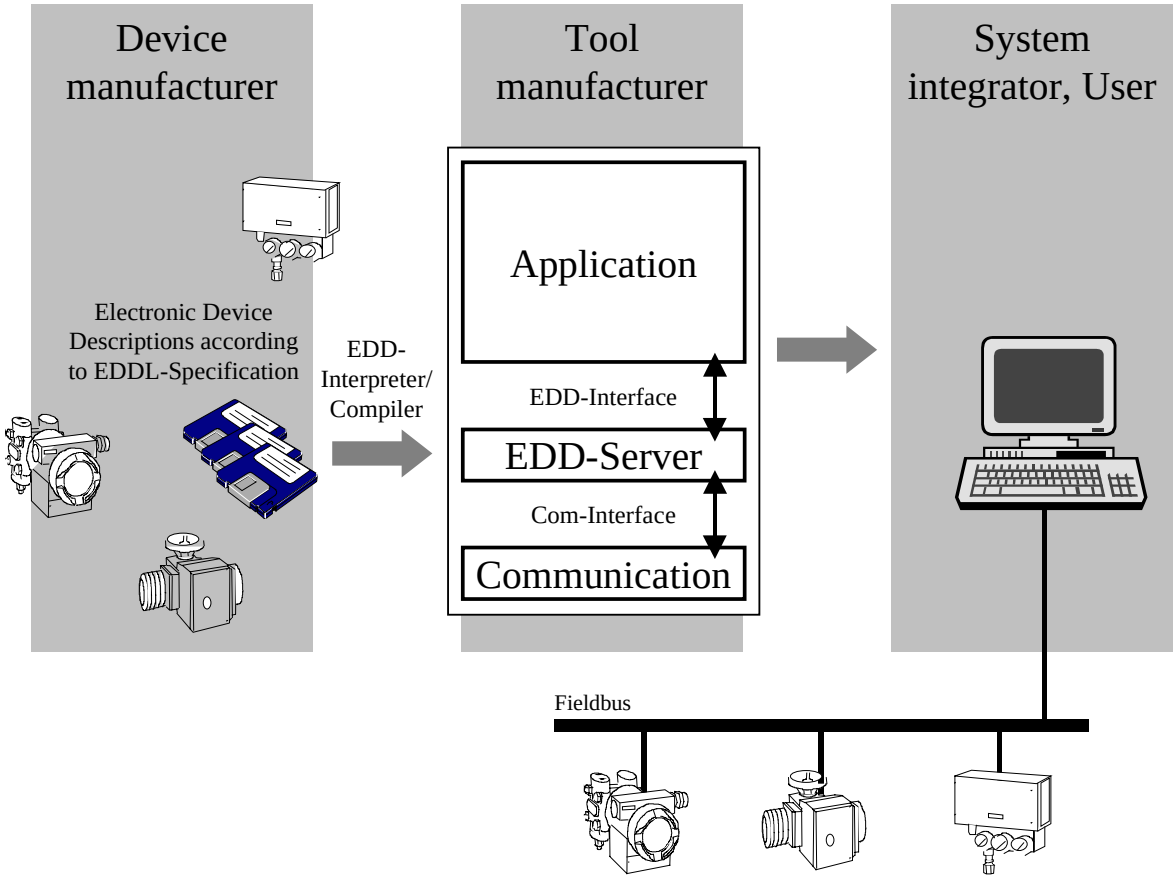


Figure 8: The integration of the EDD in the whole system

3.3 Electronic Device Description Source and Profiles

An EDD source file contains all necessary information needed to describe a field device. An EDD source file consists of two parts, standard and device specific. Standard descriptions are imported from the standard device descriptions maintained by the profile groups. The device implementor must write the device specific part. A detailed description of the syntax is found in the following chapter "Electronic Device Description Language".

4 EDD Language - Basic Elements

4.1 Introduction

The Electronic Device Description Language is a simple structured English language for describing field devices. The EDDL brings together in one place all the information a host device needs to operate with field devices. It presents this information as a clear, unambiguous, consistent description of a field device.

4.2 Preprocessor

Before processing by the Compiler, the source file is filtered through the standard C preprocessor. This filtering allows the EDD developer to use the standard C preprocessor directives such as #if, #ifdef, #endif, #define, and #include. Comments in a EDD source file are delimited with /* and */ or \\ .
The rest of this section refers to a source file that has already been processed. Experienced C programmers, however, should be very comfortable with this syntax.

4.3 Overview

There are sixteen basic constructs of the language: arrays, blocks, collections, commands, domains, item arrays, menus, methods, programs, records, refresh relations, response codes, unit relations, variable lists, variables, write as one relations.

Blocks describe the relative adressing of the parameter sets.

Connection defines multiple applications in a device.

Variables, records, and arrays describe the data contained in the device.

Menus describe how the data will be presented to a user by a host.

Methods describe the execution of complex sequence of event interactions that must take place between host devices and field devices.

Relations describe relationships between variables, records and arrays.

Item Arrays and Collections describe logical groupings of data.

Variable Lists describe logical groupings of data contained in the device that may be communicated as a group.

Commands describe the structure and the adressing of the variables in the device.

Programs specify how device executable code can be initiated by a host.

Domains can be used to download or upload moderately large amounts of data to or from a device.

Response codes specify the application specific response codes for a variable, record, array,variable list, program, or domain.

For example, some of the variables in a device are the process value, the upper and lower range values, and the upper and lower sensor limits. These variables would be described by the variable construct. Menus would describe what the users would see when they use the host to communicate with the device. The procedure used to trim the sensor as well as the procedure for reconfiguring the device would be specified by methods. Unit Relations are used to specify which variables are units codes and which variables have the units indicated by the units code. Refresh relations indicate variables that affect each other. Write-as-one relations indicate variables that are logically related and must be edited by the user simultaneously.
Each of the top level constructs, except relations and response codes, has a set of attributes associated with it. These attributes are used to define each construct. For example, a menu has the attributes: items and label. A menu is defined by specifying a definition for each of these attributes. Attributes may also have sub attributes, which refine the definition of the attribute and hence the definition of the top level construct.
The definition of an attribute may be static or dynamic. A static attribute definition never changes, while a dynamic attribute definition may change due to parameter value changes in the device. For example, an attribute that is defined one way when the device is in a certain mode and another way when it is not in that mode is a dynamic attribute definition. An attribute definition that is the same regardless of the situation is a static attribute definition.
The rest of this section describes the syntax and semantics of the Electronic Device Description Language.

4.4 Avoidance of Ambiguities in the EDD

4.4.1 Top Level Objects of equal Types and equal Identifiers

Top level objects of equal type and equal identifiers are not allowed.

4.4.2 Top Level Objects of different Types and equal Identifiers

Top level objects of different types and equal identifiers are not allowed. Example:

```
VARIABLE x
{ ... }

MENU x      // NOT ALLOWED!
{ ... }
```

4.4.3 Top Level Object containing equal Attributes

Top Level Object containing equal attributes are not allowed. Example:

```
VARIABLE x
{
    LABEL          "x";
    TYPE           INTEGER;
    CLASS          CONTAINED;
    CLASS          CONTAINED & DYNAMIC;     \\ NOT ALLOWED!
}
```

Also subattributes may not appear more than once:

```
VARIABLE y
{
    LABEL          "y";
    TYPE           INTEGER
    {
        MIN_VALUE  1;
        MAX_VALUE  2;
```



```
        MIN_VALUE  3;                          \\ NOT ALLOWED!
    }
    CLASS          CONTAINED;
    CLASS          CONTAINED & DYNAMIC;     \\ NOT ALLOWED!
}
```


4.5 Blocks

Purpose
A block construct defines the addressing scheme of a PROFIBUS device, which is organized in blocks.

Syntax

```
BLOCK name
{
    attribute, attribute, ...
}
```

where:

name is the name of the block. This name is used in the command for referencing.

attribute is one of the following block attributes:

- Required Attributes
  - Type
  - Number

4.5.1 Type Block Attribute

Purpose
A logical processing unit of software comprising an individual, named copy of the block and associated parameters specified by a block type, which persists from one invocation of the block to the next.

Syntax

```
TYPE type-definition;
```

where:

type-definition is one of the following types:

- PHYSICAL
  Hardware specific characteristics of a field device, which are associated with a resource, are made visible through the physical block. Similar to transducer blocks, they insulate function blocks from the physical hardware by containing a set of implementation independent hardware parameters.
- TRANSDUCER
  A named block consisting of one or more input, output and contained parameters. Function blocks represent the basic automation functions performed by an application which is as independent as possible of the specifics of I/O devices and the network. Each function block processes input parameters according to a specified algorithm and an internal set of contained parameters. They produce output parameters that are available for use within the same function block application or by other function block applications.
- FUNCTION
  Transducer blocks insulate function blocks from the specifics of I/O devices, such as sensors, actuators, and switches. Transducer blocks control access to I/O devices through a device independent interface defined for use by function blocks. Transducer blocks also perform functions, such as calibration and linearization, on I/O data to convert it to a device independent representation. Their interface to function blocks is defined as one or more implementation independent I/O channels.

4.5.2 Number Block Attribute

Purpose

A field device may contain several blocks which are described with the number attribute. The number attribute counts the blocks of the same type in the device management.

Syntax

```
NUMBER integer;
NUMBER name;
```

where:

integer order number of the block instance in the directory (Composite_Directory_Entries) of the same block type.

name is the value of the variable name.

4.6 Connection

Purpose

The connection command attribute specifies the name of the connection which is a reference to the connection type.

Syntax

```
CONNECTION name
{
    attribute, attribute, ...
}
```

where:

name is the name of the connection. This name is used in the command for referencing.

attribute is the following connection attribute:

- Required Attributes
  – Appinstance

4.6.1 Appinstance Connection Attribute

Purpose

Using this address model it is possible to define multiple applications in a device. Each application represents an Application Process Instance. Within an Application Process Instance it is possible to define different access levels. Further information about the addressing model can be found in the Profibus specification.

Syntax

```
APPINSTANCE integer;
```

where:

integer is the number of the application process instance. Further information about the addressing model can be found in the Profibus specification.

4.7 Variables

Purpose

VARIABLE is an EDDL construct which describes the data contained in a device.

Syntax

```
VARIABLE name
{
    attribute attribute ...
}
```

where:

name is the name of the variable. Every variable must have a name which may be used in the device description to refer to the variable.

attribute is one of the following variable attributes:

- Required Attributes
  - Class
  - Type
  - Label
- Optional Attributes
  - Constant unit
  - Handling
  - Help
  - Pre-/post-edit actions
  - Pre-/post-read actions
  - Pre-/post-write actions
  - Read timeout
  - Write timeout
  - Validity
  - Response codes

4.7.1 Class Variable Attribute

Purpose

The class attribute of a variable specifies how the variable is used by the host devices for organization and display.

Syntax

```
CLASS class-name & class-name & ... ;
```

here:

class-name identifies the variable class, and can be one of the following keywords:

INPUT Block parameters whose values can be determined by the output of another block.

OUTPUT Block parameters whose values may be accessed by another block input.

CONTAINED Block parameters that cannot be referenced by another block input or set by a block output.

DYNAMIC Variables modified by a field device without stimulus from the fieldbus network.

DIAGNOSTIC Variables that contain the device status.

SERVICE Variables in service or maintenance routines. For example, limit values that are defined for a deviation of reference measurement.

OPERATE Block parameters manipulated to control a block's operation (for instance, set point).

ALARM Variables of a block that represent the triggering limit for an alarm

TUNE Block parameters used to tune the algorithm of a block.

LOCAL Variables used locally by host devices. Local variables are not stored in a field device, but they can be sent to a field device. For example, a local variable may be used to guide the menu structure, that is, the user edits a variable and based on that value a new menu is presented. In this case, the local variable is never sent to a field device.

4.7.2 Type Variable Attribute

Purpose

A type describes the format of the variable's value.

Syntax

```
TYPE type-definition;
```

where:

type-definition is one of the following types (detailed descriptions of each type follow):

- Arithmetic Types
  - INTEGER
  - UNSIGNED_INTEGER
  - FLOAT
  - DOUBLE
- Enumeration Types
  - ENUMERATED
  - BIT_ENUMERATED
- Index Type
  - INDEX
- String Types
  - ASCII
  - PASSWORD
  - BITSTRING
- Date/Time Types
  - DATE_AND_TIME
  - TIME

4.7.2.1 Arithmetic Types

Purpose

Arithmetic variable types include the following:

- Float
- Double
- Integer
- Unsigned Integer

Variables of type float and double are single precision basic format and double precision basic format floating point numbers, as defined in ANSI/IEEE Std. 754.
Variables of type integer and unsigned integer are signed and unsigned integer numbers, respectively.

Syntax

```
FLOAT   { option option ...
        {value, description, help} ,
        {value, description, help} ,
        {value, description, help} }

DOUBLE  { option option ...
        {value, description, help} ,
        {value, description, help} ,
        {value, description, help} }

INTEGER ( size )
        { option option ...
        {value, description, help} ,
        {value, description, help} ,
        {value, description, help} }

UNSIGNED_INTEGER ( size )
        { option option ... }
        {value, description, help} ,
        {value, description, help} ,
        {value, description, help} }
```

where:

size specifies the size of the variable in octets. Size is an integer constant greater than zero and has no upper bound. This value is optional. The default is 1.

option specifies additional information about the variable related to its type. There are six arithmetic options:

- DISPLAY_FORMAT / EDIT_FORMAT
  A display format specifies how a host device will display the value of the variable. An edit format specifies how a host device will allow the variable to be edited by the user.

  ```
  DISPLAY_FORMAT string;
  EDIT_FORMAT string;
  ```

  string contains conversion specifiers for the ANSI C print function (for the display format) and ANSI C scan function (for the edit format).

- MIN_VALUE / MAX_VALUE
  Minimum and maximum values specify the range of values to which the user may set the variable. If the variable is a dynamic variable (see variable CLASS attribute), a field device can set the value of the variable outside the range specified by its minimum and maximum values.
  An arithmetic variable can have more than one minimum and maximum value. As an example, the variable can have a range just above zero and just below zero, but not exactly at zero.

  ```
  MIN_VALUE  expression;
  MAX_VALUE  expression;
  ```

  When there are multiple minimum and maximum values, an integer is appended to the keywords MIN_VALUE and MAX_VALUE. The integer must be in the range of 0 through 31. The minimum and maximum values with the same suffix form a range for the variable.
  For example, the following syntax specifies two ranges: one from -10 to -5 and another from 5 to 10:

  ```
  MIN_VALUE1 -10 ; MAX_VALUE1 -5;
  MIN_VALUE2   5 ; MAX_VALUE2 10;
  ```

- SCALING_FACTOR
  Scaling factor indicates that the actual value of the variable is not the value returned by a field device. The actual value is the value returned by a field device multiplied by a factor. Therefore, a host device must multiply the value of the variable returned by a field device with its scaling factor before it is displayed (or before it is used in any other way). This is useful for field devices that need to represent very large or very small values and for field devices that need to represent floating point values but do not have enough power for floating point arithmetic.

  ```
  SCALING_FACTOR  expression;
  ```

- DEFAULT_VALUE
  Available for all types. A variable can be preset to a constant value but can also depend on other variables. For this purpose DEFAULT_VALUE allows conditional expressions.

  ```
  DEFAULT_VALUE expression;
  ```

- INITIAL_VALUE
  Available for all types. Overwrites variable values set by DEFAULT_VALUE. A variable can be preset to a constant value. This constant value is defined with INITIAL_VALUE. The value defined with INITIAL_VALUE has a higher degree of priority as the DEFAULT_VALUE.

value (Optional) is an integer constant that specifies the value. The enumeration list is optional but if defined a value and a description is required. Equal values are not allowed.

description (Optional) is the text displayed when the variable takes on that value.

help (Optional) is text that provides a moderately extensive description of the value. The help text is intended to be used by host devices as on-line help.

There can be only one display format, one edit format, and one scaling factor. However, there can be multiple minimum and maximum values. All arithmetic options are optional.

4.7.2.2 Enumeration Types

Purpose

Enumeration type variables include the following:

Enumerated
: This variable type is an unsigned integer that has a text string associated with some or all of its values. One use for enumerated variables is to define tables.

Bit enumerated
: This variable type is an unsigned integer value that has a text string associated with some or all of its bits. One use for bit enumerated variables is defining status octets.

Syntax

```
ENUMERATED ( size )
           { option option ...
           {value, description, help} ,
           {value, description, help} ,
           {value, description, help} }
```

where:

size (Optional) specifies the size of the variable in octets. This value is an integer constant greater than zero and has no upper bound. The default is 1.

option (Optional) specifies additional information about the variable related to its type. There are two enumerated options:

- DEFAULT_VALUE
  A variable can be preset to a constant value but can also depend on other variables. For this purpose DEFAULT_VALUE allows conditional expressions.
- INITIAL_VALUE
  Overwrites variable values set by DEFAULT_VALUE. A variable can be preset to a constant value. This constant value is defined with INITIAL_VALUE. The value defined with INITIAL_VALUE has a higher degree of priority as the DEFAULT_VALUE.

value (Required) is an integer constant that specifies the value. Equal values are not allowed.

description (Required) is the text displayed when the variable takes on that value.

help (Optional) is text that provides a moderately extensive description of the value. The help text is intended to be used by host devices as on-line help.

```
BIT_ENUMERATED ( size )
               { option option ...
               {value, description, help, function, status-class, actions} ,
               {value, description, help, function, status-class, actions} ,
               {value, description, help, function, status-class, actions} }
```

where:

size (Optional) specifies the size of the variable in octets. This value is an integer constant greater than zero and has no upper bound. The default is 1.

option (Optional) specifies additional information about the variable related to its type. There are two bit enumerated options:

- DEFAULT_VALUE
  A variable can be preset to a constant value but can also depend on other variables. For this purpose DEFAULT_VALUE allows conditional expressions.
- INITIAL_VALUE
  Overwrites variable values set by DEFAULT_VALUE. A variable can be preset to a constant value. This constant value is defined with INITIAL_VALUE. The value defined with INITIAL_VALUE has a higher degree of priority as the DEFAULT_VALUE.

value (Required) is an integer constant that specifies a bit position, that is, only one bit is set in the binary representation of the value. Equal values are not allowed.

description (Required) is the text that will be displayed when that bit of the variable is set.

help (Optional) is text that provides a moderately extensive description of the bit. The help text is intended to be used by host devices as on-line help.

function (Optional) specifies the functional class of the bit. The functional class of a bit is the same as the class of a variable (see the "Class'' subsection earlier in this section). If no function is specified, the value of the function class defaults to the class of the variable. Therefore if all the bits have the same function you need only specify the class of the variable.

status class (Optional) specifies what the meaning of the bit is if it is a status bit. A status bit may belong to more than one status class. If the variable is not a status octet, the bits do not have status classes.
There are several types of status classes:

Cause
Duration
Correctability
Scope
Output
Miscellaneous

actions (Optional) specifies actions that will be performed by the host device when the bit is set. Each bit defined must specify a bit position and description. All other components are optional.

Table 2 shows the status class and the bit settings.

Cause	
HARDWARE	Hardware failure
SOFTWARE	Software failure
PROCESS	Problem with process connected to field device
MODE	Device is in a particular mode
DATA	Invalid data configuration
MISC	Miscellaneous condition
Duration	
EVENT	A one-time event
STATE	Field device is in a particular state.
Correctability	
SELF_CORRECTING	The bit will clear without further intervention

CORRECTABLE	The bit can be cleared by either a host device or the connected process
UNCORRECTABLE	The bit is neither self-correcting nor correctable
Scope	
SUMMARY	The bit is the logical combination of other bits of class detail. Each summary bit indicates a detail class
DETAIL	The bit is summarized elsewhere in a bit of class summary. Each detail bit indicates a specific status class
Miscellaneous	
MORE	There is more status available from the field device
COMM	Communications failure in the other device
IGNORE_IN_TEMPORARY_MASTER	Bit should be ignored in temporary master devices
Output Status	
BAD	Output is unreliable and should not be used for control.

Table 2: Status Classes and Bit Settings for Bit Enumerated Variables

4.7.2.3 String Types

Purpose

String variable types include the following:

ASCII
: This string type is for specifying a sequence of characters from the ISO Latin-1 character set.

Password
: This string type is intended for specifying password strings. Except for how the variable is presented to the user, password and ASCII string types are identical.

Bit String
: This string type is an ordered sequence of bits. The interpretation of the bits is unspecified.

Syntax

```
ASCII      (size) { option option ... };
{
    {value, description, help} ,
    {value, description, help} ,
    {value, description, help}
}
PASSWORD      (size)          { option option ... };
BITSTRING     (length)  { option option ... };
{
```



```
    {value, description, help} ,
    {value, description, help} ,
    {value, description, help}
}
```

where:

size is an integer constant greater than zero and specifies the number of characters from the appropriate character set. The size has no upper bound. It is important to note this specifies the number of characters in a string and not the size of the variable.

option (Optional) specifies additional information about the variable related to its type. There are two string options:

- DEFAULT_VALUE
  A variable can be preset to a constant value but can also depend on other variables. For this purpose DEFAULT_VALUE allows conditional expressions.
- INITIAL_VALUE
  Overwrites variable values set by DEFAULT_VALUE. A variable can be preset to a constant value. This constant value is defined with INITIAL_VALUE. The value defined with INITIAL_VALUE has a higher degree of priority as the DEFAULT_VALUE.

length is an integer constant greater than zero and specifies the number of bits.

value (Optional) is an integer constant that specifies the value. The enumeration list is optional but if defined a value and a description is required. Equal values are not allowed.

description (Optional) is the text displayed when the variable takes on that value.

help (Optional) is text that provides a moderately extensive description of the value. The help text is intended to be used by host devices as on-line help.

4.7.2.4 Index Type

Purpose

An index type variable is an unsigned integer which is interpreted as an index into an item array (see “Item Arrays” later in this section). An index variable may only take on the values defined by the item array, that is, the indices of the item array define the allowable values of the variable. When an index variable is presented to the user, the description of each of the indices of the item array should be displayed, not the numeric values of the indices.

Syntax

```
INDEX ( size ) item-array;
```

where:

size (Optional) specifies the size of the index type variable in octets. This value is an integer constant greater than zero and has no upper bound. The default is 1. The item array may not contain an index which exceeds the index size of the variable.

item-array specifies the item array into which the variable is an index.

4.7.2.5 Date / Time Types

Purpose

Date/Time variable types include the following:

- Date/Time
  This type is a sequence of octets representing the calendar date in both time and date. The octet representation is defined in the Profibus DPV1 specification.
- Time
  This type is sequence of octets representing time of day. The octet representation is defined in the Profibus DPV1 specification.

Syntax

```
DATE_AND_TIME;
TIME;
```

4.7.3 Constant Unit Variable Attribute

Purpose

If a variable has a units code associated with it and the units code never changes, the units code is specified by a constant unit. The constant units code is specified as the text that will be displayed along with the variable's value. A variable without a constant unit either has no units associated with it or the units are not constant.
Damping is an example of a variable whose units never change – it is always in seconds.

Syntax

```
CONSTANT_UNIT string;
```

4.7.4 Handling Variable Attribute

Purpose

Handling specifies the operations host devices may perform on the variable. There are two operations described by EDDL:

- The read operation indicates host devices can read the value of the variable from the device.
- The write operation indicates host devices can write the value of the variable to the device.

A variable without a handling attribute may be read and written by host devices.
These operations are independent of each other. Therefore, a variable may be read but not written, written but not read, both read and written, or neither read nor written.

Syntax

```
HANDLING handling & handling;
```

where:

handling is one of the following keywords:

- READ
- WRITE

4.7.5 Help Variable Attribute

Purpose

Help specifies text which provides a moderately extensive description of the variable. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

4.7.6 Label Variable Attribute

Purpose

A variable's label specifies text that host devices will display along with the variable's value. Every variable displayed by a host device needs a label.

Syntax

```
LABEL string;
```

4.7.7 Pre/Post Edit Actions Variable Attributes

Purpose

The pre/post edit actions of a variable specify actions host devices must execute when the user edits the variable.

- Pre-edit actions are executed before the variable is edited.
- Post-edit actions are executed after the variable is edited.

Syntax

```
PRE_EDIT_ACTIONS
{
    method, method, ...
}

POST_EDIT_ACTIONS
{
    method, method, ...
}
```

where:

method specifies an action the host device must execute before or after the user edits the variable.
The specified methods are executed in the order they appear at the appropriate time. If a method exits abnormally, the methods following it are not executed. If a pre-edit method aborts, the variable may not be edited.

4.7.8 Pre/Post Read Actions Variable Attributes

Purpose

The pre/post read actions of a variable specify actions host devices must execute when reading the variable from a field device.

- Pre-read actions are executed before initiating a read service request.
- Post-read actions are executed after receiving a read service confirmation.

Syntax

```
PRE_READ_ACTIONS
{
    method, method, ...
}

POST_READ_ACTIONS
{
    method, method, ...
}
```

where:

method specifies an action the host device must perform before or after reading the variable from a field device.
The specified methods are executed in the order they appear at the appropriate time. If a method exits abnormally, the methods following it are not executed. If a pre-read method aborts, the variable is not read from a field device.

4.7.9 Pre/Post Write Actions Variable Attributes

Purpose

The pre/post write actions of a variable specify actions host devices must execute when writing the variable to a field device.

Pre-write actions are executed before initiating a write service request.

Post-write actions are executed after receiving a write service confirmation.

Syntax

```
PRE_WRITE_ACTIONS
{
    method, method, ...
}

POST_WRITE_ACTIONS
{
    method, method, ...
}
```

where:

method specifies an action the host device must execute before or after writing the variable to a field device.
The specified methods are executed in the order they appear at the appropriate time. If a method exits abnormally, the methods following it are not executed. If a pre-write method aborts, the variable is not written to a field device.

4.7.10 Read/Write Timeout Variable Attributes

Purpose

A read timeout specifies the length of time, in milliseconds, host device must wait for the variable to be read from a field device.

Similarly, a write timeout specifies the length of time, in milliseconds, a host device must wait for the variable to be written to a field device.
For example, a write timeout may indicate the length of time it takes a field device to store the value of a variable. Then as long as host devices wait for the timeout to expire before reading the variable, the proper value will always be returned.

Syntax

```
READ_TIMEOUT expression;
WRITE_TIMEOUT expression;
```

An expression specified for the read or write timeout must evaluate to an integral value.

4.7.11 Validity Variable Attribute

Purpose

Although the parameter list of a function block is static, it is possible to indicate that some variables become logically nonexistent in some modes. A variable with the validity attribute defined as FALSE will be treated by a host device as though it didn't exist. This is a much different from variables with invalid values. A variable may be valid with respect to validity attribute, but have an invalid value. A variable without a validity is always considered valid.
For example, when a single sensor is connected to a temperature transmitter there is one sensor serial number, but when there are two sensors connected there are two sensor serial numbers. All the parameters associated with a missing sensor must be defined as invalid.

Syntax

```
VALIDITY boolean;
```

where:

boolean is either TRUE or FALSE.

Validity is almost always expressed using a conditional (IF, IF-ELSE, SELECT). If the validity of a variable is simply TRUE the variable is always valid; however, since this is the default it need not be specified as such. If the validity of a variable is simply FALSE the variable is never valid and should not be defined at all. Therefore, a conditional is usually used to specify that the variable is valid under certain conditions and invalid under other conditions.

4.7.12 Response Codes Variable Attribute

Purpose

Response codes specify the values that are returned from the FMS read and write services (see "Response Code Types" later in this section.).
Each variable can have its own set of response codes, because each variable is eligible for reading and writing.

Syntax

```
RESPONSE_CODES response-code-name;
```

4.7.13 Application Context

To ensure the integration of field devices into engineering environments it is useful to adapt the device (i.e. its description and therefore its appearance within host software tools) according to

specific contexts. This is done using a special variable called ApplicationContext, defined as follows:

```
VARIABLE ApplicationContext
{
    LABEL "ApplicationContext";
    CLASS LOCAL;
    TYPE BIT_ENUMERATED (4)
    {
        {0, "reserved"},
        {1, "FDT_CONFIGURATION"},
        {2, "FDT_PARAMETERIZE"},
        {3, "FDT_DIAGNOSIS"},
        {4, "FDT_MANAGEMENT"},
        {5, "FDT_OBSERVE"},
        {6, "FDT_DOCUMENTATION"},
        {7, "FDT_FORCE"},
        {8, "FDT_ASSET_MANAGEMENT"},
        {9, "reserved"},
        {10, "reserved"},
        {11, "reserved"},
        {12, "reserved"},
        {13, "reserved"},
        {14, "FDT_GMA_MAINTENANCE"},
        {15, "FDT_GMA_SPECIALIST"},
        {16, "DTM and / or vendor specific"},
        {17, "DTM and / or vendor specific"},
        {18, "DTM and / or vendor specific"},
        {19, "DTM and / or vendor specific"},
        {20, "DTM and / or vendor specific"},
        {21, "DTM and / or vendor specific"},
        {22, "DTM and / or vendor specific"},
        {23, "DTM and / or vendor specific"},
        {24, "DTM and / or vendor specific"},
        {25, "DTM and / or vendor specific"},
        {26, "DTM and / or vendor specific"},
        {27, "DTM and / or vendor specific"},
        {28, "DTM and / or vendor specific"},
        {29, "DTM and / or vendor specific"},
        {30, "DTM and / or vendor specific"},
        {31, "DTM and / or vendor specific"}
    }
}
```

More than one context can be provided, so a field of bits is used. Every EDD which is used in such engineering environments should contain this variable ApplicationContext.
The Variable ApplicationContext is set by the host environment (the engineering tool) and influences the device appearance according to pre-defined constraints and methods. E.g., the structure and the contents of menu definitions can be changed.

4.8 Menus

Purpose

A menu construct organizes parameters, methods, and other items specified in the EDDL into a hierarchical structure. A host application may use the menu items to display information to the user in an organized and consistent fashion.

Syntax

```
MENU name
{
    attribute attribute ...
}
```

where:

name is the name of the menu. Every menu must have a name which may be used in the device description to refer to the menu.

attribute is one of the following:

- Required Attributes
  - Label
  - Items
- Optional Attributes
  - Style
  - Access
  - Validity

4.8.1 Label-Menu Attribute

Purpose

The label of a menu is the text that is displayed when the menu appears as a menu item of another menu.

Syntax

```
LABEL string;
```

4.8.2 Items-Menu Attribute

Purpose

The items of a menu specify the items associated with this menus plus an optional qualifier.

Syntax

```
ITEMS
{
    menu-item , menu-item , ...
}
```

If a Menu-item occurs more than once the menu displays the item as often as it occurs. The Menu-item field may be any one of the following:

- variables
- methods
- other menus

Variables may be qualified with the following:

- (DISPLAY_VALUE)
- (READ_ONLY)
- (DISPLAY_VALUE, READ_ONLY)
- (HIDDEN)

Menus may be qualified with

- (REVIEW)

4.8.3 Style-Menu Attribute

Purpose

The style attribute specifies the type of the window. This attribute gives the manufacturer the possibility to supply special objects. For example a dialog contains a bargraph or a XY-Diagram for representing the measuring data.

Syntax

```
STYLE string
```

where:

string is one of the following items:

DIALOG for modal dialogboxes.
WINDOW for non-modal dialogboxes.
user-defined for embedding user-defined objects.

4.8.4 Access-Menu Attribute

Purpose

The access attribute defines whether the dialog communicates with the device during its lifecycle.

Syntax

```
ACCESS access-style
```

where:

access-style has the both possibilities:

- ONLINE
- OFFLINE

4.8.5 Validity-Menu Attribute

Purpose

A menu without a validity is always considered valid. A menu can be complete hidden, setting the validity to false.

Syntax

```
VALIDITY boolean;
```

where:

boolean is either TRUE or FALSE.

Validity is almost always expressed using a conditional (IF, IF-ELSE, SELECT). If the validity of a menu is simply TRUE the menu is always valid; however, since this is the default it need not be specified as such. If the validity of a menu is simply FALSE the menu is never valid and should not be defined at all. Therefore, a conditional is usually used to specify that the menu is valid under certain conditions and invalid under other conditions.

Application Handling

The menu items are presented to the user in the order they appear. For vertical menus, the first item appears on top and the last item appears on the bottom; for horizontal menus, the first item appears on the left and the last item appears on the right.
The following table specifies how the various menu-items are processed by a host application when displaying a menu item on a menu and when that menu items is selected by a user.

menu-item type	Host Application Handling
variable	**Display** The variable's label appears on the menu. If the variable is qualified with DISPLAY_VALUE, the value of the variable is displayed along with its label. **Selection** The value of the variable is presented to the user. If the variable may be modified (determined by variable's handling), the user is allowed to modify the variable. If the variable is qualified with READ_ONLY, the variable may not be modified via this menu, regardless of its handling. And if the variable is qualified with HIDDEN, it does not appear the user.
method	**Display** The method's label appears on the menu. **Selection** The method is executed.
menu	**Display** The menu's label appears on the menu. **Selection** The new menu is presented to the user. If the menu is qualified with REVIEW, the menu is presented in a manner consistent with reviewing an extensive set of data.

Table 3: Processing of menu-items

Data Access

To avoid unintentional access on the parameter of the field device, all menus are derived from these four entries.

- Horicontal menus
  - `MENU Menu_Main_Specialist`
  - `MENU Menu_Main_Maintenance`

- Vertical menus
  - `MENU Table_Main_Specialist`
  - `MENU Table_Main_Maintenance`

Starting the EDD-Interpreter, the user has to choose between specialist or maintenance and get the belonging menu-items and access rights.

4.8.6 Recommendation for the menu structure

Despite the unrestricted possibilities for the menu layout, it is useful to keep to a certain sequence. This ensures an almost completely standard user guidance for different devices.
DIN 19259 describes technical data and a classification scheme for measurement equipment in the industrial process. All EDDs should follow the structure according to this scheme. On the basis of DIN 19259, the following list provides help with the arrangement of variables for individual menus. The names used in this list must be adhered to strictly.

1. Identification
2. Application
3. Method of operation and structure
4. Input
5. Output
6. Characteristic values
   (a) Conditions of operation
   (b) Mounting conditions
   (c) Ambient conditions
7. Process conditions
8. Design
9. Display and operator interface
10. Auxiliary power
11. Certificates and approval documents
12. Ordering information
13. External standards and guidelines

For the menu arrangement, the following scheme was recommended:

- File, including save, properties, print or exit methods
- Device, including upload, download, self test or calibrate methods
- View, including measured-value display, alarm status or device status
- Tools, including configuration menus for the EDD-Tool
- Help, including help menus for the device or the EDD-Tool

The menu item “Device” contains dialogs that permit bi-directional communication with the device. That means that data are not only read but also loaded into the device. Unlike the menu item “Device”, the menu item “View” contains only passive elements such as status displays or measured value displays.

4.9 Methods

Purpose

A method describes the execution of interactions that must occur between host devices and a field device.

Syntax

```
METHOD name
{
attribute attribute ...
}
```

where:

name is the name of the method. Every method must have a name which may be used in the device description to refer to it.

attribute is one of the following method attributes

- Required Attributes
  - Class
  - Access
  - Definition
  - Label
- Optional Attributes
  - Help
  - Validity

4.9.1 Class-Method Attribute

Purpose

The class of a method specifies the affect of the method on a field device. This attribute is intended to be used by host devices to implement permission levels and organize how methods are presented.
Method classes are identical to variable classes. For more information, see the "Class'' subsection earlier in this chapter.

4.9.2 Access-Method Attribute

Purpose

The access of a method specifies wether the method implemention is using the variable values stored the device or the offline parameterset.

Syntax

```
ACCESS option;
```

where:

option can be one of the following keywords:

ONLINE specifies that the values of the variables used within the method definition are actually read from the device.

OFFLINE specifies that the values of the variables used within the method definition are read from the offline parameterset.

4.9.3 Definition-Method Attribute

Purpose

A method's definition specifies the actions to be performed by a host device. This requires a simple yet flexible means of specifying the computation of values and flow of control. The ANSI C programming language provides these features. Unfortunately, it also provides a lot of functionality that is unnecessary for most methods. To ease the burden on host device developers, the definition of a method is specified using only a subset of ANSI C.
The ANSI C subset used to specify the actions of host devices consists of simple declarations, expressions, and statements. This ANSI C subset includes the following items:

- Basic types (char, short, int, long, ...)
- Arrays (int [ ], long [ ], ...)
- Arithmetic operators (+, -, *, /, %, ...)
- Statements (if, for, switch, while, ...)

but does not include these items:

- Pointers (int *, long *, ...)
- Initializers (int x = 43;)
- Enumerations (enum {red, white, blue})
- Structures (struct { int day; int month; int year; })
- Unions (union { short sval; int ival; long lval; })

For the formal specification of the ANSI C subset allowed when specifying a method definition see "Methods'' in Appendix C.11.

Syntax

```
DEFINITION c-compound-statement
```

where:

c-compound-statement is as defined in ANSI C. Beyond it, it is possible to call user defined methods within a method.

Using Built-In Functions to Specify Method Actions

The actions specified by a method generally fall into two classes:

- User interaction. Typical user interaction actions include getting values from the user and getting acknowledgment from the user before continuing.
- Device interaction. Typical device interaction actions include sending read and write requests commands and interpreting response codes and status.

There is a library of built-in functions that can be used to specify actions taken by host devices. See chapter 4 for descriptions of the built-in functions.

Access to the attributes of a variable

Purpose

Within a method definition the attribute values can be read as follows.

Syntax

```
variable_name.attribute;
```

where:

variable_name is the name of the variable.

attribute is one of the following keywords:

- LABEL
- CONSTANT_UNIT
- HELP
- MIN_VALUE
- MAX_VALUE
- SCALING_FACTOR
- DEFAULT_VALUE
- INITIAL_VALUE

For example, runtime DEFAULT_VALUE supplies the default value of the varibale runtime.

4.9.4 Label-Method Attribute

Purpose

A method's label specifies text that host devices will display as the name of the method.

Syntax

```
LABEL string;
```

4.9.5 Help-Method Attribute

Purpose

Help specifies text which provides a moderately extensive description of the method. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

4.9.6 Validity-Method Attribute

Purpose

Validity specifies when the method is valid, that is, when it may be executed. Some field devices have methods that are only meaningful when the device is in a particular configuration. A method without a validity is always valid.

Syntax

```
VALIDITY boolean;
```

where:

boolean is either TRUE or FALSE.

Validity is almost always expressed using a conditional.
If the validity of a method is simply TRUE the method is always valid; however, since this is the default it need not be specified as such.
If the validity of a method is simply FALSE the method is never valid and should not be defined at all.
Therefore, a conditional is usually used to specify that the method is valid under certain conditions and invalid under other conditions.

4.9.7 Methods with Arguments

Purpose

Methods with arguments are used to save code, e.g. if the same method is needed but with different variables. In this case the EDDL supply function calls with arguments as specified in ANSI C. For parameter passing the mechanisms call-by-value and call-by-reference are possible.

Syntax

```
method_name(variable, variable, ... )

METHOD method_name(type var, type var, ... )
{
    DEFINITION
    {
        ...
    }
}
```

where:

method_name is the name of the method.

variable is the name of the variable defined in the EDD. All input variables in the argument list are coinstantaneous to the output variables.

type is the type (int, float, long) of the variables used in the method. The type is identical with the type of the device variable. If the EDDL type and the ANSI C type has different length (e.g. INTEGER(1) → int), the EDDL variable is casted to the ANSI C type. After the execution of the method the ANSI C type is casted back to the EDDL type. The application has to care for that after the method execution the value of the variable is within the defined min and max values.

var is the name of the variable used in the method.

4.10 Relations

Purpose

Relations specify relationships between variables. The EDDL defines the following types of relations:

- Refresh
- Unit
- Write-as-one

4.10.1 Refresh Relation

Purpose

A refresh relation allows the host device to make decisions regarding parameter value consistency when a parameter value changes. It specifies a set of block parameters which may need to be refreshed (reread from the device) whenever a block parameter from another set is modified. A block parameter can have a refresh relationship with itself, implying that the block parameter must be read after writing.
Occasionally writing a block parameter to a field device causes the field device to update the values of other block parameters. If the additional updated block parameters are dynamic, there is no conflict, because the host device should reread the parameter values from a field device each time the values are needed. However, host devices may cache the values of static block parameters. Therefore, for host devices to maintain the correct values of all static block parameters, they need to know when the field device is changing its values.

Syntax

```
REFRESH name
{
    parameter, parameter, ...
    : parameter, parameter, ..
}
```

where:

name is the name of the refresh relation. Every refresh relation must have a name which can be used in the device description to refer to it.

parameter is a block parameter. The block parameters following the colon should be reread from the device whenever one of the block parameters preceding the colon is modified.

4.10.2 Unit Relation

Purpose

A unit relation specifies a units code parameter and the block parameters with those units. When a units code parameter is modified, the block parameters with that units code should be refreshed. In this respect, a unit relation is exactly like a refresh relation. In addition, when a block parameter with a units code is displayed, the value of its units code will also be displayed.

Syntax

```
UNIT name
{
    unit-code: parameter, parameter, ...
}
```


where:

name is the name of the unit relation. Every unit relation must have a name which can be used in the device description to refer to it.

unit-code is the units code of each of the block parameters following the colon.

parameter is a block parameter associated with the units code. This value can be a variable or an array.

4.10.3 Write-As-One Relation

Purpose

A write-as-one relation informs the host device that a group of block parameters needs to be modified as a group. This relation does not necessarily mean the block parameters are written to the field device at the same time. Not all block parameters sent to the field device at the same time are necessarily part of a write-as-one relation.
If a field device requires specific block parameters to be examined and modified at the same time for proper operation, a write-as-one relation is required.

Syntax

```
WRITE_AS_ONE name
{
    parameter, parameter, ...
}
```

where:

name is the name of the write-as-one relation. Every write-as-one relation must have a name which can be used in the device description to refer to it.

parameter is a block parameter that must be modified with other members of the write-as-one relation by an application in a host device. A parameter may occur only once.

4.11 Item Arrays

Purpose

An item array is a logical group of items, such as variables or menus. Each item in the group is assigned a number, called an index. The items can be referenced from elsewhere in the device description via the item array name and the item number. Item arrays are merely groups of EDDL items and are unrelated to communication arrays (item type "ARRAY''). Communication arrays are arrays of values.

Syntax

```
ITEM_ARRAY OF item-type name
{
    attribute ...
}
```

where:

item-type specifies the type of elements in the item array. All the item array elements must be of the specified type. Following types are allowed:

- VARIABLE
- MENU
- METHOD
- REFRESH
- UNIT
- WRITE_AS_ONE
- ITEM_ARRAY OF item_type
- COLLECTION OF item_type
- RECORD
- ARRAY
- VARIABLE_LIST
- PROGRAM
- DOMAIN
- BLOCK
- COMMAND
- CONNECTION
- RESPONSE_CODES

name is the name of the item array. Every item array must have a name which may be used in the device description to refer to it.

attribute is one of the following item array attributes:

- Required Attributes
  - Elements
- Optional Attributes
  - Help
  - Label

4.11.1 Elements-Item Array Attribute

Purpose

The elements item array attribute identifies elements of an item array. Each item array element specifies one item (such as a variable or menu) in the group, and is defined by a group of four parameters (index, item, description, help).

Syntax

```
ELEMENTS
{
    index, item, description, help;
    index, item, description, help;
}
```

where:

index (Required) specifies the number by which the item may be referenced. The item array may not contain an index which exceeds the size of a variable. An index which refers to the same item-type may not occur more than once.

item (Required) is the name of the EDDL item associated with the index value.

description (Optional) provides a short description of the item.

help (Optional) specifies help text for the item.

4.11.2 Help-Item Array Attribute

Purpose

Help specifies text which provides a moderately extensive description of the item array. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

where:

string specifies the help string.

4.11.3 Label-Item Array Attribute

Purpose

An item array's label specifies text that host devices will display as the name of the item array.

Syntax

```
LABEL string;
```

where:

string specifies the help string.

4.12 Collections

Purpose

A collection is a logical group of items, such as variables or menus. Each item in the group is assigned a name. The items may be referenced from in the device description by using the collection name and the item name.

Syntax

```
COLLECTION OF item-type name
{
    attribute attribute ...
}
```

where:

item-type specifies the type of members in the collection. All the collection members must be of the specified type. The following types are allowed:

- VARIABLE
- MENU
- METHOD
- REFRESH
- UNIT
- WRITE_AS_ONE
- ITEM_ARRAY OF item_type
- COLLECTION OF item_type
- RECORD
- ARRAY
- VARIABLE_LIST
- PROGRAM
- DOMAIN
- BLOCK
- COMMAND
- CONNECTION
- RESPONSE_CODES

name is the name of the collection. Every collection must have a name which may be used in the device description to refer to it.

attribute is one of the following collection attributes (descriptions of each attribute follow):

- Required Attributes
  - Members
- Optional Attributes
  - Help
  - Label

Remark:
Item types must be completeley specified. For example, if the members of a collection contains item arrays of variables, then it must be specified as COLLECTION OF ITEM_ARRAY OF VARIABLE.

4.12.1 Members-Collection Attribute

Purpose

The members collection attribute defines members of a collection. Each collection member specifies one item (such as a variable or menu) in the group, and is defined by a group of four parameters (name, item, description, help).

Syntax

```
MEMBERS
{
    name, item, description, help;
    name, item, description, help;
}
```

where:

name (Required) specifies the name by which the item may be referenced.

item (Required) is the name of the EDDL item associated with the name value. It is not allowed to define an item more than once.

description (Optional) is a short description of the item.

help (Optional) specifies help text for the item.

4.12.2 Help-Collection Attribute

Purpose

Help specifies text which provides a moderately extensive description of the collection. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

where:

string specifies the help string.

4.12.3 Label-Collection Attribute

Purpose

A collection's label specifies text that host devices will display as the name of the collection.

Syntax

```
LABEL string;
```

string specifies the help string.

4.13 Records

Purpose

A record is a logical group of variables. Each variable in the record is assigned a EDDL variable name. Each variable may have a different data type. The variables may be referenced from elsewhere in the device description via the record name and the member name. EDDL records describe communication record objects. Therefore, from a communication perspective, the individual members of the record are not treated as individual variables, but simply as a group of variable values.

Syntax

```
RECORD name
{
    attribute attribute ...
}
```

where:

name is the name of the record. Every record must have a name which may be used in the device description to refer to it.

attribute is one of the following record attributes (descriptions of each attribute follow):

- Required Attributes
  - Members
  - Label
- Optional Attributes
  - Help
  - Response Codes

4.13.1 Members-Record Attribute

Purpose

The members record attribute defines the members of a record. Each record member specifies one EDDL variable, and is defined by a group of four parameters (name, item, description, help).

Syntax

```
MEMBERS
{
    name, item, description, help;
    name, item, description, help;
}
```

where:

name (Required) specifies the name by which the variable may be referenced through the record.

Item (Required) is the name of the EDDL item associated with the name value. It is not allowed to define an item more than once.

description (Optional) is a short description of the variable.

help (Optional) specifies help text for the variable.

4.13.2 Help-Record Attribute

Purpose

Help specifies text which provides a moderately extensive description of the record. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

4.13.3 Label-Record Attribute

Purpose

A record's label specifies text that host devices will display as the name of the record.

Syntax

```
LABEL string;
```

4.13.4 Response Codes-Record Attribute

Purpose

Response codes specify the error values that are returned from the FMS read and write services (see "Response Code Types'' later in this section). Each record can have its own set of response codes, because each record is eligible for reading and writing.

Syntax

```
RESPONSE_CODES response-code-name;
```


4.14 Arrays

Purpose

An array is a logical group of values. Each value, or element, is of the data type of an EDDL variable. An element may be referenced from elsewhere in the device description via the array name and the element index. EDDL arrays describe communication array objects. Therefore, from a communication perspective, the individual elements of the array are not treated as individual variables, but simply as individual values.

Syntax

```
ARRAY name
{
    attribute attribute ...
}
```

where:

name is the name of the array. Every array must have a name which may be used in the device description to refer to it.

attribute is one of the following array attributes (descriptions of each attribute follow):

- Required Attributes
  - Type
  - Number of Elements
  - Label
- Optional Attributes
  - Help
  - Response Codes

4.14.1 Type-Array Attribute

Purpose

The type array attribute specifies the data type and attributes of each of the elements. Therefore, the type is a reference to an EDDL variable.

Syntax

```
TYPE variable-name;
```

4.14.2 Number of Elements-Array Attribute

Purpose

The number of elements array attribute specifies the number of elements in the array as an integer constant greater than zero.

Syntax

```
NUMBER_OF_ELEMENTS integer-constant;
```


4.14.3 Help-Array Attribute

Purpose

Help specifies text which provides a moderately extensive description of the array. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

4.14.4 Label-Array Attribute

Purpose

An array's label specifies text that host devices will display as the name of the array.

Syntax

```
LABEL string;
```

4.14.5 Response Codes-Array Attribute

Purpose

Response codes specify the error values that are returned from the FMS read and write services (see "Response Code Types'' later in this section). Each array may have it's own set of response codes because each array is eligible for reading and writing.

Syntax

```
RESPONSE_CODES response-code-name;
```


4.15 Variable Lists

Purpose

A variable list is a logical group of EDDL communication objects (variables, arrays, or records). Each item in the group is assigned a name. The items may be referenced from elsewhere in the device description via the variable list name and the item name. EDDL variable lists describe predefined communication variable lists.

Syntax

```
VARIABLE_LIST name
{
    attribute attribute ...
}
```

where:

name is the name of the variable list. Every variable list must have a name which may be used in the device description to refer to it.

attribute is one of the following variable list attributes (descriptions of each attribute follow):

- Required Attributes
  - Members
- Optional Attributes
  - Help
  - Label
  - Response Codes

Variable lists can contain only variables, arrays, or records that appear as block parameters.

4.15.1 Members-Variable List Attribute

Purpose

The members variable list attribute defines the members of a variable list. Each variable list member specifies one item (variable or record) in the group, and is defined by a group of four parameters (name, item, description, help).

Syntax

```
MEMBERS
{
    name, item, description, help;
    name, item, description, help;
}
```

where:

name (Required) specifies the name by which the item may be referenced.

item (Required) is the name of the EDDL item associated with the name value. It is possible to define items of different item-types. It is not allowed to define an item more than once. Using conditional expressions in the Members-list, you have to pay attention that the name may refer to different item-types during the execution.

description (Optional) is a short description of the item.

help (Optional) specifies help text for the item.

4.15.2 Help-Variable List Attribute

Purpose

Help specifies text which provides a moderately extensive description of the variable list. This text is intended to be used by host devices as on-line help.

Syntax

```
HELP string;
```

4.15.3 Label-Variable List Attribute

Purpose

A variable list's label specifies text that host devices will display as the name of the variable list.

Syntax

```
LABEL string;
```

4.15.4 Response Codes-Variable List Attribute

Purpose

Response codes specify the error values that are returned from the FMS read and write services (see "Response Code Types" later in this section). Each variable list may have it's own set of response codes because each variable list is eligible for reading and writing.

Syntax

```
RESPONSE_CODES response-code-name;
```


4.16 Command

Purpose

Each device variable has to be adressed within a command structure. The operation of a command determines wether a variable is read or written from host to device. Furthermore the absolute or relative adressing scheme is specified by the command structure.

Syntax

```
COMMAND Name
{
    attribute; attribute; ...
}
```

where:

name is the name of the command. Every command must have a name.

attribute is one of the following variable attributes:

- Block
- Slot
- Index
- Operation
- Connection
- Module
- Response Codes
- Transaction

4.16.1 Block Command Attribute

Purpose

The block command attribute specifies the name of the block which is a reference to the block type (see also the chapter Block). Adressing variables using the block attribute is designated as relative adressing.

Syntax

```
BLOCK name;
```

4.16.2 Slot Command Attribute

Purpose

The slot command attribute specifies the number of the slot. The variables of a PROFIBUS-Device are allocated to different slots. Adressing variables using the slot attribute is designated as absolute adressing.

Syntax

```
SLOT number
SLOT name
```

where:

number is the number of the slot.

name is the value of the variable name.

4.16.3 Index Command Attribute

Purpose

A block or a slot are devided in groups. These groups are referenced with the index attribute.

Syntax

```
INDEX number;
INDEX name
```

where:

number is the number of the slot.

name is the value of the variable name.

4.16.4 Operation Command Attribute

Purpose

The operation command specifies the action the host initiate to the field device. There are three possible operations: read, write and command.

Syntax

```
OPERATION attribute;
```

where:

attribute can be one of the following:

READ Receiving the read command, the field device sends back the values of the variables listed in the transaction attribute.

WRITE Receiving the write command, the field device sets the values of the variables to the values coming from the host device.

DATA_EXCHANGE declares cyclic communication as defined in the Profibus specification. Using this type of operation you have to list the input parameter in the reply attribut and the output parameter in the request attribut of the transaction. In this operation mode the definition of slot and index are not necessary but the module reference has to be specified.

COMMAND Upon receiving a command command, the field device performs a device-specific set of actions. In methods the description are specified how these commands are to be used by host devices.

4.16.5 Connection Command Attribute

Purpose

The connection command attribute specifies the name of the connection object which is a reference to the connection type (see also the chapter Connection).

Syntax

```
CONNECTION name;
```


4.16.6 Module Command Attribute

Purpose

The module command attribute specifies the name of the module which is a reference to the module type specified in the GSD. The module contains the set of parameters which may be communicated within a cyclic channel.

Syntax

```
MODULE name;
```

4.16.7 Response Code Command Attribute

Purpose

The Response code may return values from the field device which represents status messages.

Syntax

```
RESPONSE_CODES
{
    value, type, description, help ;
    value, type, description, help ;
}
```

where:

value is an integer type and specifies the respond code value. Equal values are not allowed.

type can be one of the following:

SUCCESS The action, initiated by the command was accepted.

MISC_WARNING The action was accepted and processed by the field device but there are additional information, disconnected to the command action.

DATA_ENTRY_WARNING The action was accepted but with a slightly modified version of the data sent.

DATA_ENTRY_ERROR The action was denied because of invalid data.

MODE_ERROR The action was denied because the field device was in a mode in which the action cannot executed.

PROCESS_ERROR The action was denied because the field device was an invalid type.

MISC_ERROR The action was denied.

description is a string that specifies the displayed message when the response code is returned from the field device.

help is a text which can be used by the host device as an online help.

4.16.8 Transaction Command Attribute

Purpose

The transactions define the data set in the request and reply directive. It is possible to define more than one transaction by appending an integer to the keyword TRANSACTION (e.g. TRANSACTION2). But it is not allowed to define transactions with equal numbers or more than one transaction without a number.

Syntax

```
...
OPERATION WRITE
TRANSACTION
{
    REQUEST
    {
        data-item , data-item , ...
    }
    REPLY
    {
    }
}
...
OPERATION READ;
TRANSACTION
{
    REQUEST
    {
    }
    REPLY
    {
        data-item , data-item , ...
    }
}
```

where:

data-item is either a variable or the name of a variable. The download variables are found under REQUEST, the upload variables are found under REPLY.

4.16.8.1 Data Item Mask

Purpose

An integer variable in a request or replay may also contain a bit mask. The mask defines in which way the bits of the integer are assigned to the corresponding variables.

Syntax

```
...
OPERATION WRITE
TRANSACTION
{
    REQUEST
    {
        data-item    <integer>,
        data-item    <integer>,
        ...
    }
    REPLY
    {
    }
}
...
OPERATION READ;
TRANSACTION
{
    REQUEST
    {
```



```
    }
    REPLY
    {
        data-item    <integer>,
        data-item    <integer>,
        ...
    }
}
```

where:

integer presents the bit mask. When the LSB is set in the bit mask, the pointer shows to the next byte of the data set. The masks may contain gaps. Furthermore the data-item list may be conditioned.

4.16.8.2 Data Item Qualifier

Purpose

Variables listed in a request or reply may be qualified with the INDEX and INFO. INDEX specifies that the variable is used in the request or reply as an index into an array. INFO specifies that the variable is not actually stored in the device. The variable has an informal meaning. A variable may be qualified with INDEX and INFO. It is called a local index variable.

Syntax

```
TRANSACTION
{
    REQUEST
    {
        data-item (INFO)
    }
    REPLY
    {
    }
}

TRANSACTION
{
    REQUEST
    {
        data-item (INDEX)
    }
    REPLY
    {
    }
}

TRANSACTION
{
    REQUEST
    {
        data-item (INDEX,INFO)
    }
    REPLY
    {
    }
}
```

4.16.9 Upload-/Download-Menu

Purpose

The Upload-/Download-Menu is an EDDL construct to specify which parameters are read or written from the host to the device. These menu definitions are always top level objects.

Syntax

```
MENU download_variables
{
    LABEL    name;
    ITEMS
    {
        variable, variable, ...
    }
}

MENU upload_variables
{
    LABEL    name;
    ITEMS
    {
        variable, variable, ...
    }
}
```

where:

download_variables is the name of the download menu. The item list specifies the variables which are sent from the field device to the host. If this menu does not exist, all variables defined in the EDD are sent from the field device to the host.

upload_variables is the name of the upload menu. The item list specifies the variables which are sent from the host to the field device. If this menu does not exist, all variables defined in the EDD are sent from the host to the field device.

4.17 Programs

Purpose

Programs can be used to specify device actions that can be initiated by a host. Examples of programs include "perform self test," "Go to save state," and "go to initialized state." The program description describes a program invocation object created in the device.

Syntax

```
PROGRAM name
{
    attribute attribute ...
}
```

where:

name is the name of the program. Every program must have a name which may be used in the device description to refer to it.

attribute is one of the following program attributes (descriptions of each attribute follow):

- Optional Attributes
  - Arguments
  - Response Codes

4.17.1 Arguments-Program Attribute

Purpose

Arguments can be sent to the program during start and resume operations. The program arguments are described by the ARGUMENT attribute. An octet string containing the values of all of the arguments will be sent to the program invocation object when it is started or resumed by the application.

Syntax

```
ARGUMENT
{
    data-item, data-item ...
}
```

where:

data-item is either an unsigned integer constant or a variable.

- If a data item is an unsigned integer constant, the value of the constant appears at that position in the data field. Constant data items occupy two octets of the data field and therefore must be in the range 0 through 65535 inclusive.
- If a data item is a variable, the value of the variable appears at that position in the data field.
- If the data field of the program service is empty, the arguments can be omitted, or specified as follows:

  ```
  ARGUMENT  { }
  ```


4.17.2 Response Codes-Program Attribute

Purpose

Response codes specify the values a field device may return as program errors (see "Response Code Types'' later in this section).

Syntax

```
RESPONSE_CODES response-code-name
```

4.18 Domains

Purpose

Domains can be used to download and upload moderately large amounts of data to and from a device. The domain description describes a domain object created in the device.

Syntax

```
DOMAIN name
{
    attribute attribute ...
}
```

where:

name is the name of the domain. Every domain must have a name which may be used in the device description to refer to it.

attribute is one of the following domain attributes (descriptions of each attribute follow):

- Optional Attributes
  - Handling
  - Response Codes

4.18.1 Handling-Domain Attribute

Purpose

Handling specifies the operations host devices may perform on the domain. There are two operations:

- The read operation indicates host devices may upload the domain chunk of memory from the device.
- The write operation indicates host devices may download the domain chunk of memory to the device. A domain without a handling attribute may be read and written by host devices.

Syntax

```
HANDLING handling & handling;
```

where:

handling is one of the following keywords:

- READ
- WRITE

The read and write operations are orthogonal, that is, each operation is independent of the other. Therefore, a variable may be read but not written, written but not read, both read and written, or neither read nor written. If both keywords are used then they are linked with the ampersand &.

4.18.2 Response Codes-Domain Attribute

Purpose

Response codes specify the values a field device may return as domain download/upload errors (see "Response Code Types'' later in this section).

Syntax

```
RESPONSE_CODES response-code-name;
```


4.19 Response Codes

Purpose

Response codes specify the values a field device may return as application specific errors. Each variable, record, array, variable list, program, or domain can have its own set of response codes, because each one is eligible for FMS services.

Syntax

```
RESPONSE_CODES response-code-name
{
    value, type, description, help;
    value, type, description, help;
}
```

where:

value (Required) specifies response code value. Equal values are not allowed.

type (Required) specifies the type of the response code. Response code types specify the reasons response codes are returned. See the following table (response) for allowed response code types.

description (Required) is a short description of the response code.

help (Optional) specifies help text for the response code.

Type	Description
SUCCESS	The application layer service was accepted and processed as specified.
DATA ENTRY WARNING	The application layer service was accepted and processed with a slightly modified version of the data sent.
MISC WARNING	The application layer service was accepted and processed as specified and there is additional information, unrelated to the application layer service, in which the user might be interested.
DATA ENTRY ERROR	The application layer service was rejected because the data sent was invalid.
MODE ERROR	The application layer service was rejected because the field device was in a mode in which the application layer service could not be executed.
PROCESS ERROR	The application layer service was rejected because a process applied to the field device was invalid.
MISC ERROR	The application layer service was rejected.

Table 4: Response Code Types

4.20 Device Description Information

Purpose

The device description information attributes identify a specific device description. Electronic device description information attributes include the following:

- Manufacturer
- Device Type
- Device Revision
- EDD Revisions

Syntax

```
MANUFACTURER    integer,
DEVICE_TYPE     integer,
DEVICE_REVISION integer,
EDD_REVISION    integer
```


4.21 Output Redirection (OPEN and CLOSE Keywords)

Purpose

The output of the EDDL-Compiler is a set of objects. The OPEN and CLOSE keywords allow the developer to create various subsets of object files.

- The OPEN keyword opens an output file. Each time an OPEN keyword is processed, a new output file is opened (and created if necessary). All objects generated are written to all open files.
- The CLOSE keyword prevents further output to an open file. If a file is reopened after closing, then any following objects are appended to the file (rather than overwriting the file).

Syntax

```
OPEN filename;
    construct construct ...
CLOSE filename;
```

where:

filename is a string of letters or digits which provides the file system name.

4.22 Creating Similar Items (LIKE Keyword)

Purpose

When a new EDD item resembles an existing item, (possibly from an imported file), then the new item may be defined as "like" the first. Selected attributes of the first item may be redefined in the second.

Syntax

```
item-1 LIKE item-type item-2
{
    attribute attribute ...
}
```

where:

item-1 is the new item being defined.

item-2 is the name of a previously defined item. The location of item-2 in the EDD does not matter.

item_type is one of the following types:

- VARIABLE
- MENU
- METHOD
- ITEM_ARRAY
- ARRAY
- COLLECTION
- RECORD
- VARIABLE_LIST
- COMMAND
- CONNECTION
- PROGRAM
- DOMAIN
- RESPONSE_CODES
- BLOCK

attribute is an attribute of the newly defined item. Attributes that differ from those of the previously defined item are described using the appropriate syntax for redefining or deleting item attributes. To redefine an attribute of the item_type the keyword REDEFINE is used as shown in the following example:

```
REDEFINE    LABEL    "new label";
REDEFINE    DEFAULT_VALUE    0;
```

Redefined attributes which are not present in the original definition, are merged. It is possible to redefine all specified attributes for the item_type. See "Item Redefinitions" later in this section.

4.23 Importing Device Descriptions

Purpose

The Electronic Device Description Language constructs previously described in this section are sufficient for describing any single field device. However, additional mechanisms are required to describe multiple revisions of a field device or standard field devices which may be used by various manufacturers to develop compatible field devices.
To provide this type of functionality, one device description must be referenced by another. That is, the developer must be able to import into one device description the items (such as variables and blocks) from another device description. However, simply importing items is not sufficient. The developer must also be able to alter the definitions of the items once they are imported.
With these mechanisms the description of a new revision of a field device can often be specified by simply importing the device description of the old revision of the device, and specifying changes to a few items. This type of device description is sometimes called a delta description, because the entire device description is specified as changes to an existing device description.

Syntax

```
IMPORT MANUFACTURER integer,
DEVICE_TYPE integer,
DEVICE_REVISION integer,
EDD_REVISION integer
{
    import-keywords
    item-redefinitions
}
```

where:

MANUFACTURER integer

DEVICE_TYPE integer

DEVICE_REVISION integer

EDD_REVISION integer specify the device description from which items will be imported. There can be multiple device descriptions for a particular revision of a particular device, in which case the EDD revision distinguishes them.

import-keywords specify which items of the device description are to be imported. Only those items specified by the import keywords are actually imported. The values for the import keywords are described in the following subsection.

item-redefinitions specify how, if at all, the imported items are to be altered. An imported item may be deleted or redefined, or the attributes of an imported item may be deleted or redefined.

4.23.1 Import Keywords

Purpose

An import keyword is specified in the syntax for importing device description, and takes one of three forms, depending on what is imported. Only types and items can be imported which are defined in the imported device description.

- Importing all items, where all items of the external device description are imported. If this form is used, no other import keywords are allowed, that is, if this keyword is used, it is the one and only keyword.

- Importing items of a specified type, where only the items of the specified types are imported. If a specified type is imported, further imports of specific items of the same type are not allowed.
- Importing a specific item.

Syntax

The syntax for the import keywords is as follows:

Importing All Items

```
EVERYTHING;
```

Importing Items of a Specified Type

```
item-type & item-type & ... ;
```

where:

item-type is one of the following keywords:

- VARIABLES
- MENUS
- METHODS
- ITEM_ARRAYS
- ARRAYS
- COLLECTIONS
- RECORDS
- VARIABLE_LISTS
- COMMANDS
- PROGRAMS
- DOMAINS
- RESPONSE_CODES
- BLOCKS
- RELATIONS
- CONNECTIONS

Example

The following import keyword specifies that all the variables, commands, and methods are to be imported. Every item-type may be defined only once:

```
VARIABLES&COMMANDS&METHODS;
```

Importing a Specific Item

```
item-type item-name;
```

where:

item-type is one the keywords listed in the syntax for importing items of a specific type.

item-name is the name assigned to the item when it was defined.

Example

The following import keyword specifies that variable pv_units is imported:

```
VARIABLE pv_units;
```

4.23.2 Item Redefinitions

This subsection describes how imported items can be redefined. Only items and their attributes which are imported may be redefined or deleted. Only items which are deleted or not defined in the imported EDD may be added. Only if the imported device description contains redefined items, it is possible to redefine these items in the EDD.

4.23.2.1 Redefining Imported Blocks

This subsection describes how imported blocks may be redefined.

Deleting Blocks

An imported block may be deleted with:

```
DELETE BLOCK name;
```

Redefining Blocks

An imported block may be redefined with:

```
REDEFINE BLOCK name
{
    attribute attribute ...
}
```

All attributes of the imported block are discarded and replaced with those specified.

Deleting/Redefining Block Attributes

The attributes of an imported block may be deleted or redefined with:

```
BLOCK name
{
    DELETE keyword;
    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a block attribute.

definition is the new definition of the block attribute. The format of the definition depends on the attribute being redefined.

4.23.2.2 Redefining Imported Variables

This subsection describes how imported variables may be redefined.

Deleting Variables

An imported variable may be deleted with:

```
DELETE VARIABLE name;
```

Redefining Variables

An imported variable may be redefined with:

```
REDEFINE VARIABLE name
{
    attribute attribute ...
```



```
}
```

All attributes of the imported variable are discarded and replaced with those specified.

Deleting/Redefining Attributes of Imported Variables

The attributes of an imported variable may be deleted or redefined with:

```
VARIABLE name
{
    DELETE keyword;
    REDEFINE keyword definition
}
```

where:

keyword is one of the keywords that introduces a variable attribute.

definition is the new definition for the variable attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how attributes of imported variables can be altered.

Class Variable Attribute

The class of an imported variable may be redefined with:

```
REDEFINE CLASS class-name & class-name & ... ;
```

Constant Unit Variable Attribute

The constant unit of an imported variable may be deleted with:

```
DELETE CONSTANT_UNIT;
```

The constant unit of an imported variable may be redefined with:

```
REDEFINE CONSTANT_UNIT string;
```

Handling Variable Attribute

The handling of an imported variable may be deleted with:

```
DELETE HANDLING;
```

The handling of an imported variable may be redefined with:

```
REDEFINE HANDLING handling&handling& ...
```

Help Variable Attribute

The help of an imported variable may be deleted with:

```
DELETE HELP;
```

The help of an imported variable may be redefined with:

```
REDEFINE HELP string;
```

Label Variable Attribute

The label of an imported variable may be deleted with:

```
DELETE LABEL;
```

The label of an imported variable may be redefined with:

```
REDEFINE LABEL string;
```

Pre/Post Edit Actions Variable Attributes

The pre/post edit actions of an imported variable may be deleted with:

```
DELETE PRE_EDIT_ACTIONS;
DELETE POST_EDIT_ACTIONS;
```

The pre/post edit actions of an imported variable may be redefined with:

```
REDEFINE PRE_EDIT_ACTIONS
{
    method,method, ...
}

REDEFINE POST_EDIT_ACTIONS
{
    method,method, ...
}
```

Pre/Post Read Actions Variable Attributes

The pre/post read actions of an imported variable may be deleted with:

```
DELETE PRE_READ_ACTIONS;
DELETE POST_READ_ACTIONS;
```

The pre/post read actions of an imported variable may be redefined with:

```
REDEFINE PRE_READ_ACTIONS
{
    method,method, ...
}

REDEFINE POST_READ_ACTIONS
{
    method,method, ...
}
```

Pre/Post Write Actions Variable Attributes

The pre/post write actions of an imported variable may be deleted with:

```
DELETE PRE_WRITE_ACTIONS;
```

The pre/post write actions of an imported variable may be redefined with:

```
DELETE POST_WRITE_ACTIONS;
REDEFINE POST_WRITE_ACTIONS
{
    method,method, ...
}
```

Read/Write Time-outs Variable Attributes

The time-outs of an imported variable may be deleted with:

```
DELETE READ_TIMEOUT;
```

```
DELETE WRITE_TIMEOUT;
```

The time-outs of an imported variable may be redefined with:

```
REDEFINE READ_TIMEOUT expression;
```

```
REDEFINE WRITE_TIMEOUT expression;
```

Deleting/Redefining Arithmetic Options

If the type of an imported variable is arithmetic (integer, unsigned integer, float, or double), the arithmetic options (display/edit formats, scaling factor, min/max values) of the type can be deleted and redefined with:


```
TYPE type
{
    DELETE keyword;

    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords introducing an arithmetic option.

definition is the new definition for the arithmetic option. The format of the definition depends on the option being redefined.

Deleting/Redefining/Adding Enumeration Values

If the type of an imported variable is enumerated (enumerated or bit enumerated), the enumeration values of the type can be deleted, redefined, and extended. Only values which are defined in the imported EDD may be deleted or redefined. Furthermore only values may be added which are not yet defined in the imported EDD. The type of the added value and the imported variable must be the same.

```
TYPE type
{
    DELETE value;

    REDEFINE value-definition;

    ADD value-definition;
}
```

where:

value is one of the values of the imported variable.

value-definition is a definition of an enumeration or bit-enumeration, (see the discussion of "Enumeration Types'' in "Variables'' earlier in this section).

Validity Variable Attribute

The validity of an imported variable may be deleted with:

```
DELETE VALIDITY;
```

The validity of an imported variable may be redefined with:

```
REDEFINE VALIDITY boolean;
```

Response Codes Variable Attribute

The response codes of an imported variable may be deleted with:

```
DELETE RESPONSE_CODES;
```

The response codes of an imported variable may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.3 Redefining Imported Records

This subsection describes how to redefine imported records.

Deleting Records

An imported record may be deleted with:

```
DELETE RECORD name;
```


Redefining Records

An imported record may be redefined with:

```
REDEFINE RECORD name
{
    attribute attribute ...
}
```

All attributes of the imported record are discarded and replaced with those specified.

Deleting/Redefining Record Attributes

The attributes of an imported record may be deleted or redefined with:

```
RECORD name
{
    DELETE   keyword;
    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a record attribute.

definition is the new definition of the record attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how attributes of imported records can be altered.

Members Record Attribute

There are several ways to alter the members of an imported record.

Redefining Members

The record members may be redefined with:

```
REDEFINE MEMBERS
{
    record-member,record-member, ...
}
```

All members of the imported record are discarded and replaced with those specified.

Deleting/Redefining/Adding Members

The members of an imported record can be deleted, redefined, and extended with:

```
MEMBERS
{
    DELETE name;

    REDEFINE record-member;

    ADD record-member;
}
```

Help Record Attribute

The help of an imported record may be deleted with:

```
DELETE HELP;
```

The help of an imported record may be redefined with:

```
REDEFINE HELP string;
```


Label Record Attribute

The label of an imported record may be deleted with:

```
DELETE LABEL;
```

The label of an imported record may be redefined with:

```
REDEFINE LABEL string;
```

Response Codes Record Attribute

The response codes of an imported record may be deleted with:

```
DELETE RESPONSE_CODES ;
```

The response codes of an imported record may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.4 Redefining Imported Item Arrays

This subsection describes how to redefine imported item arrays. The item-types of the imported EDD and

Deleting Item Arrays

An imported item array may be deleted with:

```
DELETE ITEM_ARRAY name;
```

Redefining Item Arrays

An imported item array may be redefined with:

```
REDEFINE ITEM_ARRAY name
{
    attribute attribute ...
}
```

All the attributes of the imported item array are discarded and are replaced with those specified.

Deleting/Redefining Item Array Attributes

The attributes of an imported item array may be deleted or redefined with:

```
ITEM_ARRAY name
{
    DELETE keyword;

    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces an item array attribute.

definition is the new definition of the item array attribute. The format of the definition depends on the attribute being redefined.

The following syntax diagrams show how attributes of imported item arrays can be altered.

Elements Item Array Attribute

There are several ways to alter the elements of an imported item array.

Redefining the Elements

The item array elements may be redefined with:


```
REDEFINE ELEMENTS
{
    item-array-element,item-array-element, ...
}
```

All elements of the imported item array are discarded and replaced with those specified.

Restriction

The type of the elements cannot be changed when redefining an item array. That is, an item array of variables cannot be redefined as an item array of collections.

Deleting/Redefining/Adding Elements

The elements of an imported item array can be deleted, redefined, and extended with:

```
ELEMENTS
{
    DELETE index;
    REDEFINE item-array-element
    ADD item-array-element
}
```

Help Item Array Attribute

The help of an imported item array may be deleted with:

```
DELETE HELP;
```

The help of an imported item array may be redefined with

```
REDEFINE HELP string;
```

Label Item Array Attribute

The label of an imported item array may be deleted with:

```
DELETE LABEL;
```

The label of an imported item array may be redefined with:

```
REDEFINE LABEL string;
```

4.23.2.5 Redefining Imported Menus

This subsection describes how imported menus may be redefined.

Deleting Menus

An imported menu may be deleted with:

```
DELETE MENU name;
```

Redefining Menus

An imported menu may be redefined with:

```
REDEFINE MENU name
{
    attribute attribute ...
}
```

All the attributes of the imported menu are discarded and are replaced with those specified.

Deleting/Redefining Menu Attributes

The attributes of an imported menu may be deleted or redefined with:


```
MENU name
{
    DELETE keyword;
    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords introducing a menu attribute.

definition is the new definition for the menu attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how to alter the attributes of imported menus.

Label Menu Attribute

The label of an imported menu may be redefined with:

```
REDEFINE LABEL string;
```

Items Menu Attribute

The items of an imported menu may be redefined with:

```
REDEFINE ITEMS
{
    menu-item,menu-item, ...
}
```

4.23.2.6 Redefining Imported Methods

This subsection describes how imported methods may be redefined.

Deleting Imported Methods

An imported method may be deleted with:

```
DELETE METHOD name;
```

Redefining Imported Methods

An imported method may be redefined with:

```
REDEFINE METHOD name
{
    attribute attribute ...
}
```

All the attributes of the imported method are discarded and are replaced with those specified.

Deleting/Redefining Attributes of an Imported Method

The attributes of an imported method may be deleted or redefined with:

```
METHOD name
{
    DELETE keyword;

    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a method attribute.

definition is the new definition for the method attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how attributes of imported methods can be altered.

Class Method Attribute

The class of an imported method may be redefined with:

```
REDEFINE CLASS class-name&class-name& ... ;
```

Definition Method Attribute

The definition of an imported method may be redefined with:

```
REDEFINE DEFINITION c-compound-statement
```

Help Method Attribute

The help of an imported method may be deleted with:

```
DELETE HELP;
```

The help of an imported method may be redefined with:

```
REDEFINE HELP string;
```

Label Method Attribute

The label of an imported method may be redefined with:

```
REDEFINE LABEL string;
```

Validity Method Attribute

The validity of an imported method may be deleted with:

```
DELETE VALIDITY;
```

The validity of an imported method may be redefined with:

```
REDEFINE VALIDITY boolean;
```

4.23.2.7 Redefining Imported Relations

This subsection describes how to redefine imported refresh, unit, and write-as-one relations.

Deleting Relations

An imported refresh relation may be deleted with:

```
DELETE REFRESH name;
```

An imported unit relation may be deleted with:

```
DELETE UNIT name;
```

An imported write-as-one relation may be deleted with:

```
DELETE WRITE_AS_ONE name;
```

Redefining Relations

An imported refresh relation may be redefined with:

```
REDEFINE REFRESH name
{
    variable,variable, ...
        : variable, variable, ...
}
```

An imported unit relation may be redefined with:


```
REDEFINE UNIT name
{
    variable:variable,variable, ...
}
```

An imported write-as-one relation may be redefined with:

```
REDEFINE WRITE_AS_ONE name
{
    variable,variable, ...
}
```

4.23.2.8 Redefining Imported Arrays

This subsection describes how imported arrays may be redefined.

Deleting Arrays

An imported array may be deleted with:

```
DELETE ARRAY name;
```

Redefining Arrays

An imported array may be redefined with:

```
REDEFINE ARRAY name
{
    attribute attribute ...
}
```

All the attributes of the imported array are discarded and are replaced with those specified.

Deleting / Redefining Array Attributes

The attributes of an imported array may be deleted or redefined with:

```
ARRAY name
{
    DELETE   keyword;
    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces an array attribute.

definition is the new definition of the array attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how attributes of imported arrays can be altered.

Help Array Attribute

The help of an imported array may be deleted with:

```
DELETE HELP;
```

The help of an imported array may be redefined with:

```
REDEFINE HELP string;
```

Label Array Attribute

The label of an imported array may be deleted with:

```
DELETE LABEL;
```

The label of an imported array may be redefined with:


```
REDEFINE LABEL string;
```

Type Array Attribute

The type of an imported array may be redefined with:

```
REDEFINE TYPE variable-name;
```

Number of Elements Array Attribute

The number of elements of an imported array may be redefined with:

```
REDEFINE NUMBER_OF_ELEMENTS integer-constant;
```

Response Codes Array Attribute

The response codes of an imported array may be deleted with:

```
DELETE RESPONSE_CODES;
```

The response codes of an imported array may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.9 Redefining Imported Collections

This subsection describes how to redefine imported collections.

Deleting Collections

An imported collection may be deleted with:

```
DELETE COLLECTION name;
```

Redefining Collections

An imported collection may be redefined with:

```
REDEFINE COLLECTION name
{
    attribute attribute ...
}
```

All attributes of the imported collection are discarded and are replaced with those specified.

Deleting/Redefining Collection Attributes

The attributes of an imported collection may be deleted or redefined with:

```
COLLECTION name
{
    DELETE keyword;
    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a collection attribute.

definition is the new definition of the collection attribute. The format of the definition depends on the attribute being redefined.

The following syntax specification show how attributes of imported collections can be altered.

Members Collection Attribute

There are several ways to alter the members of an imported collection.

Redefining Members

The collection members may be redefined with:

```
REDEFINE MEMBERS
{
    collection-member,collection-member, ...
}
```

All members of the imported collection are discarded and replaced with those specified.

Restriction

The type of the members cannot be changed when redefining a collection, that is, a collection of variables cannot be redefined as a collection of item arrays.

Deleting/Redefining/Adding Members

The members of an imported collection can be deleted, redefined, and extended with:

```
MEMBERS
{
    DELETE name;

    REDEFINE collection-member;

    ADD collection-member
}
```

Help Collection Attribute

The help of an imported collection may be deleted with:

```
DELETE HELP;
```

The help of an imported collection may be redefined with:

```
REDEFINE HELP string;
```

Label Collection Attribute

The label of an imported collection may be deleted with:

```
DELETE LABEL;
```

The label of an imported collection may be redefined with:

```
REDEFINE LABEL string;
```

4.23.2.10 Redefining Imported Variable Lists

This subsection describes how imported variable lists may be redefined.

Deleting Variable Lists

An imported variable list may be deleted with:

```
DELETE VARIABLE_LIST name;
```

Redefining Variable Lists

An imported variable list may be redefined with:

```
REDEFINE VARIABLE_LIST name
{
    attribute attribute ...
}
```


All the attributes of the imported variable list are discarded and are replaced with those specified.

Deleting/Redefining Variable List Attributes

The attributes of an imported variable list may be deleted or redefined with:

```
VARIABLE_LIST name
{
    DELETE keyword;
    REDEFINE keyword definition
}
```

where:

keyword is one of the keywords that introduces an variable list attribute.

definition is the new definition of the variable list attribute. The format of the definition depends on the attribute being redefined.

The following syntax specifications show how the attributes of imported variable lists can be altered.

Members Variable List Attribute

There are several ways to alter the members of an imported variable list.

Redefining Members

The variable list members may be redefined with:

```
REDEFINE MEMBERS
{
    variable-list-member,record-member ,...
}
```

All members of the imported variable list are discarded and replaced with those specified.

Deleting/Redefining/Adding Members

The members of an imported variable list can be deleted, redefined, and extended with:

```
MEMBERS
{
    DELETE name ;

    REDEFINE variable-list-member

    ADD variable-list-member
}
```

Help Variable List Attribute

The help of an imported variable list may be deleted with:

```
DELETE HELP;
```

The help of an imported variable list may be redefined with:

```
REDEFINE HELP string ;
```

Label Variable List Attribute

The label of an imported variable list may be deleted with:

```
DELETE LABEL;
```

The label of an imported variable list may be redefined with:

```
REDEFINE LABEL string;
```


Response Codes Variable List Attribute

The response codes of an imported variable list may be deleted with:

```
DELETE RESPONSE_CODES;
```

The response codes of an imported variable list may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.11 Redefining Imported Programs

This subsection describes how to redefine an imported program.

Deleting Programs

An imported program may be deleted with:

```
DELETE PROGRAM name;
```

Redefining Programs

An imported program may be redefined with:

```
REDEFINE PROGRAM name
{
    attribute attribute ...
}
```

All the attributes of the imported program are discarded and are replaced with those specified.

Deleting/Redefining Program Attributes

The attributes of an imported program may be deleted or redefined with:

```
PROGRAM name
{
    DELETE keyword;

    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a program attribute.

definition is the new definition of the program attribute. The format of the definition depends on the attribute being redefined.

Arguments Program Attribute

There are several ways to alter the arguments of an imported program. The program arguments may be redefined with:

```
REDEFINE ARGUMENTS
{
    data-item,data-item,...
}
```

All arguments of the imported program are discarded and replaced with those specified. The arguments of an imported program can be deleted, redefined, and extended with:

```
ARGUMENTS
{
    DELETE data-item;

    REDEFINE data-item;
```



```
    ADD data-item;
}
```

Response Codes Program Attribute

The response codes of an imported program may be deleted with:

```
DELETE RESPONSE_CODES;
```

The response codes of an imported program may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.12 Redefining Imported Domains

This subsection describes how to redefine an imported domain.

Deleting Domains

An imported domain may be deleted with:

```
DELETE DOMAIN name;
```

Redefining Domains

An imported domain may be redefined with:

```
REDEFINE DOMAIN name
{
    attribute attribute ...
}
```

All attributes of the imported domain are discarded and replaced with those specified.

Deleting/Redefining Domain Attributes

The attributes of an imported domain may be deleted or redefined with:

```
DOMAIN name
{
    DELETE keyword;

    REDEFINE keyword definition;
}
```

where:

keyword is one of the keywords that introduces a domain attribute.

definition is the new definition of the domain attribute. The format of the definition depends on the attribute being redefined.

Handling Domain Attribute

The handling of an imported domain may be deleted with:

```
DELETE HANDLING;
```

The handling of an imported domain may be redefined with:

```
REDEFINE HANDLING handling&handling& ...;
```

Response Codes Domain Attribute

The response codes of an imported domain may be deleted with:

```
DELETE RESPONSE_CODES;
```


The response codes of an imported domain may be redefined with:

```
REDEFINE RESPONSE_CODES response-code-name;
```

4.23.2.13 Redefining Imported Response Codes

This subsection describes how to redefine imported response codes.

Deleting Response Codes

Imported response codes may be deleted with:

```
DELETE RESPONSE_CODES name;
```

Redefining Response Codes

Imported response codes may be redefined with:

```
REDEFINE RESPONSE_CODES name
{
    value,type,description,help;
    value,type,description,help;
}
```

Deleting/Redefining/Adding Response Codes

The response codes can be deleted, redefined, and extended with:

```
RESPONSE_CODES name
{
    DELETE value;
    REDEFINE value,type,description,help;
    ADD value,type,description,help;
}
```


4.24 Preprocessor Directives

Purpose

The EDDL has a number of control line commands which initiate the compiler to include files, do macro substitutions, and do conditional compilations. Preprocessing directives are lines in the EDD which start with #. The # is followed by an identifier, "the "directive name". Whitespace is also allowed before and after the # . If a #define is sufficiently long to warrant continuation on the next line, the backslash \ may be used to continue the definition.

4.24.1 Header Files

Purpose

A header file is a file containing declarations and macro definitions to be shared between several source files. Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. They are included in the EDD to supply the definitions and declarations needed to invoke system calls and libraries.
- Own header files contain declarations for interfaces between the source files of the EDD.

Each time having a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good possibility to create a header file for them.

Syntax

```
#include <FILE>
#include "FILE"
```

where:

<FILE> is the variant used for system header files. You specify directories to search for header files with the command option.

"FILE" is the variant used for header files of own programs.

Both user and system header files are included in using the preprocessing directive.

4.24.2 Macros

Purpose

A macro is a sort of abbreviation which can be defined once and then be used later for several times. #define is the directive that defines a macro. There exist several possibilities how to use the #define preprocessor.

Syntax

```
#define DefSymb
#define DefSymb NewSymb
##define DefMacro(Var1, Var2
```

where:

DefSymb defines a symbol. The existence of the symbol is checked with

#ifdef name If the symbolic name is defined (e.g. #define NAME or #define NAME Smith), from this point the followed code up to #endif is compiled.

#ifndef name If the symbolic name is not defined, the followed code up to #endif is compiled.

NewSymb is the substitute for DefSymb wherever it finds DefSymb in the EDD.

DefMacro is the name of the function with the parameters Var1 and Var2. For further information see the example in the appendix.

4.25 Conditional Expressions

Purpose

Conditional expressions are commands to declare parts of the EDD as valid or invalid during execution time. These decision depends on the values of a variable.

4.25.1 If Conditional

Purpose

An if conditional is used for specifying an attribute that has two alternative definitions. The expression specified in the if conditional is evaluated. If the result is non-zero, the attribute is specified by a then clause; otherwise, it is specified by an else clause.

Syntax

```
IF ( expression )
{
    then-clause
}
ELSE
{
    else-clause
}
```

where:

expression is the which is evaluated to determine whether the then-clause or else-clause is used to define the attribute.

then-clause is the definition for the attribute if the value of expression is non-zero. The clause structure depends on the attribute being defined. It can also take the form of another conditional.

else-clause is the definition for the attribute if the value of expression is zero. The clause structure depends on the attribute being defined. It can also take the form of another conditional.

4.25.2 Select Conditional

Purpose

A select conditional is used for specifying an attribute that has several alternative definitions. The expression in parentheses is evaluated. Then each expression following a CASE is evaluated, in order, until one evaluates to the same value as the controlling expression. If a match is found, the attribute is specified by the clause following the matching expression; otherwise, it is specified by the clause following DEFAULT.

Syntax

```
SELECT ( expression )
{
    CASE expression:
        clause
```



```
    CASE expression:
        clause
    DEFAULT:
        clause
}
```

where:

(expression) is the controlling expression against which expressions in the alternative CASE structures are evaluated.

expression is an expression which is matched against the expression in parentheses.

clause is the definition for the attribute for each case, and the default definition. The structure of each clause depends on the attribute being defined. Regardless of the attribute, each clause can also take the form of another conditional.

4.26 References

Purpose

References are used throughout a device description by items to refer to other items. For example, the pre-edit actions of a variable refers to methods defined elsewhere in the device description.
The following subsections describe conventions and syntax for references in a device description.

4.26.1 Referencing Items

The most common type of reference is simply the name of an item. A simple reference is expressed as follows:

```
item-name
```

4.26.2 Referencing Elements of a Record

The elements of a record may be referenced as:

```
record-name . member-name
```

where:

record-name is the name of a record.

member-name is one of the names associated with the elements of the record.

4.26.3 Referencing Elements Of An Array

The elements of an array may be referenced as:

```
array-name [ expression ]
```

where:

array-name is the name of an array.

4.26.4 Referencing Members of a Collection

The member of a collection may be referenced as:

```
collection-reference . member-name
```

where:

collection-reference is a reference to a collection. This reference need not be the name of a collection, only a reference to a collection. That is, collection references can be nested.

member-name is one of the names associated with the members of the collection.

4.26.5 Referencing Elements of an Item Array

The elements of an item array may be referenced as:

```
item-array-reference [ expression ]
```

where:

item-array-reference is a reference to an item array. The item-array-reference need not be the name of an item-array, only a reference to an item array, that is, item array references can be nested.

4.26.6 Referencing Members of a Variable List

The members of a variable list may be referenced as:

```
variable-list-name . member-name
```

where:

variable-list-name is the name of a variable list.

member-name is one of the names associated with the members of the variable list.

When a Variable List Member is a Record

If a member of a variable list is a record, the elements of the record may be referenced as:

```
variable-list-name . member-name . record-member
```

where:

variable-list-name is the name of a variable list.

member name is one of the names associated with the members of the variable list.

record-member is one of the names associated with the members of the record specified by member-name.

When a Variable List Member is an Array

If a member of a variable-list is an array, the element of the array may be referenced as:

```
variable-list-name . member-name [ expression ]
```

where:

variable-list-name is the name of a variable list.

member-name is one of the names associated with the members of the variable list.

4.27 Expressions

Purpose

An expression specifies the computation of a numeric value. There are three types of expressions:

1. Primary Expressions
2. Unary Expressions
3. Binary Expressions

4.27.1 Primary Expressions

Table 4 summarizes the primary expressions in the EDDL.

Primary Expression	Description
Constant	An expression whose value is identical to the value of the constant.
Parenthesized expression	An expression whose value is identical to the value of the enclosed expression.
Variable reference	An expression whose value is the value of the referenced variable. Because a variable reference may require a value read from a device, use variable references with care.

Minimum and maximum values of device variables	These primary expressions take the following form:
	`variable-name.MIN_VALUE`
	`variable-name.MAX_VALUE`
	For example, the following expression specifies the maximum value of the variable upper_range_value:
	`upper_range_value.MAX_VALUE`

Table 5: Primary Expressions

4.27.2 Unary Expressions

A unary expression consists of an operand, an expression, preceded by a unary operator. Table 5 describes the unary expressions in the EDDL.

Unary Expression	What It Specifies
-	The arithmetic negation of its operand.
~	The bitwise negation of its operand, that is, each bit of the result is the inverse of the corresponding bit of the operand. The operand of the ~ operator must have an integral value.
!	The logical negation of its operand.

Table 6:Unary Expressions

4.27.3 Binary Expressions

A binary expression consists of two operands or expressions, separated by a binary operator. If either operand has a floating point value, the other operand is converted (promoted) to a floating point value. This subsection describes the following types of binary expressions:

- Multiplicative
- Additive
- Shift
- Relational

- Equality
- Bitwise AND (&)
- Bitwise XOR (~)
- Bitwise OR (|)
- Logical AND (&&)
- Logical OR (||)

4.27.3.1 Multiplicative Operators

Multiplicative operators specify multiplication and division of operands. Table 6 describes the multiplicative operators.

Operator	What It Does
*	Specifies the multiplication of its operands.
/and%	Specifies the division of the first operand by the second operand. The result of the / operator is the quotient of the division. The result of the % operator is the remainder.

Table 7: Multiplicative Operators

4.27.3.2 Additive Operators

Additive operators specify the addition and subtraction of operands. Table 7 describes additive operators.

Operator	What It Does
+	Specifies the addition of its operands.
-	Specifies the subtraction of the second operand from the first.

Table 8: Additive Operators

4.27.3.3 Shift Operators

The << and >> operators specify a shift of the first operand by the number of bits specified by the second operand. The operands of the << and >> operators must have integral values. Table 8 describes the shift operators.

Operator	What It Does
<<	Shifts the first operand to the left. The bits shifted off are discarded, and the vacated bits are zero filled.
>>	Shifts the first operand to the right. The bits shifted off are discarded. If the first operand is less than 0, the vacated bits are one filled; otherwise they are zero filled.

Table 9: Shift Operators

4.27.3.4 Relational Operators

Relational operators (<, <=, >, >=) specify a comparison of its operands. The result of this type of an expression is 1 if the tested relationship is true, otherwise the result is 0. Table 9 describes the relational operators.

Operator	What It Does
<	Tests for the relationship „less than“.
<=	Tests for the relationship „less than or equal“.
>	Tests for the relationship „greater than“.
>=	Tests for the relationship „greater than or equal“.

Table 10: Relational Operators

4.27.3.5 Equality Operators

The equality operators are == and !=. The result of this type of an expression is 1 if the tested relationship is true, otherwise the result is 0. Table 10 describes the equality operators.

Operator	What It Does
==	Tests for the relationship „equals“.
!=	Tests for the relationship „does not equal“.

Table 11: Equality Operators

4.27.3.6 Bitwise AND Operator (&)

The & operator specifies the bitwise AND of its operands, that is, each bit of the result is set if each of the corresponding bits of the operands is set. The operands of the & operator must have integral values.

4.27.3.7 Bitwise XOR Operator (~)

The ~ operator specifies the bitwise exclusive OR of its operands, that is, each bit of the result is set if only one of the corresponding bits of the operands is set. The operands of the ~ operator must have integral values.

4.27.3.8 Bitwise OR Operator (|)

The | operator specifies the bitwise inclusive OR of its operands, that is, each bit of the result is set if either of the corresponding bits of the operands is set. The operands of the | operator must have integral values.

4.27.3.9 Logical AND Operator (&&)

The && operator specifies the boolean AND evaluation of its operands. The result of this type of expression is 1 if both of the operands are not equal to 0, otherwise the result is 0. If the first operand is equal to 0, the second operand is not evaluated.

4.27.3.10 Logical OR Operator (||)

The || operator specifies the boolean OR evaluation of its operands. The result of this type of expression is 1 if either of the operands is not equal to 0, otherwise the result is 0. If the first operand is not equal to 0, the second operand is not evaluated.

4.28 Strings

There are several ways to specify a string:

- As a string literal
- As an enumeration value string

- As a string variable
- As a dictionary reference

4.28.1 Specifying a String as a String Literal

A text string can be specified as a string literal (see "String Literals" in "Lexical Conventions" later in this section). Adjacent string literals are concatenated to form a single string literal.

4.28.2 Specifying a String as a String Variable

A string specified as a string variable is the value of the string variable.

4.28.3 Specifying a String as a Enumeration Value

Purpose

An enumeration value string is a string associated with one of the values of an enumeration variable.

Syntax

```
name ( value )
```

where:

name is the name of an enumeration variable and its values.

Example

The following enumeration value string specifies the string associated with the value 4 of the variable units_code:

```
units_code ( 4 )
```

4.28.4 Specifying a String as a Dictionary Reference

Purpose

A dictionary reference specifies a string in the standard text dictionary.

Syntax

```
[ name ]
```

where:

name is the name of a string in the standard text dictionary (see "Standard Dictionary" later in this section).

Example

The following dictionary reference specifies the string associated with the name invalid_selection in the standard text dictionary:

```
[ invalid_selection ]
```


4.29 Lexical Conventions

This section describes the lexical conventions of the language.

4.29.1 Integer Constants

An integer constant may be specified in binary, octal, decimal, or hexadecimal notation. Table 11 shows the conventions for each type of notation.

Integer ConstantType	Conventions
Binary	A non-empty sequence of the binary digits 0 and 1 preceded by either 0b or 0B.
Octal	A non-empty sequence of the digits 0 through 7 beginning with a 0.
Decimal	A non-empty sequence of the decimal digits 0 through 9, not beginning with 0.
Hexadecimal	A non-empty sequence of hexadecimal digits preceded by either 0x or 0X. The hexadecimal digits are the digits 0 through 9 and the letters a through f (or A through F) with the values 10 through 15, respectively.

Table 12: Lexical Conventions for Integer Constants

4.29.2 Floating Point Constants

Syntax

A floating point constant has four parts:

1. An integer part, a sequence of decimal digits.
2. A decimal point (.).
3. A fraction part, a sequence of decimal digits.
4. An exponent part, a possibly signed sequence of decimal digits preceded by one of the letters e or E.

Rules The following rules apply to using floating point constants:

- Either the integer part or the fraction part can be omitted, but not both.
- Either the decimal point or the exponent part can be omitted, but not both.

Example Following are examples of floating point constants:

```
59.
.87
48.93
4.8e12
```

4.29.3 String Literals

A string literal is a possibly empty sequence of characters enclosed in double quotes ("). The enclosed characters can be any ISO Latin–1 (ISO 8859–1) character except the following:

- Double quote (")
- Backslash (\)
- New line

Using Escape Sequences in String Literals

A string constant can also contain escape sequences that represent an ISO Latin-1 character. Table 12 shows the escape sequences and their results.

Escape Code	Result
'	Single quote
"	Double quote
|	Vertical bar
?	Question mark
\	Backslash
\a	Alert
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Table 13: Using Escape Sequences in String Literals

4.29.4 Using Language Codes in String Constants

There is another escape sequence called a language code. A language code consists of a vertical bar (|) followed by three decimal digits. The three digits are the same as the telephone country code.
This language code escape sequence specifies the language of the string up to the next language code. Therefore, a string literal can encapsulate all the translations of a given phrase. A string literal containing no language code is an English string. If a string literal does not contain translations for all the languages, English will be used for the unspecified languages. The table language shows the language codes that can be used in string literals.

Language Code	Language
	English
|de|	German
|fr|	French
|it|	Italian
|sp|	Spanish

Table 14: Using Language Codes in String Literals

Example The following string literal specifies the English phrase "Invalid Selection" in English and German:

```
"Invalid Selection"
"|de|Unzulässige Auswahl"
```

4.30 Standard Text Dictionary

The standard text dictionary provides a standard vocabulary for describing field devices. The dictionary is a collection of standard text strings that can be used in device descriptions.
The standard text dictionary provides the following advantages:

- The standard dictionary specifies each of the standard text strings in each of the supported languages. This provides motivation for field device developers to use the text dictionary. If field device developers use the standard text dictionary, they need not translate their text strings to any foreign languages because the text dictionary contains definitions of the text strings for all the supported languages.

- If field device developers make extensive use of the dictionary, which they will if the dictionary is complete enough, a degree of consistency across different product lines will be created. This consistency, due to the common vocabulary, will be especially apparent across similar types of field devices.
  For example, many pressure transmitters, temperature transmitters, and flow meters will use the same terminology and therefore appear similar to the users. This consistency is accomplished because a common vocabulary is used by all field device developers.

Form of the Standard Text Dictionary

The standard text dictionary takes the form of a text file. The text file consists of phrase definitions and comments. Each phrase definition is made up of three or more fields, separated by commas. The following example shows an dictionary entry:

```
[0,0] cb_time
      "Check-Back Time"
      "|de|Rückmeldezeit"
```


5 EDDL Method Built-ins Library

This appendix describes the library of built-in functions that are available to be used within EDDL methods.

5.1 ABORT_ON_ALL_COMM_STATUS

Syntax

```
void ABORT_ON_ALL_COMM_STATUS()
```

Purpose

ABORT_ON_ALL_COMM_STATUS will set all of the bits in the comm status abort mask. This will cause the system to abort the current method if the device returns any comm status value.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, RETRY_ON_COMM_STATUS, IGNORE_COMM_STATUS, RETRY_ON_ALL_COMM_STATUS, IGNORE_ALL_COMM_STATUS.

5.2 ABORT_ON_ALL_RESPONSE_CODES

Syntax

```
void ABORT_ON_ALL_RESPONSE_CODES()
```

Purpose

ABORT_ON_ALL_RESPONSE_CODES will set all of the bits in the response code abort mask. This will cause the system to abort the current method if the device returns any response code value.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_RESPONSE_CODE, RETRY_ON_RESPONSE_CODE, IGNORE_RESPONSE_CODE, RETRY_ON_ALL_RESPONSE_CODES, IGNORE_ALL_RESPONSE_CODES.

5.3 ABORT_ON_COMM_STATUS

Syntax

```
void ABORT_ON_COMM_STATUS(comm_status)
int  comm_status;
```

Purpose

ABORT_ON_COMM_STATUS will set the correct bit(s) in the comm status abort mask such that the specified comm status value will cause the method to abort.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
RETRY_ON_COMM_STATUS, IGNORE_COMM_STATUS, ABORT_ON_ALL_COMM_STATUS, RETRY_ON_ALL_COMM_STATUS, IGNORE_ALL_COMM_STATUS.

5.4 ABORT_ON_NO_DEVICE

Syntax

```
void ABORT_ON_NO_DEVICE()
```

Purpose

ABORT_ON_NO_DEVICE will set the no devices mask such that the method will be aborted if no device is found while sending a transaction.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
RETRY_ON_NO_DEVICE, IGNORE_NO_DEVICE.

5.5 ABORT_ON_RESPONSE_CODE

Syntax

```
void ABORT_ON_RESPONSE_CODE(response_code)
int  response_code;
```

Purpose

ABORT_ON_RESPONSE_CODE will set the correct bit(s) in the response code abort mask such that the specified response code value will cause the method to abort.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method.
See also:
RETRY_ON_RESPONSE_CODE, IGNORE_RESPONSE_CODE,
ABORT_ON_ALL_RESPONSE_CODES,
RETRY_ON_ALL_RESPONSE_CODES, IGNORE_ALL_RESPONSE_CODES.

5.6 DELAY

Syntax

```
void DELAY(delay_time, prompt)
int  delay_time;
char *prompt;
```

Purpose

DELAY displays the prompt and pauses for the specified number of seconds. The prompt may contain local variable values (see put_message for syntax). The delay time must be a positive number.
See also:
delay, DELAY_TIME.

5.7 DELAY_TIME

Syntax

```
void DELAY_TIME(delay_time)
int delay_time;
```

Purpose

DELAY_TIME pauses for the specified number of seconds. The delay time must be a positive number.

See also:
delay, DELAY.

5.8 IGNORE_ALL_COMM_STATUS

Syntax

```
void IGNORE_ALL_COMM_STATUS()
```

Purpose

IGNORE_ALL_COMM_STATUS will clear all of the bits in the comm status retry and abort masks. This will cause the system to ignore all bits in the comm status value.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, RETRY_ON_COMM_STATUS, IGNORE_COMM_STATUS, ABORT_ON_ALL_COMM_STATUS, RETRY_ON_ALL_COMM_STATUS.

5.9 IGNORE_ALL_RESPONSE_CODES

Syntax

```
void IGNORE_ALL_RESPONSE_CODES()
```

Purpose

IGNORE_ALL_RESPONSE_CODES will clear all of the bits in the response code retry and abort masks. This will cause the system to ignore all response code values returned from the device.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_RESPONSE_CODE, RETRY_ON_RESPONSE_CODE, IGNORE_RESPONSE_CODE, ABORT_ON_ALL_RESPONSE_CODES, RETRY_ON_ALL_RESPONSE_CODES.

5.10 IGNORE_COMM_STATUS

Syntax

```
void IGNORE_COMM_STATUS(comm_status)
int  comm_status;
```

Purpose

IGNORE_COMM_STATUS will clear the correct bit(s) in the comm status abort and retry mask such that the specified bits in the comm status value will be ignored.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, RETRY_ON_COMM_STATUS, ABORT_ON_ALL_COMM_STATUS, RETRY_ON_ALL_COMM_STATUS, IGNORE_ALL_COMM_STATUS.

5.11 IGNORE_NO_DEVICE

Syntax

```
void IGNORE_NO_DEVICE()
```

Purpose

IGNORE_NO_DEVICE will set the no device mask to show that the no device condition should be ignored while sending a transaction.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, RETRY_ON_NO_DEVICE.

5.12 IGNORE_RESPONSE_CODE

Syntax

```
void IGNORE_RESPONSE_CODE(response_code)
int  response_code;
```

Purpose

IGNORE_RESPONSE_CODE will clear the correct bit(s) in the response code masks such that the specified response code value will be ignored.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_RESPONSE_CODE, RETRY_ON_RESPONSE_CODE,
ABORT_ON_ALL_RESPONSE_CODES,
RETRY_ON_ALL_RESPONSE_CODES, IGNORE_ALL_RESPONSE_CODES.

5.13 METHODID

Syntax

```
int  METHODID(method_name)
char *method_name;
```

Purpose

METHODID will return the identifier for the method specified. A valid method name must be provided. Each method in the device description is assigned a unique identifier. This routine is used when the identifier of a method needs to be passed to the abort processing built-in functions.
Will return method identifier.

5.14 PROGID

Syntax

```
int  PROGID(progname)
char *progname;
```

Purpose

PROGID will return the identifier for the program specified. A valid program name must be provided. The ID needs to be saved in a temporary buffer for use as a parameter to another built-

in.
PROGID will return program identifier.

5.15 RETRY_ON_ALL_COMM _STATUS

Syntax

```
void RETRY_ON_ALL_COMM_STATUS()
```

Purpose

RETRY_ON_ALL_COMM_STATUS will set all of the bits in the comm status retry mask.This will cause the system to retry the current transaction if the device returns any comm status value.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, RETRY_ON_COMM_STATUS, IGNORE_COMM_STATUS, ABORT_ON_ALL_COMM_STATUS, IGNORE_ALL_COMM_STATUS.

5.16 RETRY_ON_ALL_RESPONSE_CODES

Syntax

```
void RETRY_ON_ALL_RESPONSE_CODES()
```

Purpose

RETRY_ON_ALL_RESPONSE_CODE will set all of the bits in the response code retry mask. This will cause the system to retry the current transaction if the device returns any response code value.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_RESPONSE_CODE, RETRY_ON_RESPONSE_CODE, IGNORE_RESPONSE_CODE, ABORT_ON_ALL_RESPONSE_CODES, IGNORE_ALL_RESPONSE_CODES.

5.17 RETRY_ON_COMM_STATUS

Syntax

```
void RETRY_ON_COMM_STATUS(comm_status)
int  comm_status;
```

Purpose

RETRY_ON_COMM_STATUS will set the correct bit(s) in the comm status retry mask such that the specified comm status value will cause the current transaction to be retried.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, IGNORE_COMM_STATUS, ABORT_ON_ALL_COMM_STATUS, RETRY_ON_ALL_COMM_STATUS, IGNORE_ALL_COMM_STATUS.

5.18 RETRY_ON_NO_DEVICE

Syntax

```
void RETRY_ON_NO_DEVICE()
```

Purpose

RETRY_ON_NO_DEVICE will set the no device mask such that the current transaction will be retried if no device is found while sending a transaction.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_COMM_STATUS, IGNORE_NO_DEVICE.

5.19 RETRY_ON_RESPONSE_CODE

Syntax

```
void RETRY_ON_RESPONSE_CODE(response_code)
int  response_code;
```

Purpose

RETRY_ON_RESPONSE_CODE will set the correct bit(s) in the response code retry mask such that the specified response code value will cause the current transaction to be retried.
The retry and abort masks are reset to their default values at the start of each method, so the new mask value will only be valid during the current method. See ABORT_ON_RESPONSE_CODE for default values.
See also:
ABORT_ON_RESPONSE_CODE, IGNORE_RESPONSE_CODE,
ABORT_ON_ALL_RESPONSE_CODES,
RETRY_ON_ALL_RESPONSE_CODES, IGNORE_ALL_RESPONSE_CODES.

5.20 VARID

Syntax

```
int  VARID(variable_name)
char *variable_name;
```

Purpose

VARID will return the identifier for the variable specified. A valid variable name must be provided. Each variable in the device description is assigned a unique identifier. This routine is to be used when the identifier of a variable either needs to be stored in a temporary buffer, or needs to be sent as a parameter to another built-in.
Will return variable identifier.

5.21 abort

Syntax

```
void abort()
```

Purpose

abort will display a message indicating that the method has been aborted and wait for acknowledgment from the user. Once acknowledgment has been made, the system will execute any abort methods in the abort method list, and will exit the method process.

See also:
add_abort_method(), remove_abort_method() remove_all_abort_methods(), process_abort().

5.22 acknowledge

Syntax

```
int  acknowledge(prompt)
char *prompt;
```

Purpose

acknowledge will display the prompt and wait for the enter key to be pressed. Will return the key pressed to exit the transaction.

5.23 add_abort_method

Syntax

```
int  add_abort_method(abort_method_name)
char *abort_method_name;
```

Purpose

add_abort_method will add a method to the abort method list, which is the list of methods to be executed if the current method is aborted. The abort method list can hold up to twenty methods at any one time. The methods are run in the order they are added to the list, and the same method may be added to the list more than once. The list is cleared after each method is executed.
It is important to note that the abort methods are only executed when the method is aborted, and not when you exit the method, under normal operating conditions. Methods can be aborted due to an abort mask condition when sending a transaction, or when the abort built-in is called.
Will return TRUE if the method was successfully added to the list, and FALSE if the list was full.
See also:
abort() remove_abort_method(), process_abort().

5.24 assign_str

Syntax

```
void assign_str(device_var, new_string)
char *decice_var;
char *new_string;
```

Purpose

assign_str will assign the specified string to the device variable. The variable must be valid. If neccessary, the value is casted to the type of the referenced variable.

5.25 delay

Syntax

```
void delay(delay_time, prompt, global_var_ids)
int  delay_time;
char *prompt;
int  *global_var_ids;
```

Purpose

Will display the prompt and pause for the specified number of seconds. The prompt may contain local and/or device variable values (see put_message for syntax). The delay time must be a

positive number.
See also:
DELAY, DELAY_TIME.

5.26 display

Syntax

```
void display(prompt, global_var_ids)
char *prompt;
int  *global_var_ids;
```

Purpose

This routine will display the specified message on the screen, continuously updating the dynamic variable values used in the string (see put_message for syntax). This updating will continue until the enter key is pressed.

5.27 display_comm_status

Syntax

```
void display_comm_status(comm_status_value)
int  comm_status_value;
```

Purpose

Display_comm_status will display the string associated with the specified value of the comm_status byte.
See also:
display_response_status.

5.28 display_response_status

Syntax

```
void display_response_status(response_code_value)
int  response_code_value;
```

Purpose

Display_response_status will display the string associated with the specified value of the response_code byte.
See also:
display_comm_status.

5.29 fassign

Syntax

```
int  fassign(target_var_id, new_value)
char *target_var_id; double new_value;
```

Purpose

Will assign the value specified to the target variable. The variable must be valid, and must reference a variable of type float.
Will return TRUE if the assignment was successful, and FALSE if the variable identifier was invalid.
See also:
VARID, vassign.

5.30 fvar_value

Syntax

```
double fvar_value(source_var_name)
char   *source_var_name;
```

Purpose

Will return the value of the specified variable. The variable must be valid and of type float. Will return the value of the variable specified. See also:
ivar_value, lvar_value.

5.31 get_dev_var_value

Syntax

```
int  get_dev_var_value(prompt, device_var_name)
char *prompt;
char *device_var_name;
```

Purpose

get_dev_var_value will display the specified prompt message, and allow the user to edit the value of a device variable. If the device variable is dynamic, the value will be continuously updated until a new value is entered. The edited copy of the device variable value will be updated when the new value is entered, but will not be sent to the device. This must be done explicitly using one of the send transaction routines.
The prompt may NOT contain embedded local and/or device variable values. Will return BI_SUCCESS if the variable was successfully modified, BI_ABORT if the routine was aborted, and BI_ERROR if an error occurred entering the new value or accessing the specified variable.
See also:
get_local_var_value.

5.32 get_dictionary_string

Syntax

```
int  get_dictionary_string(dict_string_name, string, max_str_len)
char *dict_string_name;
char *string;
char *max_str_len;
```

Purpose

get_dictionary_string will retrieve the dictionary string associated with the given name in the current language. If the string is not available in the current language, the English string will be retrieved. If the string is not defined in either language, an error condition occurs, and the routine will return FALSE. If the string is longer than the max_str_len, the string will be truncated. Will return TRUE if successful, FALSE if string could not be found.

5.33 get_local_var_value

Syntax

```
int  get_local_var_value(prompt, local_var_name)
char *prompt;
char *local_var_name;
```

Purpose

get_local_var_value will display the specified prompt message, and allow the user to edit the value of a local variable.
The prompt may NOT contain embedded local and/or device variable values.
Will return BI_SUCCESS if the variable was successfully modified, and BI_ERROR if an error occurred entering the new value or accessing the specified variable.
See also:
get_dev_var_value.

5.34 get_status_code_string

Syntax

```
void get_status_code_string(var_name, status_code, status_string,
status_string_length)
char *var_name;
int  status_code;
char *status_string;
int  status_string_length;
```

Purpose

Will return the status code string for the variable and status code specified. If the string is longer than the maximum length defined in status_string_length, the string is truncated. The variable identifier supplied must be valid, and the status code specified must be valid for that variable.

5.35 GET_TICK_COUNT

Syntax

```
long GET_TICK_COUNT()
```

Purpose

returns the time in milliseconds since the last system boot. It can be used for timestamps.
ATTENTION: In order to the Datatype long, the returnvalue of GET_TICK_COUNT will wrap around and start from zero after a period of 49,71026961806 days.

5.36 ivar_value

Syntax

```
int  ivar_value(source_var_name)
char *source_var_name;
```

Purpose

Will return the value of the specified variable. The variable identifier must be valid and of type integer.
Will return the value of the variable specified.
See also:
fvar_value, lvar_value.

5.37 lvar_value

Syntax

```
int  lvar_value(source_var_name)
char *source_var_name;
```

Purpose

Will return the value of the specified variable. The variable identifier must be valid and of type long.
Will return the value of the variable specified.
See also:
ivar_value, fvar_value.

5.38 process_abort

Syntax

```
void process_abort()
```

Purpose

process_abort will abort the current method, running any abort methods which are in the abort method list. Unlike the abort transaction, no message will be displayed when this routine is executed. This built-in transaction may not be run from inside an abort method.
See also:
abort, add_abort_method, remove_abort_method, remove_all_abort_methods.

5.39 put_message

Syntax

```
void put_message(message)
char *message;
```

Purpose

put_message will display the specified message on the screen.
Embedded device variables are NOT supported in this transaction.

5.40 ReadCommand

Syntax

```
void ReadCommand(name)
```

Purpose

ReadCommand reads the variables defined in the COMMAND name.

5.41 remove_abort_method

Syntax

```
int  remove_abort_method(abort_method_name)
char *abort_method_name;
```

Purpose

remove_abort_method will remove a method from the abort method list, which is the list of methods to be executed if the current method is aborted. This transaction will remove the first occurrence of the specified method in the list, starting with the first method added. If there are multiple occurrences of a specific method, only the first one is removed. Abort methods may not be removed during an abort method.
will return TRUE if the method was successfully removed from the list, and FALSE if either the method was not in the list or if this transaction was run during an abort method.
See also:
abort, add_abort_method, remove_abort_method, process_abort.

5.42 remove_all_abort_methods

Syntax

```
void remove_all_abort_methods()
```

Purpose

remove_all_abort_methods will remove all entries in the abort method list, including multiple entries for the same method. This transaction may not be run from an abort method.
See also:
abort, add_abort_method, remove_abort_method, process_abort.

5.43 rspcode_string

Syntax

```
void rspcode_string(transaction, response_code, response_string,
response_string_length)
int  transaction;
int  response_code;
char *response_string;
int  response_string_length;
```

Purpose

Will return the response code string for the transaction and response code specified. If the string is longer than the maximum length defined in response_string_length, the string is truncated. The response code specified must be valid for the indicated transaction.

5.44 sassign

Syntax

```
void sassign{destination_variable, string}
char *string;
```

Purpose

sassign will assign the specified string to the device variable. The variable must be valid. If necessary, the value is casted to the type of the referenced variable.

5.45 select_from_list

Syntax

```
int  select_from_list(prompt, option_list)
char *prompt;
char *option_list;
```

Purpose

select_from_list has the same functionality as select_from_list_wvarids, except that device variables are not allowed in the prompt string. For example:

```
int  result;
result = select_from_list("Is this correct?","Yes;No");
if (result == 0)
    { ... }
else
    { ... }
```


This transaction would display the prompt "Is this correct?'' and the two options "Yes'' and "No''. If "Yes'' is selected, a 0 is returned, and the code in the statement is executed. If "No'' is selected, a 1 is returned, and the code in the else-statement executed.

5.46 ShellExecute

Syntax

```
ShellExecute(string)
char *string;
```

Purpose

Takes the string argument and opens the specified file.

5.47 vassign

Syntax

```
int  vassign(target_var_name, source_var_id)
char *target_var_name;
char *source_var_id;
```

Purpose

Will assign the value of the source variable to the destination variable. Both variables must be valid.
Will return TRUE if the assignment was successful, and FALSE if either variable identifier was invalid.
See also:
VARID, fassign.

5.48 WriteCommand

Syntax

```
void WriteCommand(name)
```

Purpose

WriteCommand writes the variables defined in the COMMAND name to the field device.

A Example File

```
/* Example file using Electronic Device Description (EDD)              */
/* Important: This file serves as an example only, it is not normative */
/* File name: example.edd                                              */
/* 3.1. The Identification                                             */
MANUFACTURER 42,
DEVICE_TYPE 42,
DEVICE_REVISION 1,
DD_REVISION 1

VARIABLE local_variable
{
    LABEL "Local Variable";
    HELP  "Help";
    CLASS  LOCAL;
    TYPE   FLOAT
    {
        DEFAULT_VALUE 30;
        MIN_VALUE 10;
        MAX_VALUE 200;
        SCALING_FACTOR 200;
        EDIT_FORMAT "5d";
        DISPLAY_FORMAT "5d";
    }
    HANDLING READ & WRITE;
    VALIDITY TRUE;
}

BLOCK BlockIdentifier1
{
    TYPE   PHYSICAL;
    NUMBER 1;
}

VARIABLE local_variable_1
{
    LABEL  "Local Variable 1";
    CLASS  LOCAL;
    TYPE   FLOAT
    {
        DEFAULT_VALUE 20;
        MIN_VALUE 10;
        MAX_VALUE 200;
    }
    POST_EDIT_ACTIONS
    {
        postscale_variable
    }
    HANDLING READ & WRITE;
}

METHOD postscale_variable
```



```
{
    LABEL "Local Method";
    DEFINITION
    {
        float f;
        int i;

        f = fvar_value(local_variable_1);
        i = (f / 5) + 0.5;
        assign_int(variable, i);
        f = i * 5;
        assign_float(local_variable_1, f);
    }
}

COMMAND read_command
{
    SLOT 1;
    INDEX 2;
    OPERATION READ;
    TRANSACTION
    {
        REQUEST
        {
        }
        REPLY
        {
            variable1,
            variable2        <0xF0>,
            variable3        <0x08>,
            variable4        <0x07>
        }
    }
}

BLOCK physical_block
{
    TYPE   PHYSICAL;
    NUMBER 1;
}

COMMAND read_phys_blk
{
    BLOCK physical_block;
    INDEX 0;
    OPERATION READ;
    TRANSACTION
    {
        REQUEST
        {
        }
        REPLY
        {
```



```
            phys_blk_reserve, phys_blk_object,
            phys_blk_parent_class, phys_blk_class,
            phys_blk_dd_reference, phys_blk_dd_rev,
            phys_blk_profile, phys_blk_profile_rev,
            phys_blk_execution_time, phys_blk_highest_rel_offset,
            phys_blk_index_view_1, phys_blk_num_view_lists
        }
    }
}

VARIABLE VariableInCollection1
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

COLLECTION OF VARIABLE CollectionIdentifier1
{
    LABEL "Collection 1";
    HELP "Help for Collection 1";
    MEMBERS
    {
        member_1, VariableInCollection1, "description", "help";
    }
}

VARIABLE VariableInArray1
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

ITEM_ARRAY OF VARIABLE ArrayIdentifier1
{
    LABEL "Array 1";
    HELP  "Help for Array 1";
    ELEMENTS
    {
        1, VariableInArray1, "description", "help";
    }
}

VARIABLE VariableModified1
{
    CLASS LOCAL;
    TYPE  FLOAT;
    HANDLING READ & WRITE;
}

VARIABLE VariableModified2
{
```



```
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

VARIABLE VariableModified3
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

VARIABLE VariableToBeRefreshed1
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

VARIABLE VariableToBeRefreshed2
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

REFRESH Refresh1
{
    VariableModified1, VariableModified2, VariableModified3
    : VariableToBeRefreshed1, VariableToBeRefreshed2
}

VARIABLE VariableUnit
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

UNIT Unit1
{
    VariableUnit
    : VariableToBeRefreshed1, VariableToBeRefreshed2
}

VARIABLE condition
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ & WRITE;
}

VARIABLE VariableForConditional
```



```
{
    LABEL "VariableForConditional";
    CLASS CONTAINED;
    TYPE INTEGER (2);
    HANDLING IF( condition == 0x00 )
    {
        READ;
    }
    ELSE
    {
        READ & WRITE;
    }
}

VARIABLE identfier_1
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

VARIABLE identfier_2
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

MENU name
{
    LABEL  "string_A";
    HELP   "string_B";
    ACCESS ONLINE;
    STYLE  WINDOW;
    ITEMS
    {
        identfier_1,
        identfier_2
    }
}

MENU Menu_Main_Specialist
{
    LABEL "main menu";
    ITEMS
    {
        Menu_File,      /* assume to be defined somewhere else */
        Menu_Device,    /* assume to be defined somewhere else */
        Menu_View,      /* assume to be defined somewhere else */
        Menu_Options,   /* assume to be defined somewhere else */
        Menu_Help       /* assume to be defined somewhere else */
    }
}
```



```
MENU Menu_Device
{
    LABEL "Device";
    ITEMS
    {
        status,
        diagnostic,
        Online_Value
    }
}

MENU Online_Value
{
    ACCESS ONLINE;
    STYLE  BarGraph;
    LABEL  "Value";
    ITEMS
    {
        meas_value
    }
}

VARIABLE Variable1
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

VARIABLE Variable2
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

VARIABLE Variable3
{
    CLASS    LOCAL;
    TYPE     FLOAT;
    HANDLING READ;
}

MENU Table_Main_Specialist2
{
    LABEL "Test Device";
    ITEMS
    {
        Variable1,
        Menu
    }
}
```



```
MENU Menu
{
    LABEL "Menu";
    ITEMS
    {
        Variable2,
        Variable3
    }
}

/* Application Context */
VARIABLE ApplicationContext
{
    LABEL "ApplicationContext";
    CLASS LOCAL;
    TYPE BIT_ENUMERATED (4)
    {
        {0, "reserved"},
        {1, "FDT_CONFIGURATION"},
        {2, "FDT_PARAMETERIZE"},
        {3, "FDT_DIAGNOSIS"},
        {4, "FDT_MANAGEMENT"},
        {5, "FDT_OBSERVE"},
        {6, "FDT_DOCUMENTATION"},
        {7, "FDT_FORCE"},
        {8, "FDT_ASSET_MANAGEMENT"},
        {9, "reserved"},
        {10, "reserved"},
        {11, "reserved"},
        {12, "reserved"},
        {13, "reserved"},
        {14, "FDT_GMA_MAINTENANCE"},
        {15, "FDT_GMA_SPECIALIST"},
        {16, "DTM and / or vendor specific"},
        {17, "DTM and / or vendor specific"},
        {18, "DTM and / or vendor specific"},
        {19, "DTM and / or vendor specific"},
        {20, "DTM and / or vendor specific"},
        {21, "DTM and / or vendor specific"},
        {22, "DTM and / or vendor specific"},
        {23, "DTM and / or vendor specific"},
        {24, "DTM and / or vendor specific"},
        {25, "DTM and / or vendor specific"},
        {26, "DTM and / or vendor specific"},
        {27, "DTM and / or vendor specific"},
        {28, "DTM and / or vendor specific"},
        {29, "DTM and / or vendor specific"},
        {30, "DTM and / or vendor specific"},
        {31, "DTM and / or vendor specific"}
    }
}
```


B Lexic-Formal Definition

B.1 Operators

!	!=	%	%=
&	&&	&=	(
)	*	*=	+
++	+=	,	-
--	-=	.	/
/=	:	;	<
<<	<<=	<=	=
==	>	>=	>>
>>=	?	[	]
^	^=	{	|
|=	||	}	~

B.2 Keywords

ACCESS
ADD
ALARM
ALL
AO
APPINSTANCE
ARGUMENTS
ARRAY
ARRAYS
ASCII
AUTO
BAD
BIT_ENUMERATED
BITSTRING
BLOCK
BLOCKS
break
CASE
case
char
CLASS
COLLECTION
COLLECTIONS
COMMAND
COMMANDS
COMM_ERROR
CONNECTION
CONSTANT_UNIT
CONTAINED
continue
CORRECTABLE
DATA
DATA_ENTRY_ERROR
DATA_ENTRY_WARNING
DATA_EXCHANGE
DATE_AND_TIME
DD_REVISION
DEFAULT
default
DEFAULT_VALUE
DEFINITION
DELETE
DETAIL
DEVICE_REVISION
DEVICE_TYPE
DIAGNOSTIC
DIALOG
DISPLAY_FORMAT
DISPLAY_VALUE
do
DOMAIN
DOUBLE
double
DV
DYNAMIC
EDD_REVISION
EDIT_FORMAT
ELEMENTS
ELSE
else
ENUMERATED
EVENT
EVERYTHING
FALSE
FLOAT
float
for
FUNCTION
GOOD
HANDLING
HARDWARE
HELP
HIDDEN
IF
if
IGNORE_IN_HANDHELD
IMPORT
INDEX
INFO
INITIAL_VALUE
INPUT
int
INTEGER
ITEM_ARRAY
ITEMS
LABEL
LIKE
LOCAL
long
MANUAL
MANUFACTURER
MAX_VALUE
MEMBERS
MENU
MENUS
METHOD
METHODS
MIN_VALUE
MISC
MISC_ERROR
MISC_WARNING
MODE
MODE_ERROR
MODULE
MORE
NUMBER
NUMBER_OF_ELEMENTS
OF
OFFLINE
ONLINE
OPERATE
OPERATION
OUTPUT
PASSWORD

PHYSICAL	POST_EDIT_ACTIONS	POST_READ_ACTIONS
POST_WRITE_ACTIONS	PRE_EDIT_ACTIONS	PRE_READ_ACTIONS
PRE_WRITE_ACTIONS	PROCESS	PROCESS_ERROR
PROGRAM	READ	READ_ONLY
READ_TIMEOUT	RECORD	REDEFINE
REDEFINITIONS	REFRESH	RELATIONS
REPLY	REQUEST	RESPONSE_CODES
return	REVIEW	SCALING_FACTOR
SELECT	SELF_CORRECTING	SERVICE
short	signed	SLOT
SOFTWARE	STATE	STYLE
SUCCESS	SUMMARY	switch
TIME	TRANSACTION	TRANSDUCER
TRUE	TUNE	TV
TYPE	UNCORRECTABLE	UNIT
unsigned	UNSIGNED_INTEGER	VALIDITY
VARIABLE	VARIABLE_LIST	VARIABLES
while	WINDOW	WRITE
WRITE_AS_ONE	WRITE_TIMEOUT	

B.3 Terminals

```
DEFINE digit          = { 0-9 } .
       bin_digit          = { 0 1 } .
       non_zero_digit = { 1-9 '-' } .
       oct_digit          = { 0-7 } .
       hex_digit          = { 0-9abcdefABCDEF } .
       letter             = { a-zA-Z } .
       escapes            = { '"?afnrtv'\' } .
       ISOLatin1char  = - { " } .

/* Integer */
(0b|0B) bin_digit +               /* binaer */
non_zero_digit digit *        /* dezimal */
"0" oct_digit *                          /* octal */
(0x|0X) hex_digit +               /* hexadezimal */

/* real zahl */
digit*"."digit+((E|e){+\-}?digit+)?

/* string */
\" ISOLatin1char *  \"

/* character */
\' ISOLatin1char \'

/* Identifier */
letter (letter|digit|_)*
```


C Syntax-Formal Definition

C.1 Device Description Information

```
device_description
        = identification definition_list

identification
        = manufacturer ',' device_type ',' device_revision ',' DD_revision '

manufacturer
        = 'MANUFACTURER' Integer

device_type
        = 'DEVICE_TYPE' Integer

device_revision
        = 'DEVICE_REVISION' Integer

DD_revision
        = 'DD_REVISION' Integer
        = 'EDD_REVISION' Integer

definition_list
        = definition
        = definition_list definition

definition
        = item
        = imported_description
        = like

item
        = array
        = block
        = collection
        = command
        = connection
        = domain
        = item_array
        = menu
        = method
        = program
        = record
        = refresh_relation
        = response_codes_definition
        = unit_relation
        = variable
        = variable_list
        = write_as_one_relation
```


C.2 Array

```
array
        = 'ARRAY' Identifier '{' array_attribute_list '}'

array_attribute_list
        = array_attribute_listR

array_attribute_listR
        = array_attribute
        = array_attribute_listR array_attribute

array_attribute
        = array_type                                            /* M */
        = array_size                                            /* M */
        = required_label                                        /* M */
        = help                                                  /* O */
        = response_codes                                        /* O */

array_type
        = 'TYPE' variable_reference ';'

array_size
        = 'NUMBER_OF_ELEMENTS' Integer ';'
```

C.3 Block

```
block
        = 'BLOCK' Identifier '{' block_attribute_list '}'

block_attribute_list
        = block_attribute_listR

block_attribute_listR
        = block_attribute
        = block_attribute_listR block_attribute

block_attribute
        = block_type                                            /* M */
        = block_number                                          /* M */

block_type
        = 'TYPE' 'PHYSICAL' ';'
        = 'TYPE' 'TRANSDUCER' ';'
        = 'TYPE' 'FUNCTION' ';'

block_number
        = 'NUMBER' Integer ';'
        = 'NUMBER' expr ';'
```


C.4 C-Grammer

```
c_primary_expr
        = '[' Identifier ']'
        = Identifier
        = c_constant
        = string_literal
        = '(' c_expr ')'

c_constant
        = Integer
        = RealConst
        = CharacterConst

c_postfix_expr
        = c_primary_expr
        = c_postfix_expr '[' c_expr ']'
        = c_postfix_expr '(' ')'
        = c_postfix_expr '(' c_argument_expr_list ')'
        = c_postfix_expr '.' Identifier
        = c_postfix_expr '.' 'DEFAULT_VALUE'
        = c_postfix_expr '.' 'INITIAL_VALUE'
        = c_postfix_expr '++'
        = c_postfix_expr '--'

c_argument_expr_list
        = c_assignment_expr
        = c_argument_expr_list ',' c_assignment_expr

c_unary_expr
        = c_postfix_expr
        = '++' c_unary_expr
        = '--' c_unary_expr
        = c_unary_operator c_postfix_expr

c_unary_operator
        = '+'
        = '-'
        = '~'
        = '!'

c_multiplicative_expr
        = c_unary_expr
        = c_multiplicative_expr '*' c_unary_expr
        = c_multiplicative_expr '/' c_unary_expr
        = c_multiplicative_expr '%' c_unary_expr

c_additive_expr
        = c_multiplicative_expr
        = c_additive_expr '+' c_multiplicative_expr
```



```
        = c_additive_expr '-' c_multiplicative_expr

c_shift_expr
        = c_additive_expr
        = c_shift_expr '<<' c_additive_expr
        = c_shift_expr '>>' c_additive_expr

c_relational_expr
        = c_shift_expr
        = c_relational_expr '<' c_shift_expr
        = c_relational_expr '>' c_shift_expr
        = c_relational_expr '>=' c_shift_expr
        = c_relational_expr '<=' c_shift_expr

c_equality_expr
        = c_relational_expr
        = c_equality_expr '==' c_relational_expr
        = c_equality_expr '!=' c_relational_expr

c_and_expr
        = c_equality_expr
        = c_and_expr '&' c_equality_expr

c_exclusive_or_expr
        = c_and_expr
        = c_exclusive_or_expr '^' c_and_expr

c_inclusive_or_expr
        = c_exclusive_or_expr
        = c_inclusive_or_expr '|' c_exclusive_or_expr

c_logical_and_expr
        = c_inclusive_or_expr
        = c_logical_and_expr '&&' c_inclusive_or_expr

c_logical_or_expr
        = c_logical_and_expr
        = c_logical_or_expr '||' c_logical_and_expr


c_conditional_expr
        = c_logical_or_expr
        = c_logical_or_expr '?' c_logical_or_expr ':' c_conditional_expr
c_assignment_expr
        = c_conditional_expr
        = c_unary_expr c_assignment_operator c_assignment_expr

c_assignment_operator
        = '='
        = '*='
        = '/='
        = '%='
        = '+='
```



```
        = '-='
        = '>>='
        = '<<='
        = '&='
        = '^='
        = '|='

c_expr
        = c_assignment_expr
        = c_expr ',' c_assignment_expr

c_constant_expr
        = c_conditional_expr

c_declaration
        = c_declaration_specifiers ';'
        = c_declaration_specifiers c_declarator_list ';'

c_declaration_specifiers
        = c_type_specifier
        = c_type_specifier c_declaration_specifiers

c_declarator_list
        = c_declarator
        = c_declarator_list ',' c_declarator

c_declarator
        = Identifier
        = c_declarator '[' ']'
        = c_declarator '[' c_constant_expr ']'

c_type_specifier
        = 'char'
        = 'short'
        = 'int'
        = 'long'
        = 'signed'
        = 'unsigned'
        = 'float'
        = 'double'

c_statement
        = c_labeled_statement
        = c_compound_statement
        = c_expr_statement
        = c_selection_statement
        = c_iteration_statement
        = c_jump_statement

c_labeled_statement
        = 'case' c_constant_expr ':' c_statement
        = 'default' ':' c_statement
```



```
c_compound_statement
        = '{' '}'
        = '{' c_statement_list '}'
        = '{' c_declaration_list '}'
        = '{' c_declaration_list c_statement_list '}'

c_declaration_list
        = c_declaration
        = c_declaration_list c_declaration
c_statement_list
        = c_statement
        = c_statement_list c_statement

c_expr_statement
        = ';'
        = c_expr ';'

c_selection_statement
        = 'if' '(' c_expr ')' c_statement
        = 'if' '(' c_expr ')' c_statement 'else' c_statement
        = 'switch' '(' c_expr ')' c_statement

c_iteration_statement
        = 'while' '(' c_expr ')' c_statement
        = 'do' c_statement 'while' '(' c_expr ')' ';'
        = 'for' '('        ';'        ';'        ')' c_statement
        = 'for' '('        ';'        ';' c_expr ')' c_statement
        = 'for' '('        ';' c_expr ';'        ')' c_statement
        = 'for' '('        ';' c_expr ';' c_expr ')' c_statement
        = 'for' '(' c_expr ';'        ';'        ')' c_statement
        = 'for' '(' c_expr ';'        ';' c_expr ')' c_statement
        = 'for' '(' c_expr ';' c_expr ';'        ')' c_statement
        = 'for' '(' c_expr ';' c_expr ';' c_expr ')' c_statement

c_jump_statement
        = 'continue' ';'
        = 'break' ';'
        = 'return' ';'
        = 'return' c_expr ';'
```

C.5 Collection

```
collection
        = 'COLLECTION' 'OF' item_type Identifier '{' collection_attribute_list '}'

collection_attribute_list
        = collection_attribute_listR

collection_attribute_listR
        = collection_attribute
        = collection_attribute_listR collection_attribute
```



```
collection_attribute
        = members                                               /* M */
        = help                                                  /* O */
        = optional_label                                        /* O */

members
        = 'MEMBERS' '{' members_specifier_list '}'

members_specifier_list
        = members_specifier_listR

members_specifier_listR
        = members_specifier
        = members_specifier_listR members_specifier

members_specifier
        = member_list
        = 'IF' '(' expr ')' '{' members_specifier_list '}'
        = 'IF' '(' expr ')' '{' members_specifier_list '}'
          'ELSE' '{' members_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' members_selection_list '}'

member_list
        = member_listR

member_listR
        = member
        = member_listR member

member
        = Identifier ',' reference ';'
        = Identifier ',' reference ',' description_string ';'
        = Identifier ',' reference ',' description_string ',' help_string ';'

members_selection_list
        = members_selection
        = members_selection_list members_selection


members_selection
        = 'CASE' expr ':' members_specifier_list
        = 'DEFAULT' ':' members_specifier_list
```

C.6 Command

```
command
        = 'COMMAND' Identifier '{' command_attribute_list '}'
        = 'COMMAND' Identifier '{' '}'

command_attribute_list
        = command_attribute_listR
```



```
command_attribute_listR
        = command_attribute
        = command_attribute_listR command_attribute

command_attribute
        = command_address
        = command_number
        = operation
        = transaction
        = 'RESPONSE_CODES' '{' response_codes_specifier_list '}'
        = 'CONNECTION' Identifier ';'
        = 'MODULE' Identifier ';'


command_address
        = 'SLOT' Integer ';'
        = 'SLOT' Identifier ';'
        = 'INDEX' Integer ';'
        = 'BLOCK' Identifier ';'

command_number
        = 'NUMBER' command_number_specifier ';'

command_number_specifier
        = Integer ';'
        = 'IF' '(' expr ')' '{' command_number_specifier '}' 'ELSE'
          '{' command_number_specifier '}'
        = 'SELECT' '(' expr ')' '{' command_number_selection_list '}'

command_number_selection_list
        = command_number_selection_listR

command_number_selection_listR
        = command_number_selection
        = command_number_selection_listR command_number_selection

command_number_selection
        = 'CASE' expr ':' command_number_specifier
        = 'DEFAULT' ':' command_number_specifier
operation
        = 'OPERATION' operation_specifier

operation_specifier
        = 'READ' ';'
        = 'WRITE' ';'
        = 'COMMAND' ';'
        = 'DATA_EXCHANGE' ';'
        = 'IF' '(' expr ')' '{' operation_specifier '}' 'ELSE' '{' operation_specifier '}'
        = 'SELECT' '(' expr ')' '{' operation_selection_list '}'

operation_selection_list
        = operation_selection_listR
```



```
operation_selection_listR
        = operation_selection
        = operation_selection_listR operation_selection

operation_selection
        = 'CASE' expr ':' operation_specifier
        = 'DEFAULT' ':' operation_specifier

transaction
        = 'TRANSACTION' '{' transaction_specifier_list '}'
        = 'TRANSACTION' Integer '{' transaction_specifier_list '}'

transaction_specifier_list
        = transaction_specifier_listR


transaction_specifier_listR
        = transaction_specifier
        = transaction_specifier_listR transaction_specifier

transaction_specifier
        = request
        = reply
        = 'RESPONSE_CODES' '(' reference ')'

request
        = 'REQUEST' '{' data_items_specifier_list '}'
        = 'REQUEST' '{' '}'

reply
        = 'REPLY' '{' data_items_specifier_list '}'
        = 'REPLY' '{' '}'

data_items_specifier_list
        = data_items_specifier_listR

data_items_specifier_listR
        = data_items_specifier
        = data_items_specifier_listR data_items_specifier

data_items_specifier
        = data_items_list
        = 'IF' '(' expr ')' '{' data_items_specifier_list '}'
          'ELSE' '{' data_items_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' data_items_selection_list '}'

data_items_list
        = data_items_listR

data_items_listR
        = data_items
        = data_items_listR ',' data_items
```



```
data_items
        = Integer
        = variable_reference
        = variable_reference '<' Integer '>'
        = variable_reference '(' data_items_qualifiers ')'
        = variable_reference '<' Integer '>' '(' data_items_qualifiers ')'

data_items_qualifiers
        = data_items_qualifiers_

data_items_qualifiers_
        = data_items_qualifier
        = data_items_qualifiers_ ',' data_items_qualifier

data_items_qualifier
        = 'INDEX'
        = 'INFO'

data_items_selection_list
        = data_items_selection_listR

data_items_selection_listR
        = data_items_selection
        = data_items_selection_listR data_items_selection

data_items_selection
        = 'CASE' expr ':' data_items_specifier_list
        = 'DEFAULT' ':' data_items_specifier_list
```

C.7 Connection

```
connection
        = 'CONNECTION' Identifier '{' connection_attribute_list '}'

connection_attribute_list
        = connection_attribute_listR

connection_attribute_listR
        = connection_attribute
        = connection_attribute_listR connection_attribute

connection_attribute
        = 'APPINSTANCE' Integer                                 /* M */
```

C.8 Domain

```
domain
        = 'DOMAIN' Identifier '{' domain_attribute_list '}'

domain_attribute_list
        = domain_attribute_listR
```



```
domain_attribute_listR
        = domain_attribute
        = domain_attribute_listR domain_attribute

domain_attribute
        = handling                                          /* O */
        = response_codes                                    /* O */
```

C.9 Expression

```
primary_expr
        = reference '.' 'MIN_VALUE'
        = reference '.' 'MAX_VALUE'
        = reference '.' MIN_VALUE_Integer
        = reference '.' MAX_VALUE_Integer
        = reference
        = RealConst
        = Integer
        = '(' expr ')'
postfix_expr
        = primary_expr
        = postfix_expr '++'
        = postfix_expr '--'

unary_expr
        = postfix_expr
        = '++' unary_expr
        = '--' unary_expr
        = unary_operator multiplicative_expr

unary_operator
        = '+'
        = '-'
        = '~'
        = '!'
        = '&'

multiplicative_expr
        = unary_expr
        = multiplicative_expr '*' unary_expr
        = multiplicative_expr '/' unary_expr
        = multiplicative_expr '%' unary_expr

additive_expr
        = multiplicative_expr
        = additive_expr '+' multiplicative_expr
        = additive_expr '-' multiplicative_expr

shift_expr
        = additive_expr
        = shift_expr '<<' additive_expr
        = shift_expr '>>' additive_expr
```



```
relational_expr
        = shift_expr
        = relational_expr '<' shift_expr
        = relational_expr '>' shift_expr
        = relational_expr '>=' shift_expr
        = relational_expr '<=' shift_expr

equality_expr
        = relational_expr
        = equality_expr '==' relational_expr
        = equality_expr '!=' relational_expr

and_expr
        = equality_expr
        = and_expr '&' equality_expr

exclusive_or_expr
        = and_expr
        = exclusive_or_expr '^' and_expr


inclusive_or_expr
        = exclusive_or_expr
        = inclusive_or_expr '|' exclusive_or_expr

logical_and_expr
        = inclusive_or_expr
        = logical_and_expr '&&' inclusive_or_expr

logical_or_expr
        = logical_and_expr
        = logical_or_expr '||' logical_and_expr

conditional_expr
        = logical_or_expr
        = logical_or_expr '?' expr ':' conditional_expr

assignment_expr
        = conditional_expr
        = unary_expr assignment_operator assignment_expr

assignment_operator
        = '='
        = '*='
        = '/='
        = '%='
        = '+='
        = '-='
        = '>>='
        = '<<='
        = '&='
        = '^='
```



```
        = '|='

expr
        = assignment_expr
        = expr ',' assignment_expr
```

C.10 Imported EDD

```
imported_description
        = 'IMPORT' identification '{' imports '}'
        = 'IMPORT' identification '{' imports redefinitions '}'

imports
        = 'EVERYTHING' ';'
        = item_import_list

item_import_list
        = item_import_listR

item_import_listR
        = item_import
        = item_import_listR item_import

item_import
        = item_import_by_type ';'
        = item_import_by_name ';'

item_import_by_type
        = import_item_type
        = item_import_by_type '&' import_item_type

import_item_type
        = 'VARIABLES'
        = 'METHODS'
        = 'MENUS'
        = 'RELATIONS'
        = 'COLLECTIONS'
        = 'COMMANDS'
        = 'ARRAYS'
        = 'RESPONSE_CODES'
        = 'BLOCKS'
       = 'ITEM_ARRAYS'
       = 'RECORDS'
       = 'VARIABLE_LIST'
       = 'PROGRAMS'
       = 'DOMAINS'
       = 'CONNECTIONS'

item_import_by_name
        = item_type Identifier

redefinitions
```



```
        = 'REDEFINITIONS' '{' redefinition_list '}'

redefinition_list
        = redefinition_listR

redefinition_listR
        = redefinition
        = redefinition_listR redefinition

redefinition
        = block_redefinition
        = variable_redefinition
        = menu_redefinition
        = command_redefinition
        = method_redefinition
        = write_as_one_redefinition
        = refresh_redefinition
        = unit_redefinition
        = item_array_redefinition
        = collection_redefinition
        = response_codes_definition_redefinition
        = record_redefinition
        = array_redefinition
        = variable_list_redefinition
        = program_redefinition
        = domain_redefinition
        = connection_redefinition
```

C.11 Item array

```
item_array
        = 'ITEM_ARRAY' 'OF' item_type Identifier '{'item_array_attribute_list '}'

item_type
        = 'VARIABLE'
        = 'MENU'
        = 'METHOD'
        = 'REFRESH'
        = 'UNIT'
        = 'WRITE_AS_ONE'
        = 'ITEM_ARRAY' 'OF' item_type
        = 'COLLECTION' 'OF' item_type
        = 'RECORD'
        = 'ARRAY'
        = 'VARIABLE_LIST'
        = 'PROGRAM'
        = 'DOMAIN'
        = 'RESPONSE_CODES'
        = 'BLOCK'
        = 'COMMAND'
        = 'CONNECTION'
```



```
item_array_attribute_list
        = item_array_attribute_listR

item_array_attribute_listR
        = item_array_attribute
        = item_array_attribute_listR item_array_attribute

item_array_attribute
        = elements                                              /* M */
        = help                                                  /* M */
        = optional_label                                        /* O */

elements
        = 'ELEMENTS' '{' elements_specifier_list '}'

elements_specifier_list
        = elements_specifier_listR

elements_specifier_listR
        = elements_specifier
        = elements_specifier_listR elements_specifier

elements_specifier
        = element_list
        = 'IF' '(' expr ')' '{' elements_specifier_list '}'
        = 'IF' '(' expr ')' '{' elements_specifier_list '}'
          'ELSE' '{' elements_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' elements_selection_list '}'

element_list
        = element_listR


element_listR
        = element
        = element_listR element

element
        = Integer ',' reference ';'
        = Integer ',' reference ',' description_string ';'
        = Integer ',' reference ',' description_string ',' help_string ';'

elements_selection_list
        = elements_selection_listR

elements_selection_listR
        = elements_selection
        = elements_selection_listR elements_selection

elements_selection
        = 'CASE' expr ':' elements_specifier_list
        = 'DEFAULT' ':' elements_specifier_list
```



```
optional_label
        = 'LABEL' string_specifier
```

C.12 Like

```
like
        = Id1: Identifier 'LIKE' 'VARIABLE' Id2: Identifier
          '{' variable_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'MENU' Id2: Identifier
          '{' menu_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'METHOD' Id2: Identifier
          '{' method_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'ITEM_ARRAY' 'OF' item_type Id2: Identifier
          '{' item_array_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'COLLECTION' 'OF' item_type Id2: Identifier
          '{' collection_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'RESPONSE_CODES' Id2: Identifier
          '{' response_code_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'BLOCK' Id2: Identifier
          '{' block_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'RECORD' Id2: Identifier
          '{' record_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'ARRAY' Id2: Identifier
          '{' array_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'VARIABLE_LIST' Id2: Identifier
          '{' variable_list_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'COMMAND' Id2: Identifier
          '{' command_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'CONNECTION' Id2
          '{' connection_attribute_redefinition_list '}'
         = Id1: Identifier 'LIKE' 'PROGRAM' Id2: Identifier
          '{' program_attribute_redefinition_list '}'
        = Id1: Identifier 'LIKE' 'DOMAIN' Id2: Identifier
          '{' domain_attribute_redefinition_list '}'
```

C.13 Menu

```
menu
        = 'MENU' Identifier '{' menu_attribute_list '}'

menu_attribute_list
        = menu_attribute_listR

menu_attribute_listR
        = menu_attribute
        = menu_attribute_listR menu_attribute

menu_attribute
        = required_label                                        /* M */
        = menu_items
```



```
        = menu_access
        = menu_style
        = help                                                  /* O */
        = validity                                              /* O */

menu_items
        = 'ITEMS' '{' '}'
        = 'ITEMS' '{' menu_item_list '}'

menu_item_list
        = menu_item_listR

menu_item_listR
        = menu_item
        = menu_item_listR ',' menu_item

menu_item
        = menu_item_item
        = 'IF' '(' expr ')' '{' menu_item_list '}'
        = 'IF' '(' expr ')' '{' menu_item_list '}' 'ELSE' '{' menu_item_list '}'
        = 'SELECT' '(' expr ')' '{' menu_item_list_selection_list '}'

menu_item_item
        = reference
        = reference '(' 'REVIEW' ')'
        = reference '(' variable_qualifier_list ')'

variable_qualifier_list
        = variable_qualifier
        = variable_qualifier_list ',' variable_qualifier

variable_qualifier
        = 'DISPLAY_VALUE'
        = 'READ_ONLY'
        = 'HIDDEN'

menu_item_list_selection_list
        = menu_item_list_selection_listR


menu_item_list_selection_listR
        = menu_item_list_selection
        = menu_item_list_selection_list menu_item_list_selection

menu_item_list_selection
        = 'CASE' expr ':' menu_item_list
        = 'DEFAULT' ':' menu_item_list

menu_access
        = 'ACCESS' 'ONLINE' ';'
        = 'ACCESS' 'OFFLINE' ';'

menu_style
```



```
        = 'STYLE' 'WINDOW' ';'
        = 'STYLE' 'DIALOG' ';'
        = 'STYLE'  string ';'
```

C.14 Method

```
method
        = 'METHOD' Identifier '{' method_attribute_list '}'
        = 'METHOD' Identifier method_parameter_list'{' method_attribute_list '}'

method_parameter_list
        = '(' method_parameter_listR ')'

method_parameter_listR
        = method_parameter
        = method_parameter_listR ',' method_parameter

method_parameter
        = method_parameter_type Identifier

method_parameter_type
        = 'float'
        = 'int'
        = 'long'

method_attribute_list
        = method_attribute_listR

method_attribute_listR
        = method_attribute
        = method_attribute_listR method_attribute

method_attribute
        = variable_class                                        /* O */
        = method_definition                                     /* M */
        = optional_label                                        /* O */
        = method_access
        = help                                                  /* O */
        = validity                                              /* O */


method_access
        = 'ACCESS' 'OFFLINE' ';'
        = 'ACCESS' 'ONLINE' ';'

method_definition
        = 'DEFINITION' c_compound_statement
```

C.15 Open-Close

```
open
        = 'OPEN' filename
```



```
close
        = 'CLOSE' filename

filename
        = Identifier
```

C.16 Program

```
program
        = 'PROGRAM' Identifier '{' program_attribute_list '}'

program_attribute_list
        = program_attribute_listR

program_attribute_listR
        = program_attribute
        = program_attribute_listR program_attribute

program_attribute
        = arguments                                             /* O */
        = response_codes                                        /* O */

arguments
        = 'ARGUMENTS' '{' '}'
        = 'ARGUMENTS' '{' arguments_specifier_list '}'

arguments_specifier_list
        = arguments_specifier_listR

arguments_specifier_listR
                = arguments_specifier
                = arguments_specifier_listR arguments_specifier

arguments_specifier
        = argument_list
        = 'IF' '(' expr ')' '{' arguments_specifier_list '}'
        = 'IF' '(' expr ')' '{' arguments_specifier_list '}'
          'ELSE' '{' arguments_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' arguments_selection_list '}'

argument_list
        = argument_listR
argument_listR
        = argument
        = argument_listR ',' argument

argument
        = Integer
        = RealConst
        = variable_reference
```



```
arguments_selection_list
        = arguments_selection_listR

arguments_selection_listR
        = arguments_selection
        = arguments_selection_listR arguments_selection

arguments_selection
        = 'CASE' expr ':' arguments_specifier_list
        = 'DEFAULT' ':' arguments_specifier_list
```

C.17 Records

```
record
        = 'RECORD' Identifier '{' record_attribute_list '}'

record_attribute_list
        = record_attribute_listR

record_attribute_listR
        = record_attribute
        = record_attribute_listR record_attribute

record_attribute
        = members                                               /* M */
        = required_label                                        /* M */
        = help                                                  /* O */
        = response_codes                                        /* O */
```

C.18 Redefinition

```
command_redefinition
        = 'DELETE' 'COMMAND' Identifier ';'
        = 'REDEFINE' 'COMMAND' Identifier '{' '}'
        = 'REDEFINE' 'COMMAND' Identifier '{' command_attribute_list '}'
        = 'COMMAND' Identifier '{' command_attribute_redefinition_list '}'

command_attribute_redefinition_list
        = command_attribute_redefinition
        = command_attribute_redefinition_list command_attribute_redefinition

command_attribute_redefinition
        = command_address_redefinition
        = command_number_redefinition
        = command_operation_redefinition
        = command_transaction_redefinition
        = command_connection_redefinition
        = command_response_codes_redefinition
        = command_module_redefinition

command_address_redefinition
        = 'DELETE' 'SLOT' ';'
```



```
        = 'DELETE' 'INDEX' ';'
        = 'DELETE' 'BLOCK' ';'
        = 'REDEFINE' 'SLOT' Integer ';'
        = 'REDEFINE' 'SLOT' Identifier ';'
        = 'REDEFINE' 'INDEX' Integer ';'
        = 'REDEFINE' 'BLOCK' Identifier ';'

command_number_redefinition
        = 'DELETE' 'NUMBER' ';'
        = 'REDEFINE' 'NUMBER' command_number_specifier ';'

command_operation_redefinition
        = 'DELETE' 'OPERATION' ';'
        = 'REDEFINE' 'OPERATION' operation_specifier ';'

command_transaction_redefinition
        = 'DELETE' 'TRANSACTION' ';'
        = 'DELETE' 'TRANSACTION' Integer ';'
        = 'REDEFINE' 'TRANSACTION' '{' transaction_specifier_list '}'
        = 'REDEFINE' 'TRANSACTION' Integer '{' transaction_specifier_list '}'

command_connection_identifier_redefinition
        = 'DELETE' 'CONNECTION' ';'
        = 'REDEFINE' 'CONNECTION' Identifier ';'

command_response_codes_redefinition
        = 'DELETE' 'RESPONSE_CODES' ';'
        = 'REDEFINE' 'RESPONSE_CODES' '{' response_codes_specifier_list '}'
        = 'RESPONSE_CODES' '{' response_codes_redefinition_list '}'

command_module_redefinition
        = 'DELETE' 'MODULE' ';'
        = 'REDEFINE' 'MODULE' Identifier ' ;'


connection_redefinition
        = 'DELETE' 'CONNECTION' Identifier ';'
        = 'REDEFINE' 'CONNECTION' Identifier '{' connection_attribute_list '}'


write_as_one_redefinition
        = 'DELETE' 'WRITE_AS_ONE' Identifier ';'
        = 'REDEFINE' 'WRITE_AS_ONE' Identifier '{' variable_reference_list '}'


block_redefinition
        = 'DELETE' 'BLOCK' Identifier ';'
        = 'REDEFINE' 'BLOCK' Identifier '{' '}'
        = 'REDEFINE' 'BLOCK' Identifier '{' block_attribute_list '}'
        = 'BLOCK' Identifier '{' block_attribute_redefinition_list '}'

block_attribute_redefinition_list
        = block_attribute_redefinition
```



```
        = block_attribute_redefinition_list block_attribute_redefinition

block_attribute_redefinition
        = block_type_redefinition
        = block_number_redefinition

block_type_redefinition
        = 'REDEFINE' block_type

block_number_redefinition
        = 'REDEFINE' block_number

variable_redefinition
        = 'DELETE' 'VARIABLE' Identifier ';'
        = 'REDEFINE' 'VARIABLE' Identifier '{' '}'
        = 'REDEFINE' 'VARIABLE' Identifier '{' variable_attribute_list '}'
        = 'VARIABLE' Identifier '{' variable_attribute_redefinition_list '}'

variable_attribute_redefinition_list
        =
        = variable_attribute_redefinition_list variable_attribute_redefinition

variable_attribute_redefinition
        = variable_class_redefinition
        = handling_redefinition
        = help_redefinition
        = constant_unit_redefinition
        = required_label_redefinition
        = pre_edit_actions_redefinition
        = post_edit_actions_redefinition
        = pre_read_actions_redefinition
        = post_read_actions_redefinition
        = pre_write_actions_redefinition
        = post_write_actions_redefinition
        = read_timeout_redefinition
        = write_timeout_redefinition
        = type_redefinition
        = response_codes_reference_redefinition
        = validity_redefinition
        = default_value_redefinition
        = initial_value_redefinition

variable_class_redefinition
        = 'REDEFINE' variable_class

handling_redefinition
        = 'DELETE' 'HANDLING' ';'
        = 'REDEFINE' handling

help_redefinition
        = 'DELETE' 'HELP' ';'
        = 'REDEFINE' help
```



```
constant_unit_redefinition
        = 'DELETE' 'CONSTANT_UNIT' ';'
        = 'REDEFINE' constant_unit

required_label_redefinition
        = 'REDEFINE' required_label

pre_edit_actions_redefinition
        = 'DELETE' 'PRE_EDIT_ACTIONS' ';'
        = 'REDEFINE' pre_edit_actions

post_edit_actions_redefinition
        = 'DELETE' 'POST_EDIT_ACTIONS' ';'
        = 'REDEFINE' post_edit_actions

pre_read_actions_redefinition
        = 'DELETE' 'PRE_READ_ACTIONS' ';'
        = 'REDEFINE' pre_read_actions

post_read_actions_redefinition
        = 'DELETE' 'POST_READ_ACTIONS' ';'
        = 'REDEFINE' post_read_actions

pre_write_actions_redefinition
        = 'DELETE' 'PRE_WRITE_ACTIONS' ';'
        = 'REDEFINE' pre_write_actions

post_write_actions_redefinition
        = 'DELETE' 'POST_WRITE_ACTIONS' ';'
        = 'REDEFINE' post_write_actions

read_timeout_redefinition
        = 'DELETE' 'READ_TIMEOUT' ';'
        = 'REDEFINE' read_timeout

write_timeout_redefinition
        = 'DELETE' 'WRITE_TIMEOUT' ';'
        = 'REDEFINE' write_timeout

type_redefinition
        = 'TYPE' type_redefinitions
        = 'REDEFINE' type

type_redefinitions
        = 'INTEGER' '{' arithmetic_option_redefinition_list '}'
        = 'UNSIGNED_INTEGER' '{' arithmetic_option_redefinition_list '}'
        = 'FLOAT' '{' arithmetic_option_redefinition_list '}'
        = 'DOUBLE' '{' arithmetic_option_redefinition_list '}'
        = 'ENUMERATED' '{' enumeration_redefinition_list '}'
        = 'BIT_ENUMERATED' '{' bit_enumeration_redefinition_list '}'

arithmetic_option_redefinition_list
        = arithmetic_option_redefinition
```



```
        = arithmetic_option_redefinition_list arithmetic_option_redefinition

arithmetic_option_redefinition
        = display_format_redefinition
        = edit_format_redefinition
        = scaling_factor_redefinition
        = minimum_value_redefinition
        = maximum_value_redefinition
        = default_value_redefinition
        = initial_value_redefinition

display_format_redefinition
        = 'DELETE' 'DISPLAY_FORMAT' ';'
        = 'REDEFINE' display_format

edit_format_redefinition
        = 'DELETE' 'EDIT_FORMAT' ';'
        = 'REDEFINE' edit_format

scaling_factor_redefinition
        = 'DELETE' 'SCALING_FACTOR' ';'
        = 'REDEFINE' scaling_factor

minimum_value_redefinition
        = 'DELETE' 'MIN_VALUE' ';'
        = 'DELETE' MIN_VALUE_Integer ';'
        = 'REDEFINE' minimum_value

maximum_value_redefinition
        = 'DELETE' 'MAX_VALUE' ';'
        = 'DELETE' MAX_VALUE_Integer ';'
        = 'REDEFINE' maximum_value

enumeration_redefinition_list
        = enumeration_redefinition
        = enumeration_redefinition_list enumeration_redefinition

enumeration_redefinition
        = 'DELETE' Integer ';'
        = 'REDEFINE' enumerator
        = 'ADD' enumerator

bit_enumeration_redefinition_list
        = bit_enumeration_redefinition
        = bit_enumeration_redefinition_list bit_enumeration_redefinition

bit_enumeration_redefinition
        = 'DELETE' Integer ';'
        = 'REDEFINE' bit_enumerator
        = 'ADD' bit_enumerator

response_codes_reference_redefinition
        = 'DELETE' 'RESPONSE_CODES' ';'
```



```
        = 'REDEFINE' response_codes

validity_redefinition
        = 'DELETE' 'VALIDITY' ';'
        = 'REDEFINE' validity

default_value_redefinition
        = 'DELETE' 'DEFAULT_VALUE' ';'
        = 'REDEFINE' default_value

initial_value_redefinition
        = 'DELETE' 'INITIAL_VALUE' ';'
        = 'REDEFINE' initial_value

menu_redefinition
        = 'DELETE' 'MENU' Identifier ';'
        = 'REDEFINE' 'MENU' Identifier '{' '}'
        = 'REDEFINE' 'MENU' Identifier '{' menu_attribute_list '}'
        = 'MENU' Identifier '{' menu_attribute_redefinition_list '}'

menu_attribute_redefinition_list
        = menu_attribute_redefinition_list menu_attribute_redefinition

menu_attribute_redefinition
        = required_label_redefinition
        = menu_items_redefinition
        = menu_access_redefinition
        = menu_style_redefinition
        = help_redefinition

menu_items_redefinition
        = 'REDEFINE' menu_items

menu_access_redefinition
        = 'REDEFINE' menu_access

menu_style_redefinition
        = 'REDEFINE' menu_style

method_redefinition
        = 'DELETE' 'METHOD' Identifier ';'
        = 'REDEFINE' 'METHOD' Identifier '{' '}'
        = 'REDEFINE' 'METHOD' Identifier '{' method_attribute_list '}'
        = 'METHOD' Identifier '{' method_attribute_redefinition_list '}'

method_attribute_redefinition_list
        = method_attribute_redefinition
        = method_attribute_redefinition_list method_attribute_redefinition

method_attribute_redefinition
        = variable_class_redefinition
        = method_definition_redefinition
        = required_label_redefinition
```



```
        = help_redefinition
        = validity_redefinition

method_definition_redefinition
        = 'REDEFINE' method_definition

refresh_redefinition
        = 'DELETE' 'REFRESH' Identifier ';'
        = 'REDEFINE' 'REFRESH' Identifier '{' '}'
        = 'REDEFINE' 'REFRESH' Identifier '{' refresh_specifier '}'

unit_redefinition
        = 'DELETE' 'UNIT' Identifier ';'
        = 'REDEFINE' 'UNIT' Identifier '{' '}'
        = 'REDEFINE' 'UNIT' Identifier '{' unit_specifier '}'

item_array_redefinition
        = 'DELETE' 'ITEM_ARRAY' Identifier ';'
        = 'REDEFINE' 'ITEM_ARRAY' 'OF' item_type Identifier '{' '}'
        = 'REDEFINE' 'ITEM_ARRAY' 'OF' item_type Identifier
          '{' item_array_attribute_list '}'
        = 'ITEM_ARRAY' 'OF' item_type Identifier
          '{' item_array_attribute_redefinition_list '}'

item_array_attribute_redefinition_list
        = item_array_attribute_redefinition
        = item_array_attribute_redefinition_list item_array_attribute_redefinition

item_array_attribute_redefinition
        = elements_redefinition
        = help_redefinition
        = optional_label_redefinition

elements_redefinition
        = 'ELEMENTS' '{' element_redefinition_list '}'
        = 'REDEFINE' elements

element_redefinition_list
        = element_redefinition
        = element_redefinition_list element_redefinition

element_redefinition
        = 'DELETE' Integer ';'
        = 'REDEFINE' element
        = 'ADD' element

optional_label_redefinition
        = 'DELETE' 'LABEL' ';'
        = 'REDEFINE' optional_label

collection_redefinition
        = 'DELETE' 'COLLECTION' Identifier ';'
        = 'REDEFINE' 'COLLECTION' 'OF' item_type Identifier '{' '}'
```



```
        = 'REDEFINE' 'COLLECTION' 'OF' item_type Identifier
          '{' collection_attribute_list '}'
        = 'COLLECTION' 'OF' item_type Identifier
          '{' collection_attribute_redefinition_list '}'

collection_attribute_redefinition_list
        = collection_attribute_redefinition
        = collection_attribute_redefinition_list collection_attribute_redefinition

collection_attribute_redefinition
        = members_redefinition
        = help_redefinition
        = optional_label_redefinition

members_redefinition
        = 'MEMBERS' '{' member_redefinition_list '}'
        = 'REDEFINE' members

member_redefinition_list
        = member_redefinition
        = member_redefinition_list member_redefinition
member_redefinition
        = 'DELETE' Identifier ';'
        = 'REDEFINE' member
        = 'ADD' member

record_redefinition
        = 'DELETE' 'RECORD' Identifier ';'
        = 'REDEFINE' 'RECORD' Identifier '{' record_attribute_list '}'
        = 'RECORD' Identifier '{' record_attribute_redefinition_list '}'

record_attribute_redefinition_list
        = record_attribute_redefinition
        = record_attribute_redefinition_list record_attribute_redefinition

record_attribute_redefinition
        = help_redefinition
        = required_label_redefinition
        = response_codes_reference_redefinition
        = members_redefinition

array_redefinition
        = 'DELETE' 'ARRAY' Identifier ';'
        = 'REDEFINE' 'ARRAY' Identifier '{' array_attribute_list '}'
        = 'ARRAY' Identifier '{' array_attribute_redefinition_list '}'

array_attribute_redefinition_list
        = array_attribute_redefinition
        = array_attribute_redefinition_list array_attribute_redefinition

array_attribute_redefinition
        = array_type_redefinition
        = number_of_elements_redefinition
```



```
        = help_redefinition
        = required_label_redefinition
        = response_codes_reference_redefinition

array_type_redefinition
        = 'REDEFINE' array_type

number_of_elements_redefinition
        = 'REDEFINE' number_of_elements

response_codes_definition_redefinition
        = 'DELETE' 'RESPONSE_CODES' Identifier ';'
        = 'REDEFINE' 'RESPONSE_CODES' Identifier '{' '}'
        = 'REDEFINE' 'RESPONSE_CODES' Identifier '{' response_codes_specifier_list '}'
        = 'RESPONSE_CODES' Identifier '{' response_code_redefinition_list '}'


response_code_redefinition_list
        = response_code_redefinition
        = response_code_redefinition_list response_code_redefinition

response_code_redefinition
        = 'DELETE' Integer ';'
        = 'REDEFINE' response_code
        = 'ADD' response_code

variable_list_redefinition
        = 'DELETE' 'VARIABLE_LIST' Identifier ';'
        = 'REDEFINE' 'VARIABLE_LIST' Identifier '{' variable_list_attribute_list '}'
        = 'VARIABLE_LIST' Identifier '{' variable_list_attribute_redefinition_list '}'

variable_list_attribute_redefinition_list
        = variable_list_attribute_redefinition
        = variable_list_attribute_redefinition_list variable_list_attribute_redefinition

variable_list_attribute_redefinition
        = help_redefinition
        = optional_label_redefinition
        = response_codes_reference_redefinition
        = members_redefinition

program_redefinition
        = 'DELETE' 'PROGRAM' Identifier ';'
        = 'REDEFINE' 'PROGRAM' Identifier '{' program_attribute_list '}'
        = 'PROGRAM' Identifier '{' program_attribute_redefinition_list '}'

program_attribute_redefinition_list
        = program_attribute_redefinition
        = program_attribute_redefinition_list program_attribute_redefinition

program_attribute_redefinition
        = arguments_redefinition
        = response_codes_reference_redefinition
```



```
arguments_redefinition
        = 'DELETE' 'ARGUMENTS' ';'
        = 'REDEFINE' arguments

domain_redefinition
        = 'DELETE DOMAIN' Identifier ';'
        = 'REDEFINE DOMAIN' Identifier '{' domain_attribute_list '}'
        = 'DOMAIN' Identifier '{' domain_attribute_redefinition_list '}'

domain_attribute_redefinition_list
        = domain_attribute_redefinition
        = domain_attribute_redefinition_list domain_attribute_redefinition

domain_attribute_redefinition
        = handling_redefinition
        = response_codes_reference_redefinition
```

C.19 References

```
reference
        = Identifier
        = reference '[' expr ']'
        = reference '(' argument_list ')'
        = reference '.' Identifier
        = 'BLOCK' '.' Identifier

variable_reference
        = reference

menu_reference
        = reference

method_reference
        = reference

item_array_reference
        = reference

collection_reference
        = reference

response_codes_reference
        = reference

refresh_reference
        = reference

unit_reference
        = reference

block_reference
```



```
        = reference
```

C.20 Relation

```
refresh_relation
        = 'REFRESH' Identifier '{' refresh_specifier '}'

refresh_specifier
        = left: variable_reference_list ':' right: variable_reference_list

variable_reference_list
        = variable_reference_listR
variable_reference_listR
        = variable_reference
        = variable_reference_listR variable_reference
        = variable_reference_listR ',' variable_reference

unit_relation
        = 'UNIT' Identifier '{' unit_specifier '}'

unit_specifier
        = variable_reference ':' variable_reference_list

write_as_one_relation
        = 'WRITE_AS_ONE' Identifier '{' variable_reference_list '}'
```

C.21 Response Code

```
response_codes_definition
        = 'RESPONSE_CODES' Identifier '{' response_codes_specifier_list '}'

response_codes_specifier_list
        = response_codes_specifier_listR

response_codes_specifier_listR
        = response_codes_specifier
        = response_codes_specifier_listR response_codes_specifier

response_codes_specifier
        = response_code_list
        = 'IF' '(' expr ')' '{' response_codes_specifier_list '}'
        = 'IF' '(' expr ')' '{' response_codes_specifier_list '}'
          'ELSE' '{' response_codes_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' response_codes_selection_list '}'

response_code_list
        = response_code_listR

response_code_listR
        = response_code
        = response_code_listR response_code
```



```
response_code
        = Integer ',' response_code_type ',' description_string ',' help_string ';'
        = Integer ',' response_code_type ',' description_string ';'

response_code_type
        = 'SUCCESS'
        = 'MISC_WARNING'
        = 'DATA_ENTRY_WARNING'
        = 'DATA_ENTRY_ERROR'
        = 'MODE_ERROR'
        = 'PROCESS_ERROR'
        = 'MISC_ERROR'

response_codes_selection_list
        = response_codes_selection_listR

response_codes_selection_listR
        = response_codes_selection
        = response_codes_selection_listR response_codes_selection

response_codes_selection
        = 'CASE' expr ':' response_codes_specifier_list
        = 'DEFAULT' ':' response_codes_specifier_list
```

C.22 Variable

```
variable
        = 'VARIABLE' Identifier '{' variable_attribute_list '}'

variable_attribute_list
        = variable_attribute_listR

variable_attribute_listR
        = variable_attribute
        = variable_attribute_listR variable_attribute

variable_attribute
        = variable_class                                        /* M */
        = type                                                  /* M */
        = required_label                                        /* M */
        = constant_unit                                         /* O */
        = handling                                              /* O */
        = help                                                  /* O */
        = pre_edit_actions                                      /* O */
        = post_edit_actions                                     /* O */
        = pre_read_actions                                      /* O */
        = post_read_actions                                     /* O */
        = pre_write_actions                                     /* O */
        = post_write_actions                                    /* O */
        = read_timeout                                          /* O */
        = write_timeout                                         /* O */
        = response_codes                                        /* O */
```



```
        = validity                                              /* O */
        = default_value
        = initial_value

variable_class
        = 'CLASS' variable_class_definition ';'

variable_class_definition
        = variable_class_keyword
        = variable_class_definition '&' variable_class_keyword
variable_class_keyword
        = 'INPUT'
        = 'OUTPUT'
        = 'CONTAINED'
        = 'DYNAMIC'
        = 'DIAGNOSTIC'
        = 'SERVICE'
        = 'OPERATE'
        = 'ALARM'
        = 'TUNE'
        = 'LOCAL'

required_label
        = 'LABEL' required_string_specifier

required_string_specifier
        = string ';'
        = 'IF' '(' expr ')' '{' required_string_specifier '}'
          'ELSE' '{' required_string_specifier '}'
        = 'SELECT' '(' expr ')' '{' required_string_selection_list '}'


string
        = string_literal
        = variable_reference
        = variable_reference '(' Integer ')'
        = '[' Identifier ']'

string_literal
        = string_const
        = string_literal string_const

required_string_selection_list
        = required_string_selection_listR

required_string_selection_listR
        = required_string_selection
        = required_string_selection_listR required_string_selection

required_string_selection
        = 'CASE' expr ':' required_string_specifier
        = 'DEFAULT' ':' required_string_specifier
```



```
constant_unit
        = 'CONSTANT_UNIT' string_specifier

string_specifier
        = string ';'
        = 'IF' '(' expr ')' '{' string_specifier '}'
        = 'IF' '(' expr ')' '{' string_specifier '}' 'ELSE' '{' string_specifier '}'
        = 'SELECT' '(' expr ')' '{' string_selection_list '}'

string_selection_list
        = string_selection_listR

string_selection_listR
        = string_selection
        = string_selection_listR string_selection

string_selection
        = 'CASE' expr ':' string_specifier
        = 'DEFAULT' ':' string_specifier

handling
        = 'HANDLING' handling_specifier

handling_specifier
        = handling_definition ';'
        = 'IF' '(' expr ')' '{' handling_specifier '}'
        = 'IF' '(' expr ')' '{' handling_specifier '}' 'ELSE' '{' handling_specifier '}'
        = 'SELECT' '(' expr ')' '{' handling_selection_list '}'

handling_definition
        = handling_definition_

handling_definition_
        = handling_keyword
        = handling_definition_ '&' handling_keyword

handling_keyword
        = 'READ'
        = 'WRITE'

handling_selection_list
        = handling_selection_listR

handling_selection_listR
        = handling_selection
        = handling_selection_listR handling_selection

handling_selection
        = 'CASE' expr ':' handling_specifier
        = 'DEFAULT' ':' handling_specifier

help
        = 'HELP' string_specifier
```



```
pre_edit_actions
        = 'PRE_EDIT_ACTIONS' '{' actions_specifier_list '}'

post_edit_actions
        = 'POST_EDIT_ACTIONS' '{' actions_specifier_list '}'

pre_read_actions
        = 'PRE_READ_ACTIONS' '{' actions_specifier_list '}'

post_read_actions
        = 'POST_READ_ACTIONS' '{' actions_specifier_list '}'

pre_write_actions
        = 'PRE_WRITE_ACTIONS' '{' actions_specifier_list '}'

post_write_actions
        = 'POST_WRITE_ACTIONS' '{' actions_specifier_list '}'

actions_specifier_list
        = actions_specifier_listR

actions_specifier_listR
        = actions_specifier
        = actions_specifier_listR actions_specifier

actions_specifier
        = method_reference_list
        = 'IF' '(' expr ')' '{' actions_specifier_list '}'
        = 'IF' '(' expr ')' '{' actions_specifier_list '}'
          'ELSE' '{' actions_specifier_list '}'
        = 'SELECT' '(' expr ')' '{' actions_selection_list '}'

method_reference_list
        = method_reference_listR

method_reference_listR
        = method_reference
        = method_reference_listR ',' method_reference

actions_selection_list
        = actions_selection_listR

actions_selection_listR
        = actions_selection
        = actions_selection_listR actions_selection

actions_selection
        = 'CASE' expr ':' actions_specifier_list
        = 'DEFAULT' ':' actions_specifier_list

read_timeout
        = 'READ_TIMEOUT' expr_specifier
```



```
write_timeout
        = 'WRITE_TIMEOUT' expr_specifier
expr_specifier
        = expr ';'
        = 'IF' '(' expr ')' '{' expr_specifier '}'
        = 'IF' '(' expr ')' '{' expr_specifier '}' 'ELSE' '{' expr_specifier '}'
        = 'SELECT' '(' expr ')' '{' expr_selection_list '}'

expr_selection_list
        = expr_selection_listR

expr_selection_listR
        = expr_selection
        = expr_selection_listR expr_selection

expr_selection
        = 'CASE' expr ':' expr_specifier
        = 'DEFAULT' ':' expr_specifier

type
        = 'TYPE' type_specifier

type_specifier
        = arithmetic_type
        = enumerated_type
        = index_type
        = string_type
        = bitstring_type
        = date_time_type

arithmetic_type
        = float_type
        = double_type
        = integer_type
        = unsigned_integer_type


float_type
        = 'FLOAT' ';'
        = 'FLOAT' '{' arithmetic_option_list '}'

double_type
        = 'DOUBLE' ';'
        = 'DOUBLE' '{' arithmetic_option_list '}'

integer_type
        = 'INTEGER' ';'
        = 'INTEGER' '(' Integer ')' ';'
        = 'INTEGER' '{' arithmetic_option_list '}'
        = 'INTEGER' '(' Integer ')' '{' arithmetic_option_list '}'

unsigned_integer_type
```



```
        = 'UNSIGNED_INTEGER' ';'
        = 'UNSIGNED_INTEGER' '(' Integer ')' ';'
        = 'UNSIGNED_INTEGER' '{' arithmetic_option_list '}'
        = 'UNSIGNED_INTEGER' '(' Integer ')' '{' arithmetic_option_list '}'

arithmetic_option_list
        = arithmetic_option_listR

arithmetic_option_listR
        = arithmetic_option
        = arithmetic_option_listR arithmetic_option

arithmetic_option
        = display_format
        = edit_format
        = scaling_factor
        = minimum_value
        = maximum_value
        = default_value
        = initial_value
        = enumerator_list

display_format
        = 'DISPLAY_FORMAT' string_specifier

edit_format
        = 'EDIT_FORMAT' string_specifier

scaling_factor
        = 'SCALING_FACTOR' expr_specifier

minimum_value
        = 'MIN_VALUE' expr_specifier
        = MIN_VALUE_Integer expr_specifier

MIN_VALUE_Integer
        = 'MIN_VALUE1'
        = 'MIN_VALUE2'
        = 'MIN_VALUE3'
        = 'MIN_VALUE4'
        = 'MIN_VALUE5'

maximum_value
        = 'MAX_VALUE' expr_specifier
        = MAX_VALUE_Integer expr_specifier

MAX_VALUE_Integer
        = 'MAX_VALUE1'
        = 'MAX_VALUE2'
        = 'MAX_VALUE3'
        = 'MAX_VALUE4'
        = 'MAX_VALUE5'
```



```
enumerated_type
        = 'ENUMERATED' '{' enumerators_list '}'
        = 'ENUMERATED' '(' Integer ')' '{' enumerators_list '}'
        = 'BIT_ENUMERATED' '{' bit_enumerators_list '}'
        = 'BIT_ENUMERATED' '(' Integer ')' '{' bit_enumerators_list '}'

enumerators_list
        = enumerators_listR

enumerators_listR
        = enumerators_items
        = enumerators_listR enumerators_items

enumerators_items
        = default_value
        = initial_value
        = enumerators

enumerators
        = enumerator_list
        = 'IF' '(' expr ')' '{' enumerators_list '}'
        = 'IF' '(' expr ')' '{' enumerators_list '}' 'ELSE' '{' enumerators_list '}'
        = 'SELECT' '(' expr ')' '{' enumerators_list_selection_list '}'

enumerator_list
        = enumerator_listR

enumerator_listR
        = enumerator
        = enumerator_listR ',' enumerator

enumerator
        = '{' Integer ',' description_string '}'
        = '{' Integer ',' description_string ',' help_string '}'

description_string
        = string

help_string
        = string


enumerators_list_selection_list
        = enumerators_list_selection_listR

enumerators_list_selection_listR
        = enumerators_list_selection
        = enumerators_list_selection_list enumerators_list_selection

enumerators_list_selection
        = 'CASE' expr ':' enumerators_list
        = 'DEFAULT' ':' enumerators_list
```



```
bit_enumerators_list
        = bit_enumerators_listR

bit_enumerators_listR
        = bit_enumerators_items
        = bit_enumerators_listR bit_enumerators_items

bit_enumerators_items
        = default_value
        = initial_value
        = bit_enumerators

bit_enumerators
        = bit_enumerator_list
        = 'IF' '(' expr ')' '{' bit_enumerators_list '}'
        = 'IF' '(' expr ')' '{' bit_enumerators_list '}' 'ELSE' '{' bit_enumerators_list '}'
        = 'SELECT' '(' expr ')' '{' bit_enumerators_list_selection_list '}'

bit_enumerator_list
        = bit_enumerator_listR

bit_enumerator_listR
        = bit_enumerator
        = bit_enumerator_listR ',' bit_enumerator

bit_enumerator
        = '{' Integer ',' description_string '}'
        = '{' Integer ',' description_string ',
          ' help_string '}'
        = '{' Integer ',' description_string ',
          ' variable_class_definition '}'
        = '{' Integer ',' description_string ',
          ' status_class '}'
        = '{' Integer ',' description_string ',
          ' method_reference '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' variable_class_definition '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' status_class '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' method_reference '}'
        = '{' Integer ',' description_string ',
          ' variable_class_definition ',' status_class '}'
        = '{' Integer ',' description_string ',
          ' variable_class_definition ',' method_reference '}'
        = '{' Integer ',' description_string ',
          ' status_class ',' method_reference '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' variable_class_definition ',' status_class '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' variable_class_definition ',' method_reference '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' status_class ',' method_reference '}'
```



```
        = '{' Integer ',' description_string ',
          ' variable_class_definition ',' status_class ',' method_reference '}'
        = '{' Integer ',' description_string ',
          ' help_string ',' variable_class_definition ',' status_class ',
          ' method_reference '}'

status_class
        = status_class_keyword
        = status_class '&' status_class_keyword

status_class_keyword
        = 'HARDWARE'
        = 'SOFTWARE'
        = 'PROCESS'
        = 'MODE'
        = 'DATA'
        = 'MISC'
        = 'EVENT'
        = 'STATE'
        = 'SELF_CORRECTING'
        = 'CORRECTABLE'
        = 'UNCORRECTABLE'
        = 'SUMMARY'
        = 'DETAIL'
        = 'MORE'
        = 'COMM_ERROR'
        = 'IGNORE_IN_HANDHELD'
        = 'DV'          '(' output_mode ')'
        = 'TV'          '(' output_mode ')'
        = 'AO'          '(' output_mode ')'
        = 'ALL'         '(' output_mode ')'
        = 'DV'  Integer '(' output_mode ')'
        = 'TV'  Integer '(' output_mode ')'
        = 'AO'  Integer '(' output_mode ')'
        = 'ALL' Integer '(' output_mode ')'

output_mode
        = reliability '&' mode
        = mode '&' reliability

reliability
        = 'AUTO'
        = 'MANUAL'

mode
        = 'GOOD'
        = 'BAD'

bit_enumerators_list_selection_list
        = bit_enumerators_list_selection_listR

bit_enumerators_list_selection_listR
        = bit_enumerators_list_selection
```



```
        = bit_enumerators_list_selection_listR bit_enumerators_list_selection

bit_enumerators_list_selection
        = 'CASE' expr ':' bit_enumerators_list
        = 'DEFAULT' ':' bit_enumerators_list

index_type
        = 'INDEX' item_array_reference ';'
        = 'INDEX' item_array_reference '{' '}'
        = 'INDEX' item_array_reference '{' string_option_list '}'
        = 'INDEX' '(' Integer ')' item_array_reference ';'
        = 'INDEX' '(' Integer ')' item_array_reference '{' '}'
        = 'INDEX' '(' Integer ')' item_array_reference '{' string_option_list '}'

string_type
        = 'ASCII' '(' Integer ')' ';'
        = 'ASCII' '(' Integer ')' '{' '}'
        = 'ASCII' '(' Integer ')' '{' string_option_list '}'
        = 'PASSWORD' '(' Integer ')' ';'
        = 'PASSWORD' '(' Integer ')' '{' '}'
        = 'PASSWORD' '(' Integer ')' '{' string_option_list '}'

string_option_list
        = string_option_listR

string_option_listR
        = string_option
        = string_option_listR string_option

string_option
        = default_value
        = initial_value

bitstring_type
        = 'BITSTRING' '(' Integer ')' ';'

date_time_type
        = 'DATE_AND_TIME' ';'
        = 'DATE_AND_TIME' '{' '}'
        = 'DATE_AND_TIME' '{' string_option_list '}'
        = 'TIME' ';'
        = 'TIME' '{' '}'
        = 'TIME' '{' string_option_list '}'
        = 'TIME' '(' Integer ')' ';'
        = 'TIME' '(' Integer ')' '{' '}'
        = 'TIME' '(' Integer ')' '{' string_option_list '}'


response_codes
        = 'RESPONSE_CODES' response_codes_reference ';'
        = 'RESPONSE_CODES' expr_specifier

validity
```



```
        = 'VALIDITY' boolean_specifier

boolean_specifier
        = boolean ';'
        = 'IF' '(' expr ')' '{' boolean_specifier '}'
        = 'IF' '(' expr ')' '{' boolean_specifier '}' 'ELSE' '{' boolean_specifier '}'
        = 'SELECT' '(' expr ')' '{' boolean_selection_list '}'

boolean
        = 'TRUE'
        = 'FALSE'

boolean_selection_list
        = boolean_selection_listR

boolean_selection_listR
        = boolean_selection
        = boolean_selection_listR boolean_selection

boolean_selection
        = 'CASE' expr ':' boolean_specifier
        = 'DEFAULT' ':' boolean_specifier

default_value
        = 'DEFAULT_VALUE' expr_specifier
        = 'DEFAULT_VALUE' string ';'

initial_value
        = 'INITIAL_VALUE' expr_specifier
        = 'INITIAL_VALUE' string ';'
```

C.23 Variable List

```
variable_list
        = 'VARIABLE_LIST' Identifier '{' variable_list_attribute_list '}'

variable_list_attribute_list
        = variable_list_attribute_listR

variable_list_attribute_listR
        = variable_list_attribute
        = variable_list_attribute_listR variable_list_attribute

variable_list_attribute
        = members                                               /* M */
        = help                                                  /* O */
        = optional_label                                        /* O */
        = response_codes                                        /* O */
```


D List of Manufacturers

List Of Manufacturer*	DEVICE_MAN_ID (Hexadecimal)
ABB Automation	0x1A
ACCUTECH	0x5E
Acromag	0x1
Allen Bradley	0x2
Ametek	0x3
Analog Devices	0x4
Anderson Instrument Company	0x5A
Apparatebau Hundsbach	0x71
Applied System Technologies	0x41
Arcom Control Systems	0x3C
ASCO	0x102
Beckman	0x6
Bell Microsensor	0x7
BESTA	0x66
Betz	0x46
Bopp & Reuther Heinrichs	0x6C
Bourns	0x8
Bristol Babcock	0x9
Brooks Instrument	0x0A
BTG	0x55
Bürkert	0x78
Camille Bauer	0x2B
Chessell	0x0B
Combustion Engineering	0x0C
Daniel Industries	0x0D
Delta	0x0E
Dieterich Standard	0x0F
Dohrmann	0x10
Draeger	0x52
Drexelbrook	0x4E
Druck	0x47
Elcon Instruments	0x49
Elsag Bailey	0x5
Elsag Bailey	0x12
Elsag Bailey	0x16
EMCO	0x4A
Endress & Hauser	0x11
Exac Corporation	0x3A

List Of Manufacturer*	DEVICE_MAN_ID (Hexadecimal)
Fireye	0x44
Fisher Controls	0x13
Flow Measurement	0x5F
Flowdata	0x51
Foxboro	0x14
Foxboro Eckardt	0x3F
Fuji	0x15
Harold Beck and Sons	0x68
HELIOS	0x59
Honeywell	0x17
INOR	0x5B
ITT Barton	0x18
Jordan Controls	0x6E
KAMSTRUP	0x60
Kay Ray/Sensall	0x19
KDG Mobrey	0x3B
Knick	0x61
Krohne	0x45
K-TEK	0x50
Leeds & Northrup	0x1B
Leslie	0x1C
Magnetrol	0x56
Masoneilan-Dresser	0x65
Measurement Technology	0x40
Measurex	0x1E
Meridian Instruments	0x54
Micro Motion	0x1F
Milltronics	0x58
Moore Industries	0x20
Moore Products	0x21
M-System Co.	0x1D
MTS Systems Corp.	0x63
Neles Controls	0x57
Nuovo Pignone	0x38
Ohkura Electric	0x22
Ohmart	0x67
Oval	0x64
Paine	0x23
Peek Measurement	0x27

List Of Manufacturer*	DEVICE_MAN_ID (Hexadecimal)
PEPPERL+FUCHS	0x5D
PR Electronics	0x6D
Princo	0x3D
Promac	0x39
Raytek	0x53
rittmeyer instrumentation	0x69
ROBERTSHAW	0x5C
Rochester Instrument Systems	0x24
Ronan	0x25
Rosemount	0x26
Rosemount Analytic	0x2E
Rossel Messtechnik	0x6A
Rüeger	0x100
Saab Tank Control	0x4F
Samson	0x42
Schlumberger	0x28
Sensall	0x29
SICK	0x101
Siemens	0x2A
Smar	0x3E
SOR	0x48
Sparling Instruments	0x43
Termiflex Corporation	0x4B
Toshiba	0x2C
Transmation	0x2D
US ELECTRIC MOTORS	0x70
VAF Instruments	0x4C
Valcom s.r.l.	0x6F
Valmet	0x2F
Valtek	0x30
Varec	0x31
VEGA	0x62
Viatran	0x32
Weed	0x33
Westinghouse	0x34
Westlock Controls	0x4D
WIKA	0x6B
Xomox	0x35
Yamatake	0x36

List Of Manufacturer*	DEVICE_MAN_ID (Hexadecimal)
Yokogawa	0x37

E Description of the EDDL-Syntax using Unified Modeling Language

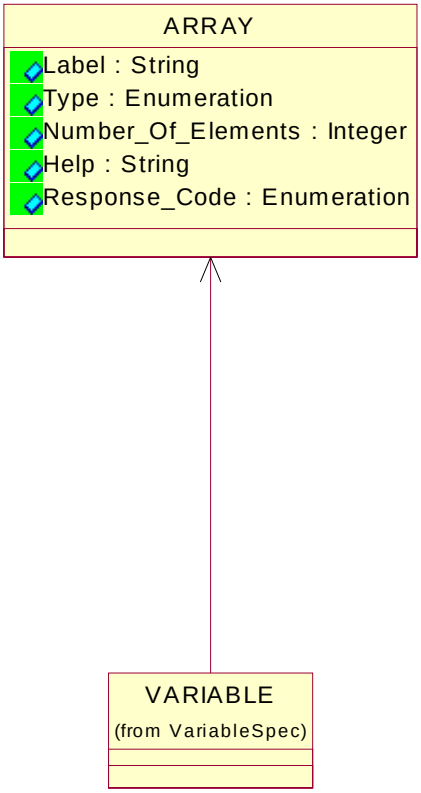


Figure 9: Array

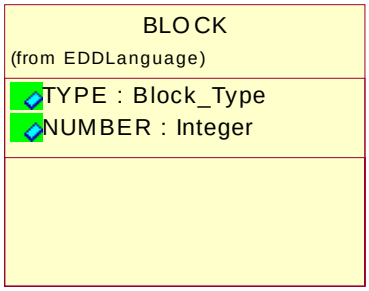


Figure 10: Block

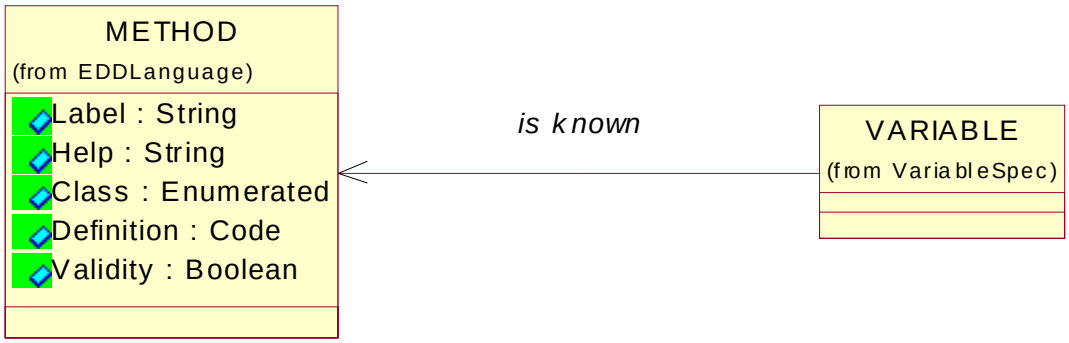


Figure 11: Method

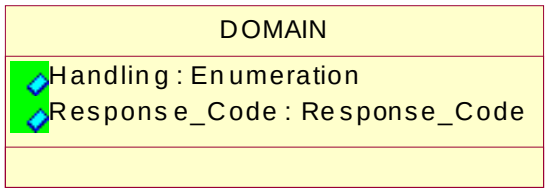


Figure 12: Domain

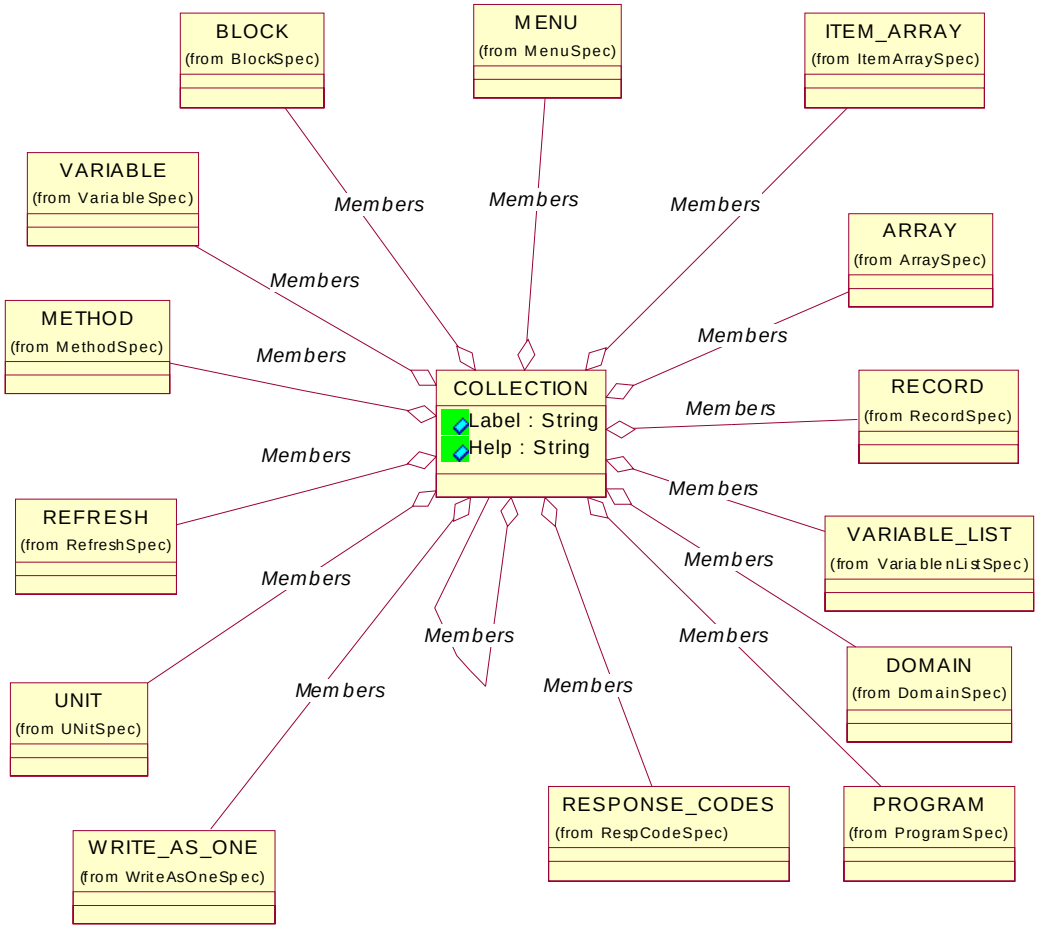


Figure 13: Collection

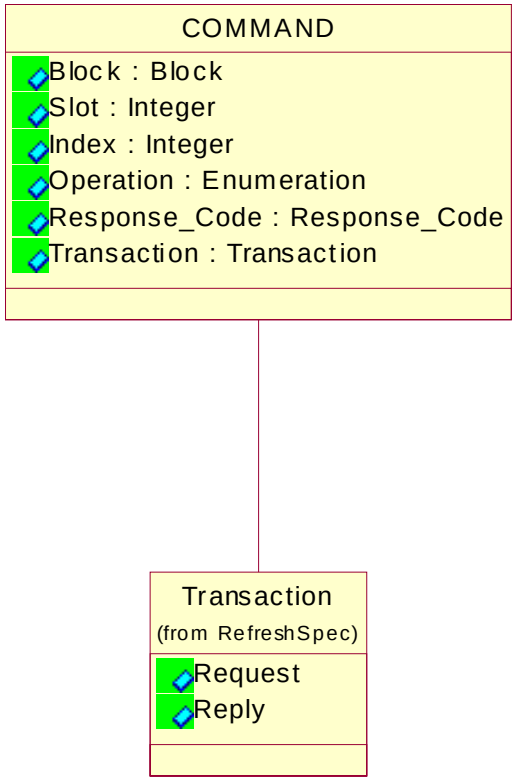


Figure 14: Command

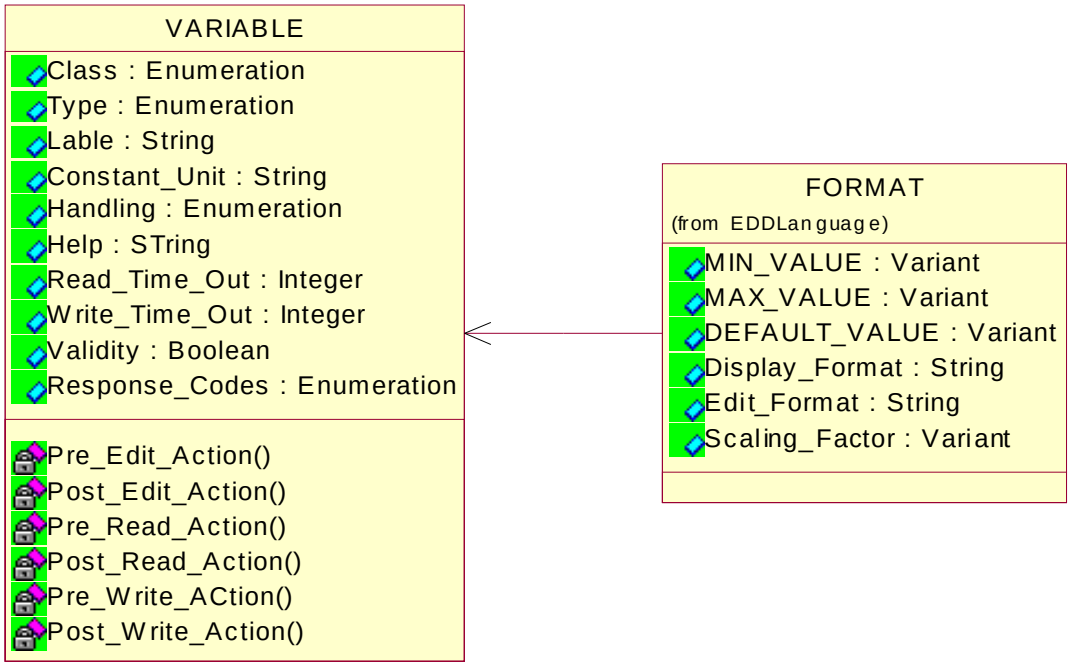


Figure 15: Variable

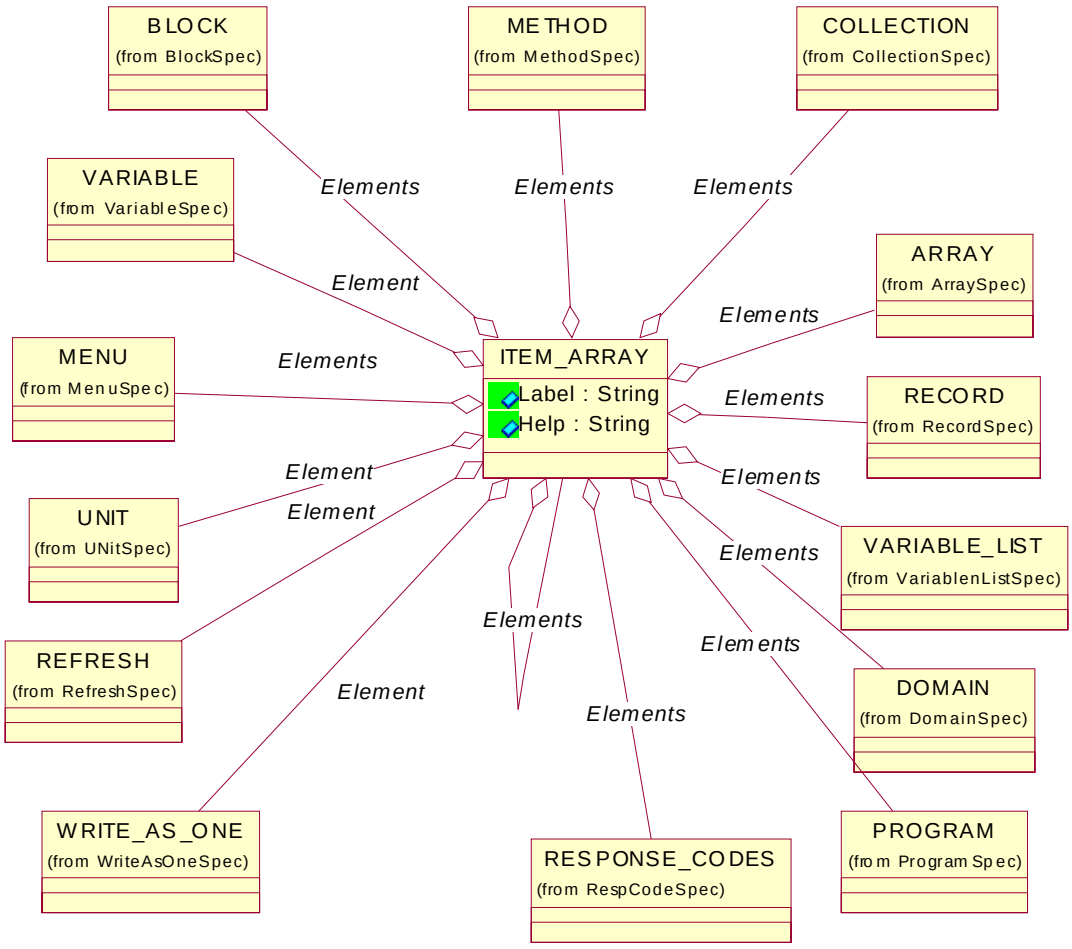


Figure 16: Item Array

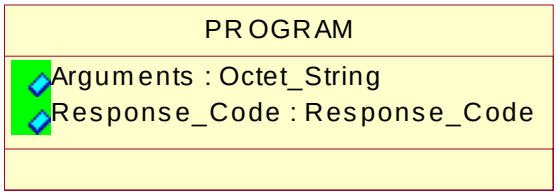


Figure 17: Program

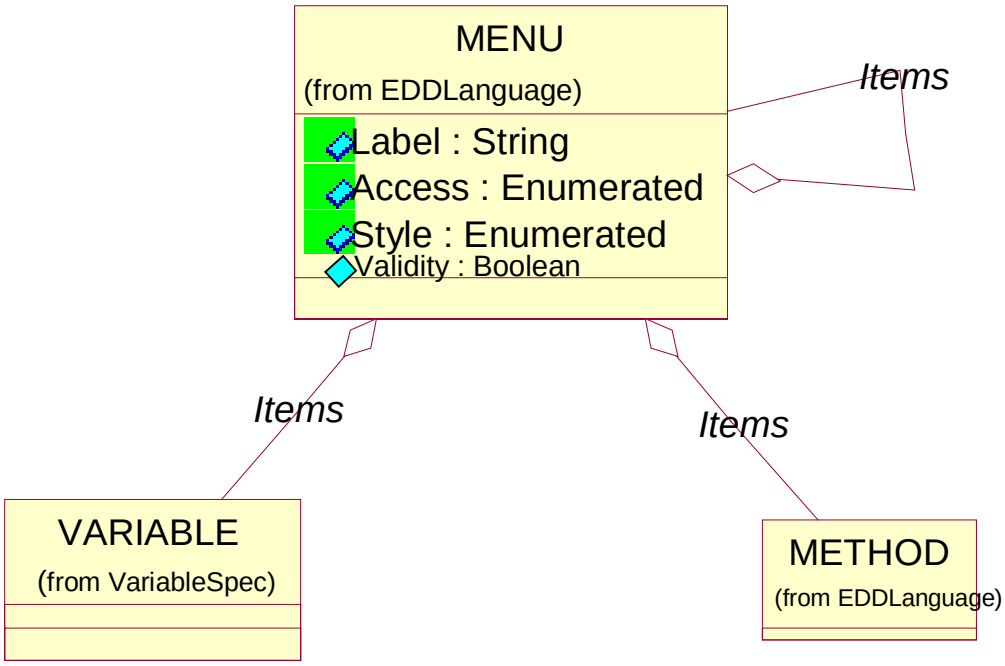


Figure 18: Menu

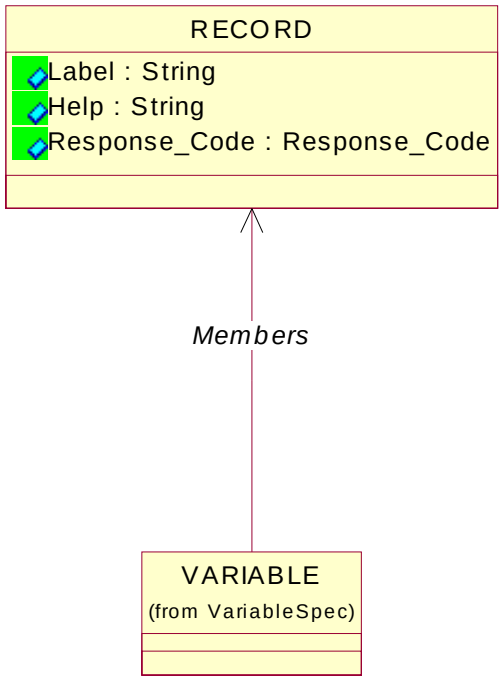


Figure 19: Record

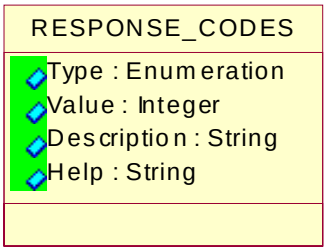


Figure 20: Response Code

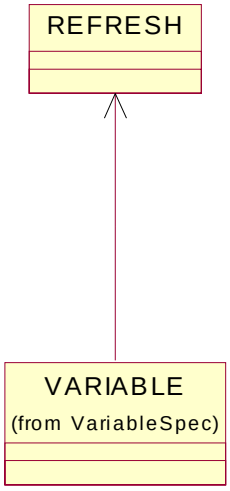


Figure 21: Refresh

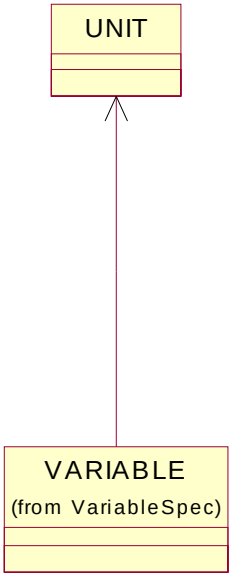


Figure 22: Unit

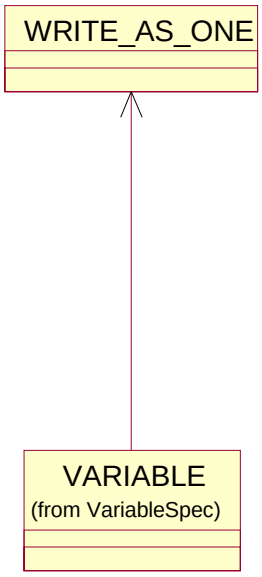


Figure 23: Write As One

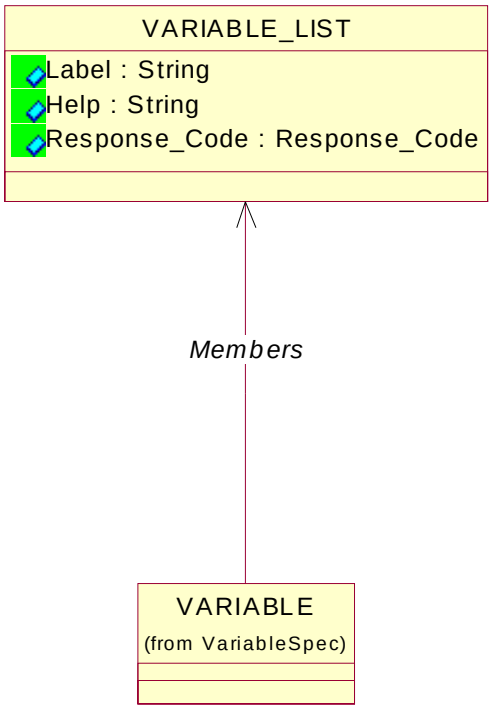


Figure 24: Variable List

PROFIBUS Nutzerorganisation e.V.
Haid-und-Neu-Str. 7
D-76131 Karlsruhe

Phone: ++ 721 / 96 58 590
Fax: ++ 721 / 96 58 589
PROFIBUS_International@compuserve.com
www.profibus.com