Consultative Committee for Space Data Systems

REPORT CONCERNING SPACE DATA SYSTEM STANDARDS

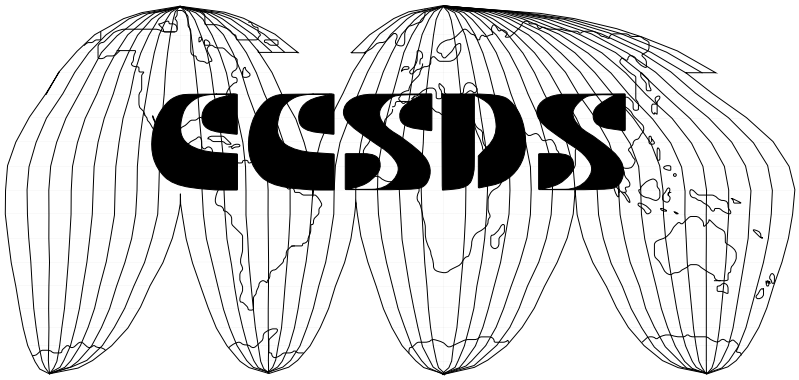
THE DATA DESCRIPTION LANGUAGE EAST—

A TUTORIAL

CCSDS 645.0-G-1

GREEN BOOK

May 1997



AUTHORITY

Issue:	Green Book, Issue 1
Date:	May 1997
Location:	São José dos Campos São Paulo, Brazil

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and reflects the consensus of technical panel experts from CCSDS Member Agencies. The procedure for review and authorization of CCSDS Reports is detailed in the *Procedures Manual for the Consultative Committee for Space Data Systems* [12].

This document is published and maintained by:

CCSDS Secretariat
Program Integration Division (Code MG)
National Aeronautics and Space Administration
Washington, DC 20546, USA

FOREWORD

This Report is a companion book to Reference [1] and contains rationale and explanatory material for the Recommendation in Reference [1].

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Report is therefore subject to CCSDS document management and change control procedures which are defined in reference [12]. Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/ccsds/

Questions relating to the contents or status of this document should be addressed to the CCSDS Secretariat at the address indicated on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were

- Agenzia Spaziale Italiana (ASI)/Italy.
- British National Space Centre (BNSC)/United Kingdom.
- Canadian Space Agency (CSA)/Canada.
- Centre National d'Etudes Spatiales (CNES)/France.
- Deutsche Forschungsanstalt für Luft- und Raumfahrt e.V. (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- National Aeronautics and Space Administration (NASA)/USA.
- National Space Development Agency of Japan (NASDA)/Japan.
- Russian Space Agency (RSA)/Russian Federation.

Observer Agencies

- Austrian Space Agency (ASA)/Austria.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- Centro Tecnico Aeroespacial (CTA)/Brazil.
- Chinese Academy of Space Technology (CAST)/China.
- Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
- Communications Research Laboratory (CRL)/Japan.
- Danish Space Research Institute (DSRI)/Denmark.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Federal Service of Scientific, Technical & Cultural Affairs (FSST&CA)/Belgium.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Industry Canada/Communications Research Centre (CRC)/Canada.
- Institute of Space and Astronautical Science (ISAS)/Japan.
- Institute of Space Research (IKI)/Russian Federation.
- KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- MIKOMTEK: CSIR (CSIR)/Republic of South Africa.
- Korea Aerospace Research Institute (KARI)/Korea.
- Ministry of Communications (MOC)/Israel.
- National Oceanic & Atmospheric Administration (NOAA)/USA.
- National Space Program Office (NSPO)/Taipei.
- Swedish Space Corporation (SSC)/Sweden.
- United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

Document	Title	Date	Status/Remarks
CCSDS 645.0-G-1	Report Concerning Space Data System Standards: The Data Description Language EAST—A Tutorial, Issue 1	May 1997	Original Issue

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

Figure Page

Example

CONTENTS (continued)

CONTENTS (continued)

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

Panel 2 of the Consultative Committee for Space Data Systems (CCSDS) is involved in information interchange issues. The Standard Formatted Data Unit (SFDU) concept is intended to allow the automation of information interchange between and among different environments (see reference [4]).

Intrinsic to the SFDU specification is the use of Data Description Record (DDR) to specify the representation of the interchanged data. Because of the wide diversity of operating systems and machine representations for numerics, the understanding of data coming from another agency or archives can only be reached by using a rigorous notation/language that provides a complete, non-ambiguous logical and physical description. EAST (Enhanced Ada Subset) is one of the recommended languages for data description records.

This document is intended to assist in the use of EAST, proposed as a description language. It explains how and why one would use this language to interchange data and data descriptions.

This document describes the usage of the EAST language, its format and construction rules as well as suggested practices. The chosen acronym (Enhanced Ada SubseT) suggests that EAST is based on a subset of the Ada programming language, which is the declarative part. The use of EAST does not preclude the use of any language for the application accessing the data, because in most of the cases, the use of a parser and an interpreter is needed. See 5.4 and Annex C for more explanations and a list of the available tools.

Most users will be able to use the language after reading this document.

1.2 REQUIREMENTS AND THEIR RATIONALES

This section has been developed from the document “Language Usage in Information Interchange” (see reference [2]), which lists the Requirements for a Data Interchange Language (DIL). The CCSDS believes that the general features of a language to support the description of data being interchanged shall be:

R1. Good Readability

Rationale: Users not specialized in computing must be able to understand descriptions of data to be processed, with a minimal effort.

R2. Support of basic data types

Rationale: As a minimum, the “atomic” types of character and numeric real and integer must be supported within the language. Additionally, the chosen language set should allow Boolean, bit and complex types.

R3. Data type definition capabilities

Rationale: Data type definition is the ability of the language to define and name user data types, to classify into families of data (date, temperature_in_degree_Celsius, distance_in_kilometers...).

R4. Data type structuring capabilities

Rationale: Data type structuring is the ability of the language to describe the logical relationship of "atomic" data items.

R5. Separation of the description from the data

Rationale: This is the ability to physically separate the description of the data from the data itself, so that the description can be updated and reused independently of the data.

R6. Physical representation capabilities

Rationale: Physical representation is the ability of the language to specify the bit pattern representation of data to be transported. This representation must specify not only basic data types, but also how the implementation producing the information represents these types.

1.3 DOCUMENT STRUCTURE

This document is intended to explain to potential users the description capabilities of EAST. It provides information for the effective use of EAST. Readers who will also be reading the EAST formal specification (reference [1]) may find it useful to read this document first, in order to have more examples and justifications of the EAST syntactic rules.

This document is structured as follows:

- Section 2 presents an overview of the context and why is a Data Interchange Language useful.
- Section 3 provides information and examples about EAST capabilities and how they can be used to satisfy data description requirements.
- Section 4 proposes some uses of EAST descriptions.
- Section 5 suggests some general practices which make the data description easier, identifies usages which may cause difficulties.
- Section 6 is a discussion of correspondence between requirements and EAST capabilities.

- Annex A contains acronyms used in this document.
- Annex B contains some EAST usage rules identified in this document.
- Annex C lists some of the tools that can be provided to check, generate, parse and analyze EAST descriptions.
- Annex D provides examples of data descriptions generated with an existing interactive tool.
- Annex E provides a compliance matrix according to data description requirements.
- Annex F provides a comparison between Ada and EAST.

1.4 DEFINITIONS

1.4.1 TERMS

The terms used throughout this document are listed in Annex A. They are also explained in the text when encountered for the first time.

1.4.2 CONVENTIONS

EAST is not case sensitive, but for the sake of readability, we adopted the following conventions in the document:

- EAST keywords are provided using lowercase letters;
- user type names or user variable names are provided using uppercase letters.

As a tutorial, this document explains the EAST syntactic rules, in providing examples, notes or answers to questions. Different categories of users will read this tutorial:

- application users, who are interested in knowing if EAST meets their requirements, if tools support the EAST technology, etc.;
- programmers, who require many examples to follow in implementing specifications;
- Ada programmers, who are interested in the references to the Ada language, the differences between the two languages, etc.

Three levels of detail corresponding to the categories above are provided in this document. The following convention applies throughout the document: the notes that are addressed to readers who have some knowledge of Ada, are indicated by a (Fn) sign; the text of the notes is gathered in Annex F.

1.5 REFERENCES

[1] *The Data Description Language EAST Specification (CCSD0010)*. Recommendation for Space Data Systems Standards, CCSDS 644.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, May 1997.

[2] *Language Usage in Information Interchange Tutorial*. Report Concerning Space Data Systems Standards, CCSDS 642.1-G-1. Green Book. Issue 1. Washington, D.C.: CCSDS, October 1989.

[3] *Packet Telemetry*. Recommendation for Space Data Systems Standards, CCSDS 102.0-B-4. Blue Book. Issue 4. Washington, D.C.: CCSDS, November 1995.

[4] *Standard Formatted Data Units—Structure and Construction Rules*. Recommendation for Space Data Systems Standards, CCSDS 620.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, May 1992.

[5] *ASCII Encoded English (CCSD0002*). Recommendation for Space Data Systems Standards, CCSDS 643.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, November 1992.

[6] *Information Processing—8-Bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1*. International Standard, ISO 8859-1:1987. Geneva: ISO, 1987.

[7] *Information Technology—Programming Languages—Ada*. International Standard, ISO/IEC 8652:1995. Geneva: ISO, 1995.

[8] *Binary Floating Point Arithmetic*. American National Standard, ANSI/IEEE 754-1985 (R1991). New York: ANSI, 1985.

[9] *The Data Description Language EAST—List of Conventions*. Report Concerning Space Data Systems Standards, CCSDS 646.0-G-1. Green Book. Issue 1, May 1997.

[10] *Information Technology—Programming Languages—FORTRAN*. International Standard, ISO/IEC 1539:1991. Geneva: ISO, 1991.

[11] *Standard Formatted Data Units—Control Authority Procedures*. Recommendation for Space Data Systems Standards, CCSDS 630.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, June 1993.

[12] *Procedures Manual for the Consultative Committee for Space Data Systems*. CCSDS A00.0-Y-7. Yellow Book. Issue 7. Washington, D.C.: CCSDS, November 1996.

2 OVERVIEW

2.1 CONTEXT

Before reading the technical content of the tutorial, the reader may want to know what is useful for a description language and why a formal language (like EAST) is more advantageous to a user than other languages (like the English language, for example).

Space agencies produce a large amount of data, which are immediately investigated, or stored, or exchanged, etc. A data item that is not described is useless. Indeed, how can it be used, if no one knows what it represents, how long it is, if it is a scalar or something else?

Data description languages are therefore highly necessary to space agencies to manage and maintain their vast amounts of data. Some data description languages are available: the natural language English is also recommended by the CCSDS (reference [5]) to supply information within the Standard Formatted Data Unit (SFDU) environment.

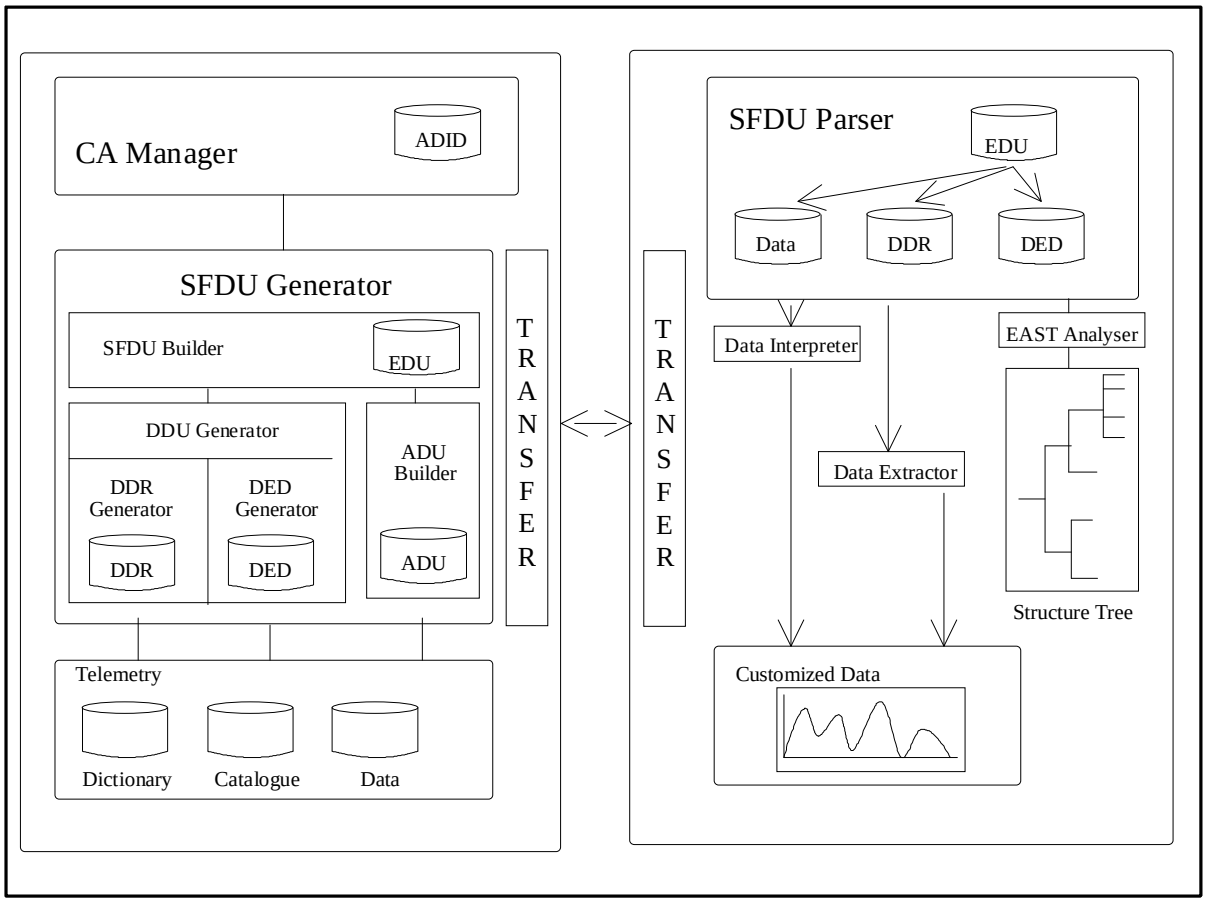
The use of a formal language instead of a natural language has the following advantage: it is a machine interpretable language that allows the interpretation of data in an automated fashion. Tools can therefore be implemented to provide help to users for:

- the description of data;
- the interpretation of data;
- the conversion of data to another format;
- the filtering of data to extract the useful information;
- the rehabilitation of old data, etc.

The use of a formal language that provides a physical separation of data and data description has the following additional advantage: it does not interfere with the data that it describes. It does not impose any format to the data. Such a formal language is able to describe the data as they actually are. It is therefore able to describe “historical” data, as well as “future” data.

EAST is a language that meets these requirements. It allows the automated interpretation of the data and the description of “historical” and “future” data.

EAST is a proposed language for the production of Data Description Records (DDR) in the SFDU context. Figure 2-1 describes the general context of data exchange.



Legend:

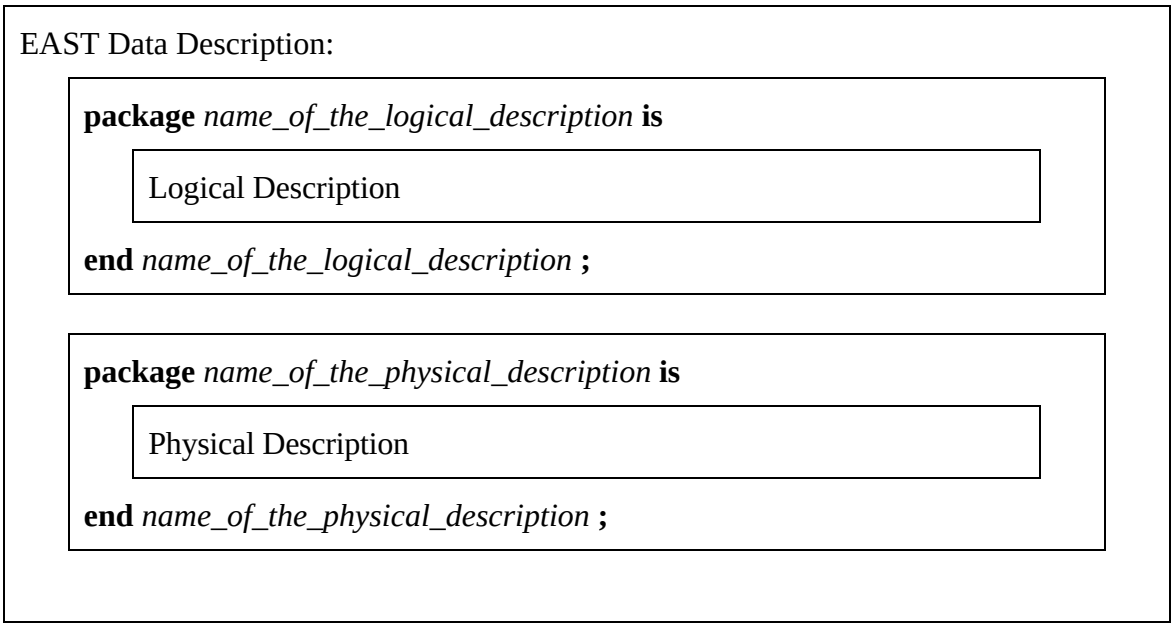
ADID	Authority and Description Identifier
ADU	Application Data Unit
CA	Control Authority
DDU	Description Data Unit
DED	Data Entity Dictionary
EDU	Exchange Data Unit

Figure 2-1: Data Exchange in SFDU Context

A more detailed description of the SFDU context is provided in the SFDU - Structure and Construction Rules Book (reference [4]).

2.2 ORGANIZATION OF THE INFORMATION CONVEYED BY EAST

An EAST data description is composed of two parts, called packages: the first one is called "logical description" and the second one is called "physical description".



a) Logical Description

The logical part provides syntactic information and in some way semantic information. It provides a large part of the information needed by an application user to understand the data he has to deal with.

The logical part gives a name to every data item; i.e., it provides some meaning of the data item (e.g., a COUNTER, a MEASUREMENT, a SATELLITE_IDENTIFIER, an ACTIVITY_FLAG, etc.).

It describes the nature of every data item (e.g., it is a whole number, or a real number, or a character string, or a bit string, etc.).

It gives syntactic information to every data item (e.g., it is a positive 16-bit integer, or a 32-bit real with a range of values from 0.1 to 1.0, or a 20 character string, or two-bit enumeration with three permitted values, ON, OFF, ERROR, etc.).

The logical part also includes elements for the structuring of data; e.g.:

- a date is made of an integer representing the year, an integer representing the month and an integer representing the day;
- the repetition of 100 measurements defines a data block, etc.

It also provides the ordering of the data items.

Subsection 3.2 of this document describes the logical part of an EAST data description.

b) Physical Description

The physical part provides pure syntactic information. It is a detailed description, i.e., a bit level description that ensures a non-ambiguous interpretation of the data. This part should only be used by the tool in charge of the data interpretation or of other processing.

It provides machine dependent characteristics that determine the coding of real numbers, the coding of integers, the way of storing tables, etc. For example:

- the location of the sign bit, the location of the exponent and the location of the mantissa, and also the standard used to build the real (e.g., the IEEE standard—see reference [8]) are provided for any real;
- the location of the most significant bit and the location of the least significant bit are provided for any integer.

While EAST physical descriptions are written in a human readable language, manually reading the physical part is not recommended, since it is likely to be long and complicated.

Subsection 3.3 of this document describes the physical part of an EAST data description.

A tool based on a Graphical User Interface (see Annex C) is considered to be highly necessary to the user, so that he can write data descriptions in EAST without any knowledge of the syntax. Nevertheless this document intends to explain how to use the syntax of EAST. If a user wishes to generate EAST descriptions by hand (e.g., for testing purposes) another tool (see Annex C) is necessary to check the correctness of the generated description.

A tool that parses EAST data descriptions and interprets data should also be used by application users to access the data.

Figure 2-2 illustrates the tools that are recommended:

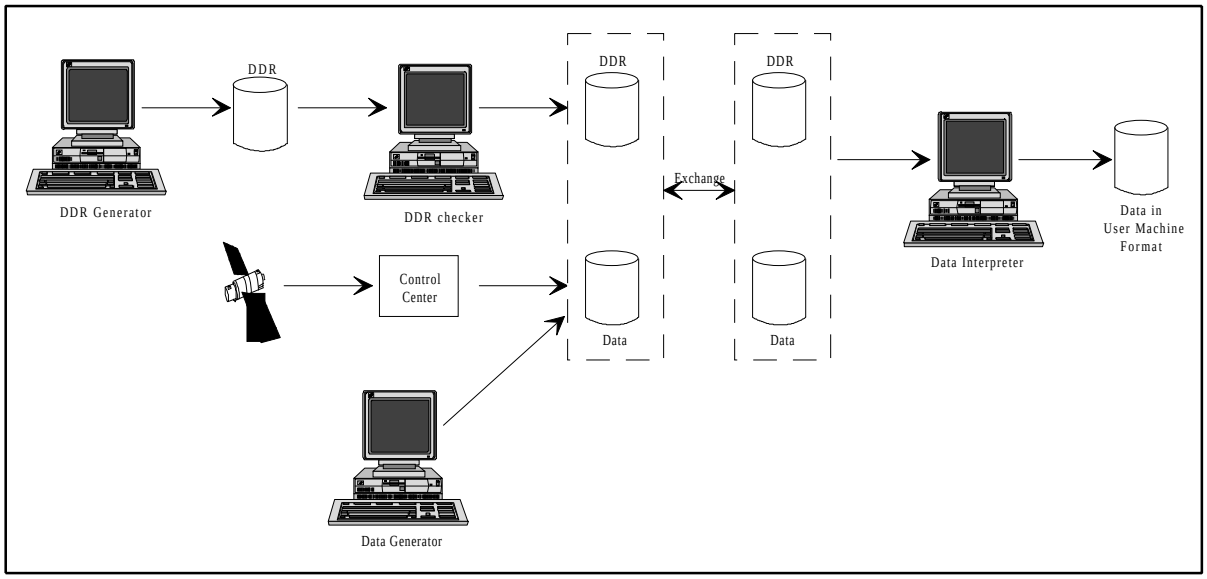


Figure 2-2: Data and Data Description Records (DDR)

2.3 SCOPE OF THE DATA TO BE DESCRIBED

Data Descriptions may be at once the concern of people in many different contexts: project, telemetry, telecommand, data processing or other results storage.

Whatever the context, data must be described and could be represented using a tree structure. The structure of the data is represented by the “branches” of the tree, and the elementary or scalar data by the “leaves”.

The following subsections present two examples of data, taken from different contexts (telemetry and space mechanics), that are used throughout the document to illustrate the EAST syntax.

The telemetry context provides many examples of exchanged data.

The figure 2-3 gives, as an example, the format of a source packet (see reference [3]).

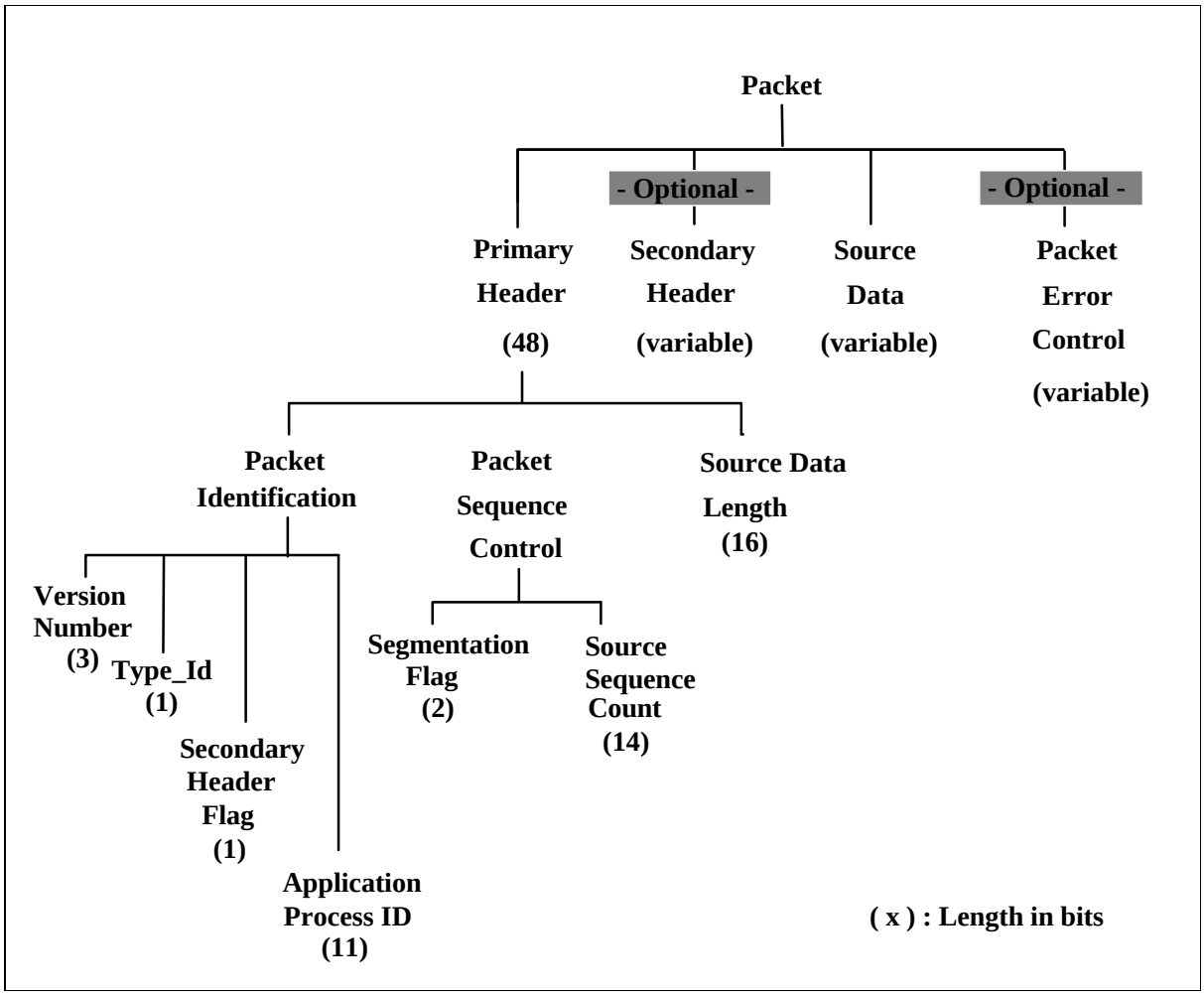


Figure 2-3: Version 1 "Source Packet" Format

A branch or a leaf may be optional, depending on the value of another leaf. In the example, "Secondary Header" is present or absent, depending on the value of "Secondary Header Flag". A leaf may be:

- a bit string, with specific bit patterns, representing a limited set of values (e.g., the "Segmentation Flag", which identifies the status of the packet);
- a whole number, with a predefined length (e.g., the 16 bit field "Source Data Length");
- a real number, as illustrated in the following example.

Figure 2-4 provides an example of the space mechanics context:

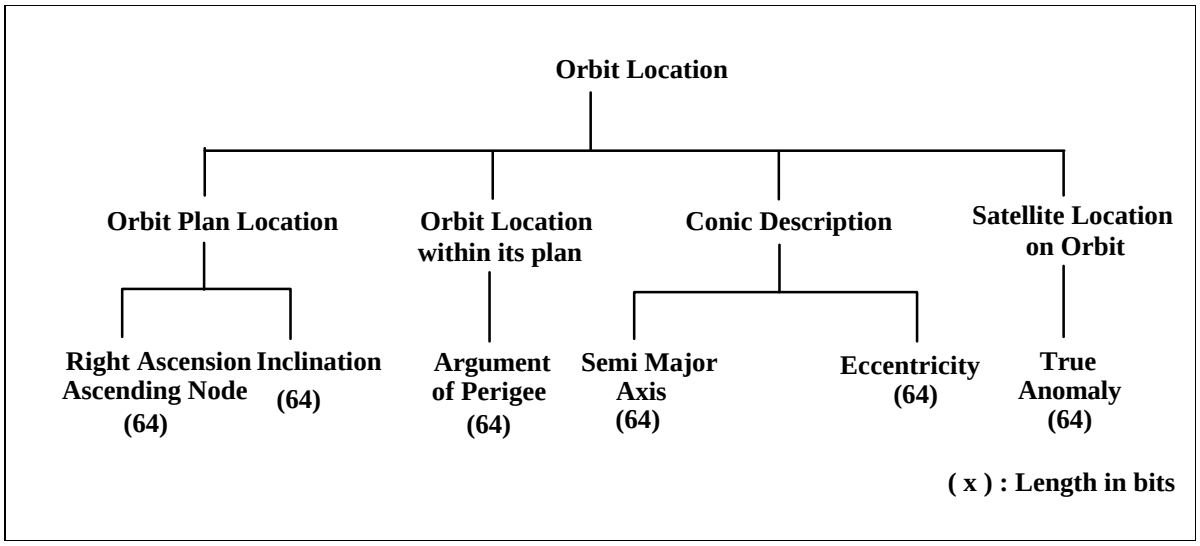


Figure 2-4: Orbit Location

Figure 2-5 provides imaginary telemetry data, which could be, for example, the source data of the packet (see figure 2-3).

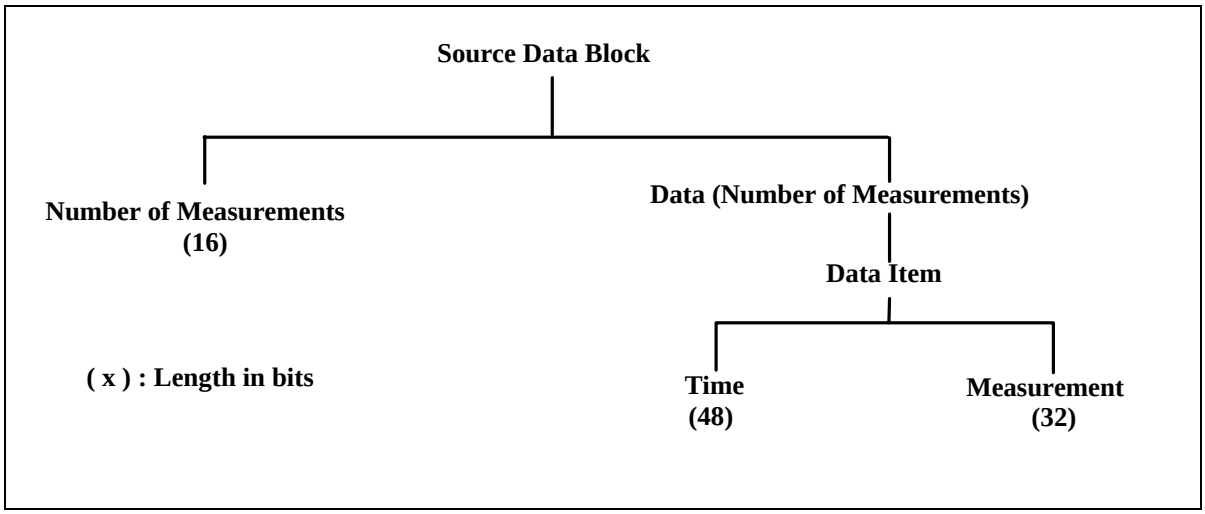


Figure 2-5: Source Data Block

The number of elements of a branch may depend on the value of another leaf. In the example, the number of measurements determines the number of data items.

A Data Description Language must therefore make the description of scalar data easy and must allow the data to be structured to varying levels of complexity.

3 PRODUCING EAST DATA DESCRIPTIONS

This section deals with the production of EAST data descriptions:

- subsection 3.1 describes the lexical elements used in any part of an EAST description;
- subsection 3.2 describes the information conveyed by the logical description part;
- subsection 3.3 describes the information conveyed by the physical description part;
- subsection 3.4 describes the relation between the logical information and the physical information.

3.1 LEXICAL ELEMENTS OF EAST DATA DESCRIPTIONS

The text of a data description is a sequence of lexical elements, each composed of characters. The 128 first characters of the "Latin Alphabet No. 1" character set (see reference [6]) are allowed in an EAST description. A **lexical element** is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character string, a string literal, or a comment. The rules of composition are given in this section.

In some cases an explicit separator is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A **separator** is any of a space character, a control character, or the end of a line.

- A space character is a separator except within a comment, a string literal, or a space character literal.
- Control characters other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.
- The end of a line is always a separator. It is understood to occur upon encountering the following conditions:
  - a Carriage Return, when it is not followed by a Line Feed;
  - a Carriage Return/Line Feed pair, regardless of what follows;
  - a Line Feed, when it is not followed by a Carriage Return;
  - a Line Feed/Carriage Return pair, regardless of what follows.

A **delimiter** is either one of the following special characters:

& ' ( ) * + , - . / : ; < = > |

or one of the following compound delimiters each composed of two adjacent special characters:

=> .. ** := /= >= <= << >> <>

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string literal, character literal, or numeric literal.

A **comment** starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a description.

An **identifier** is a character string composed of letters, digits and underline characters. All characters of an identifier are significant, including any underline character inserted between a letter or a digit and an adjacent letter or digit. Identifiers differing in the use of corresponding upper- and lowercase letters are considered to be the same.

A **numeric literal** is a character string, composed of letters, digits and underline characters, that represents a numeric value.

See reference [1] for a detailed definition of the lexical elements.

3.2 LOGICAL DESCRIPTIONS

3.2.1 OVERVIEW

In order to list some of the usual kinds of transported data, both contexts (telemetry and space mechanics) presented in the previous section are used to illustrate the information conveyed in EAST logical descriptions.

The logical part of an EAST description provides the information that is required by an application user to understand the data. Each data item is described using a programming language concept, called type. A type is a model, defined once, that is used to create many occurrences of the model.

A type has a name: this name, if well chosen, is a way to indicate the meaning of the model.

A type has a nature: the model may represents a whole number or a real number, or a character string, etc. The syntax used to define a type varies according to the nature of the type. Depending on the nature of the type, some additional information is given; in most of the cases, the list (or the range) of permitted values is defined.

There are two kinds of types:

- The basic types, also called atomic types, that are elementary types. EAST allows the definition of enumeration types (see 3.2.2), integer types (see 3.2.4) and real types (see 3.2.5). EAST provides a predefined basic type: character (see 3.2.3).

- The composite types, also called structuring types, that are composed of basic types or composite types. EAST allows the aggregation of elements with the definition of record types (see 3.2.6 and 3.2.11) and the repetition of elements with the definition of array types (see 3.2.7). EAST provides a predefined composite type: character string (see 3.2.3).

When defined, a type can be used to define other types:

- by aggregation for the definition of composite types;

- by restriction for the definition of subtypes (see 3.2.8).

A type can be also used to define occurrences of the data:

- subsection 3.2.9 describes the definition and the use of variables;

- subsection 3.2.10 describes the definition and the use of constants.

The logical part of an EAST data description is composed of two sections: one for the definition of the data models (also called section for the declaration of types) and one for the definition of data occurrences (also called section for the declaration of variables). The data models are defined using type, subtype and constant declarations, while the data occurrences are defined using variable and constant declarations.

The first definition of a variable delimits the two sections. Any declaration that occurs before the first variable definition belongs to the section for the declaration of types. Any declaration that occurs after the first variable definition (including the first variable declaration itself) belongs to the section for the declaration of variables.

The logical part is structured as follows:

Package name_of_logical_description_part is

Section for the Declaration of Types: Definition of the Data Models

-- declarations for types, subtypes and constants

Section for the Declaration of Variables: Definition of the Data Occurrences

-- declarations for variables and constants

End name_of_logical_description_part;

Some of the frequently asked questions about the logical description part of EAST data descriptions are summarized in 3.2.13.

3.2.2 ENUMERATION TYPES

Enumeration types are used each time the description of a limited set of values is needed.

In a telemetry context, a block of data usually begins with a synchronization signal. This signal consists of a specific bit pattern, which represents a value. This synchronization field may have, for example, a 16-bit length. It may be filled with two alternative values, for example one for a primary block (called PRIMARY_SYNCHRO) and one for a secondary block (called SECONDARY_SYNCHRO). This signal is described using an EAST enumeration type.

```
type SYNCHRONIZATION_VALUE is ( PRIMARY_SYNCHRO,
                                SECONDARY_SYNCHRO);
```

Example 3-1: Enumeration Type Declaration

In the example, "PRIMARY_SYNCHRO" represents the name (i.e., the meaning) given to one of the enumeration literals associated with the type. If there is a need to express the bit pattern of this synchronization value (for example: the hexadecimal value 0A08), the enumeration representation clauses of the EAST Syntax allow a user to formulate this integer value, in four possible ways(F1):

- 2#0000101000001000#[1] -- binary integer
- 2568 -- decimal integer
- 8#5010# -- octal integer
- 16#0A08# -- hexadecimal integer

In the same way, if the bit pattern of the other synchronization value, SECONDARY_SYNCHRO, is the hexadecimal value CD04, then this integer value can be expressed in four possible ways:

- 2#1100110100000100# -- binary integer
- 52484 -- decimal integer
- 8#146404# -- octal number
- 16#CD04# -- hexadecimal integer

NOTE – If no base is specified, 10 is the default base.

[1] The syntax is: base # value #

The following example illustrates enumeration representation clauses using hexadecimal values:

```
for SYNCHRONIZATION_VALUE use ( PRIMARY_SYNCHRO      => 16#0A08#,
                                SECONDARY_SYNCHRO    => 16#CD04#);
```

Example 3-2: Enumeration Representation Clauses Declaration

The EAST length clauses are used to specify the field length (in bits)[F2]. In the example, the synchronization signal is a 16-bit field:

```
for SYNCHRONIZATION_VALUE'size use 16; -- bits
```

Example 3-3: Length Clause Declaration

NOTE – 16 bits are necessary to code the hexadecimal value CD04.

The use of enumeration types can be extended to other applications. Each time a data item (variable) can have a limited set of values, an enumeration type is advisable. The definition of an enumeration type provides indeed the meaning of every value of the type. The use of enumeration types improves the semantic meaning of the descriptions (see 5.2).

As an example, in the figure 2-3, the Segmentation Flag field is used to indicate the status of a long message-oriented source packet that has been broken into shorter communications-oriented segments. This status may have four different values, corresponding to the four segment types: one value for a first segment, one value for a continuation segment, one value for a last segment and one value for an unsegmented packet. This status is described by the following EAST declarations:

```
type STATUS is ( CONTINUATION_SEGMENT,
                 FIRST_SEGMENT,
                 LAST_SEGMENT,
                 UNSEGMENTED_PACKET);
for STATUS'size use 2;
for STATUS use ( CONTINUATION_SEGMENT => 2#00#,
                 FIRST_SEGMENT => 2#01#,
                 LAST_SEGMENT => 2#10#,
                 UNSEGMENTED_PACKET => 2#11#);
```

Example 3-4: Complete Enumeration Type Definition

In the same example, the “Version Number” field is used to specify the packet format. At present only 2 versions of the packet are permitted, but 3 bits are reserved. This packet format is also described using an enumeration type, as follows:

```
type VERSION is (VERSION_1 , VERSION_2);
for VERSION'size use 3; -- bits
for VERSION use (VERSION_1 => 2#000# , VERSION_2 => 2#100#);
```

Example 3-5: Complete Enumeration Type Definition

NOTE – The field length of the enumeration type VERSION could have been specified with a minimal value: 1 bit is necessary to implement 2 enumeration literals. But it is not an absolute necessity to specify a minimal length clause. In this case, by adding new version numbers, future variations of the source packet structure become possible.

Enumeration literals may also be character literals. The following example defines the ROMAN_NUMERAL type:

```
type ROMAN_NUMERAL is (‘I’ , ‘V’ , ‘X’ , ‘L’ , ‘C’ , ‘D’ , ‘M’);
for ROMAN_NUMERAL use ( ‘I’ => 1, ‘V’ => 5, ‘X’ => 10, ‘L’ => 50, ‘C’ =>100,
                                    ‘D’ => 500, ‘M’ => 1000);
for ROMAN_NUMERAL'size use 16;
```

Example 3-6: Enumeration Type Declaration using Characters

Rules about the Usage of Enumeration Types

Rule 1 The enumeration literals listed in an enumeration type definition are identifiers or character literals. See section 3.2.1.2. of reference [1].

Rule 2 The size of an enumeration type must always be provided; i.e., a length clause is mandatory. See section 3.2.4.1. of reference [1].

Rule 3 An enumeration representation clause is optional. See section 3.2.4.2. of reference [1].

Rule 4(F3) If there is an enumeration representation clause, then each literal of the enumeration type must be provided with a unique bit pattern. The numeric value associated with this bit pattern must satisfy the ordering relation of the type (i.e., must increase). If no enumeration representation clause is provided, then default integer codes are presumed for binary encoded enumeration types: the value of the first listed enumeration literal is zero; the value for each other enumeration literal is one more than for its predecessor in the list. If no enumeration representation clause is provided, the enumeration type is maybe ASCII encoded according to the physical part of the EAST description (see 3.3.5). See sections 3.2.4.2. and 3.3.3.2 of reference [1].

A special case of enumeration types: the Booleans(F4)

EAST can be used to describe specific Boolean types, which gives more powerful semantic meaning than the classic "Boolean" type available in some programming languages. A classic Boolean type answers a question (it is TRUE or FALSE). But it is more meaningful to express the kind of question the Boolean answers. The following examples provide an illustration of the expressiveness of specific Boolean types:

```
type SATELLITE_STATUS is (HIDDEN , VISIBLE);
for SATELLITE_STATUS use (HIDDEN => 0 , VISIBLE => 1);
for SATELLITE_STATUS'size use 1; -- bit

type PRESENCE_FLAG is (ABSENT, PRESENT);
for PRESENCE_FLAG use (ABSENT => 0 , PRESENT => 1);
for PRESENCE_FLAG'size use 8; -- bits

type PACKET_TYPE is (TELEMETRY, TELECOMMAND);
for PACKET_TYPE use (TELEMETRY => 0 , TELECOMMAND => 1);
for PACKET_TYPE'size use 1; -- bit
```

Example 3-7: Some Substitutes to Boolean Types

NOTE – The PRESENCE_FLAG is an enumeration type requiring 1 bit, but the size is given as 8 bits: this is a case of a forced size.

3.2.3 CHARACTER TYPES AND CHARACTER STRING TYPES

EAST provides the predefined type “CHARACTER”. This type has been defined as an eight-bit coded enumeration type composed of the 256 ISO8859-1 (Latin Alphabet No. 1) coded characters, containing 191 printable characters. The full character set is defined in Annex B of the EAST Specification document (reference [1]) and in the Latin Alphabet No. 1 document (reference [6]).

NOTE – A predefined EAST type is a type provided by EAST that can be used without having been declared.

Other character types may be derived (or subtyped) from the predefined EAST CHARACTER type. For example, a character type which only accepts capital letters is defined as follows:

```
subtype CAPITAL_LETTER is CHARACTER range ‘A’ .. ‘Z’;
```

Example 3-8: Character Type Declaration

For the description of character strings, EAST provides the predefined type “STRING”. A possible way to use the predefined EAST STRING type is to subtype it, i.e., to rename it and optionally to specify the size of the actual character string.

In the next example, the EAST declaration defines a 32 character string type:

```
subtype NAME is STRING (1 .. 32);
```

Example 3-9: Character String Type Declaration

More explanations about subtypes are provided in 3.2.8.

Rule 5 The types CHARACTER and STRING[F5] do not have to be declared in a data description. They are predefined types of EAST. See section 3.2.1.1. of reference [1].

3.2.4 INTEGER TYPES

Integer types are used to describe whole numbers.

The definition of an integer type specifies the range of values taken into account. In the first example (see figure 2-3), the “Source Sequence Count” field is a 14-bit field, which contains a straight sequential count (modulo 16384) of each generated packet. Such a counter is described using an EAST integer type as follows:

```
type COUNTER is range 0 .. 16383;
```

Example 3-10: Integer Type Declaration

In this example, 0 is the lower bound, i.e., the minimum value of the type, and 16383 is the upper bound, i.e., the maximum value of the type.

The EAST length clauses must be used to specify the integer size:

```
for COUNTER'size use 14; -- bits
```

Example 3-11: Length Clause Declaration

Data products are often dated. The CCSDS recommends standard dates and times. For example, the DATE_YMD provides the year, the month and the day within the Gregorian calendar. The binary representation of this type is described using integer types as follows:

```
type YEAR is range 0 .. 9999;
for YEAR'size use 16;

type MONTH is range 1 .. 12;
for MONTH'size use 8;

type DAY is range 1 .. 31;
for DAY'size use 8;
```

Example 3-12: Complete Integer Type Declarations

Rule about the Usage of Integer Types

Rule 6 The size of an integer type must always be specified. See section 3.2.4. of reference [1].

3.2.5 REAL TYPES

The definition of a real type specifies the number of significant digits and may specify additionally the range of values taken into account. The EAST length clauses must be used to specify the real size.

In the figure 2-4, the orbit location is defined by the semi-major axis in kilometers, the eccentricity, the inclination in degrees, the argument of perigee in degrees, the right ascension in ascending node in degrees and the true anomaly in degrees. All these measurements are described using floating point reals. But the range of values or the precision of the measurements is not the same for kilometers or degrees, as shown in the following example:

```
type ANGULAR_DEGREE is digits 8 range -180.0 .. 180.0;-- the range is specified
for ANGULAR_DEGREE'size use 64; -- bit

type KILOMETERS is digits 15;    -- the range is not specified
for KILOMETERS'size use 64;
```

Example 3-13: Complete Real Type Declarations

NOTE – If the real type declaration specifies no range, the range is by default the largest range that can be implemented within the specified number of bits accommodating the number of significant digits. The default range also depends on the convention used to represent the binary values of the real types (see 3.3.4).

Rule about the Usage of Real Types

Rule 7 The size of a real type must always be specified. See section 3.2.4. of reference [1].

3.2.6 RECORD TYPES

As it is often necessary to aggregate different types of data, EAST provides a concept of record to structure “atomic” data items or even other aggregations. The aggregated data items are explicitly named, when they are used in a record type definition. Every component of an aggregation is therefore declared as follows:

Data_Instance_Name: Data_Type_Name;

NOTE – If the Data_Type_Name corresponds to an array type for which the number of elements is not specified at definition time, then the Data_Type_Name is followed in the component declaration by explicit indices that specify the actual number of elements (see 3.2.7)

In the case of the figure 2-3, the leftmost branch of the tree would be:

```
-- “Atomic” types
type VERSION is (VERSION_1 , VERSION_2);
for VERSION'size use 3;
for VERSION use (VERSION_1 => 2#000#, VERSION_2 => 2#100#);

type PACKET_TYPE is (TELEMETRY , TELECOMMAND);
for PACKET_TYPE'size use 1;
for PACKET_TYPE use (TELEMETRY => 0 , TELECOMMAND => 1);

type PRESENCE_FLAG is (ABSENT , PRESENT);
for PRESENCE_FLAG'size use 1;
for PRESENCE_FLAG use (ABSENT => 0, PRESENT => 1);

type PROCESS_IDENTIFICATION is (WORKING , IDLE);
for PROCESS_IDENTIFICATION'size use 11;
for PROCESS_IDENTIFICATION use (   WORKING => 2#00000000000#,
                                   IDLE => 2#11111111111);
                                   .../...
```

Example 3-14: Record Type Declaration (1 of 2)

```
                                .../...
-- Structuring type
type PACKET_IDENTIFICATION_TYPE is
        record
                VERSION_NUMBER : VERSION;
                PACKET : PACKET_TYPE;
                SECONDARY_HEADER_FLAG : PRESENCE_FLAG;
                APPLICATION_PROCESS_ID : PROCESS_IDENTIFICATION;
        end record;
for PACKET_IDENTIFICATION_TYPE'size use 16;
```

Example 3-14: Record Type Declaration (2 of 2)

NOTE – As for atomic data type specification (enumeration, integer and real types) a length clause specifies the size of a record.

Within a structuring type, it is also possible to specify that a kind of data is present in some cases. For example, the SECONDARY_HEADER of figure 2-3 is only present if the SECONDARY_HEADER_FLAG has the value PRESENT:

```
type PACKET_HEADER (
        SECONDARY_HEADER_FLAG: PRESENCE_FLAG := PRESENT ) is
        record
                PRIMARY_HEADER: PRIMARY_HEADER_TYPE;
                -- see previous record type declaration
                case SECONDARY_HEADER_FLAG is
                        when PRESENT =>
                                SECONDARY_HEADER: SECONDARY_HEADER_TYPE;
                        when ABSENT =>
                                null; -- no corresponding field
                end case;
        end record;
```

Example 3-15: Record Type Declaration with Optional Field

NOTES

1 In this example, the instance SECONDARY_HEADER_FLAG of the type PRESENCE_FLAG (that has the default value PRESENT) determines (i.e., discriminates) the presence of another component (SECONDARY_HEADER). This component is called a discriminant.

2 In this example, the length of the record depends on the value of "SECONDARY_HEADER_FLAG" (the discriminant). In one case, the length is the length of the discriminant + the length of the primary header; in the other one, the length of the discriminant + the length of the primary header + the length of the secondary header. No length clause is therefore provided for the record.

Rules about the usage of record types

Rule 8 A component on which depends the existence of other components is called a discriminant for the record type. The alternative lists of components are called variants of the record. See section 3.2.1.6. of reference [1].

Rule 9 A length clause must be provided for a record, every time it is possible. In some cases, no length clause can be provided for the record, because the length is undefined. See section 3.2.4.1. of reference [1].

The EAST syntax requires a default value for each discriminant (if any) in a record type declaration. A default value does not preclude any possible value for the discriminant of the actual data. In the case of the type “PACKET_HEADER”, the default value could have been ABSENT or PRESENT (or in fact any allowed value for the enumeration type “PRESENCE_FLAG”).

Another rule about the usage of record types

Rule 10 If a record contains one or more discriminants, it is mandatory to provide a default discriminant value for each of them. See section 3.2.1.6. of reference [1].

3.2.7 ARRAY TYPES

Array types are used to describe repetitions. If a telemetry contains a repetition of a certain number of measurements or blocks of measurements, this number being either a constant or a variable, this repetition would be defined using an array type. The next example defines an array type which has a constant number of elements:

```
type MEASUREMENT is digits 4; -- real type with a precision of 4 significant digits
for MEASUREMENT'size use 32;

type TEN_MEASUREMENT_BLOCK_TYPE
      is array (1 .. 10) of MEASUREMENT;
for TEN_MEASUREMENT_BLOCK_TYPE'size use 320; -- 10*32 bits
```

Example 3-16: Array Type Declaration with a Constant Number of Elements

In this example, a length clause specifies the size of the array, because its size is known; i.e., the array has a constant number of elements. The declaration of an instance of this type is:

```
BLOCK : TEN_MEASUREMENT_BLOCK_TYPE;
```

Example 3-17: Array Instance Declaration

If there is a need to describe data of the same type for which the number of occurrences varies, then an array with unlimited size (specified with “range <>”) is the suitable structure. The next example defines an array type which has a variable number of elements:

```
type NUMBER is range 0 .. 65535;
for NUMBER'size use 16;

type UNLIMITED_MEASUREMENT_BLOCK_TYPE
      is array (NUMBER range <>) of MEASUREMENT;
```

Example 3-18: Array Type Declaration with a Variable Number of Elements

NOTES

1 In this example, no length clause specifies the size of the array, because its size is not known.

2 The expression “NUMBER range <>” means that the range (or the index) of the array in an instance of this array type will be defined by a range of NUMBER (i.e., by two values of the type NUMBER which specify the bounds of the array index: “1 .. 10” or “1 .. 100” in example 3-19).

Using this array type declaration, possible declarations would be:

```
A_10_MEASUREMENT_BLOCK :
      UNLIMITED_MEASUREMENT_BLOCK_TYPE (1 .. 10);

A_100_MEASUREMENT_BLOCK :
      UNLIMITED_MEASUREMENT_BLOCK_TYPE (1 .. 100);
```

Example 3-19: Array Instance Declarations

When the number of elements of an array is specified "at run time" within the data block, the suitable EAST structure to describe it is a record type including a component that specifies the size of the array and the array itself.

```
-- atomic type declarations
type NUMBER is range 0 .. 65535;
for NUMBER'size use 16;

type MEASUREMENT is digits 4;
for MEASUREMENT'size use 32;

-- array type definition
type UNLIMITED_MEASUREMENT_BLOCK_TYPE is array
      (NUMBER range <>) of MEASUREMENT;

-- structuring type definition
type DATA_SET(NUMBER_OF_ELEMENTS : NUMBER := 1) is record
      DATA : UNLIMITED_MEASUREMENT_BLOCK_TYPE
            (1 .. NUMBER_OF_ELEMENTS);
end record;
```

Example 3-20: Use of an Array Type with a Variable Number of Elements

The declaration of a datum of this type is:

```
A_DATA_SET : DATA_SET;(F6)
```

Example 3-21: Array Instance Declaration

NOTE – The instance A_DATA_SET corresponds to a data item of the type DATA_SET, the number of elements being not known "at definition time" but only specified at "run time". A data instance is composed of a number of elements and measurements (as many as specified by the leading number).

A special use of arrays with variable number of elements

When unlimited array types (also called unconstrained array types) are limited in a declaration, null arrays can serve to specify that there is no data of this particular type within a given data set. To declare null arrays, the value of the lower bound of the array index has to be greater than the upper bound.

Using the record type declaration of the previous example:

```
A_DATA_SET : DATA_SET(0);   -- zero element
```

Example 3-22: Null Array Declaration

NOTE – In this case, the lower bound (1) of the index of the array DATA is greater than the upper bound (0), which means that the array has no component. Note that the type NUMBER must allow values less than the lower bound of the array index.

Rules about the usage of array types

Rule 11 A length clause must be provided for an array, every time it is possible. For unconstrained array types, no length clause can be provided because they have an undefined number of elements. The number of elements is specified at the declaration of a data of this type. See section 3.2.4.1. of reference [1].

Rule 12 In the case of an unconstrained array, the constraint (i.e., the number of elements) is given to the instance at its declaration. See section 3.2.1.5. of reference [1].

Rule 13 If the lower bound of an index range is greater than the upper bound, the corresponding array row/column has no component. See section 3.2.1.5. of reference [1].

3.2.8 SUBTYPES

EAST allows the specification of user types to describe families of data. EAST provides a facility to define sub-families, i.e., families of the same type but with a different spectrum of data. In other words, EAST allows the definition of subtypes to restrict the set of values of the initial type. The subtyping can be used to rename types, if the initial set of values is not restricted in the subtype declaration. See below some examples:

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
for DAY'size use 8;
subtype RESTING_DAY is DAY range SAT .. SUN;
subtype WORKING_DAY is DAY range MON .. FRI;

type DEGREE is digits 15;
for DEGREE'size use 64;
subtype ANGULAR_DEGREE is DEGREE range -180.0 .. 180.0;
subtype CELSIUS_DEGREE is DEGREE range -273.0 .. 100000000000000.0;

subtype IDENTIFIER is STRING; -- renaming subtype
```

Example 3-23: Subtype Declarations

NOTES

1 No length clauses are provided in subtype definitions.

2 A subtype is considered to be a type, so it is permissible to subtype subtypes.

3.2.9 VARIABLES

Variables are used to name explicitly the exchanged data. Variables are declared using types that have been previously declared.

There is a different use of type declarations and variable (data) declarations. A type declaration is used to describe the kind of a “generic datum”. The declaration of a variable is used to explicitly declare a physical occurrence of a datum of this type. For example, the declaration of ANGULAR_DEGREE, which is a 64 bit real type, can be used to declare many data occurrences (INCLINATION, TRUE_ANOMALY, etc.).

```
-- declaration of types
type ANGULAR_DEGREE is digits 8 range -180.0 .. 180.0;
for ANGULAR_DEGREE'size use 64;
type KILOMETERS is digits 15;
for KILOMETERS'size use 64;

-- declaration of variables
SEMI_MAJOR_AXIS : KILOMETERS;
INCLINATION : ANGULAR_DEGREES;
ARGUMENT_OF_PERIGEE : ANGULAR_DEGREES;
RIGHT_ASCENSION_IN_ASCENDING_NODE : ANGULAR_DEGREES;
TRUE_ANOMALY : ANGULAR_DEGREES;
```

Example 3-24: Declaration of Variables

3.2.10 CONSTANTS

EAST provides a facility for expressing constants. Constants of any type may be declared. A constant is not a type. A constant has a static value. See below some examples of constant declarations:

```
-- type declarations
type PACKET_TYPE is (TELEMETRY , TELECOMMAND);
for PACKET_TYPE use (TELEMETRY => 0 , TELECOMMAND => 1);
for PACKET_TYPE'size use 1;

type NUMBER is range 0 .. 65535;
for NUMBER'size use 16;

-- constant declarations
DORIS_PACKET : constant PACKET_TYPE := TELEMETRY;
        -- an enumeration constant
MAX_NUMBER : constant NUMBER := 255;
        -- an integer constant
```

Example 3-25: Constant Declarations

Constants can be used mainly for two purposes:

- as range bounds in type or subtype definitions, or in other constant declarations (the final purpose being to be used as range bounds); in this case, they are declared in the section of type declarations;

- as markers (i.e., end-delimiters of repetitions) when defined in the section for the declaration of variables.

3.2.10.1 Use of constants in the section of type declarations

The constant MAX_NUMBER, defined in the example 3-25, can be used in other type or constant definitions as follows:

```
type MEASUREMENT_BLOCK_TYPE
      is array (1 .. MAX_NUMBER) of MEASUREMENT;
      -- use of the constant in an array type declaration

type MEASUREMENT_COUNTER is range 0 .. MAX_NUMBER;
      -- use of the constant in an integer type declaration

MEAN_NUMBER : constant NUMBER := (MAX_NUMBER + 1) / 2;
      -- use of a constant in another constant declaration
```

Example 3-26: Use of Constants (1)

A constant that is declared in the section of type declarations is either an integer constant, a real constant or an enumeration constant.

A number declaration is a special form of a constant declaration with no specified type. See below some examples:

```
PI : constant := 3.1415926536;      -- a real number
ZERO : constant := 0;               -- an integer number
```

Example 3-27: Number Declarations

A number can be used in a constant definition as follows:

```
RIGHT_ANGLE : constant := PI/2;
      -- use of a constant in another constant declaration
```

Example 3-28: Use of Constants (2)

Non-typed constants (or numbers) can also be used in range definitions. It is an EAST facility for the definition of types. The following definitions,

```
MIN : constant := 1;
MAX : constant := 255;
type VALUE is range MIN .. MAX;
```

Example 3-29: Use of Non-typed Constants (1)

are strictly equivalent to the following definition:

```
type VALUE is range 1 .. 255;
```

Example 3-30: Use of Non-typed Constants (2)

No implementation types are associated with these numbers. In the previous examples, the main difference between MAX_NUMBER (defined in the example 3-25) and MAX (defined in the example 3-29) is the knowledge or non-knowledge of the representation of the values: MAX_NUMBER is a 16-bit unsigned integer, while MAX is physically undefined (i.e., is just a logical concept).

3.2.10.2 Use of constants in the section for the declaration of variables

A constant can be used as a marker when its definition occurs after a declaration of a variable. In this case, the following convention is applicable:

Rule 14 The variable that is declared immediately before the constant occurs an undetermined number of times, the last instance being followed by the constant value. See section 3.2.3.2.2. of reference [1].

The following example illustrates the use of markers:

```
-- Section of Data Type Declarations
type DEGREE is digits 15;
for DEGREE’size use 64;
.../...

-- Section of Data Occurrence Declarations
MEASUREMENT : DEGREE;
END_OF_MEASUREMENT_BLOCK : constant STRING := “END”;
```

Example 3-31: Use of Constants as Markers

Figure 3-1 represents the data described by the previous EAST description:

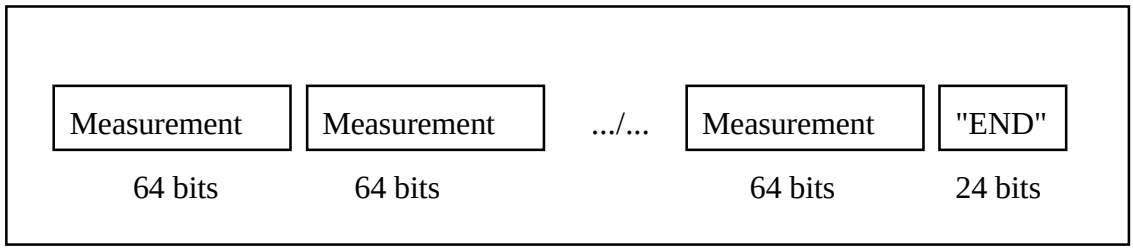


Figure 3-1: Data Block ended by a Marker

A constant that is declared in the section for the declaration of variables is an integer constant, an enumeration constant, a character constant or a character string constant.

NOTE – Real constants are not allowed as markers because markers should be unambiguously recognized. The representation of a real value is a floating point representation, and is as such an approximation of the original value.

A special case of markers : EOF

A special marker is the "end of file". This marker is encountered at the end of the data file. It does not correspond to any bit pattern to be found in the data. File management systems do not indeed restore the "end of file".

It is encountered at the end of the data file. The following convention is adopted: the type of the Marker is an EAST predefined type, called EOF. No explicit value is associated with this constant since this value is unknown. This is the only case of a constant declaration where the value is absent.

NOTE – The EOF marker can only be used once in an EAST description. It is the last declaration of the logical description part.

The next example presents the description of a data file that contains a header and n values (n, being undetermined).

```
HEADER : HEADER_TYPE; -- any type

VALUE : COEFFICIENT;  -- COEFFICIENT is a real type defined in 3.2.1.4 as:
                      -- type COEFFICIENT is digits 10 range 0.0 .. 1.0;

END_OF_COEFFICIENTS : constant EOF ;
```

Example 3-32: EOF Marker Declaration

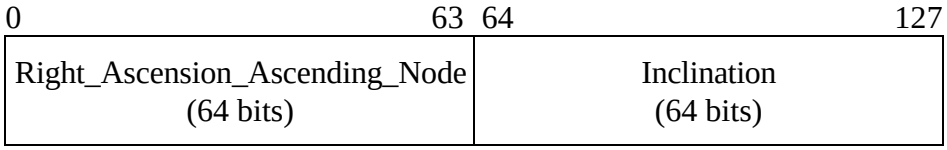
Only typed constants are allowed as markers.

3.2.11 RECORD REPRESENTATION CLAUSES

Subsection 3.2.6 recommends record types for the specification of structured data. In order to specify the exact location of data items in a record, EAST provides record representation clauses[F7].

A record representation clause specifies the storage representation of the record, that is, the relative position and the size of the record components (including discriminants if any).

The following example illustrates a simple kind of record and its associated representation clauses:



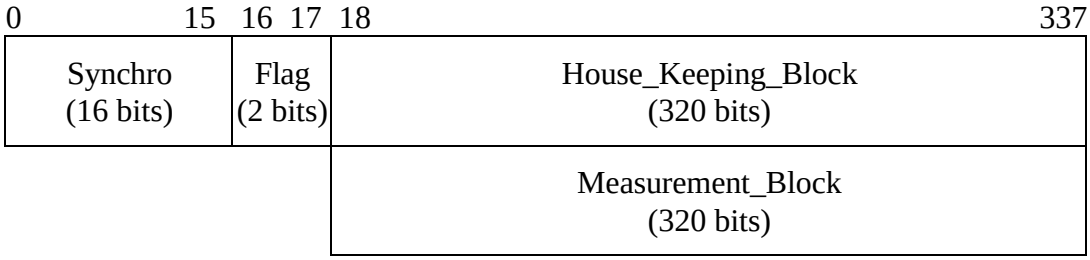

```
type DEGREE is digits 15;
for DEGREE'size use 64;

type ORBIT_PLAN_LOCATION is record
      RIGHT_ASCENSION_ASCENDING_NODE : DEGREE;
      INCLINATION : DEGREE;
end record;
for ORBIT_PLAN_LOCATION use
      record
            RIGHT_ASCENSION_ASCENDING_NODE      at 0 range 0 .. 63;
            INCLINATION                         at 0 range 64 .. 127;
      end record;
for ORBIT_PLAN_LOCATION'size use 128; -- bits
```

Example 3-33: Complete Record Type Declaration

This example illustrates the fact that there is no gap between components of a record, and that the component locations do not overlap.

NOTE – The expression “at 0” in the component locations means that the range that follows the expression is specified relatively to the beginning (i.e., location 0) of the record. More explanations about this expression are provided on page 3-32.

The next example shows that component locations may overlap if they do not belong to the same alternative list of components:



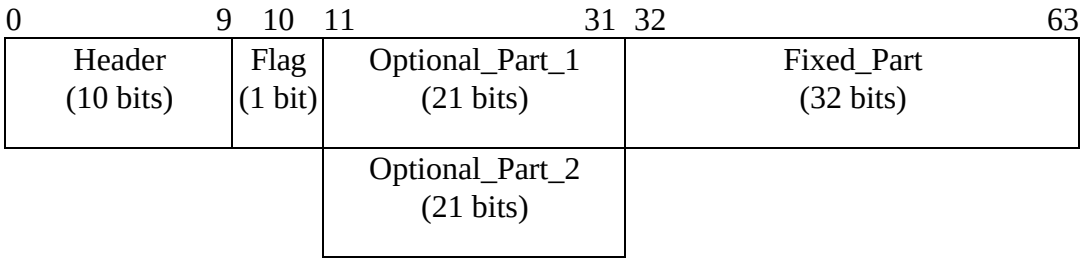

```
type ACTIVITY_FLAG is (HOUSE_KEEPING, MEASUREMENT, INCIDENT);
for ACTIVITY_FLAG'size use 2;

type STRUCTURE (FLAG : ACTIVITY_FLAG := MEASUREMENT) is record
        SYNCHRO: SYNCHRONIZATION_VALUE;
        case FLAG is
                when HOUSE_KEEPING =>
                        HOUSE_KEEPING_BLOCK : BLOCK_TYPE;
                when MEASUREMENT =>
                        MEASUREMENT_BLOCK : BLOCK_TYPE;
                when others =>
                        null;
        end case;
end record;
for STRUCTURE use
record
        SYNCHRO at 0 range 0 .. 15;
        FLAG at 0 range 16 .. 17;
        HOUSE_KEEPING_BLOCK at 0 range 18 .. 337;
        MEASUREMENT_BLOCK at 0 range 18 .. 337;
end record;
for STRUCTURE'size use 336; -- bits
```

Example 3-34: Complete Record Type Declaration with Variants

The clause “when others =>” is mandatory if all the discriminant values are not explicitly named in the record type definition. It represents all the other discriminant values. In other words, all the occurrences of the discriminant must be named, either explicitly or implicitly, in a case statement.

NOTE – In this example, “when others =>” could have been replaced by “when INCIDENT =>”.

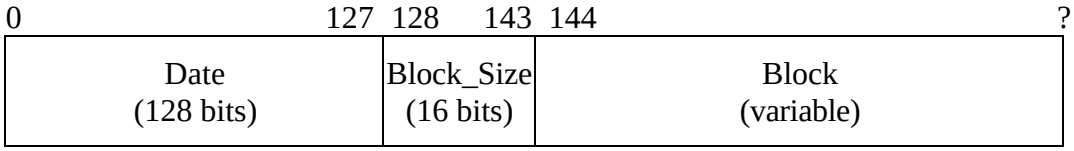
One of the most interesting uses of record representation clauses is illustrated in the following example, which explains that component representation clauses do not have to appear in the same order as the declaration order. The EAST Syntax requires indeed the fixed elements to be declared before the optional ones in a structure; nevertheless, in record representation clauses, one or more elements of the fixed part are allowed to be put after a variant part, if and only if the variant part has a constant length.




```
type STATUS is (OPEN, CLOSED);
for STATUS'size use 1;

type STRUCTURE ( FLAG : STATUS := OPEN) is record
        HEADER : HEADER_TYPE;
        FIXED_PART : FIXED_PART_TYPE;
        case FLAG is
                when OPEN =>
                        OPTIONAL_PART_1 : OPTIONAL_PART_1_TYPE;
                when CLOSED =>
                        OPTIONAL_PART_2 : OPTIONAL_PART_2_TYPE;
        end case;
end record;
-- But to describe the actual data type,
-- the use of record representation clauses is necessary.
for STRUCTURE use
record
        HEADER at 0 range 0 .. 9;
        FLAG at 0 range 10 .. 10;
        OPTIONAL_PART_1 at 0 range 11 .. 31;
        OPTIONAL_PART_2 at 0 range 11 .. 31;
        FIXED_PART at 0 range 32 .. 63;
end record;
for STRUCTURE'size use 64; -- bits
```

Example 3-35: Use of Record Representation Clauses

The next example illustrates a record for which the size is not known a priori, and for which the record representation clause is partly provided.



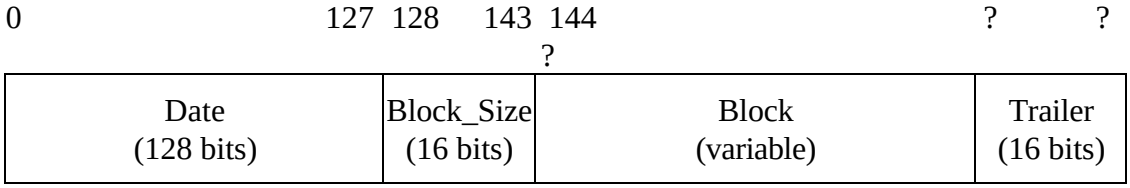

```
type STRUCTURE ( BLOCK_SIZE : NUMBER := 1) is record
      DATE : DATE_FORMAT;
      BLOCK : UNLIMITED_MEASUREMENT_BLOCK_TYPE (1 .. BLOCK_SIZE);
end record;

for STRUCTURE use
record
      DATE at 0 range 0 .. 127;
      BLOCK_SIZE at 0 range 128 .. 143;
end record;
```

Example 3-36: Incomplete Record Representation Clause (1) Declaration

In this case, a representation clause cannot be given for the block, because the size (i.e., the location of the end of the block) is not known a priori. A length clause cannot therefore be provided.

By default, all components of a data block are contiguous. In this case, no representation clause is provided for the component BLOCK, but its location begins by default at bit 144.

The next example is another illustration of a record for which the record representation clause is partly provided.



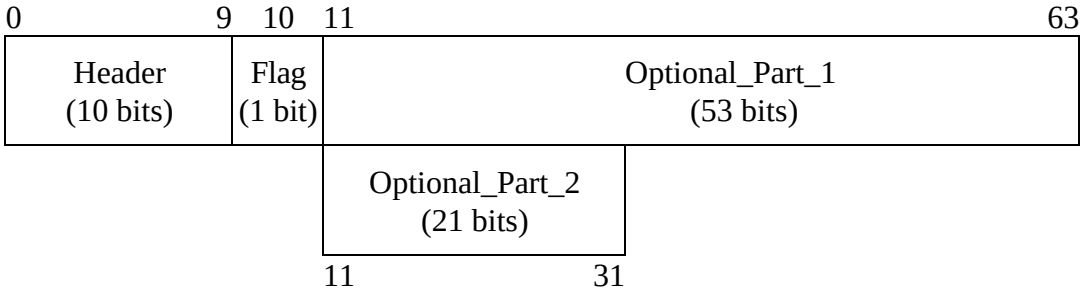

```
type STRUCTURE ( BLOCK_SIZE : NUMBER := 1) is record
      DATE : DATE_FORMAT;
      BLOCK : UNLIMITED_MEASUREMENT_BLOCK_TYPE (1 .. BLOCK_SIZE);
      TRAILER : TRAILER_TYPE;
end record;

for STRUCTURE use
record
      DATE at 0 range 0 .. 127;
      BLOCK_SIZE at 0 range 128 .. 143;
end record;
```

Example 3-37: Incomplete Record Representation Clause (2) Declaration

As in the previous example, a representation clause cannot be given for the block, because the size is not known a priori. A representation clause cannot be given for the trailer also, because its exact location is not known at definition time. Its size is known (16 bits) but it follows a component which has no representation clause. The trailer begins when the block ends.

In this case, the order of the components is not fully determined by the record representation clause. The order of the components, for which a representation clause is missing, is determined according to the order of these components within the record type definition.

The following example illustrates a record for which the size is not known a priori, but for which a record representation clause completely specifies the storage representation of the record.

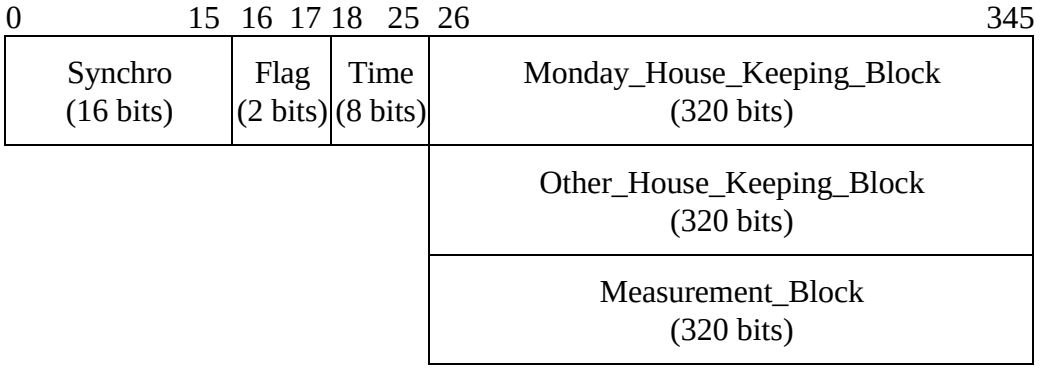



```
type STATUS is (OPEN, CLOSED);
for STATUS'size use 1;

type STRUCTURE ( FLAG : STATUS := OPEN) is record
        HEADER : HEADER_TYPE;
        case FLAG is
                when OPEN =>
                        OPTIONAL_PART_1 : OPTIONAL_PART_1_TYPE;
                when CLOSED =>
                        OPTIONAL_PART_2 : OPTIONAL_PART_2_TYPE;
        end case;
end record;

for STRUCTURE use
record
        HEADER at 0 range 0 .. 9;
        FLAG at 0 range 10 .. 10;
        OPTIONAL_PART_1 at 0 range 11 .. 63;
        OPTIONAL_PART_2 at 0 range 11 .. 31;
end record;
```

Example 3-38: Complete Record Representation Clause Declaration

A length clause cannot be provided for the whole structure, because the size of the optional part is not known a priori (53 bits or 21 bits).

A component can depend, at the same time, on the values of many discriminants. The next example illustrates a record with two discriminants. Some components depend, at the same time, on the values of two enumeration components.




```
type ACTIVITY_FLAG is (HOUSE_KEEPING, MEASUREMENT, INCIDENT);
for ACTIVITY_FLAG'size use 2;

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
for DAY'size use 8;

type STRUCTURE (    FLAG : ACTIVITY_FLAG := MEASUREMENT;
                    TIME : DAY := MON ) is record
      SYNCHRO: SYNCHRONIZATION_VALUE;
      case FLAG is
            when HOUSE_KEEPING =>
                  case TIME is
                        when MON =>
                              MONDAY_HOUSE_KEEPING_BLOCK : BLOCK_TYPE;
                        when TUE .. SUN =>
                              OTHER_HOUSE_KEEPING_BLOCK : BLOCK_TYPE;
                  end case;
            when MEASUREMENT=>
                  MEASUREMENT_BLOCK : BLOCK_TYPE;
            when others =>
                  null;
      end case;
end record;
                                                .../...
```

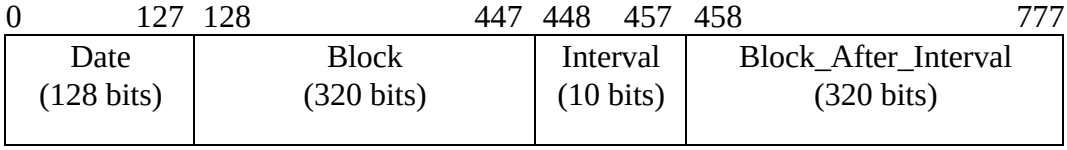
Example 3-39: Complete Record Type Declaration with 2 Discriminants (1 of 2)

```
                                .../...
for STRUCTURE use
record
        SYNCHRO at 0 range 0 .. 15;
        FLAG at 0 range 16 .. 17;
        TIME at 0 range 18 .. 25;
        MONDAY_HOUSE_KEEPING_BLOCK at 0 range 26 .. 345;
        OTHER_HOUSE_KEEPING_BLOCK at 0 range 26 .. 345;
        MEASUREMENT_BLOCK at 0 range 26 .. 345;
end record;
for STRUCTURE'size use 346; -- bits
```

Example 3-39: Complete Record Type Declaration with 2 Discriminants (2 of 2)

When the activity is House Keeping, the House Keeping Block varies with the Time values. When the activity is Measurement, the Measurement Block does not depend on the Time. When the activity is Incident, nothing else is present in the data.

The storage location of a component, relative to the start of the record, has been expressed until now in bits (the expression after the keyword at has been set to 0). For large structures, the values of expressions given after the reserved word range can be huge (see next example).




```
type STRUCTURE is record
        DATE : DATE_FORMAT;
        BLOCK : MEASUREMENT_BLOCK_TYPE;
        INTERVAL: TIME_COUNTER;
        BLOCK_AFTER_INTERVAL: MEASUREMENT_BLOCK_TYPE;
end record;
```

```
for STRUCTURE use
record
        DATE at 0 range 0 .. 127;
        BLOCK at 0 range 128 .. 447;
        INTERVAL at 0 range 448 .. 457;
        BLOCK_AFTER_INTERVAL at 0 range 458 .. 777;
end record;

for STRUCTURE 'size use 778; -- bits
```

Example 3-40: Big Record Type Declaration

So the EAST syntax also allows one to express the relative position of a component in a distance, called *word*, to which a number of bits is added. EAST allows two units for the distance: a 16-bit word or a 32-bit word.

The location of BLOCK_AFTER_INTERVAL of the last example is:

- the 459th bit; or
- the 11th bit in the word 28 (i.e., in the 29th word), if the chosen unit is a 16-bit word; or
- the 11th bit in the word 14 (i.e., in the 15th word), if the chosen unit is a 32-bit word.

The distance is specified using two predefined identifiers: WORD_16_BITS and WORD_32_BITS(F8) that always represent 16 bits, respectively 32 bits, on any architecture.

Using this facility, the previous example becomes:

```
type STRUCTURE is record
        DATE : DATE_FORMAT;
        BLOCK : MEASUREMENT_BLOCK_TYPE;
        INTERVAL: A_10_BIT_INTEGER;
        BLOCK_AFTER_INTERVAL : MEASUREMENT_BLOCK_TYPE;
end record;

for STRUCTURE use
record
        DATE at 0 * WORD_32_BITS range 0 .. 127;
        BLOCK at 4 * WORD_32_BITS range 0 .. 319;
        INTERVAL at 14 * WORD_32_BITS range 0 .. 10;
        BLOCK_AFTER_INTERVAL at 14 * WORD_32_BITS range 11 .. 331;
end record;

for STRUCTURE 'size use 778; -- bits
```

Example 3-41: Big Record Type Declaration Using Word Facility

Rules about the usage of record representation clauses

Rule 15 The clause "when others =>" is mandatory if all the discriminant values are not explicitly named in the record type definition. See section 3.2.16. of reference [1].

Rule 16 Component locations must not overlap, except if the components belong to distinct variants (i.e., belong to different alternative lists of components). See section 3.2.4.3. of reference [1].

Rule 17 The EAST Syntax requires the declaration of the fixed elements before the optional ones in a structure. See section 3.2.4.3. of reference [1].

Rule 18 Record representation clauses allow one or more elements of the fixed part to be placed after a variant part, if and only if the variant part has a constant length. See section 3.2.4.3. of reference [1].

Rule 19 A record representation clause must be provided every time it is possible. For variable components, representation clauses cannot be provided[2]. See section 3.2.4.3.of reference [1].

Rule 20 The order of record components is determined by the record representation clause. If the record representation clause is incomplete, the order of the components that have no representation clause is determined from the order within the record type definition. See section 3.2.4.3. of reference [1].

3.2.12 VIRTUAL COMPONENTS

As seen in 3.2.6, record types represent structured data. Some of these structures are variable. The variable part might be located deep in the structure, so that it is hidden from the root of the structure. It may be informative to announce, at the root level, that a structure is variable and what causes its variability. This is achieved by the use of discriminants (see 3.2.6).

But the duplication of discriminants at the highest level implies the presence, in the exchanged data, of data occurrences corresponding to the duplicated discriminants (see 3.2.9). To satisfy this need in spite of the implications, EAST proposes to prefix such discriminants with "VIRTUAL_". The following rule is then applicable:

Rule 21 Each component identifier which begins with "VIRTUAL_" does not represent any data occurrence. See section 3.2.1.6. of reference [1].

---

[2] The EAST Interpreter, used to access the data, has to compute the size of dynamic variables.

NOTE – In consequence, the use of the prefix "VIRTUAL_" is reserved for this specific use.

The example, presented in section 2 and taken from the telemetry context, illustrates the use of virtual components. It presents a packet, which may be considered as a data block, containing a primary header, an optional secondary header, a source data block and some other data. The primary header is composed of a packet identification block, a sequence control block and the source data length. The packet identification may be precisely described by the version number, the type identification, the secondary header flag and the application process identification, etc.

Some components discriminate the presence or the size of other components: the secondary header flag discriminates the presence of the secondary header, and the source data length discriminates the size of the source data that are exchanged.

Figure 3-2 presents a packet structure.

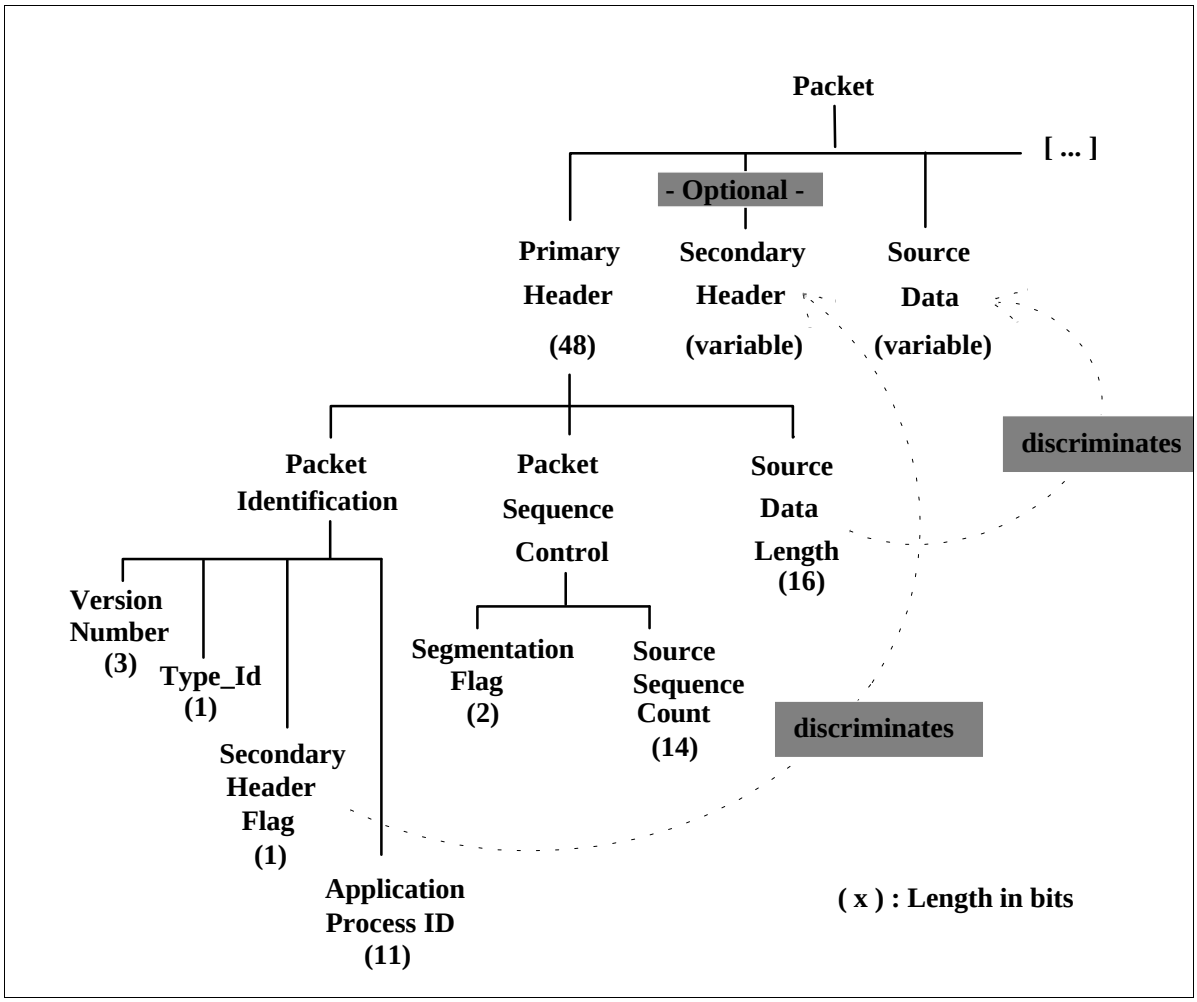


Figure 3-2: Discriminants in Version 1 "Source Packet" Format

This description can be formalized using EAST as follows:

```
-- basic data types used in the first branch
type VERSION is (VERSION_1, VERSION_2);
for VERSION use (VERSION_1 => 2#000#, VERSION_2 => 2#100#);
for VERSION'size use 3;

type PACKET_TYPE is (TELEMETRY , TELECOMMAND);
for PACKET_TYPE use (TELEMETRY => 0, TELECOMMAND => 1);
for PACKET_TYPE'size use 1;

type PRESENCE_FLAG is (ABSENT , PRESENT);
for PRESENCE_FLAG use (ABSENT => 0 , PRESENT => 1);
for PRESENCE_FLAG'size use 1;

type PROCESS_IDENTIFICATION is (WORKING , IDLE);
for PROCESS_IDENTIFICATION'size use 11;
for PROCESS_IDENTIFICATION use (    WORKING => 2#00000000000#,
                                    IDLE => 2#11111111111);

-- structuring type for the Packet Identification
type PACKET_IDENTIFICATION_TYPE is record
        VERSION_NUMBER : VERSION;
        TYPE_ID : PACKET_TYPE;
        SECONDARY_HEADER_FLAG : PRESENCE_FLAG;
        APPLICATION_PROCESS_ID : PROCESS_IDENTIFICATION;
end record;
for PACKET_IDENTIFICATION_TYPE use
record
        VERSION_NUMBER at 0 range 0 .. 2;
        TYPE_ID at 0 range 3 .. 3;
        SECONDARY_HEADER_FLAG at 0 range 4 .. 4;
        APPLICATION_PROCESS_ID at 0 range 5 .. 15;
end record;
for PACKET_IDENTIFICATION_TYPE'size use 16;
                                    .../...
```

Example 3-42: EAST logical description of Version 1 “Source Packet” Format (1 of 3)

```
                                        .../...
-- basic data types used in the second branch
type STATUS is ( CONTINUATION_SEGMENT,
                 FIRST_SEGMENT,
                 LAST_SEGMENT,
                 UNSEGMENTED_PACKET);
for STATUS'size use 2;
for STATUS use ( CONTINUATION_SEGMENT => 2#00#,
                 FIRST_SEGMENT => 2#01#,
                 LAST_SEGMENT => 2#10#,
                 UNSEGMENTED_PACKET => 2#11#);

type COUNTER is range 0 .. 16383;
for COUNTER'size use 14 bits;

-- structuring type for the Packet Sequence Control
type PACKET_SEQUENCE_CONTROL_TYPE is record
      SEGMENTATION_FLAG : STATUS;
      SOURCE_SEQUENCE_COUNT : COUNTER;
end record;
for PACKET_SEQUENCE_CONTROL_TYPE use
record
      SEGMENTATION_FLAG at 0 range 0 .. 1;
      SOURCE_SEQUENCE_COUNT at 0 range 2 .. 15;
end record;
for PACKET_SEQUENCE_CONTROL_TYPE 'size use 16;

-- basic data types used in the other branches
type NUMBER is range 0 .. 65535;
for NUMBER'size use 16;

type OCTET is range 0 .. 255;
for OCTET'size use 8;

-- structuring types
type DATA_ARRAY is array (NUMBER range <>) of OCTET;

subtype SECONDARY_HEADER_TYPE is DATA_ARRAY ( 1 .. 4);
                                        .../...
```

Example 3-42: EAST logical description of Version 1 “Source Packet” Format (2 of 3)

```
                                        .../...
type PRIMARY_HEADER_TYPE is record
     PACKET_IDENTIFICATION : PACKET_IDENTIFICATION_TYPE;
     PACKET_SEQUENCE_CONTROL : PACKET_SEQUENCE_CONTROL_TYPE;
     SOURCE_DATA_LENGTH : NUMBER;
end record;
for PRIMARY_HEADER_TYPE use
record
     PACKET_IDENTIFICATION at 0 range 0 .. 15;
     PACKET_SEQUENCE_CONTROL at 0 range 16 .. 31;
     SOURCE_DATA_LENGTH at 0 range 32 .. 47;
end record;
for PRIMARY_HEADER_TYPE'size use 48;

type PACKET_FORMAT_TYPE(
     VIRTUAL_SECONDARY_HEADER_FLAG : PRESENCE_FLAG := PRESENT;
     -- point to the second header flag located in the first branch
     VIRTUAL_SOURCE_DATA_LENGTH : NUMBER := 256)
     -- point to the source data length located in the third branch
is record
     PRIMARY_HEADER : PRIMARY_HEADER_TYPE;
     case VIRTUAL_SECONDARY_HEADER_FLAG is
       when ABSENT =>
     SOURCE_DATA_0 : DATA_ARRAY (1 .. VIRTUAL_SOURCE_DATA_LENGTH);
       when PRESENT =>
     SECONDARY_HEADER : SECONDARY_HEADER_TYPE;
     SOURCE_DATA_1 : DATA_ARRAY (1 .. VIRTUAL_SOURCE_DATA_LENGTH);
     end case;
end record;
for PACKET_FORMAT_TYPE use
record
     PRIMARY_HEADER at 0 range 0 .. 47;
     SECONDARY_HEADER at 0 range 48 .. 79;
end record;

PACKET : PACKET_FORMAT_TYPE;
```

Example 3-42: EAST logical description of Version 1 “Source Packet” Format (3 of 3)

The two virtual components "VIRTUAL_SECONDARY_HEADER_FLAG" and "VIRTUAL_SOURCE_DATA_LENGTH" do not really exist in the exchanged data block. They serve as a link between other data:

- VIRTUAL_SECONDARY_HEADER_FLAG is supposed to have the value of the SECONDARY_HEADER_FLAG field in the PACKET IDENTIFICATION block, and conditions the existence of the SECONDARY_HEADER block. It serves as a link between these two fields.

- VIRTUAL_SOURCE_DATA_LENGTH is supposed to have the value of the SOURCE_DATA_LENGTH field in the PRIMARY HEADER and conditions the size of the SOURCE DATA block. It serves as a link, too.

As an example, an occurrence of the variable PACKET could be:

- Virtual_Secondary_Header_Flag = PRESENT, i.e., the data item called Secondary_Header_Flag (located in the 5th bit of the data occurrence) has the value PRESENT.
- Virtual_Source_Data_Length = 0, i.e., the data item called Source_Data_Length (located from the 33rd bit through the 48th bit) has the value 0.
- PRIMARY_HEADER : 48 bits
- SECONDARY_HEADER : 32 bits
- SOURCE_DATA : 0 bit

The size of this occurrence is, in this case, 80 bits (the virtual components being absent in any data occurrence).

----------or----------

- Virtual_Secondary_Header_Flag = ABSENT, i.e., the data item called Secondary_Header_Flag (located in the 5th bit of the data occurrence) has the value ABSENT.
- Virtual_Source_Data_Length = 10, i.e., the data item called Source_Data_Length (located from the 33rd bit through the 48th bit) has the value 10.
- PRIMARY_HEADER : 48 bits
- SECONDARY_HEADER : 0 bit
- SOURCE_DATA : 80 bits

The size of this occurrence is, in this case, 128 bits (the virtual components being absent in any data occurrence).

Example 3-43: Occurrences of Version 1 "Source Packet" Format

The convention of virtual variables is used to extend the descriptive capabilities of EAST. It allows one to write more about the exchanged data, keeping the same structuring of the data.

NOTE – The two components “SOURCE_DATA_0” and “SOURCE_DATA_1” represent the same data. But their location in the exchanged data block is different. That is why they must have a different name.

Rule 22 EAST forbids identical names in a record. See section 3.2.1.6. of reference [1].

3.2.13 FREQUENTLY ASKED QUESTIONS

Question 1 Are length clauses mandatory?

Answer 1 They are mandatory for basic data types (enumeration, integer and real types). They are mandatory for aggregation types (array and record types) if the size of the type is known at definition time.

Question 2 Are Enumeration Representation Clauses mandatory?

Answer 2 They are highly recommended for binary encoded enumeration types, but not mandatory. If not provided, default bit patterns are assumed: the integer value 0 is associated with the first enumeration literal, 1 is associated with the second one, and so on for every enumeration literal of the enumeration type.

They are forbidden for ASCII encoded enumeration types.

Question 3 Must an enumeration length clause exactly fit the bit patterns associated with enumeration literals (provided or assumed)?

Answer 3 No. The size of an enumeration literal may be greater than needed. For example, an enumeration type defined by three alternative values can be mapped on a 2-bit type, but it could be defined with an 8-bit type. In this case, the three corresponding binary integer values are stored in the 8-bit field as an 8-bit integer.

Question 4 Integer types and enumeration types can be used as discriminants in record types, either to determine the size of arrays or to determine the presence of any kind of component. Must these discriminants have a binary representation, or can they have an ASCII representation?

Answer 4 Both representations, binary and ASCII, are allowed to discriminate types. In particular, a string may be used as a discriminant, if the possible occurrences of that string are well identified, i.e., define an enumeration type or an integer type (see 3.2.5).

Question 5 Why are length clauses not provided for subtypes?

Answer 5 A subtype defines restrictions (in the range) of an existing basic type. The size of the subtype is the same as the size of the type. It is therefore not necessary (and not allowed by the EAST syntax) to specify a length clause for a subtype because the type already has its length clause. Warning: one must define a new type if the size of the type, defined by the restricted values, is not the same as the size of the original type.

Question 6 Why are default values mandatory for discriminants?

Answer 6 Types allow the declaration of variables as seen in 3.2.9. In the case of variable structures (i.e., records with discriminants), the corresponding variables do not have to be static structures in assigning explicit values to the discriminants. The following data declaration is not recommended because it sets forever the structure of the data which is supposed to be variable:

```
HEADER : PACKET_HEADER( ABSENT );
-- This data has a constant structure
```

The suitable data declaration is:

```
HEADER : PACKET_HEADER; -- This data may have one of the two
structures
```

Question 7 Are Record Representation Clauses mandatory?

Answer 7 Yes, but they may be partially provided in some cases (e.g., if components are of variable size or if components follow a variable size component).

3.3 PHYSICAL DESCRIPTIONS

3.3.1 OVERVIEW

The physical description must be self sufficient. The receiving machine, called *destination*, does not have to know the emitting machine, called *source*. It does not have to refer to any documentation about the source either. So the physical representation must provide all data-storage related characteristics of the source, so that the destination is able to interpret the received data. These characteristics are:

- the way of storing arrays on the medium, which, for multi-dimensional arrays, indicates whether the first or last index varies first when considering the elements stored on the medium (this characteristic is detailed in 3.3.2);

- the way of storing octets on the medium, which, for multi-octet elements, indicates whether the most significant octet or the least significant octet is the first stored on the medium (this characteristic is detailed in 3.3.3);

- the binary representation of logically defined basic types, which, for specific elements (integer and real), provides a bit-level description as well as the standard used to compute the numeric values from the bit-description (this characteristic is detailed in 3.3.4);

- the ASCII representation of logically defined basic types (integer, real and enumeration types), which provides the number of characters used for their representation and, for ASCII enumeration types, provides also the list of the ASCII permitted values (this characteristic is detailed in 3.3.5).

The physical part is structured as follows:

Package name_of_physical_description_part is

Features applicable to the whole data collection

-- Way of storing octets and arrays

Representation of basic types

-- Binary and ASCII representations

Relation between logical names and physical representations

-- Relation type definition

End name_of_physical_description_part;

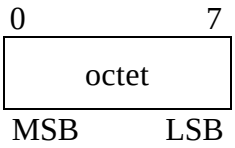
Some of the frequently asked questions about the physical description part of EAST data descriptions are summarized in 3.3.6.

The only basic types for which a physical description must be provided are those defined in the logical description part. In some cases (see 3.4.2), the physical description part is empty.

The convention, adopted in this document, for the data representation on the medium is the following one:

- In multi-octet elements, the first octet is drawn in the leftmost position and is called “Octet Zero”. Successive octets are assigned successively larger numbers.

- Within an octet, the first bit (which is the Most Significant Bit or MSB) is drawn in the leftmost position and is called “Bit Zero”. The Least Significant Bit (LSB) is in the rightmost position.



3.3.2 ARRAY STORAGE METHOD

The way of storing multidimensional arrays on the medium is one of the machine-dependent characteristics. This descriptive element is easy to understand: host machines have different ways of storing multi-dimensional arrays on the medium sometimes due to generating languages. The identified ways of storing arrays are either first_index_first or last_index_first. This attribute indicates how the sequence of the array elements is organized: in the first case, the first index varies first; in the second case, the last index varies first.

In the following example, a two-dimensional matrix of integer elements is defined:

```
-- Data types declarations
type ELEMENT is range 0 .. 200 ;
for ELEMENT'size use 8 ; -- bits

type MATRIX is array(1 .. 10 , 1 .. 10) of ELEMENT ;
for MATRIX'size use 800 ; -- bits

-- Data occurrence declaration
IMAGE : MATRIX;
```

Example 3-44: Two dimensional Matrix

How will a data interpreter use the array storage information to access the data?

To access the element IMAGE(1,5) an interpreter uses the following algorithm:

If A is the address of the first element of IMAGE on the medium (the first element being IMAGE(1,1)):

- If the array storage method is *last_index_first*, then the element IMAGE(1,5) is the fifth element of the array. The address of this element is therefore (A + 4 * 8) because there are four elements before the fifth element and the size of one element is 8 bits, as specified by the representation clause of the ELEMENT type.

- If the array storage method is *first_index_first*, then the element IMAGE(1,5) is the forty-first element of the array. The address of this element is therefore (A + 41 * 8).

The array storage method is information obviously necessary for the destination to interpret the received data. It is provided in the physical description using an enumeration type:

```
type ARRAY_STORAGE_METHOD is ( FIRST_INDEX_FIRST,
                               LAST_INDEX_FIRST) ;
```

Template 3-1 of the Physical Description

Using this declaration, the actual way of storing the array is provided:

```
ARRAY_STORAGE : constant ARRAY_STORAGE_METHOD :=
                               FIRST_INDEX_FIRST; -- for example
```

Example 3-45: Array storage

This declaration is applicable to the whole description.

By default (i.e. if this declaration is not available in the EAST description), the array storage is FIRST_INDEX_FIRST.

3.3.3 OCTET STORAGE METHOD

Another characteristic of the source is the way of storing octets on the medium. This characteristic is a tricky point of the physical description.

A machine is said to be big-endian or little-endian depending on whether the MSB is in the lowest or highest addressed octet, i.e. in the first or last transmitted octet.

For a big-endian representation, the MSB is in the first transmitted octet, i.e. in the first octet on the medium, while it is in the last transmitted octet, i.e. in the last octet on the medium for a little-endian representation.

The little-endian representation for a data element can be viewed as storing the bits from least to most significant bit order, but then re-ordering the bits (from most to least significant) within each octet when output to some medium.

This machine-dependent characteristic is very important for a correct interpretation of the data. Its definition is given for multi-octets data elements, but is still applicable for every data element, whatever its length and its position (on octet boundary or not) within the data set.

The octet storage method has some impacts on basic types (enumeration, integer and real) as well as on aggregation types too (records and arrays).

The following is an example of a simple record used to illustrate the differences introduced by the source on the generated data:

Value (16 bits) = 1345	Factor (8 bits) = 8

The logical description of this data structure is the following one:

```
type FACTOR_TYPE is range -10 .. 10;
for FACTOR_TYPE'size use 8; -- bits

type VALUE_TYPE is range 0 .. 65535;
for VALUE_TYPE'size use 16; -- bits

type STRUCTURE is
record
      VALUE : VALUE_TYPE;
      FACTOR : FACTOR_TYPE;
end record;
for STRUCTURE use
record
      VALUE at 0 range 0 .. 15;
      FACTOR at 0 range 16 .. 23;
end record;
for STRUCTURE 'size use 24; -- bits

DATA_STRUCTURE : STRUCTURE;
```

Example 3-46: Record with Elements on Octet Boundaries

On a SUN (or on any other big-endian architecture), the data structure corresponds to the following hexadecimal dump on the medium:

05 41 08

On a PC (or on any other little-endian architecture), the data structure corresponds to the following hexadecimal dump on the medium:

41 05 08

How will a data interpreter use the octet storage information to access the data?

The data that have been written by a SUN are interpreted using the following algorithm:

16#05 41 08# is read from the medium.
16#05 41 08# is also 2#0000 0101 0100 0001 0000 1000#
According to the record representation clauses of STRUCTURE: VALUE is located from bit 0 through bit 15: 2#0000 0101 0100 0001#, i.e., has the value 1345,
FACTOR is located from bit 16 through bit 23: 2#0000 1000#, i.e., has the value 8.

The data that have been written by a PC are interpreted using the following algorithm:

16#41 05 08# is read from the medium.
41 is the least significant octet, while 05 is the most significant octet, because of the behavior of the PC architecture when writing data on a medium. A data interpreter has therefore to invert the octet first within each data item. The data structure becomes: 16#05 41 08# which is also 2#0000 0101 0100 0001 0000 1000#
According to the record representation clauses of STRUCTURE: VALUE is located from bit 0 through bit 15: 2#0000 0101 0100 0001#, i.e., has the value 1345,
FACTOR is located from bit 16 through bit 23: 2#0000 1000#, i.e., has the value 8.

The following is another example of a simple record used to illustrate the differences introduced by the source on the generated data. In this case, the data items do not begin on an octet boundary.

Version (2 bits) = 1	Value (16 bits) = 1345	Factor (6 bits) = 8

The logical description of this data structure is the following one:

```
type VERSION_TYPE is (ZERO, ONE, TWO);
for VERSION_TYPE use (ZERO => 0, ONE => 1, TWO => 2);
for VERSION_TYPE'size use 2; -- bits

type FACTOR_TYPE is range -10 .. 10;
for FACTOR_TYPE'size use 6; -- bits

type VALUE_TYPE is range 0 .. 65535;
for VALUE_TYPE'size use 16; -- bits

type STRUCTURE is
record
      VERSION : VERSION_TYPE;
      VALUE : VALUE_TYPE;
      FACTOR : FACTOR_TYPE;
end record;
for STRUCTURE use
record
      VERSION at 0 range 0 .. 1;
      VALUE at 0 range 2 .. 17;
      FACTOR at 0 range 18 .. 23;
end record;
for STRUCTURE 'size use 24; -- bits

DATA_STRUCTURE : STRUCTURE;
```

Example 3-47: Record with Elements not on Octet Boundaries

On a SUN (or on any other big-endian architecture), the data structure corresponds to the following hexadecimal dump on the medium:

41 50 48

On a PC (or on any other little-endian architecture), the data structure corresponds to the following hexadecimal dump on the medium:

05 15 20

How will a data interpreter use the octet storage information to access the data?

The data that have been written by a SUN are interpreted using the following algorithm:

16#41 50 48# is read from the medium.
16#41 50 48# is also 2#0100 0001 0101 0000 0100 1000#
According to the record representation clauses of STRUCTURE: VERSION is located from bit 0 through bit 1: 2#01#, i.e., has the value 1 (= ONE), VALUE is located from bit 2 through bit 17: 2#0000 0101 0100 0001#, i.e., has the value 1345, FACTOR is located from bit 18 through bit 23: 2#001000#, i.e., has the value 8.

The data that have been written by a PC are interpreted using the following algorithm:

16#05 15 20# is read from the medium.
In this case, the data items are not on octet boundaries. A simple octet inversion is not applicable. The solution is to invert each bit within each octet of the data structure, and then invert each bit within each data item.
16#05 15 20# is also 2#00000101 00010101 00100000#2
After the first bit inversion, the data structure becomes:
2#10100000 10101000 00000100#
After the other bit inversions, the data structure becomes:
2#01 0000010101000001 001000#
According to the record representation clauses of STRUCTURE: VERSION is located from bit 0 through bit 1: 2#01#, i.e., has the value 1 (= ONE), VALUE is located from bit 2 through bit 17: 2#0000 0101 0100 0001#, i.e., has the value 1345, FACTOR is located from bit 18 through bit 23: 2#001000#, i.e., has the value 8.

The octet storage method is a way to indicate how to interpret the data. It is provided in the physical description using an enumeration type:

```
type BIT_ORDER is ( HIGH_ORDER_FIRST,    -- big-endian representation
                    LOW_ORDER_FIRST) ;   -- little-endian representation
```

Template 3-2 of the Physical Description

Using this declaration, the actual way of storing octet is provided:

```
OCTET_STORAGE : constant BIT_ORDER := HIGH_ORDER_FIRST; -- for example
```

Example 3-48: Octet storage

This declaration is applicable to the whole description.

By default (i.e. if this declaration is not available in the EAST description), the octet storage is HIGH_ORDER_FIRST.

3.3.4 BINARY REPRESENTATION OF SCALAR TYPES

There is no predefined scalar type[F9] (except CHARACTER type) provided by EAST for user data descriptions, so each scalar type must be explicitly defined by the user in the data description.

a) Enumeration types

Enumeration types are defined by their possible values and their size in bits. A binary enumeration value is an integer value that is expected to be represented as a bit string that respects a standard format, defined as follows:

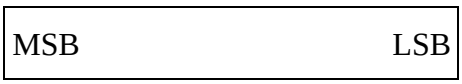
No binary representation is provided for a binary enumeration type. Negative values are represented in a two's complement form.

b) Integer types

Integer types are defined by their range and their size in bits. There are two kinds of integer types:

- the integer types, which are independent of the machine integer types (e.g., a 13-bit integer type);
- the integer types that can map with the integer types of the host machine, called "machine integer types" (e.g., a 16-bit signed integer type).

In the first case, the integer types are not expected to be mapped on existing machine integer types. No binary representation is provided. The bit pattern of such integer types is supposed to respect a standard format, defined as follows:

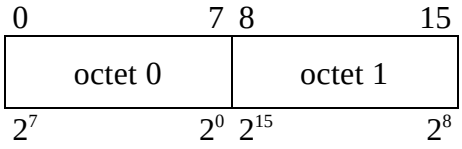


Warning:

- If the range of the integer type allows negative values, then a sign bit is present and is then the leftmost bit. The sign convention is then by default the two's complementation.
- If the range of the integer type does not allow negative values, then there is no sign bit.

In the second case, the size is a multiple of octets, and a binary representation must therefore be specified. The binary representation of an integer type specifies the sign convention which indicates the complementation, if any, and the location of the bits from the MSB to the LSB, the sign location, if any, being the MSB.

Let's take the example of a 16-bit integer generated by a PC (i.e., for which a binary representation is mandatory), which has the following bit pattern on a medium:



The most significant bit (if the integer is unsigned) or the sign bit (if it is signed) is the 9^{th} bit encountered (bit 8). Then, a less significant bit is the 10^{th} bit encountered (bit 9) and so on till the 16^{th} bit. And then from bit 0 through bit 7, the bit 7 being the least significant bit of the integer. The bit ordering (from MSB to LSB) can also be expressed in a simple manner using ranges: (8 , 15) and (0 , 7).

How will a data interpreter use the binary representation of this integer to retrieve the value?

To compute an element coded with this 16-bit integer type, an interpreter uses the following algorithm:

- If the integer is identified as a signed integer, then the sign bit is bit 8 and the MSB is bit 9. If it is identified as an unsigned integer, the MSB is bit 8.

- If the integer is identified as a signed integer and if the sign bit has the value 1, the interpreter must complement the bit string.

- The value is computed in multiplying each bit with its weight (the weight decreases from 2^{15} (or 2^{14} if a sign bit is present) for the MSB, to 2^0 for the LSB) and adding the result.

The description of an integer binary representation using the EAST syntax is the following one:

```
type INTEGER_PHYSICAL_DESCRIPTION
        (NUMBER_OF_SUBFIELDS : SUBFIELD_NUMBER := 1)
is record
        COMPLEMENT : SIGN_CONVENTION;
        LOCATION : LOCATION_OF_FIELD(1 .. NUMBER_OF_SUBFIELDS);
end record;
```

Template 3-3 of the Physical Description

SIGN_CONVENTION is defined as follows:

```
type SIGN_CONVENTION is ( UNSIGNED, SIGN_AND_MAGNITUDE,
        ONES_COMPLEMENT, TWOS_COMPLEMENT);
```

Template 3-4 of the Physical Description

- UNSIGNED is used for unsigned integer types.

- SIGN_AND_MAGNITUDE is used when the integer is interpreted as a sign bit location followed by a positive quantity. A MSB ‘1’ means a negative integer and a MSB ‘0’ means a positive integer.

- ONES_COMPLEMENT is used when , the MSB being ‘1’, the absolute value of the negative integer is computed in inverting each bit (‘1’ becomes ‘0’ and ‘0’ becomes ‘1’).

- TWOS_COMPLEMENT is used when , the MSB being ‘1’, the absolute value of the negative integer is computed in inverting each bit (‘1’ becomes ‘0’ and ‘0’ becomes ‘1’) and adding ‘1’ to the LSB.

LOCATION_OF_FIELD is defined as an array of intervals declaring the location of subfields (these subfields are used to define the exact location of the integer bits).

```
type NATURAL_NUMBER is range 0 .. 65535;

type LOCATION_OF_SUBFIELD is record
        BEGINNING_AT_BIT_NUMBER : NATURAL_NUMBER;
        ENDING_AT_BIT_NUMBER : NATURAL_NUMBER;
end record;

MAXIMUM_NUMBER_OF_SUBFIELDS : constant := 255;

type SUBFIELD_NUMBER is range
        1 .. MAXIMUM_NUMBER_OF_SUBFIELDS;

type LOCATION_OF_FIELD is array (SUBFIELD_NUMBER range <>)
        of LOCATION_OF_SUBFIELD;
```

Template 3-5 of the Physical Description

BEGINNING_AT_BIT_NUMBER of the first element of the array LOCATION_OF_FIELD is supposed to be the bit number of the MSB or the sign bit number, if any. Bit numbers continue in sequence until ENDING_AT_BIT_NUMBER of the last element of LOCATION_OF_FIELD, which is supposed to be the bit number of the LSB.

NOTES

1 The MAXIMUM_NUMBER_OF_SUBFIELDS is set to 255. The number of subfields that are necessary to locate the bits of an integer can be up to 255. It is an arbitrary value that is big enough to cover all the identified architectures.

2 The upper bound of NATURAL_NUMBER is set to 65535. It is an arbitrary value that seems to be large enough in this context.

Each time the bits of an integer are not contiguously located on the medium from the MSB to the LSB (see the previous example), several subfields are necessary to locate the bits of the integer.

The binary representation of a 16 bit signed integer on PC is:

```
Binary_Representation : constant INTEGER_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_SUBFIELDS => 2,
        COMPLEMENT => TWOS_COMPLEMENT,
        LOCATION => (       1 => (8,15) , -- first subfield (bit 8 through 15)
                            2 => (0,7))); -- second subfield (bit 0 through 7)
```

Example 3-49: Binary Integer Type Physical Description (1)

In this example, elements 1 and 2 of the LOCATION component are assigned to values (8,15) and (0,7). The whole binary representation indicates therefore that the sign bit is the 9th bit encountered (bit 8). Then, the most significant bit is the 10th bit encountered (bit 9), then, a less significant bit is the 11th bit encountered (bit 10), and so on till the 16th bit, and then from bit 0 through bit 7, bit 7 being the least significant bit of the integer.

The binary representation of a 16 bit unsigned integer on PC is:

```
Binary_Representation : constant INTEGER_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_SUBFIELDS => 2,
        COMPLEMENT => UNSIGNED,
        LOCATION => (       1 => (8,15) , -- first subfield (bit 8 through 15)
                            2 => (0,7))); -- second subfield (bit 0 through 7)
```

Example 3-50: Binary Integer Type Physical Description (2)

In this example, elements 1 and 2 of the LOCATION component are assigned to values (8,15) and (0,7). The whole binary representation indicates therefore that the most significant bit is the 9th bit encountered (bit 8). Then, a less significant bit is the 10th bit encountered (bit 9), and so on till the 16th bit, and then from bit 0 through bit 7, bit 7 being the least significant bit of the integer.

NOTES

1 The ranges can be ordered backwards as well, (15,8) and (7,0), if this is the way that the bits are numbered by the machine architecture, i.e., if the first transmitted bit of an octet is the LSB, and not the MSB (as it is supposed to be).

2 The name of the constant used to identify the binary representation (Binary_Representation) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in a package.

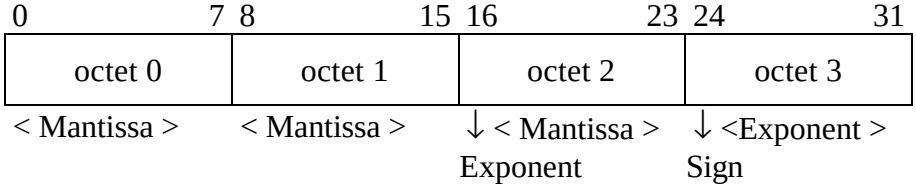
c) Real types

All real types, when binary encoded, must have a binary representation. The binary representation of a real type on the medium specifies the sign bit number, the sign convention, the exponent base, the bias used, the location of the exponent and the location of the mantissa.

A convention or standard gives the way to use the bit pattern to compute the associated numeric value. The binary representation of a real type also specifies therefore the convention used for its generation.

NOTE – The convention only concerns real representation on the medium, so different format representations can be expressed in the same Data Description Record even though they could not have existed on the same machine at the same time. A block of data processed on a given machine could have been inserted in another block of data processed in another machine.

Let's take the example of a 32-bit real generated by a PC (IEEE convention - see reference [8]) which has the following bit pattern on a medium:



The most significant bit of the exponent is the 26th bit encountered (bit 25). Then from bit 26 through bit 31 the bits encountered are less significant, and bit 16 is the least significant bit of the exponent.

In the same way, the most significant bit of the mantissa is the 18th bit encountered (bit 17). Then from bit 18 through bit 23, and from bit 8 through bit 15, and from bit 0 through bit 7, the bits encountered are less significant, bit 7 being the least significant bit of the mantissa.

The bit ordering of the exponent (from MSB to LSB) can also be expressed in a simple manner using ranges: (25 , 31) and (16 , 16). In the same way, the bit ordering of the mantissa (from MSB to LSB) can be expressed using ranges: (17,23), (8,15), (0,7).

How will a data interpreter use the binary representation of this real to retrieve the value?

To compute an element coded with the 32 bit real type, an interpreter uses the following algorithm:

- The exponent is computed by multiplying each bit by its weight (the weight decreases from 2^7 for the MSB, to 2^0 for the LSB) and adding the result.

- The mantissa is computed by dividing each bit by an increasing weight (the weight increases from 2^1 for the MSB of the mantissa to 2^{23} for the LSB) and adding the result.

- The value is computed according to the formula: $(-1)^S * 1.M * 2^{(E - 127)}$, where S is the value of the sign bit, M is the calculated mantissa and E is the calculated exponent.

The following is a description of a real binary representation using the EAST syntax:

```
type REAL_PHYSICAL_DESCRIPTION(
   NUMBER_OF_SUBFIELDS_IN_EXPONENT : SUBFIELD_NUMBER := 1;
   NUMBER_OF_SUBFIELDS_IN_MANTISSA : SUBFIELD_NUMBER := 1)
is record
   CONVENTION_USED : LIST_OF_RECOGNIZED_CONVENTIONS;
   SIGN_BIT_NUMBER : NATURAL_NUMBER ;
   COMPLEMENT : SIGN_CONVENTION;
   EXPONENT_BASE : NATURAL_NUMBER ;
   BIAS : NATURAL_NUMBER ;
   LOCATION_OF_EXPONENT : LOCATION_OF_FIELD
               (1 .. NUMBER_OF_SUBFIELDS_IN_EXPONENT);
   LOCATION_OF_MANTISSA : LOCATION_OF_FIELD
               (1 .. NUMBER_OF_SUBFIELDS_IN_MANTISSA);
end record;
```

Template 3-6 of the Physical Description

LIST_OF_RECOGNIZED_CONVENTIONS is defined as a list of ADIDs representing the permitted conventions. Reference [9] provides the names of the recognized conventions and associated ADIDs, and for each of them, the algorithm to be used, to compute the original value, according to the information stored in the real binary representation. As an example, this type could be defined as follows:

```
type LIST_OF_RECOGNIZED_CONVENTIONS is (FCSTC000, FCSTC001);
```

Example 3-51: List of Conventions

NOTE – This type shall contain at least the ADIDs, used in the current Data Description. It is not mandatory to find in this definition the exhaustive list of the registered conventions, i.e., the exhaustive list of relevant ADIDs.

SIGN_CONVENTION is the enumeration type previously defined for the INTEGER_PHYSICAL_DESCRIPTION, and applied to the mantissa. The BIAS is to be subtracted from the exponent, and the EXPONENT_BASE is raised to the power of the biased exponent.

Each time the bits of the exponent or of the mantissa are not contiguously located on the medium from the MSB to the LSB (see the previous example), several subfields are necessary to locate these bits.

The associated representation of the previous real example is:

```
Binary_Representation : constant REAL_PHYSICAL_DESCRIPTION :=
(NUMBER_OF_SUBFIELDS_IN_EXPONENT => 2,
NUMBER_OF_SUBFIELDS_IN_MANTISSA => 3,
CONVENTION_USED => FCSTC000, -- IEEE754
SIGN_BIT_NUMBER => 24,
COMPLEMENT => SIGN_AND_MAGNITUDE,
EXPONENT_BASE => 2,
BIAS => 127,
LOCATION_OF_EXPONENT => ( 1 => (25,31) , -- 1st subfield
                          2 => (16,16)), -- 2nd subfield
LOCATION_OF_MANTISSA => ( 1 => (17,23) , -- 1st subfield
                          2 => (8,15) , -- 2nd subfield
                          3 => (0,7))); -- 3rd subfield
```

Example 3-52: Binary Real Type Physical Description

In this example, the LOCATION_OF_EXPONENT component, elements 1 and 2 are assigned to values (25,31) and (16,16). This expresses the range of the bit numbering in the exponent subfield. In the same way, the LOCATION_OF_MANTISSA component, elements 1, 2 and 3 are assigned to values (17,23), (8,15) and (0,7). This expresses the range of the bit numbering in the mantissa subfield.

NOTE – The name of the constant used to identify the binary representation (Binary_Representation) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in a package.

3.3.5 ASCII REPRESENTATION OF SCALAR TYPES

In order to increase the portability of data, some users may wish to store types as ASCII encoded types and not as binary types (enumeration types, integer types or real types). An ASCII Encoded type is a character string type with a specific format, that depends on the nature of the type (enumeration, integer or real).

There is no distinction made in the logical part of an EAST description between binary and ASCII encoded types. The actual representation of the types is provided in the physical part of the description. By default, a type is a binary encoded type. An ASCII representation must be associated with the type name, if the type is ASCII encoded.

a) ASCII Encoded Enumeration

An ASCII Encoded Enumeration is a character string representing an enumeration value. The ASCII representation of an enumeration type provides all the character strings associated with all the enumeration literals of the type.

The ASCII representation of an enumeration uses the following types:

```
type STRING_LIST is array(NATURAL_NUMBER  range <>,
NATURAL_NUMBER  range <>) of
   CHARACTER;

type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION (
   NUMBER_OF_OCCURRENCES : NATURAL_NUMBER  := 0;
   NUMBER_OF_CHARACTERS : NATURAL_NUMBER  := 0) is record
   REPRESENTATION : STRING_LIST (  1 .. NUMBER_OF_OCCURRENCES,
                                   1 .. NUMBER_OF_CHARACTERS);
end record;
```

Template 3-7 of the Physical Description

For example, an enumeration type which has two permitted values, “TM” and “TC”, indicating a Telemetry Packet or a Telecommand Packet, can be described in the logical part as follows:

```
type PACKET_TYPE is (TELEMETRY, TELECOMMAND);
for PACKET_TYPE'size use 16; -- bits
```

Example 3-53: ASCII Enumeration Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep : constant ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
    (NUMBER_OF_OCCURRENCES => 2, NUMBER_OF_CHARACTERS => 2,
    REPRESENTATION => ("TM" , "TC"));
```

Example 3-54: ASCII Enumeration Type Physical Description

Rule 23 The number of characters used to encode the enumeration type must be the same for every enumeration literal of the type. See section 3.3.3.2. of reference [1].

Rule 24 All characters (i.e., the 256 characters of the "Latin Alphabet No. 1"—see reference [6]) are allowed and significant, including the space character. See section 3.3.3.2. of reference [1].

Rule 25 The physical representations of the enumeration literals are provided in the order of their declaration in the logical part. See section 3.3.3.2. of reference [1].

NOTE – The name of the constant used to identify the ASCII representation (ASCII_Rep) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in a package.

The relation between the logical definition of the enumeration type "PACKET_TYPE" and its physical description "ASCII_Rep" is made in creating a connection between the two names (see 3.4.2).

b) ASCII Encoded Decimal Integer

An ASCII Encoded Decimal Integer is a character string representing an integer value. The format of the character string corresponding to an ASCII encoded decimal integer is described in Figure 3-3:

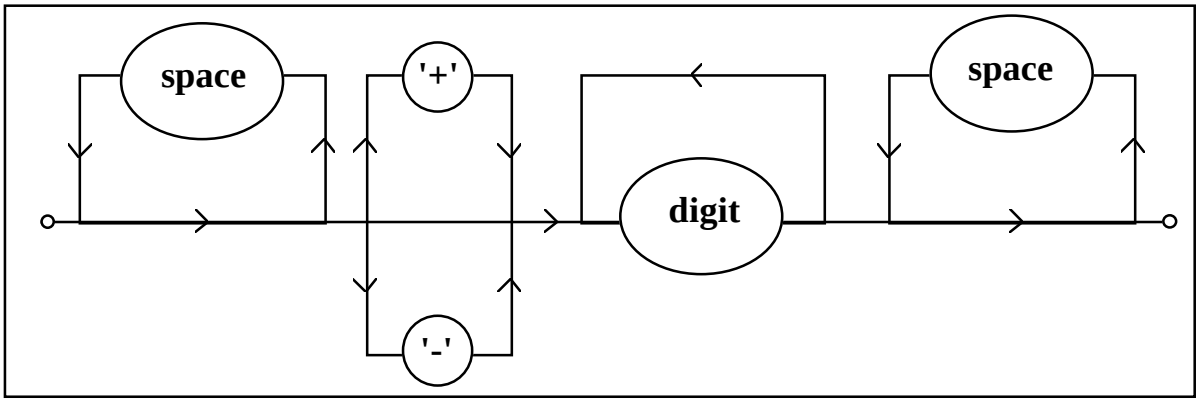


Figure 3-3: ASCII Encoded Decimal Integer Format

A digit is one of the following characters: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.

The ASCII representation of an integer type specifies the number of characters used for the integer values. The ASCII representation of an integer uses the following type:

```
type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is record
   NUMBER_OF_CHARACTERS : NATURAL_NUMBER ;
end record;
```

Template 3-8 of the Physical Description

For example, a 5-character ASCII decimal integer type can be described in the logical part as follows:

```
type COUNTER is range -1 .. 16383;
for COUNTER'size use 40; -- bits, i.e., 5 characters
```

Example 3-55: ASCII Integer Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep : constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
   (NUMBER_OF_CHARACTERS => 5);
```

Example 3-56: ASCII Integer Type Physical Description

Possible occurrences of this integer type are:

- “ -1”
- “ 205”
- “ 8451”
- “11001”

NOTE – The name of the constant used to identify the ASCII representation (ASCII_Rep) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in a package.

c) ASCII Encoded Decimal Real

An ASCII Encoded Decimal Real is a string representing a real value. The format of the character string corresponding to an ASCII encoded decimal real is described in Figure 3-4:

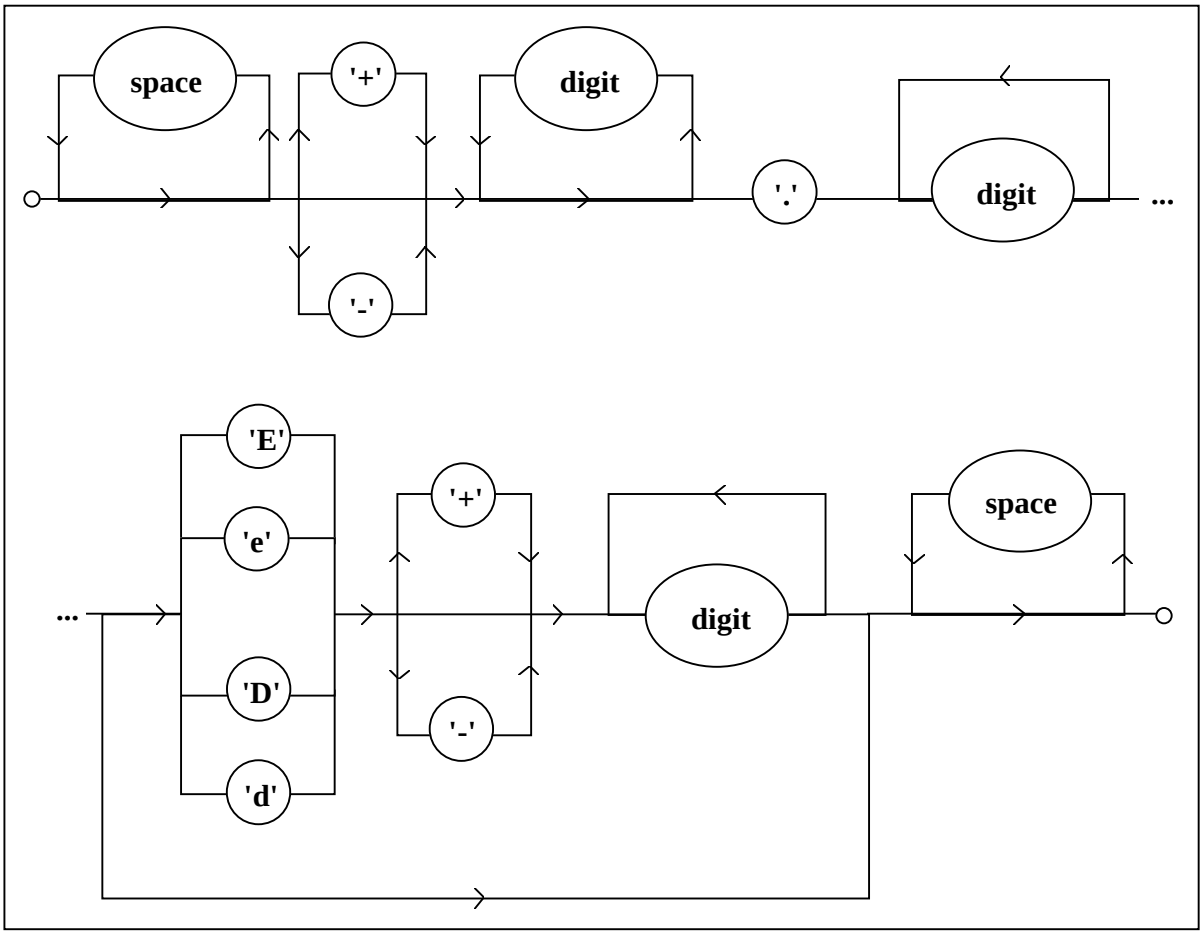


Figure 3-4: ASCII Encoded Decimal Real Format

A digit is one of the following characters: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’.

NOTES

1 Only the normalized ASCII encoded numbers can be described using EAST. There is no convention for the ASCII representation of infinite values (“+INF”, “-INF” or “+ ∞”, “- ∞”) and no representation for “NaN” (Not a Number).

2 In the FORTRAN 90 Numeric Editing section (see reference [10]), it is specified that the decimal point is optional in the F notation. In the same way, the ‘E’ or ‘D’ may be omitted in the E and D notation. These features have not been retained because the ASCII real value must be “readable”, i.e., understandable without any other information but the number of characters.

The ASCII representation of a real type specifies the number of characters used for the real values. The ASCII representation of a real uses the same type as the one used for the representation of integer:

```
type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is record
   NUMBER_OF_CHARACTERS : NATURAL_NUMBER ;
end record;
```

Template 3-9 of the Physical Description

For example, an 11-character ASCII decimal real type can be described in the logical part as follows:

```
type KILOMETERS is digits 5;
for KILOMETERS'size use 88; -- bits
```

Example 3-57: ASCII Real Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep : constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
   (NUMBER_OF_CHARACTERS => 11);
```

Example 3-58: ASCII Real Type Physical Description

Possible occurrences of this real type are:

- “ 1.2674E+03”
- “   -128.56”
- “ -1.3689E-8”

NOTE – The name of the constant used to identify the ASCII representation (ASCII_Rep) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in a package.

3.3.6 FREQUENTLY ASKED QUESTIONS

Question 1 Is the octet storage method significant for the bit storage?

Answer 1 The octet storage method, defined by the bit ordering, is useful for the interpretation of multi-octet elements as well as for the interpretation of "small" elements, i.e., elements with a length less than 8 bits.

Question 2 Why is no binary representation provided for a binary enumeration type?

Answer 2 An enumeration value is an integer value. The bit ordering is sufficient to deduce the binary representation of the enumeration because it specifies indirectly the location of the most significant bit and the location of the least significant bit: if the octet storage method is "high order first", then the first encountered bit is the most significant bit of the enumeration; if it is the "low order first", then the octets must be first inverted, to have the most significant bit of the enumeration in the first encountered bit. The sign convention, if needed, is the two's complementation.

Question 3 How is a real that is generated using a non-recognized convention described?

Answer 3 The list of recognized conventions is not an exhaustive list (see reference [9]). This list shall be extended, if necessary. In the case of a "new" real convention, the binary representation must be provided to the relevant Member Agency Control Authority Office (MACAO) for registration and definition of a new ADID; the "CONVENTION_USED" field must be filled with the "new" convention ADID. The document [9] should be upgraded within a relatively short delay.

3.4 ORGANIZATION OF EAST DATA DESCRIPTION RECORDS

As seen in the previous subsections (3.2 and 3.3), EAST contributes to a complete data description. EAST organizes the description in two units, called *packages*, the first one being the logical data description package and the second one the physical data description package.

Subsection 3.4.1 summarizes the list of information items provided in the logical part and gives an example of an EAST logical package.

Subsection 3.4.2 summarizes the list of information items provided in the physical part and gives an example of an EAST physical package (associated with the logical one presented in 3.4.1).

3.4.1 LOGICAL DATA DESCRIPTION PACKAGE

The first package contains the logical description of all data types used to declare an occurrence of the exchanged data. This logical description is written using the EAST syntax, as described in 3.2 and specified in reference [1].

The logical description must include the following EAST statements:

- A mandatory statement beginning with the keyword "*package*" followed by an EAST identifier, which is supposed to be the name of the logical data description part, and followed by the keyword "*is*". This statement is the first one of the logical package.

- Declaration of constants used in the rest of the description.

- User type declarations and their associated representation clauses, which describe the syntax of data items and the relationship of these data items. Atomic data types (enumeration types, integer types, real types and character string subtypes) must be declared before any aggregation data types (array types and record types) that make use of them.

- Declaration of variables (and constants), which represent one actual data occurrence. The order of the declarations must correspond to the order of the contiguous data items in any block instance described by this logical description. The exchanged data block contains n ($n \geq 1$) contiguous occurrences of the described data.

- A mandatory statement beginning with the keyword "*end*" followed by the name of the logical data description part (the same one as the one after the keyword "*package*" at the beginning of the description), and followed by the character ";". This statement is the last one of the logical package.

The order of the declarations of the logical package is not free. An EAST definition must appear before it is used(F10).

CAUTION – A type declaration does not correspond to any data occurrence. Only the declared variables correspond to the data that are to be exchanged.

Some readers may find it laborious to declare types and then variables. Why is it not sufficient to provide only type declarations for a data description?

The following example illustrates the different meaning intrinsic to a type declaration and to a declaration of a variable.

```
package logical_CNES_description_01 is

-- type declarations
type BULLETIN_KIND is (CARTESIAN , KEPLERIAN);
for BULLETIN_KIND'size use 72; -- bits

type YEAR is range 1900 .. 2100;
for YEAR'size use 32; -- bits

type MONTH is range 1 .. 12;
for MONTH'size use 32; -- bits

type DAY_OF_MONTH is range 1 .. 31;
for DAY_OF_MONTH'size use 32; -- bits

type HOUR is range 0 .. 23;
for HOUR'size use 32; -- bits

type MINUTE is range 0 .. 59;
for MINUTE'size use 32;-- bits
                                        .../...
```

Example 3-59: Complete Logical Description (1 of 4)

```
type SECOND is digits 5 range 0.0000 .. 59.999;
for SECOND'size use 32; -- bits

type KILOMETERS is digits 12 range 0.00000000000 .. 999999.999999;
for KILOMETERS'size use 64; -- bits

type KM_SEC is digits 12 range 0.00000000000 .. 999999.999999;
for KM_SEC'size use 64; -- bits

type RATIO is digits 12 range 0.00000000000 .. 1.00000000000;
for RATIO'size use 64; -- bits

type ANGULAR_DEGREE is digits 12 range 0.0 .. 360.0;
for ANGULAR_DEGREE'size use 64; -- bits

type EPOCH_TIME is record
        Experiment_Year      : YEAR;
        Experiment_Month     : MONTH;
        Experiment_Day       : DAY_OF_MONTH;
        Experiment_Hour      : HOUR;
        Experiment_Minute    : MINUTE;
        Experiment_Second    : SECOND;
end record;
for EPOCH_TIME use
record
        Experiment_Year              at 0 * WORD_32_BITS range 0 .. 31;
        Experiment_Month             at 1 * WORD_32_BITS range 0 .. 31;
        Experiment_Day               at 2 * WORD_32_BITS range 0 .. 31;
        Experiment_Hour              at 3 * WORD_32_BITS range 0 .. 31;
        Experiment_Minute            at 4 * WORD_32_BITS range 0 .. 31;
        Experiment_Second            at 5 * WORD_32_BITS range 0 .. 31;
end record;
for EPOCH_TIME'size use 192; -- bits
                                                .../...
```

Example 3-59: Complete Logical Description (2 of 4)

```
type BULLETIN (Kind : BULLETIN_KIND := CARTESIAN) is record
        case Kind is
                when CARTESIAN =>
                        State_Position_Element_X_Axis : KILOMETERS;
                        State_Position_Element_Y_Axis : KILOMETERS;
                        State_Position_Element_Z_Axis : KILOMETERS;
                        State_Velocity_Element_X_Axis : KM_SEC;
                        State_Velocity_Element_Y_Axis : KM_SEC;
                        State_Velocity_Element_Z_Axis : KM_SEC;
                when KEPLERIAN =>
                        Semi_Major_Axis : KILOMETERS;
                        Eccentricity : RATIO;
                        Inclination : ANGULAR_DEGREE;
                        Right_Ascencion_Ascending_Node : ANGULAR_DEGREE;
                        Argument_of_Perigee : ANGULAR_DEGREE;
                        True_Anomalie : ANGULAR_DEGREE;
        end case;
end record;
for BULLETIN use
record
        Kind at 0 * WORD_32_BITS range 0 .. 71;
        State_Position_Element_X_Axis at 2 * WORD_32_BITS range 8 .. 71[3];
        State_Position_Element_Y_Axis at 4 * WORD_32_BITS range 8 .. 71;
        State_Position_Element_Z_Axis at 6 * WORD_32_BITS range 8 .. 71;
        State_Velocity_Element_X_Axis at 8 * WORD_32_BITS range 8 .. 71;
        State_Velocity_Element_Y_Axis at 10 * WORD_32_BITS range 8 .. 71;
        State_Velocity_Element_Z_Axis at 12 * WORD_32_BITS range 8 .. 71;
        Semi_Major_Axis at 2 * WORD_32_BITS range 8 .. 71;
        Eccentricity at 4 * WORD_32_BITS range 8 .. 71;
        Inclination at 6 * WORD_32_BITS range 8 .. 71;
        Right_Ascencion_Ascending_Node at 8 * WORD_32_BITS range 8 .. 71;
        Argument_of_Perigee at 10 * WORD_32_BITS range 8 .. 71;
        True_Anomaly at 12 * WORD_32_BITS range 8 .. 71;
end record;
for BULLETIN'size use 392; -- bits
                                                .../...
```

Example 3-59: Complete Logical Description (3 of 4)

[3] All the components after “kind” are not located on a (32 bits) word boundary. That is why a component is located from the 8th bit of the word to the 71th, which is also the 7th of two words later.

```
-- declaration of variables
TIME : EPOCH_TIME;
BULLETIN_AT_THAT_TIME : BULLETIN;
INTERVAL : SECOND;
BULLETIN_AFTER_INTERVAL : BULLETIN;

end logical_CNES_description_01;
```

Example 3-59: Complete Logical Description (4 of 4)

In this example, it may be noted that:

- many types are declared, only three of those are used to declare the variables corresponding to the actual exchanged data;
- one of these types is an atomic data type (SECOND) used:
  - for the definition of a complex data type (EPOCHTIME);
  - for the declaration of a data item (Interval);
- the exchanged data set contains two data of the same type (BULLETIN).

So there is no one-to-one relationship between type declarations and data occurrences. Only variables can describe what kind of data is actually exchanged or stored. In this case, the described data set is a concatenation of an Epochtime, a Bulletin, an Interval and another Bulletin.

3.4.2 PHYSICAL DATA DESCRIPTION PACKAGE

The second package contains the physical description as specified in 3.3 It includes the following EAST statements in the following order:

- a mandatory statement beginning with the keyword "*package*" followed by an EAST identifier, which is supposed to be the name of the physical data description part, and by the keyword "*is*";

- two optional statements giving the array storage method, as specified in 3.3.2;

- two optional statements giving the octet storage method, as specified in 3.3.3;

- optional statements giving the actual representations of scalar types, that is type declarations as specified in 3.3.4 and 3.3.5, and constant declarations providing the actual representations;

- a set of optional statements giving the association of basic (i.e. scalar) type names and their actual representations (this point is further developed);

- a mandatory statement beginning with the keyword "*end*" followed by the name of the physical data description part, and followed by the character ";".

Representations of scalar types are provided in the physical description part. However, the relationship between these representations and the type names which are given in the logical description part still have to be specified. This is achieved with the following declarations.

An enumeration type declaration must be present to give all the basic type names defined in the logical description part, i.e., all integer type names, all real type names and some enumeration type names (the ASCII one). Every name is prefixed by "USER_TYPE_" in the following list:

```
type BASIC_TYPE_NAMES is (USER_TYPE_xxx , USER_TYPE_yyy ,
                          USER_TYPE_zzz);
```

Template 3-10 of the Physical Description

where xxx, yyy, zzz are the names of the basic types.

The different representations used are declared using the following constant declarations:

```
Binary_Representation_01 : constant INTEGER_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS => n,
      COMPLEMENT =>m ,
      LOCATION => (      1 => (r,s) ,
                         ...
                         n => (t,u)));

Binary_Representation_02 : constant REAL_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS_IN_EXPONENT => n1,
      NUMBER_OF_SUBFIELDS_IN_MANTISSA => n2,
      CONVENTION_USED => FCSTC000, -- IEEE754
      SIGN_BIT_NUMBER => z,
      COMPLEMENT => m,
      EXPONENT_BASE => d,
      BIAS => i,
      LOCATION_OF_EXPONENT =>   (1 => (r,s) ,
                                 ...
                                 n1 => (t,u)),
      LOCATION_OF_MANTISSA =>   (1 => (v,w) ,
                                 ...
                                 n2 => (x,y)));

ASCII_Representation_01 : constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_CHARACTERS => d);

ASCII_Representation_02 : constant
      ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_CHARACTERS => j,
      NUMBER_OF_OCCURRENCES => k,
      REPRESENTATION => ("...", ...));
```

Example 3-60: Template for ASCII and Binary Physical Descriptions

where n, n1, n2 are the numbers of subfields; r, s, t, u, v, w, x, y, z are bit numbers; m indicates the sign convention; d, i, j and k are positive numbers.

Finally the relation between the scalar type names and their representations is specified as follows:

```
type RELATION(choice : BASIC_TYPE_NAMES) is record
      case choice is
            when USER_TYPE_xxx =>
                  PHYS_xxx : INTEGER_PHYSICAL_DESCRIPTION :=
                        Binary_Representation_01;
            when USER_TYPE_yyy =>
                  PHYS_yyy : REAL_PHYSICAL_DESCRIPTION :=
                        Binary_Representation_02;
            when USER_TYPE_zzz =>
                  PHYS_zzz : ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
                        ASCII_Representation_01;
      end case;
end record;
```

Example 3-61: Template for Relation Type Definition

The integer physical representation Binary_Representation_01 is associated with the logical type named “xxx”, which has been previously defined in the associated logical package as seen in 3.2. In the same way, the real physical representation Binary_Representation_02 is associated with the logical type named “yyy”, and the ASCII physical representation ASCII_Representation_01 is associated with the logical type named “zzz”.

NOTES

1 The syntax of the type RELATION is not free. In particular the number of components and their names (PHYS_...) are imposed by the number of enumeration literals of the type BASIC_TYPE_NAMES and the names of these literals.

2 The names of the physical representation are free. The names Binary_Representation_xx and ASCII_Representation_xx are just examples. The only applicable rule is that all the physical representation names must be different.

Rules about the content of the physical data description package:

Rule 26 The array storage is optional (ARRAY_STORAGE_METHOD type and ARRAY_STORAGE constant) if there is no multi-dimensional array in the logical part, or if the method is FIRST_INDEX_FIRST (default value).

Rule 27 The octet storage is optional (BIT_ORDER type and OCTET_STORAGE constant) if the method is HIGH_ORDER_FIRST (default value).

Rule 28 The type REAL_PHYSICAL_DESCRIPTION is optional if there is no binary representation for real type to provide, i.e. if there is no binary real type in the logical part.

Rule 29 The type INTEGER_PHYSICAL_DESCRIPTION is optional if there is no binary representation for integer type to provide, i.e. if there is no binary integer type in the logical part or if there are all considered as machine-independent integers (unsigned integers or two's complement signed integers).

Rule 30 The type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION is optional if there is no ASCII representation for enumeration type to provide, i.e. if there is no ASCII enumeration type in the logical part.

Rule 31 The type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is optional if there is no ASCII representation for integer or real type to provide, i.e. if there is no ASCII integer type and no ASCII real type in the logical part.

Rule 32 The types BASIC_TYPE_NAMES and RELATION are optional if there is no representation to provide.

For all these rules, see also section 3.3.5. of reference [1].

The following example provides a physical data description package, which could be associated with the logical data description package presented in the previous example. This logical package defined some basic types:

- one enumeration type (BULLETIN_KIND);
- five 16 bit integer types (YEAR, MONTH, DAY_OF_MONTH, HOUR, MINUTE);
- one 32 bit real type (SECOND);
- four 64 bit real types (KILOMETERS, KM_SEC, RATIO, ANGULAR_DEGREE).

These eleven basic types are described in the physical package. They are named in the enumeration type BASIC_TYPE_NAMES, and their binary or ASCII representations are provided (five different representations). The following physical description assumes that the data have been generated on a SUN host machine (with the IEEE convention for the real generation).

In order to distinguish the templates from the user defined parts, the adopted convention in the example is that bold text represents the templates, i.e., the constant information.

```
package physical_CNES_description_01 is

type ARRAY_STORAGE_METHOD is (        FIRST_INDEX_FIRST ,
                                      LAST_INDEX_FIRST) ;
ARRAY_STORAGE : constant ARRAY_STORAGE_METHOD :=
                                      FIRST_INDEX_FIRST;

type BIT_ORDER is (HIGH_ORDER_FIRST , LOW_ORDER_FIRST) ;
OCTET_STORAGE : constant BIT_ORDER := HIGH_ORDER_FIRST;

type LOCATION_OF_SUBFIELD is record
      BEGINNING_AT_BIT_NUMBER : NATURAL_NUMBER ;
      ENDING_AT_BIT_NUMBER : NATURAL_NUMBER ;
end record;

MAXIMUM_NUMBER_OF_SUBFIELDS : constant := 255;

type SUBFIELD_NUMBER is range 1 .. MAXIMUM_NUMBER_OF_SUBFIELDS;

type LOCATION_OF_FIELD is array (SUBFIELD_NUMBER range <>)
      of LOCATION_OF_SUBFIELD;

type SIGN_CONVENTION is (UNSIGNED, SIGN_AND_MAGNITUDE,
      ONES_COMPLEMENT, TWOS_COMPLEMENT);

type LIST_OF_RECOGNIZED_CONVENTIONS is (FCSTC000, FCSTC001);

                                      .../...
```

Example 3-62: Complete Physical Description (1 of 4)

```
type INTEGER_PHYSICAL_DESCRIPTION
      (NUMBER_OF_SUBFIELDS : SUBFIELD_NUMBER := 1)
is record
      COMPLEMENT : SIGN_CONVENTION;
      LOCATION : LOCATION_OF_FIELD(1 .. NUMBER_OF_SUBFIELDS);
end record;

type STRING_LIST is array(NATURAL_NUMBER  range <>,
NATURAL_NUMBER  range <>) of CHARACTER;

type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION (
      NUMBER_OF_OCCURRENCES : NATURAL_NUMBER  := 0;
      NUMBER_OF_CHARACTERS : NATURAL_NUMBER  := 0) is record
      REPRESENTATION : STRING_LIST (1 .. NUMBER_OF_OCCURRENCES,
                                    1 .. NUMBER_OF_CHARACTERS);
end record;

type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is record
      NUMBER_OF_CHARACTERS : NATURAL_NUMBER ;
end record;

type REAL_PHYSICAL_DESCRIPTION(
      NUMBER_OF_SUBFIELDS_IN_EXPONENT : SUBFIELD_NUMBER := 1;
      NUMBER_OF_SUBFIELDS_IN_MANTISSA : SUBFIELD_NUMBER := 1)
is record
      CONVENTION_USED : LIST_OF_RECOGNIZED_CONVENTIONS;
      SIGN_BIT_NUMBER : NATURAL_NUMBER ;
      COMPLEMENT : SIGN_CONVENTION;
      EXPONENT_BASE : NATURAL_NUMBER ;
      BIAS : NATURAL_NUMBER ;
      LOCATION_OF_EXPONENT : LOCATION_OF_FIELD
            (1 .. NUMBER_OF_SUBFIELDS_IN_EXPONENT);
      LOCATION_OF_MANTISSA : LOCATION_OF_FIELD
            (1 .. NUMBER_OF_SUBFIELDS_IN_MANTISSA);
end record;
```

.../...

Example 3-62: Complete Physical Description (2 of 4)

```
type BASIC_TYPE_NAMES is ( USER_TYPE_BULLETIN_KIND,
      USER_TYPE_YEAR, USER_TYPE_MONTH, USER_TYPE_DAY_OF_MONTH,
      USER_TYPE_HOUR, USER_TYPE_MINUTE, USER_TYPE_SECOND,
      USER_TYPE_KILOMETERS, USER_TYPE_KM_SEC, USER_TYPE_RATIO,
      USER_TYPE_DEGREE);

-- actual binary representations
Binary_Representation_01 : constant INTEGER_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS => 1, COMPLEMENT => TWOS_COMPLEMENT,
      LOCATION => (1 => (0,31)));

ASCII_Representation_01 : constant
ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_OCCURRENCES => 2, NUMBER_OF_CHARACTERS => 9,
      REPRESENTATION => ("CARTESIAN" , "KEPLERIAN"));

Binary_Representation_02 : constant REAL_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS_IN_EXPONENT => 1,
      NUMBER_OF_SUBFIELDS_IN_MANTISSA => 1,
      CONVENTION_USED => FCSTC000, -- IEEE754
      SIGN_BIT_NUMBER => 0,
      COMPLEMENT => SIGN_AND_MAGNITUDE,
      EXPONENT_BASE => 2,
      BIAS => 1023,
      LOCATION_OF_EXPONENT =>    (1 => (1,11)) ,
      LOCATION_OF_MANTISSA =>    (1 => (12,63)));

Binary_Representation_03 : constant REAL_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS_IN_EXPONENT => 1,
      NUMBER_OF_SUBFIELDS_IN_MANTISSA => 1,
      CONVENTION_USED => FCSTC000, -- IEEE754,
      SIGN_BIT_NUMBER => 0,
      COMPLEMENT => SIGN_AND_MAGNITUDE,
      EXPONENT_BASE => 2,
      BIAS => 127,
      LOCATION_OF_EXPONENT =>    (1 => (1,8)) ,
      LOCATION_OF_MANTISSA =>    (1 => (9,31)));
```

.../...

Example 3-62: Complete Physical Description (3 of 4)

```
type RELATION(choice : BASIC_TYPE_NAMES) is record
case choice is
when USER_TYPE_BULLETIN_KIND =>
  PHYS_BULLETIN_KIND : ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
       ASCII_Representation_01;
when USER_TYPE_YEAR=>
  PHYS_YEAR : INTEGER_PHYSICAL_DESCRIPTION :=
       Binary_Representation_01;
when USER_TYPE_MONTH=>
  PHYS_MONTH : INTEGER_PHYSICAL_DESCRIPTION :=
       Binary_Representation_01;
when USER_TYPE_DAY_OF_MONTH=>
  PHYS_DAY_OF_MONTH : INTEGER_PHYSICAL_DESCRIPTION :=
       Binary_Representation_01;
when USER_TYPE_HOUR=>
  PHYS_HOUR : INTEGER_PHYSICAL_DESCRIPTION :=
       Binary_Representation_01;
when USER_TYPE_MINUTE=>
  PHYS_HOUR : INTEGER_PHYSICAL_DESCRIPTION :=
       Binary_Representation_01;
when USER_TYPE_SECOND =>
  PHYS_SECOND: REAL_PHYSICAL_DESCRIPTION :=
       Binary_Representation_03;
when USER_TYPE_KILOMETERS=>
  PHYS_KILOMETERS: REAL_PHYSICAL_DESCRIPTION :=
       Binary_Representation_02;
when USER_TYPE_KM_SEC =>
  PHYS_KM_SEC: REAL_PHYSICAL_DESCRIPTION :=
       Binary_Representation_02;
when USER_TYPE_RATIO =>
  PHYS_RATIO : REAL_PHYSICAL_DESCRIPTION :=
       Binary_Representation_02;
when USER_TYPE_DEGREE =>
  PHYS_DEGREE : REAL_PHYSICAL_DESCRIPTION :=
       Binary_Representation_02;
end case;
end record;

end physical_CNES_description_01;
```

Example 3-62: Complete Physical Description (4 of 4)

4 USING EAST DATA DESCRIPTION RECORD

This section provides an example to illustrate the use of an EAST Data Description Record, from an end user point of view. Subsection 4.1 explains how to use the logical description, while 4.2 explains how to use the physical description.

4.1 USING LOGICAL DESCRIPTIONS

The logical part of a data description is required by a user to define the application that will consume (or make use of) the data. It contains the information relative to the nature of the received data. A user may be helped by a tool (see 5.4 and annex C) to extract some data from a data block. The user must be able to identify (or name) the data item from which he wants to get the value. For that purpose, EAST provides the dot notation.

The described data correspond to the objects (variables or constants) declared in the second section of the logical package. A name denotes either a declared object or a subcomponent of a declared object.

The following example gives a simple logical description package:

```
package logical_CNES_description_02 is

-- first section: declaration of types
type MEASUREMENT is range 0 .. 65000;
for MEASUREMENT'size use 16; -- bits
type SECOND is digits 8 range 0.0000000 .. 60.000000;
for SECOND'size use 64; -- bits
type DATED_MEASUREMENT is record
        TEMPERATURE : MEASUREMENT;
        DATE_OFFSET : SECOND;
end record;
for DATED_MEASUREMENT use
record
        TEMPERATURE at 0 range 0 .. 15;
        DATE_OFFSET at 0 range 16 .. 79;
end record;
for DATED_MEASUREMENT'size use 80; -- bits
type MEASUREMENT_BLOCK is array (1..10,1..10) of DATED_MEASUREMENT;
for MEASUREMENT_BLOCK'size use 8000; -- bits

-- second section: declaration of variables
SOURCE_DATA : MEASUREMENT_BLOCK;
end logical_CNES_description_02;
```

Example 4-1: Complete Logical Description

The data object contained in the data block is “SOURCE_DATA”. The access path to the complete data set is:

“SOURCE_DATA” -- 100 of DATED_MEASUREMENT

The access path to subcomponents are:

“SOURCE_DATA(1..10, 2)” -- 10 of DATED_MEASUREMENT
or “SOURCE_DATA(3,5)” -- 1 DATED_MEASUREMENT
or “SOURCE_DATA(5, 1 .. 5).DATE_OFFSET” -- 5 of SECOND
or “SOURCE_DATA(1,1).TEMPERATURE” -- 1 MEASUREMENT

NOTES

1 The dot notation (“.”) is used to select one component of a record.

2 The slice notation (“(1 .. 5)”) is used to select a number of contiguous elements of an array.

3 The dot notation and slice notation can be composed to access elements of a record contained in an array, or elements of an array contained in a record.

4.2 USING PHYSICAL DESCRIPTIONS

The physical part of a data description is required by an interpretation tool, or a specific decommutation program, to retrieve the values of the described data. The following example gives a possible physical description package associated with the previous logical description package.

```
package physical_CNES_description_02 is

type ARRAY_STORAGE_METHOD is (FIRST_INDEX_FIRST ,
LAST_INDEX_FIRST) ;
ARRAY_STORAGE : constant ARRAY_STORAGE_METHOD :=
FIRST_INDEX_FIRST;

type BIT_ORDER is (HIGH_ORDER_FIRST , LOW_ORDER_FIRST) ;
OCTET_STORAGE : constant BIT_ORDER := HIGH_ORDER_FIRST;

type LOCATION_OF_SUBFIELD is record
      BEGINNING_AT_BIT_NUMBER : NATURAL_NUMBER ;
      ENDING_AT_BIT_NUMBER : NATURAL_NUMBER ;
end record;

MAXIMUM_NUMBER_OF_SUBFIELDS : constant := 255;

type SUBFIELD_NUMBER is range 1 .. MAXIMUM_NUMBER_OF_SUBFIELDS;

type LOCATION_OF_FIELD is array (SUBFIELD_NUMBER range <>)
            of LOCATION_OF_SUBFIELD;

type SIGN_CONVENTION is (UNSIGNED, SIGN_AND_MAGNITUDE,
            ONES_COMPLEMENT, TWOS_COMPLEMENT);

type LIST_OF_RECOGNIZED_CONVENTIONS is (FCSTC000, FCSTC001);

type INTEGER_PHYSICAL_DESCRIPTION
      (NUMBER_OF_SUBFIELDS : SUBFIELD_NUMBER := 1)
is record
      COMPLEMENT : SIGN_CONVENTION;
      LOCATION : LOCATION_OF_FIELD(1 .. NUMBER_OF_SUBFIELDS);
end record;
```

.../...

Example 4-2: Complete Physical Description (1 of 3)

```
type REAL_PHYSICAL_DESCRIPTION(
      NUMBER_OF_SUBFIELDS_IN_EXPONENT : SUBFIELD_NUMBER := 1;
      NUMBER_OF_SUBFIELDS_IN_MANTISSA : SUBFIELD_NUMBER := 1)
is record
      CONVENTION_USED : LIST_OF_RECOGNIZED_CONVENTIONS;
      SIGN_BIT_NUMBER : NATURAL_NUMBER ;
      COMPLEMENT : SIGN_CONVENTION;
      EXPONENT_BASE : NATURAL_NUMBER ;
      BIAS : NATURAL_NUMBER ;
      LOCATION_OF_EXPONENT : LOCATION_OF_FIELD
            (1 .. NUMBER_OF_SUBFIELDS_IN_EXPONENT);
      LOCATION_OF_MANTISSA :    LOCATION_OF_FIELD
            (1 .. NUMBER_OF_SUBFIELDS_IN_MANTISSA);
end record;

type BASIC_TYPE_NAMES is ( USER_TYPE_MEASUREMENT,
USER_TYPE_SECOND);

-- actual binary representations
Binary_Representation_01 : constant INTEGER_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS => 1, COMPLEMENT => UNSIGNED,
      LOCATION => (1 => (0,15)));
Binary_Representation_02 : constant REAL_PHYSICAL_DESCRIPTION :=
      (NUMBER_OF_SUBFIELDS_IN_EXPONENT => 1,
      NUMBER_OF_SUBFIELDS_IN_MANTISSA => 1,
      CONVENTION_USED => FCSTC000, -- IEEE754
      SIGN_BIT_NUMBER => 0,
      COMPLEMENT => SIGN_AND_MAGNITUDE,
      EXPONENT_BASE => 2,
      BIAS => 1023,
      LOCATION_OF_EXPONENT =>    (1 => (1,11)) ,
      LOCATION_OF_MANTISSA =>    (1 => (12,63)));

                                                .../...
```

Example 4-2: Complete Physical Description (2 of 3)

```
type RELATION(choice : BASIC_TYPE_NAMES) is record
    case choice is
      when USER_TYPE_MEASUREMENT =>
            PHYS_MEASUREMENT : INTEGER_PHYSICAL_DESCRIPTION :=
                  Binary_Representation_01;
      when USER_TYPE_SECOND =>
            PHYS_SECOND: REAL_PHYSICAL_DESCRIPTION :=
                  Binary_Representation_02;
    end case;
end record;

end physical_CNES_description_02;
```

Example 4-2: Complete Physical Description (3 of 3)

The value of the data item named "SOURCE_DATA(1,1).TEMPERATURE" is retrieved according to the following algorithm:

> SOURCE_DATA(1,1) is the first element of the array. TEMPERATURE is located on the first 16 bits of an element of the array SOURCE_DATA(1,1). Temperature is therefore located on the first 16 bits of the data block. These 16 bits correspond to an unsigned integer, whose binary representation indicates that the most significant bit is the first encountered (bit 0) and that the least significant bit is the last bit encountered (bit 15).

The value of the data item named "SOURCE_DATA(3,5).DATE_OFFSET" is retrieved according to the following algorithm:

> SOURCE_DATA(3,5) is the 43^{th} element of the array because the array storage method specifies that the first index varies first. DATE_OFFSET is located on the last 64 bits of an element of the array (i.e., from bit 16 to bit 79). The size of an element of the array is 80 bits. The DATE_OFFSET of the 43^{th} element is therefore located from bit (42*80 + 16) to bit (42*80 + 79), i.e., from bit 3376 to bit 3439. These 64 bits correspond to an IEEE real whose binary representation indicates that the exponent is located from bit 1 to bit 11 and that the mantissa is located from bit 12 to bit 63.

5 RECOMMENDED PRACTICES AND LIMITATIONS

Since EAST is based on the Ada programming language, there are a number of restrictions and practices, which are described in 5.1 and 5.2.

5.1 RESERVED KEYWORDS

Since the EAST syntax is fully compatible with the Ada syntax, it might be possible to include an EAST description in an Ada application. In order to keep the compatibility, it is recommended that EAST reserved keywords as well as pure Ada reserved keywords not be used.

Below are listed the EAST reserved keywords which are, for some of them, Ada reserved keywords too, and for others, pure EAST reserved identifiers. The other Ada keywords are listed too.

5.1.1 EAST (AND ADA) KEYWORDS

array	digits	is	package	type
at				
	end	null	range	use
case			record	
constant	for	of		when
		others	subtype	

5.1.2 PURE EAST RESERVED IDENTIFIERS

virtual_...	word_32_bits	word_16_bits

NOTE − Identifiers of the physical part (INTEGER_PHYSICAL_DESCRIPTION, USER_TYPE_..., RELATION, etc.) are not EAST reserved identifiers, because they are allowed in the logical part of the description for the definition of types and variables.

5.1.3 PURE ADA (AND NOT EAST) KEYWORDS

abort	delta	if	pragma	tagged
abs	do	in	private	task
abstract			procedure	terminate
accept	else	limited	protected	then
access	elsif	loop		
aliased	entry		raise	until
all	exception	mod	rem	
and	exit		renames	while
		new	requeue	with
begin	function	not	return	
body			reverse	xor
	generic	or		
declare	goto	out	select	
delay			separate	

NOTE – These keywords are ADA95 keywords (see reference [7]).

5.2 RECOMMENDED USAGE OF THE EAST SYNTAX

Section 3 provides information on how to define types, using the EAST syntax, in order to logically describe data. Nevertheless, some recommendations are summarized in this section to enhance the readability of the data descriptions. An EAST description might be syntactically correct but might have at the same time a very poor expressiveness.

EAST is not intended to define formal semantics, but the grouping of types in structures and the naming of types convey a large amount of semantic information to the human reader.

Type Names

Type Names should provide indications on the data that they describe. For example, a type called SIXTEEN_BIT_INTEGER_TYPE gives less semantic information than a type called ORBIT_COUNTER, or SYNCHRONIZATION_VALUE, although all these types are implemented using a 16 bit integer.

Acronyms should be avoided: for example, the enumeration literals “Telemetry” and “Telecommand” should be preferred to “TM” and “TC”, although the actual representations are the character strings: “TM” and “TC”.

Type Natures

Enumeration types should be used in preference to integer types, each time the integer values have a particular meaning. For example, a type called MONTH, which is used to define a date, can be considered as an integer with the range: 1 to 12. It can also be defined as an enumeration type that has 12 alternative values: JANUARY, FEBRUARY, ..., DECEMBER. In this case, an enumeration representation clause specifies the enumeration values from 1 to 12.

Type Structuring

The grouping of types into structures is recommended each time data types are in relationship together: a repetition of measurements should be described by an array. The elements of a date (year, month, day, etc.) should be aggregated into a structure (e.g., called DATE).

5.3 IDENTIFIED LIMITATIONS OF EAST TO DESCRIBE DATA

The following limitations have been identified:

a) Discrete discriminants

In EAST, the type of a discriminant must be discrete. Because of this fact, only enumeration types and integer types are allowed to discriminate records: the binary and the ASCII representations are allowed, so it is possible to have character strings as discriminants when using the ASCII representation of enumeration types.

Other types cannot be considered to be discriminants; e.g., a component of a record cannot depend on the value of a real, of an array, of a record, etc.

- Real types have been banished because the floating point representation (that is an approximation of the actual value) forbids any comparison between real values.

- Array types and record types are aggregation types; the need relative to a record or to an array as a discriminant corresponds to components that depend at the same time on the value of many other components. This can be translated into multiple discriminants. Subsection 3.2.11 explains how to use more than one discriminant to discriminate the same component in a record.

b) Static discriminants

A discriminant must be a static expression, which means that it cannot result from a computation. For example, EAST does not allow the specification of a component A that exists if the value of a component B plus the value of the component C has a given value.

This problem is an algorithmic problem. It can be solved at data design time. During the data design process, it is easy to add a component within the data block that represents the result of the computation, and to make this component the discriminant.

c) Multiple ranges

Multiple ranges are not provided. So it is impossible to describe, using EAST, an integer type whose values vary between 0 and 5 and then between 10 and 15, for example. And it is therefore impossible to specify that the index of an array varies between 0 and 5, and between 10 and 15.

Since integer types are used to describe whole numbers resulting from measurements of the real world, it would be curious if the measured phenomenon were not continuous. If the integer values correspond to something else that is discontinuous, then an enumeration type is possibly appropriate to describe these values. It is easy, using enumeration clauses, to specify a gap between two consecutive enumeration literals.

d) Characters

The only character set retained in EAST is the "Latin Alphabet No. 1" character set (see reference [6]).

e) Array Storage applicability

The array storage method is applicable to the whole description. If a data set is composed of data from different sources, all the arrays must be stored in the same way.

5.4 USE OF TOOLS

This section contains a brief list of useful tools in an EAST context. More details about available tools are provided in Annex C.

An *EAST Data Description Generator* is necessary to assist in designing and generating automatically an EAST compliant description of data, relieving the user from concern about the EAST syntax.

The EAST language is based on the Ada syntax. But it contains additional semantic information, so it is possible using EAST to describe most of the data that are to be exchanged. That is the reason why an Ada Compiler is not sufficient to ensure the consistency of an EAST Data Description Record. An *EAST Syntax Checker* might therefore be useful.

The use of EAST as a Data Description Language does not preclude the use of programming languages other than Ada for the application accessing the data. In most cases, a software interface between the application and the data is necessary, i.e., a tool that parses and analyses the Data Description Record, and sometimes a tool that converts the data to the “right” format, i.e., a readable format for the application. Such a tool is called a *Data Interpreter*. Ada is a privileged language (but not the only possible language) for an application accessing data described using EAST, because in some cases EAST data specifications can be included in the Ada application code (with some modifications that are the addition of specific Ada language instructions to the Ada compiler like “pragma pack” and the suppression of some representation clauses used to promote compiler independence in EAST).

6 EAST AND DATA DESCRIPTION LANGUAGE REQUIREMENTS

This section is a discussion on the compliance of the EAST language with the requirements that a Data Description Language shall be designed to satisfy.

These requirements and their rationales are listed in 1.2.

R1. Good readability

The readability is not intrinsic to the EAST language. The recommended usage of the EAST language relative to the type naming and type structuring (see 5.2) enhances the readability of EAST data descriptions.

R2. Support of basic types

The EAST language supports character, enumeration, integer and real types.

R3. Data type definition capabilities

The EAST language supports the programming language concept, called type, that defines a model, defined once, that can be used to create many occurrences of the model.

R4. Data type structuring capabilities

The EAST language supports array and record types. An array type is the relationship of homogeneous data items, i.e., describes a repetition of data items of the same type. A record is the relationship of heterogeneous data items, i.e., describes an ordered aggregation of data items of any type.

R5. Separation of the description from the data

EAST descriptions are physically separated from the data to which they are related.

R6. Physical representation capabilities

The EAST physical packages specify the bit pattern representation of the described data.

These requirements are high-level requirements, specified in the document “Language Usage in Information Interchange” (see reference [2]). Additional detailed level requirements and EAST compliance are listed in Annex E.

ANNEX A

ACRONYMS AND GLOSSARY

This annex defines key acronyms and the glossary of terms which are used throughout this Report to describe the Data Description Language EAST.

A 1 ACRONYMS

ADID	Authority and Description IDentifier
ADU	Application Data Unit
ASCII	American Standard Code for Information Interchange
CA	Control Authority
CCSDS	Consultative Committee for Space Data Systems
DDL	Data Description Language
DDR	Data Description Record
DDU	Description Data Unit
DED	Data Entity Dictionary
DIL	Data Interchange Language
EAST	Enhanced Ada SubseT
EDU	Exchange Data Unit
MACAO	Member Agency Control Authority Office
MSB	Most Significant Bit
LSB	Least Significant Bit
SFDU	Standard Formatted Data Unit

A 2 GLOSSARY OF TERMS

ADID: in the context of EAST, an ADID is an identifier of the EAST recommendation within the CCSDS organization. See Reference [4].

Array type: an array type is a composite type whose components are all of the same type. Components are selected by indexing.

Based literal: a based literal is a numeric literal expressed in a form that specifies the base explicitly.

Bit string: a bit string is a sequence of bits, each having the value 0 or 1.

Character literal: a character literal is formed by enclosing a graphic character between two apostrophe characters.

Character type: a character type is an enumeration type that represents a character set.

Comment: a comment starts with two adjacent hyphens and extends up to the end of the line.

Composite type: a composite type is a collection of components of the same or different types.

Constant: a constant is a keyword that indicates that the identifier it qualifies has a unique and specified value.

Constrained array: a constrained array is an array with a constant number of elements.

Delimiter: a delimiter is one of the following compositions of special characters: & ' ( ) * + , - . / : ; < = > | => .. ** := /= >= <= << >> <>

Discrete type: a discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case statements and as array indexes.

Discriminant: a discriminant is a component of a record type whose value influences the structure of this record.

Elementary type: an elementary type does not have components.

Enumeration representation clause: an enumeration representation clause specifies the bit pattern for each literal of the corresponding enumeration type.

Enumeration type: an enumeration type is defined by the list of its values, called enumeration literals, which may be identifiers or character literals. All values for a given enumeration type are different.

Identifier: an identifier is composed of letters, digits and underline characters.

Length clause: a length clause specifies the amount of storage in bits associated with a type.

Lexical element: a lexical element is either a delimiter, an identifier, a numeric literal, a string literal or a comment.

Literal: a literal is a value represented by its value itself instead of an identifier. A literal can be specialized as a numeric literal, an enumeration literal, a character literal, or a string literal.

Marker: a marker is a constant value provided by a data description. This value will be found in the data as an end-delimiter of a repetition.

Numeric literal: a numeric literal is the value of a number, expressed by means of characters.

Object: an object is either a constant or a variable.

Predefined type: a predefined type is a type provided by EAST, that is, a type that can be used in any EAST description without being previously declared.

Record representation clause: a record representation clause specifies the storage representation of the record type on the medium, that is, the order, position and size of record components (including discriminants, if any).

Record type: a record type is a composite type consisting of zero or more named components, possibly of different types.

Representation clause: representation clauses specify the mapping between types of the language and their physical representation.

Scalar type: scalar types are discrete types and real types.

Separator: a separator is any of a space character, a control character or the end of a line (see 3.1).

String literal: a string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets.

Subtype: a subtype is a type together with a constraint, which constrains the values of the type to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

Type: a type is a named set of characteristics. This name can be used to define sets of values.

Unconstrained array: an unconstrained array is an array with a variable number of elements.

Variable: a variable is an identifier that represents a data item occurrence.

Variant part: a variant part of a record specifies alternative record components, dependent on the discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.

Virtual Discriminant: a virtual discriminant is a discriminant that is not included in the composite type that it discriminates.

ANNEX B

SYNTAX RULES

This annex contains the usage rules identified in this document, followed by the page number to which they refer.

ANNEX C

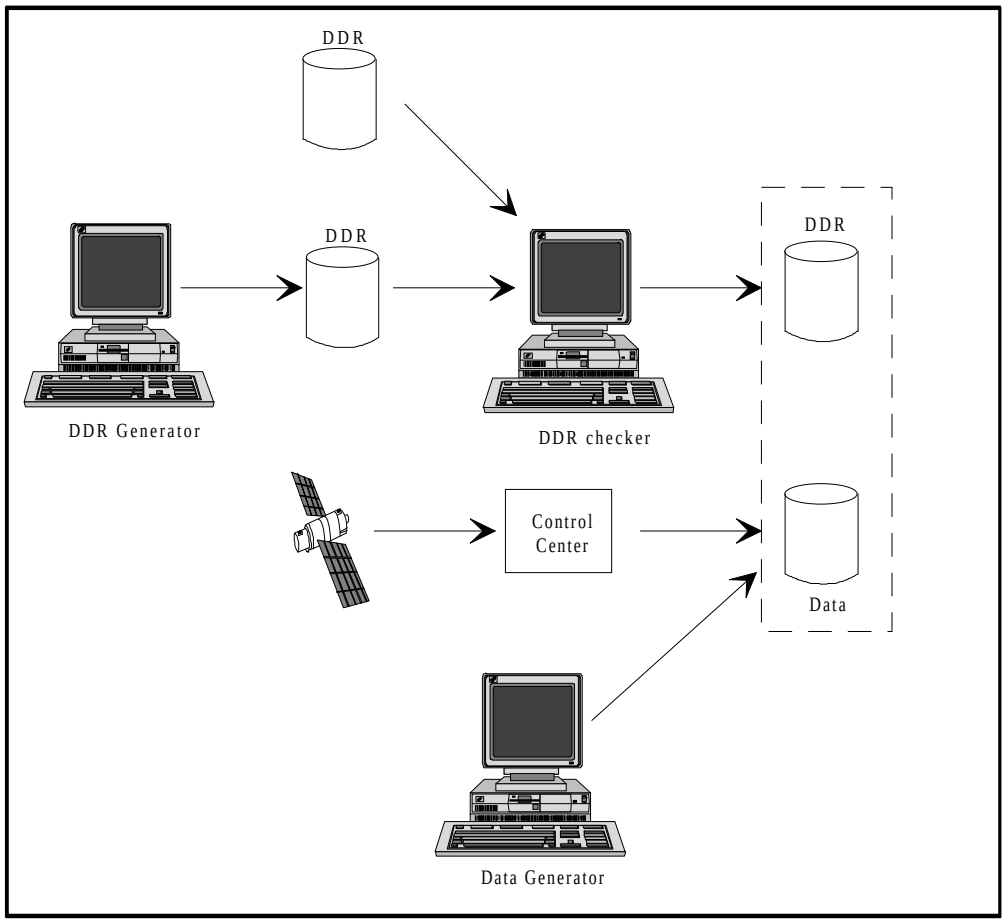
TOOLS FOR AN EAST ENVIRONMENT

This annex lists some of the available tools that are useful to check, generate, parse and analyze EAST DDRs.

The first group of tools is useful to the data definers, while the second group is useful to end users.

For additional information about the available tools, please contact the CCSDS Secretariat or the relevant MACAO (see reference [11]).

C 1 DATA DEFINER TOOLS



C 1.1 DATA DESCRIPTION RECORD GENERATOR

The description of data, using EAST, can easily be automated: in particular, the physical description which follows precise rules, as described in 3.3. A tool that automatically generates DDRs (including logical and physical description) is available. This tool, based on a Graphical User Interface, allows users, without any notion of the EAST language, to describe data logically, and the tool automatically generates a physical description according to the nature of a selected host machine. This physical description does not have to be visible to the user. On the contrary, the logical description is easy to understand (see 3.2) and is readable by any user.

C 1.2 DATA DESCRIPTION RECORD SYNTAX CHECKER

Most of the EAST DDRs should be produced by a tool that automatically generates correct DDRs. Nevertheless an EAST Syntax Checker might be useful to check the syntax of an EAST DDR.

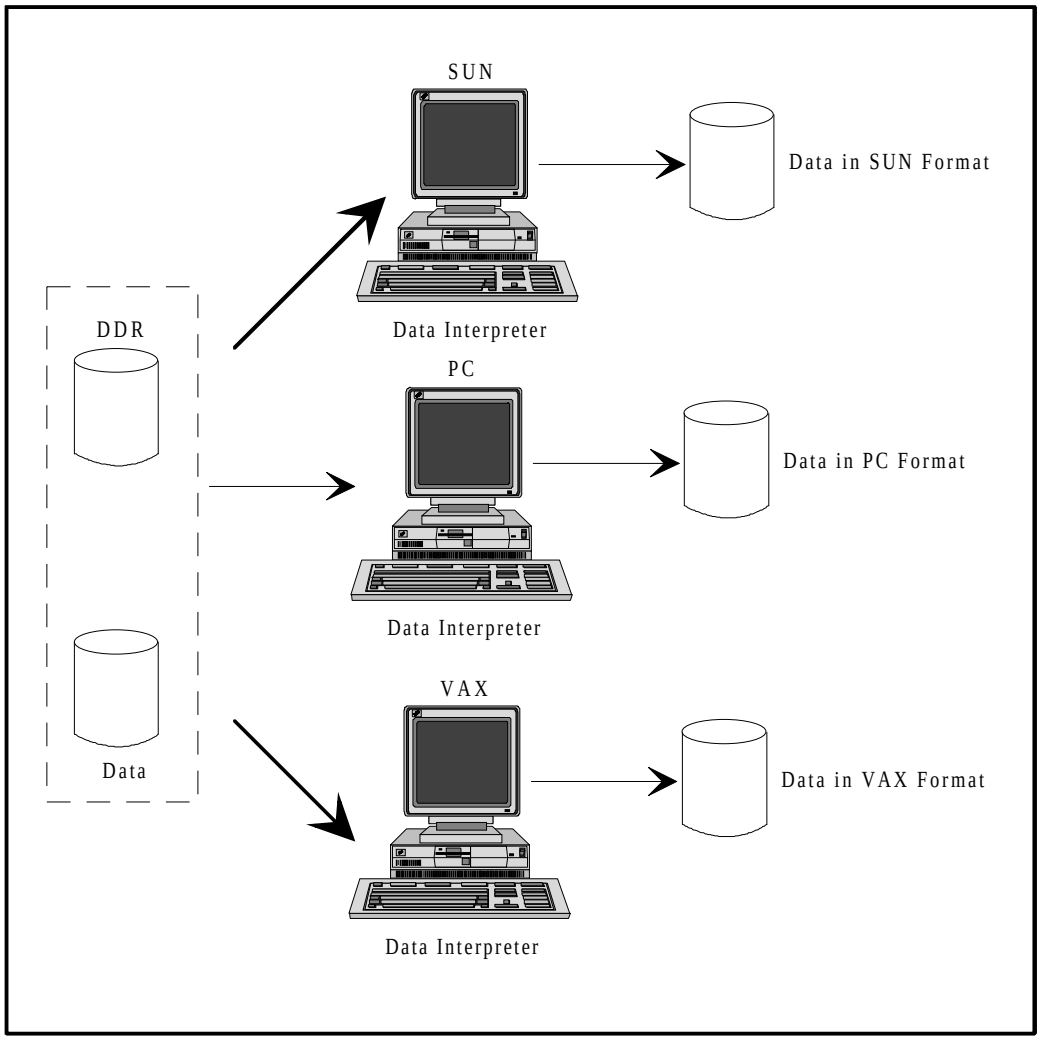
If no syntax checker is available, the EAST description can be passed through an Ada compiler, because of the full compatibility of the EAST syntax with the Ada syntax. But one must keep in mind that such a verification does not ensure that the description is correct. If the description includes non-EAST features that are Ada features, the Ada compiler will not identify these.

C 1.3 DATA GENERATOR

Some users may have to describe data that will be generated in the future and does not yet exist. In this case, in addition to a data description tool, a data generation tool could be useful. This tool would generate data on a medium exactly as the user has described them, regardless of the host machine used to generate.

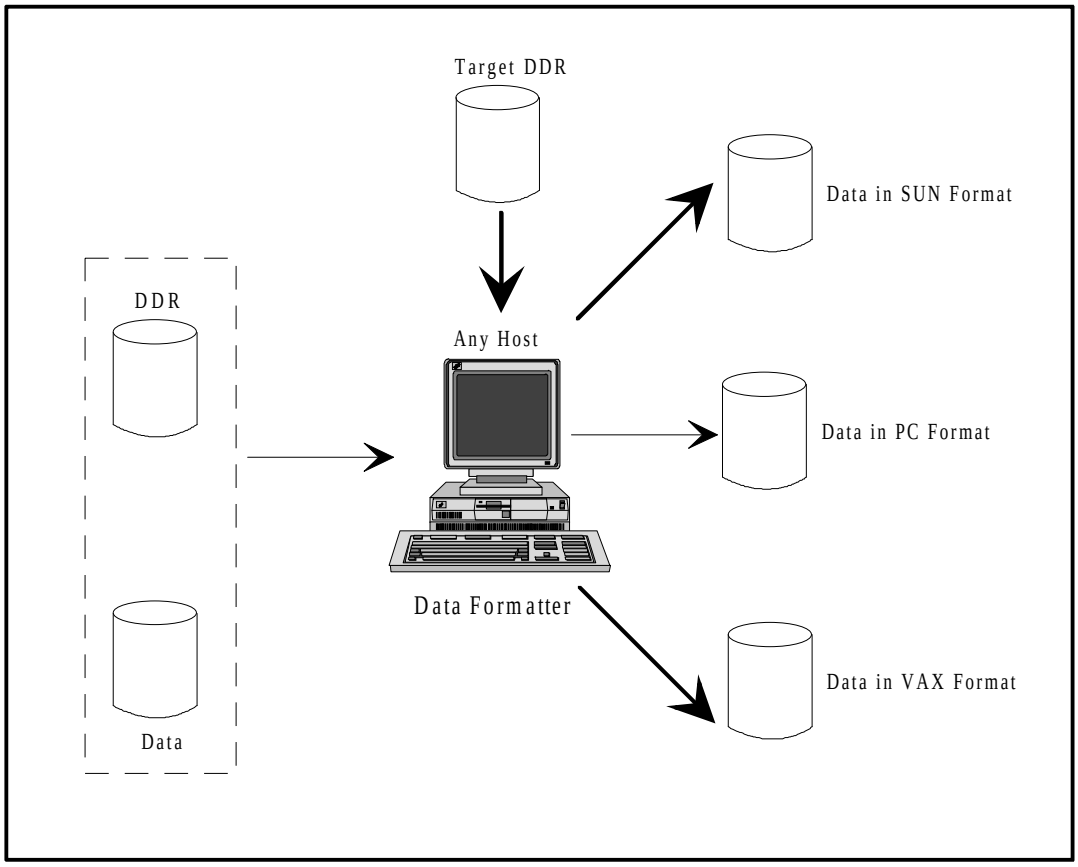
C 2 END USER TOOLS

C 2.1 DATA INTERPRETER



Generally, it is necessary to have an Interpretation tool that parses, analyses, and converts data according to the host machine that will make use of the data. Such a tool is available. This tool analyses an EAST DDR, and offers access services from an application to data blocks described by this DDR.

C 2.2 DATA FORMATTER



If an interpretation tool is not available on a target machine, the use of a data formatter allows the interpretation of the data and their generation in a new format, regardless of the host machine used for the formatting process. This tool is mainly useful for the rehabilitation of historical data (e.g., to change some “strange” 60-bit reals into IEEE754 64 bit reals).

ANNEX D

DATA DESCRIPTION RECORD EXAMPLES

This annex contains an example of a DDR in EAST.

The following EAST DDR provides the description of an imaginary (but realistic) telemetry.

A textual description, a graphical representation, and finally an EAST description are provided as follows:

D 1 TEXTUAL DESCRIPTION

The telemetry is a flow of FORMATs. Each FORMAT is preceded by a SYNCHRO bit pattern (DF3 hexadecimal bit pattern). Each FORMAT is composed of 28 LINEs.

Each LINE begins with a field that identifies which scientific INSTRUMENT (CAMERA, ALTIMETER or INTERFEROMETER) data are provided by this LINE.

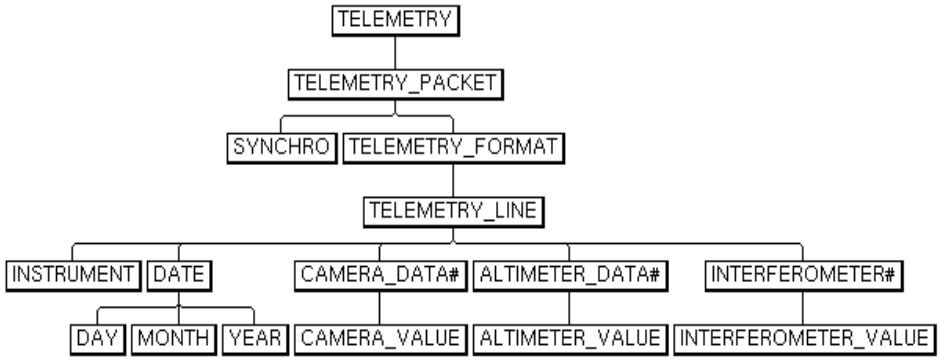
The first LINE field is followed by a DATE related to the first INSTRUMENT VALUE provided by the LINE. This DATE is composed of three sub-fields (DAY, MONTH and YEAR).

The end of the LINE (following INSTRUMENT and DATE) depends on the INSTRUMENT:

- in a CAMERA line there are 40 (8 bits) VALUEs from the CAMERA;
- in an ALTIMETER line there are 20 (16 bits) VALUEs from the ALTIMETER;
- in an INTERFEROMETER line there are 10 (32 bits) VALUEs from the ALTIMETER.

So, each VALUE field has a constant (320 bits) length.

D 2 GRAPHICAL DESCRIPTION



NOTE – The # sign after a field identifier means that this field is optional depending on the value of another field. Here each value field presence depends on the value of the instrument field.

D 3 FORMAL EAST DESCRIPTION

```
package logical_TELEMETRY is

type A_SYNCHRO_PATTERN is ( SYNCHRO_PATTERN ) ;
for A_SYNCHRO_PATTERN use ( SYNCHRO_PATTERN => 16#DF3# ) ;
for A_SYNCHRO_PATTERN'size use 12 ; -- bits

type AN_INSTRUMENT is ( CAMERA , ALTIMETER , INTERFEROMETER ) ;
for AN_INSTRUMENT use (  CAMERA => 0 , ALTIMETER => 1 ,
                                 INTERFEROMETER => 2 ) ;
for AN_INSTRUMENT'size use 2 ; -- bits

type A_DAY_IN_A_MONTH is range 1 .. 31 ;
for A_DAY_IN_A_MONTH'size use 5 ; -- bits

type A_MONTH is (  JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
                   AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER ) ;
for A_MONTH use (  JANUARY => 1 , FEBRUARY => 2 , MARCH => 3 , APRIL => 4 ,
                   MAY => 5 , JUNE => 6 , JULY => 7 , AUGUST => 8 ,
                   SEPTEMBER => 9 , OCTOBER => 10, NOVEMBER => 11,
                   DECEMBER => 12) ;
for A_MONTH'size use 4 ; -- bits

type A_YEAR is range 1950 .. 2100 ;
for A_YEAR'size use 12 ; -- bits
```

```
type A_DD_MM_YY_DATE is record
        DAY : A_DAY_IN_A_MONTH ;
        MONTH : A_MONTH ;
        YEAR : A_YEAR ;
end record ;
for A_DD_MM_YY_DATE'size use 21 ; -- bits
for A_DD_MM_YY_DATE use record
        DAY at 0 * WORD_32_BITS range 0 .. 4 ;
        MONTH at 0 * WORD_32_BITS range 5 .. 8 ;
        YEAR at 0 * WORD_32_BITS range 9 .. 20 ;
end record ;

type A_CAMERA_VALUE is range 0 .. 255 ;
for A_CAMERA_VALUE'size use 8 ; -- bits

type CAMERA_VALUE_INDEX is range 1 .. 40 ;
for CAMERA_VALUE_INDEX'size use 6 ; -- bits

type CAMERA_DATA_VALUES is array ( CAMERA_VALUE_INDEX ) of
        A_CAMERA_VALUE ;
for CAMERA_DATA_VALUES'size use 320 ; -- bits

type AN_ALTIMETER_VALUE is range 0 .. 65535 ;
for AN_ALTIMETER_VALUE'size use 16 ; -- bits

type ALTIMETER_VALUE_INDEX is range 1 .. 20 ;
for ALTIMETER_VALUE_INDEX'size use 5 ; -- bits

type ALTIMETER_DATA_VALUES is array ( ALTIMETER_VALUE_INDEX ) of
        AN_ALTIMETER_VALUE ;
for ALTIMETER_DATA_VALUES'size use 320 ; -- bits

type AN_INTERFEROMETER_VALUE is range 0 ..  2147483646 ;
for AN_INTERFEROMETER_VALUE'size use 32 ; -- bits

type INTERFEROMETER_VALUE_INDEX is range 1 .. 10 ;
for INTERFEROMETER_VALUE_INDEX'size use 4 ; -- bits

type INTERFEROMETER_DATA_VALUES is array
        ( INTERFEROMETER_VALUE_INDEX) of AN_INTERFEROMETER_VALUE;
for INTERFEROMETER_DATA_VALUES'size use 320 ; -- bits
```

```
type A_TELEMETRY_LINE ( INSTRUMENT : AN_INSTRUMENT := CAMERA) is
record
        DATE : A_DD_MM_YY_DATE ;
        case INSTRUMENT is
                when CAMERA =>
                        CAMERA_DATA : CAMERA_DATA_VALUES ;
                when ALTIMETER =>
                        ALTIMETER_DATA : ALTIMETER_DATA_VALUES ;
                when INTERFEROMETER =>
                        INTERFEROMETER : INTERFEROMETER_DATA_VALUES ;
                when OTHERS =>
                        null ;
        end case ;
end record ;
for A_TELEMETRY_LINE use record
        INSTRUMENT at 0 * WORD_32_BITS range 0 .. 1 ;
        DATE at 0 * WORD_32_BITS range 2 .. 22 ;
        CAMERA_DATA at 0 * WORD_32_BITS range 23 .. 342 ;
        ALTIMETER_DATA at 0 * WORD_32_BITS range 23 .. 342 ;
        INTERFEROMETER at 0 * WORD_32_BITS range 23 .. 342 ;

end record ;

type A_TELEMETRY_LINE_NUMBER is range 1 .. 28 ;
for A_TELEMETRY_LINE_NUMBER'size use 5 ; -- bits

type A_TELEMETRY_FORMAT is array ( A_TELEMETRY_LINE_NUMBER ) of
        A_TELEMETRY_LINE ;

type A_TELEMETRY_PACKET is record
        SYNCHRO : A_SYNCHRO_PATTERN ;
        TELEMETRY_FORMAT : A_TELEMETRY_FORMAT ;
end record ;
for A_TELEMETRY_PACKET use record
        SYNCHRO at 0 * WORD_32_BITS range 0 .. 11 ; -- bits
end record ;

TELEMETRY_PACKET : A_TELEMETRY_PACKET ;

end logical_TELEMETRY ;


package physical_TELEMETRY is

-- of no use

end physical_TELEMETRY;
```

NOTE – The physical part of an EAST description usually includes the representation of scalar types, the way of storing arrays on the medium and the way of storing octets on the medium. In this example, there are no reals, no machine-dependent integers and no ASCII encoded scalar types, so the part related to the representation of scalar types is useless. The arrays are all one-dimensional arrays, so the information related to the way of varying indices is useless. The way of storing octets is by default HIGH_ORDER_FIRST, so this information item can also be missing.

ANNEX E

COMPLIANCE MATRIX

This annex provides a compliance matrix according to detailed data description requirements.

Requirements	**EAST Compliance**
General features	
Naming of base data types	yes
Aggregation of base data types	yes
Naming of aggregations	yes
Aggregation of aggregations	yes
Choice (by discriminant) of base data types	yes
Immediately preceding discriminant	yes
Remote preceding discriminant	yes
Choice (by discriminant) of aggregations	yes
Immediately preceding discriminant	yes
Remote preceding discriminant	yes
Calculated discriminant	no
Arrays of base data types	yes
Arrays of aggregations	yes
Arrays of >2 dimensions	yes
Features by base data type	
Integers (binary)	yes
Complement types supported	yes
0’s	yes
1’s	yes
2’s	yes
Bit ordering	yes
Random or special ordering	yes
Byte ordering (MSB, LSB)	yes
Other byte ordering	no
Maximum and minimum value specifiable	yes
Multiple ranges specifiable	no
Character-coded numerical	yes

Enumerated	yes
Highly flexible naming	yes
Size in bits specifiable	yes
Bit ordering	yes[4]
Byte ordering	yes
Physical representation	yes
Logical	no[5]
User definitions of TRUE and FALSE	yes
Real	yes
Size in bits specifiable	yes
Maximum and minimum value specifiable	yes
Multiple ranges specifiable	no
Multiple locations of exponent/mantissa	yes
Algorithm used is specifiable	yes
Exponent position	yes
Size in bits	yes
Bit ordering	yes
Byte ordering	yes
Bias is specifiable	yes
Mantissa position	yes
Size in bits	yes
Bit ordering	yes
Byte ordering	yes
Standard representations	
IEEE754	yes
DEC VAX	yes
IBM “3081”	yes
Mil_STD_1750A	yes
others	yes[6]

[4] The bit ordering and byte ordering are linked together and specified in EAST using the type “BIT_ORDER”.

[5] The logical type is not predefined in EAST, but can be user-defined with the enumerated type.

[6] Every new standard can be accommodated in EAST.

Bit string	yes
Size in bits specifiable	yes
Bit ordering	yes[4]
Byte ordering	yes
Permitted bit patterns	yes
Naming of bit pattern	yes
Octet string	yes
Size in octets	yes
Octet ordering	yes
Non sequential order	no
Permitted values	no
Null	yes
Character string	yes
Size in characters specifiable	yes
Single byte characters	yes
Double byte characters	no
Different character sets	yes
Definable subsets	yes
Permitted values	yes
Other useful types	
Times	no[7]
Dates	no[7]

Discriminants by value of

Integer	yes
Enumerated	yes
Logical	yes
Bit string	yes
Octet string	no
Character string	no
Aggregations	no

[7] Not predefined in EAST but user-definable.

Operations involving discriminants

Logical operations on several discriminants	yes
Arithmetic operations on several discriminants	no
Select an element on a multi-value discriminant	yes
If - Then -Else / Case	yes
Do until a discriminant reaches a limit	no
Do While	no
Do until	no
Repeat until a calculated discriminant is equaled	no

Software support

Tool to generate description	yes
Tool to generate in conformity with data from description	no
Tool to check description syntax	no
Tool to parse description and associated data, and build data structure tree	yes
Tool to browse data structure tree and present/return values	yes
Library to access data structures/values from software (callable tools)	prototype

ANNEX F

COMPARISON BETWEEN ADA AND EAST

This annex provides the main differences between the Ada programming language and EAST. It is mainly addressed to Ada programmers.

F1 on page 3-5 The Ada syntax allows any positive integer for the base; EAST indeed restricts the possible base values to 2, 8, 10 and 16.

F2 on page 3-6 In Ada, the value of a length clause specifies an upper bound for the number of bits to be allocated to instances of the given type. In EAST, the value specifies the exact number of bits that any instance of the given type occupies.

F3 on page 3-7 The rule 4 states that numeric value must increase: this EAST syntax rule is inherited from the Ada programming language and has been maintained for compatibility reasons.

F4 on page 3-8 The Ada predefined type “BOOLEAN”, which is in Ada a particular enumeration type, is not provided in EAST as a predefined EAST type, because this Ada type is implementation-defined. The bit patterns associated with the values “TRUE” and “FALSE” depend on the host machine. The length of the predefined type BOOLEAN also depends on the host machine.

F5 on page 3-9 The predefined types CHARACTER and STRING are exactly the same as in Ada (see reference [7]).

F6 on page 3-16 The declaration of data instances that have a variable number of elements is correct because of the default discriminant required by rule 10. In Ada, the default discriminant is a requirement only if there is an unconstrained part in the variant structure. In EAST, the default discriminant is always mandatory. Therefore, in EAST, every instantiation does not have to include a discriminant value although the variant structure is required to have a discriminant.

F7 on page 3-24 In Ada, a record representation clause specifies the storage representation of records in memory, that is the order, position and size of record components in memory of a given machine. In EAST, the record representation clause specifies the actual storage representation on the medium.

F8 on page 3-32 If the logical part of and EAST description should be incorporated in an Ada program, the following statements should be first added:

with System; -- before the keyword package

and the definition of the chosen distance for record representation clauses:

WORD_16_BITS : constant = 16/System.Storage_Unit;
or
WORD_32_BITS : constant = 32/System.Storage_Unit;

The statement "with System;" is referred to the System package implemented on an Ada Compiler. The expression "16/System.Storage_Unit" represents 16 bits while the expression "32/System.Storage_Unit" represents 32 bits, independently of the machine configuration.

EAST only provides two numbers WORD_16_BITS and WORD_32_BITS, and not for example WORD_8_BITS or WORD_64_BITS. If the user wants to define one of the two last numbers and use it in a record representation clause, he must be aware, that the use of an 8 bit word is not portable on any architecture (e.g., not allowed on a 1750-A machine that has a 16 bit architecture). In the same way, a 64 bit word is not used on usual machines.

F9 on page 3-51 The Ada programming language offers some predefined types and subtypes as character, string, Boolean, integer, float, natural, positive, etc.

Some of them are *completely defined*, like CHARACTER and STRING. So they can be widely used.

Some other ones (BOOLEAN, INTEGER, FLOAT, etc.) are *implementation-defined*: their binary representations (sign position, etc.) depend on the implementation. Such implementation-defined types must therefore be banished in any logical description. They are therefore not provided by EAST for the definition of data. There is no use restriction of these types in the physical part of an EAST description.

The subtypes of implementation-defined types must be banished too:

subtype NATURAL **is** INTEGER **range** 0 .. INTEGER'LAST;
subtype POSITIVE **is** INTEGER **range** 1 .. INTEGER'LAST;

E.g., the INTEGER predefined type is a 16 bit integer on a PC-DOS machine and a 32 bit integer on a SUN-UNIX machine.

F10 on page 3-65 An EAST definition must appear before it is used. This rule is inherited from Ada.

INDEX