ASN.1 and BER Reference
DRAFT 3.4

January 17, 1994

Michael F. Gering

IBM Corporation
Dept. E78, Building 673
P. O. Box 12195
RTP, NC 27609
External: (919) 254-4420
Tie Line: 444-4420
VNET: GERING at RALVM6
internet: gering@ralvm6.vnet.ibm.com

Contents

Overview

1.0 ASN1BER Package

User's Guide

Chapter 2. ASN.1 Compiler 2-1
2.1 Source File 2-1
2.1.1 Script and Bookmaster 2-1
2.1.2 Note to Users of EBCDIC machines 2-2
2.1.3 Syntactic Extensions 2-2
2.1.4 Structure Considerations 2-2
2.1.5 Language Limitations 2-2
2.2 Error File 2-2
2.3 *TBL* file 2-3
2.4 ASN1 - VM/CMS Version 2-4
2.5 ASN1 - DOS and OS/2 Version 2-5
2.6 ASN1 - AIX Version 2-6

Chapter 3. Invoking the TBLXLATE Translator 3-1
3.1 *C* file 3-1
3.2 H File 3-1
3.3 TBLXLATE - VM/CMS Version 3-2
3.4 TBLXLATE - DOS and OS/2 Version 3-2
3.5 TBLXLATE - AIX Version 3-2

Chapter 4. The ENCDEC Sample Program 4-1
4.1 ENCDEC - VM/CMS Version 4-4
4.2 ENCDEC - DOS and OS/2 Version 4-5
4.3 ENCDEC - AIX Version 4-5

Chapter 5. ASN.1 Syntax 5-1
5.1 ASN.1 Character Set 5-1
5.2 General Rules 5-1
5.3 Reserved Words 5-2
5.4 Syntactic Guards 5-2
5.5 Type Reference Item 5-2
5.6 Any Table Reference Item 5-4
5.7 Value Reference Item 5-4
5.8 Identifier Item 5-5
5.9 Other Items 5-6
5.9.1 Value Reference Item 5-6
5.9.2 Module Reference Item 5-6
5.9.3 Comment Item 5-6
5.9.4 Number Item 5-6
5.9.5 Binary String Item 5-7
5.9.6 Hexadecimal String Item 5-7
5.9.7 Character String Item (cstring) 5-8
5.9.8 Assignment Item 5-8
5.10 Module Definition 5-8

5.10.1 Module Identification 5-9
5.10.2 Default Tagging 5-9
5.10.3 Module Body Syntax 5-9
5.10.4 Module Example 5-10
5.11 Type Assignment 5-11
5.12 Value Assignment 5-12
5.13 Any Table Assignment 5-12
5.14 Referencing Types, Values and Any Table Definitions 5-13
5.15 Built-In Types 5-14
5.16 BOOLEAN 5-14
5.17 INTEGER 5-15
5.18 ENUMERATED 5-16
5.19 BIT STRING 5-17
5.20 OCTET STRING 5-18
5.21 NULL 5-18
5.22 SEQUENCE 5-19
5.23 SEQUENCE OF 5-21
5.24 SET 5-21
5.25 SET OF 5-22
5.26 CHOICE 5-22
5.27 Selection Type 5-23
5.28 Tagged Types 5-23
5.29 ANY 5-25
5.30 OBJECT IDENTIFIER 5-26
5.31 Character String Types 5-28
5.32 NumericString 5-28
5.33 PrintableString 5-28
5.34 Other Character String Types 5-29
5.35 Generalized Time 5-30
5.36 Universal Time 5-30
5.37 External 5-30
5.38 Object Descriptor 5-31
5.39 Real Number 5-31
5.40 Subtypes 5-32

Chapter 6. Basic Encoding Rules 6-1
6.1 Basic Structure 6-1
6.2 Identifier Field 6-1
6.3 Length Field 6-2
6.4 Contents Field 6-3
6.5 BOOLEAN Value 6-3
6.6 INTEGER And ENUMERATED Value 6-3
6.7 BIT STRING Value 6-3
6.8 OCTET STRING Value 6-4
6.9 Null Value 6-4
6.10 Sets and Sequence Values 6-4
6.11 Tagged Values 6-4
6.12 OBJECT IDENTIFIER Value 6-5

Programmer's Guide

Chapter 7. Writing User Encoding Routines 7-1

Chapter 8. Runtime Library 8-1

8.1 Global Variables 8-1
8.1.1 modules 8-1
8.1.2 freepdus 8-1
8.1.3 usrencfn 8-1
8.1.4 version_string 8-2
8.2 Routines by Category 8-2
8.2.1 Initialization/Termination 8-2
8.2.2 Searching/Accessing 8-3
8.2.3 Data Conversion 8-3
8.2.4 Encoding/Decoding 8-3
8.2.5 Formating 8-3
8.3 Alphabetical Listing 8-3
8.3.1 addmodule 8-3
8.3.2 basetype 8-4
8.3.3 decber 8-5
8.3.4 decode 8-5
8.3.5 encode 8-6
8.3.6 findmodule 8-7
8.3.7 findtbl 8-7
8.3.8 findtype 8-8
8.3.9 findval 8-8
8.3.10 firstmodule 8-9
8.3.11 freepdu 8-10
8.3.12 hexstr 8-10
8.3.13 initmodules 8-11
8.3.14 INTEGERtolong 8-11
8.3.15 longtoINTEGER 8-12
8.3.16 lngs2OID 8-12
8.3.17 newpdu 8-13
8.3.18 OIDtolngs 8-14
8.3.19 printpdu 8-14
8.3.20 PrintUnresolvedImports 8-15
8.3.21 readdatfile 8-15
8.3.22 resolveallimports 8-16
8.3.23 resolveimports 8-17
8.3.24 setusrencfn 8-17
8.3.25 typestr 8-18
8.3.26 unloadmodule 8-18

Chapter 9. The Metatable Structure 9-1
9.1 The Role of the ASN.1 Compiler 9-1
9.2 The Role of the TBLXLATE Program 9-1
9.3 Structure of a TBL File 9-2
9.3.1 Structure of Module Definitions 9-2
9.4 Structure of the *DAT* File 9-6
9.5 Structure of Type Assignments 9-7
9.6 Structure of Types 9-7
9.6.1 BOOLEAN 9-7
9.6.2 INTEGER 9-7
9.6.3 ENUMERATED 9-8
9.6.4 BIT STRING 9-8
9.6.5 OCTET STRING 9-9
9.6.6 NULL 9-9
9.6.7 SEQUENCE 9-9
9.6.8 SEQUENCE OF 9-10

9.6.9 SET . . . 9-10
9.6.10 SET OF . . . 9-11
9.6.11 CHOICE . . . 9-11
9.6.12 Selection Type . . . 9-11
9.6.13 Tagged Type . . . 9-11
9.6.14 Any Type . . . 9-12
9.6.15 OBJECT IDENTIFIER Type . . . 9-12
9.6.16 REAL Type . . . 9-12
9.6.17 Defined Type . . . 9-13
9.6.18 Character String and Useful Types . . . 9-13
9.7 Structure of Subtypes . . . 9-13
9.7.1 SubtypeSpec . . . 9-14
9.7.2 Single Value Constraint . . . 9-14
9.7.3 Value Range Constraint . . . 9-14
9.7.4 Size Constraint . . . 9-15
9.7.5 Contained Subtype . . . 9-15
9.7.6 Permitted Alphabet . . . 9-15
9.7.7 InnerType Constraints . . . 9-15
9.7.8 Unimplemented Types . . . 9-15
9.8 Structure of Value Assignments . . . 9-15
9.9 Structure of Values . . . 9-15
9.9.1 Number . . . 9-15
9.9.2 OBJECT IDENTIFIER Value . . . 9-16
9.9.3 BOOLEAN Value . . . 9-16
9.9.4 NULL Value . . . 9-16
9.9.5 Hex String Value . . . 9-16
9.9.6 Binary String Value . . . 9-16
9.9.7 Character String Value . . . 9-16
9.9.8 CHOICE Value . . . 9-16
9.9.9 ANY Value . . . 9-16
9.9.10 Named Number Values . . . 9-17
9.9.11 Unimplemented Values . . . 9-17
9.10 Structure of Table Assignments and References . . . 9-17
9.10.1 Any Table Assignment . . . 9-17
9.10.2 Any Table References . . . 9-17

Chapter 10. SORTRES . . . 10-1

Chapter 11. References . . . 11-1

Appendix A. ASN1 Compiler Messages . . . A-1

Appendix B. *H* File Example . . . B-1

Appendix C. *C* File Example . . . C-1

Appendix D. Universal Tag Assignments . . . D-1

List of Abbreviations . . . X-1

Index . . . X-3

Figures

1-1. ASN1BER Package - Data Flow Diagram 2
4-1. Example ASN.1 Source File 4-2
4-2. Example ASN1 Invocation 4-1
4-3. Example TBLXLATE Invocation 4-3
4-4. Example ENCDEC Invocation - 1 4-3
4-5. Example ENCDEC Invocation - 2 4-4
5-1. Standard ASN.1 Reserved Words 5-3
5-2. Additional Reserved Words 5-4
5-3. Syntactic Guard Examples 5-5
9-1. Example Symbol Table 9-3
B-1. Example H File B-1
C-1. Example C File C-2

Overview

Abstract Syntax Notation Number One and Basic Encoding Rules are two widely used standards for representing application data which are communicated between cooperating systems. These standards were first defined for use by OSI presentation layer and application layer standards and have since been applied to other data communication specifications including TCP/IP, local area network, and proprietary protocols.

ASN.1 and BER represent different, though related, aspects of data. ASN.1 is a formal notation for describing syntactic and semantic aspects of data. ASN.1 is an *abstract* representation because it does not specify how data are represented in a local computer nor does it specify how data are represented when they are communicated between systems. For two applications to meaningfully communicate with each other, they must first agree on this level of abstraction. The second requirement is that they agree on a way to represent information when it is transferred between them. That is, the applications must agree on a *transfer syntax*. The third form of data representation is the *local syntax*, in which form each application directly manipulates the data. The local syntax depends on the application programming language, the compiler for the programming language, and the computer architecture on which the application runs. In general, the relationship between abstract, transfer, and local syntax representations is many to many to many. For data communications specifications, the abstract and transfer syntaxes are the only relevant representations. ASN.1 is only one possible abstract representation, BER is only one possible transfer representation. For implementations of data communications specifications, all three representations are relevant. As noted, the local representation of data depends on individual languages, compilers, and computer architectures, and so is not completely standardized.

An example helps to illustrate the distinctions and usefulness of these three types of data representations. ASN.1 defines a datatype, `INTEGER`, to represent the positive and negative whole numbers. The BER, or transfer syntax, representation for an `INTEGER` is a twos complement binary number with the most significant bit transmitted first using the fewest possible number of octets. The local syntax for an `INTEGER` may differ according to the programming language, compiler manufacturer, and computer architecture. The C language offers several choices: `int`, `long`, and `short`. Depending on the particular C compiler and its target machine, these data types have different numerical limits and different internal representations. For one compiler and target machine, an `int` might be represented as a two byte, twos complement binary number with the least significant byte first.

1.0 ASN1BER Package

The ASN1BER package consists of programs that deal with all three aspects of data representation. Specifically, it provides programs and source code for:

- ASN1 for compiling the ASN.1 language
- Runtime library routines to encode and decode BER data
- TBLXLATE, a utility to port data definitions from the machine on which the ASN.1 compilation occurs to the machine on which the user application runs
- SORTRES, a utility to help port the ASN.1 compiler to other host machines
- ENCDEC, a sample program to demonstrate the runtime library routines
- Sample ASN.1 source definitions

The data flow diagram in Figure 1-1 on page 2 shows these how these components relate to the application development environment.

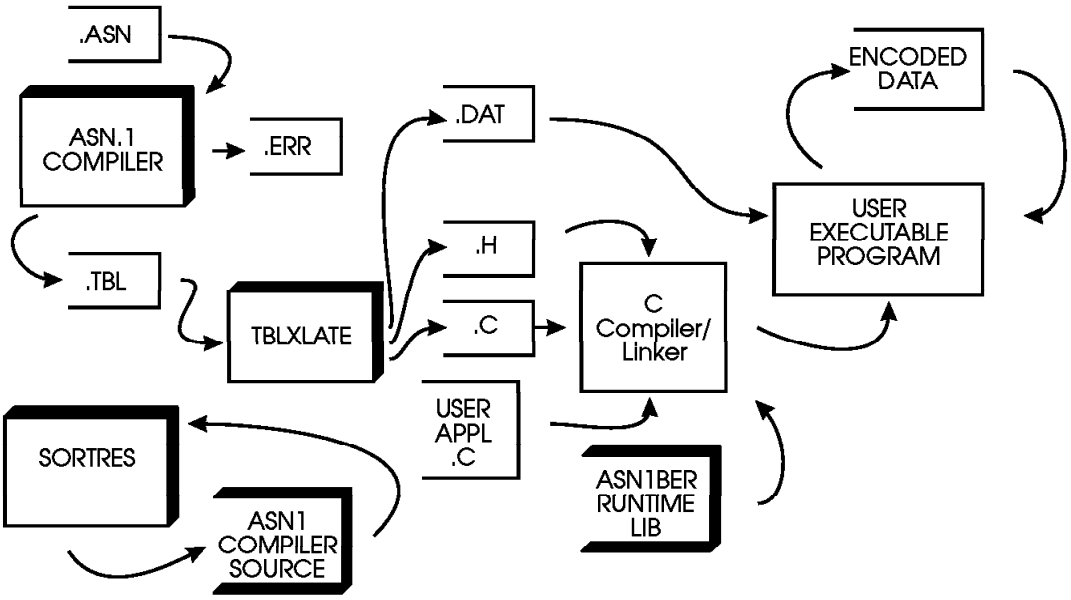


Figure 1-1. ASN1BER Package - Data Flow Diagram

In the data flow diagram, the components with shadows are part of the ASN1BER package.

First, the ASN.1 compiler takes a file, labelled *ASN*, containing ASN.1 source and produces two output files: an error messages file, labelled *ERR*, and an intermediate file, labelled *TBL*.

The TBLXLATE program takes an intermediate file, labelled *TBL*, and translates it into three output files: a metatable data file, labelled *DAT*, a C language header file, labelled *H*, and a metatable C language file, labelled *C*.

The *C* and *H* files may, but need not be, incorporated with a user-written application program. The ASN1BER package includes a runtime library which is used by the user-written application to help it encode and decode data. A C language compiler/linker creates an executable form of the user-written application. Note that the *DAT* file is not compiled or link-edited with the user-written application.

At runtime, the user-written application can load (and unload) *DAT* files, as required, to encode and decode data according to the ASN.1 definitions represented in the *DAT* file. This is the dynamic style. If the program's use of the ASN.1 definitions is more static in nature, the *C* files provide equivalent information; they would be link-edited with the user-written application instead of having the *DAT* files loaded and unloaded. Note that a combination of *DAT* and *C* files may be used by the same user-written application.

The ENCDEC program which is supplied in the ASN1BER package is a sample application to encode/decode data. As such, it is an example of what would be a user-written application.

The SORTRES program is not used for the application development process. Instead, it is a utility program which facilitates porting the ASN1 compiler program to different systems. It takes a file containing a list of reserved words, like "INTEGER", and sorts it according to the native collating sequence on the new system. Unless you are porting the compiler itself to a new system, you don't need to worry about this program.

Each of these programs is explained in following chapters.

User's Guide

Chapter 2. ASN.1 Compiler

The ASN.1 compiler program takes a single file containing abstract syntax notation as source input and produces several output files:

- Error
- *TBL*

2.1 Source File

The source file contains one or more ASN.1 definitions which may be intermixed with non-ASN.1 text. Normally, ASN.1 definitions are combined within ASN.1 modules. Each module definition contains zero or more type definitions, value definitions and comments. Type and value definitions may occur outside module definitions, in which case the compiler treats them as though they were declared as part of a global module named "GLOBAL-MODULE".

2.1.1 Script and Bookmaster

To facilitate use of ASN.1 definitions within IBM Script and Bookmaster documents, the compiler provides the `SCRIPT` option. With the `SCRIPT` option in effect:

- The compiler skips all source lines outside of ASN.1 sections. An ASN.1 section is introduced with a source line of the form ".* asn.1 on". The ".*" must begin in columns one and two, "asn.1" and "on" are separated by one or more spaces and may be upper or lower case. An ASN.1 section is terminated with a source line of the form ".* asn.1 off". Note that these two lines appear as comments to the Script or Bookmaster processor. A source file may have multiple ASN.1 sections with no importance placed on where the section breaks occur with respect to the ASN.1 definitions.
- Within an ASN.1 section, the following symbols are allowed, but not required, in place of their single character equivalents:
  - `&lbrc.` or `&lbrace.` for {
  - `&rbrc.` or `&rbrace.` for }
  - `&lbrk.` or `&lbracket.` for [
  - `&rbrk.` or `&rbracket.` for]
  - `&gml.` for :
  - `&colon.` for :
  - `&period.` or `&dot.` for .
  - `&eq.` or `&equals.` or `&eqsym.` for =
  - `&percent.` for %
  - `&tab.` or `&rbl.` for a space
  - `&us.` for _ (underscore)
- The above substitutions are made inside character string constants, but are not made within ASN.1 comments.
- Within an ASN.1 section, the compiler will ignore any line which begins with a period or a colon in column one. If you absolutely need to begin a line with a period or a colon which is part of an ASN.1 symbol, use `&period.` or `&colon.` to avoid having the compiler ignore the line.

Note that the above facilities are only in effect when the SCRIPT option is invoked. Otherwise, the source file is assumed to contain *only* valid ASN.1 statements without script symbols and ASN.1 sections.

2.1.2 Note to Users of EBCDIC machines

EBCDIC machines have some difficulty handling modern (post 1970) applications which use characters like "[" and "]". For these machines, the ASN1 compiler accepts two EBCDIC characters for "[":

- X'AD' - usually displays correctly on printer devices
- X'BA' - usually displays correctly on terminals

The compiler accepts these two EBCDIC characters for "]":

- X'BD' - usually displays correctly on printer devices
- X'BB' - usually displays correctly on terminals

2.1.3 Syntactic Extensions

In addition to the standard ASN.1 syntax, extensions are provided which enable the user application to efficiently handle the ANY DEFINED BY and EXTERNAL datatypes. These extensions are provided in a standard-compatible way to prevent other ASN.1 compilers from complaining about their presence. See Chapter 5, “ASN.1 Syntax” on page 5-1 for details about the supported ASN.1 syntax.

2.1.4 Structure Considerations

If feasible, structuring all the application's module definitions into one source file is preferable for three reasons. First, the compiler is able to perform additional semantic checks to ensure type and value definitions are compatible with references to them from modules other than where they are declared. Second, the compiler is able to perform certain optimizations on type and value references when the type and value definitions are in different modules. Third, the compiled output will occupy less storage at application runtime since the modules all share the same symbol table, i.e. strings representing type and value references are not replicated when they occur in multiple modules which are compiled from the same source file.

2.1.5 Language Limitations

The ASN.1 macro language is not supported.

2.2 Error File

The Error file lists errors, warning and informational messages about the ASN.1 compilation. Each message indicates the line and column number of the related source file, and the severity of the message.

The message severity ranges from 1 to 4:

- Level 1 messages are informational in nature
- Level 2 messages are warnings that indicate runtime limitations
- Level 3 messages are warnings that indicate questionable use of the ASN.1 language
- Level 4 messages are errors in the ASN.1 source or internally detected compiler errors

The /W compiler option controls the suppression of severity level 1 through 3 messages.

2.3 *TBL* file

The *TBL* file contains the metatable and symbol table definitions corresponding to the ASN.1 source file. The *TBL* format is character data that are intended to be portable. This allows the ASN.1 compiler to be hosted on one type of computer to generate metatable and symbol table information for use by the user application on a different type of computer. Because the *TBL* file format is not efficient for the runtime environment, the TBLXLATE utility is used to translate the *TBL* format into the *DAT*, C, and H formats. The *DAT*, C, and H formats depend on the target machine and C compiler implementation. Hence, they are not generally portable.

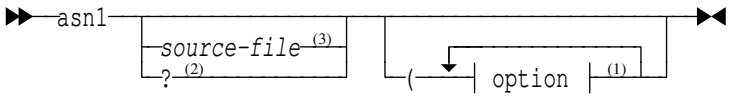
The runtime library contains routines to dynamically load *DAT* files. This approach offers several advantages to the runtime environment:

- The application need only load those modules it needs, thus avoiding wastage of storage in its runtime environment
- Changing the abstract syntax does not require that the application be recompiled and link edited

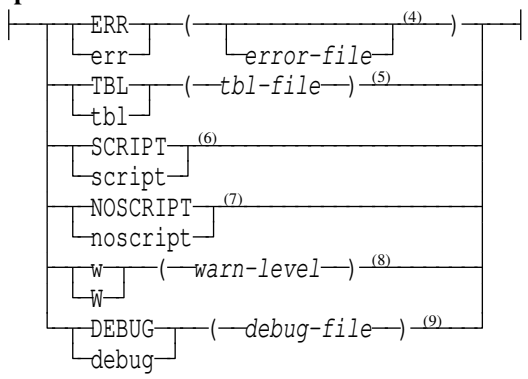
2.4 ASN1 - VM/CMS Version

The syntax for invoking the ASN.1 compiler from a VM/CMS command line is:

ASN1 - VM/CMS



option:



Notes:

1 The options may appear in any order. The characters in the option keywords are case insensitive. An option may be specified more than once, in which case the last specification overrides previous specifications.

2 The `?` option causes syntax help information to be displayed.

3 The *source-file* specifies the filename, type and mode for the ASN.1 input file. The default is the *standard input* file, which is normally the console. Note that you signal the *end of file* condition by entering "/*" on the console in columns one and two. See 3.1, "*C* file" on page 3-1 for information about the C file format and contents.

4 The `err` option specifies the output file for error messages. The *error-file* option, if specified, identifies the file name, filetype and filemode for the output file. The default error output is directed to the *standard output* (*stdout*) file, which is normally the console. The *standard output* file is also indicated by the absence of *error-file*. See 2.2, "Error File" on page 2-2 for information about the error file format and contents.

5 The `tbl` option specifies the output file for the metatable data in the *TBL* format. The *tbl-file* parameter indicates the filename, file type, and file mode for the output file. The default is to produce no *TBL* file. See 2.3, "*TBL* file" on page 2-3 for information about the *TBL* file format and contents.

6 The `script` option specifies that the input file contains Script or Bookmaster formatting controls. See 2.1, "Source File" on page 2-1 for more information about processing the input file.

7 The `noscript` option specifies that the input file does not contain Script or Bookmaster formatting controls. See 2.1, "Source File" on page 2-1 for more information about processing the input file.

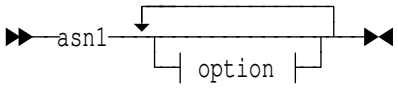
8 The `w` option sets the minimum message warning level. Values range from 1 to 4, and the default is 3. Level 4 messages are considered errors and cannot be suppressed.

9 The `debug` option turns on debugging information for the compiler. The output is sent to the *debug-file*, if it is specified. If *debug-file* is not specified, the output is directed to *standard output*, which is normally the console.

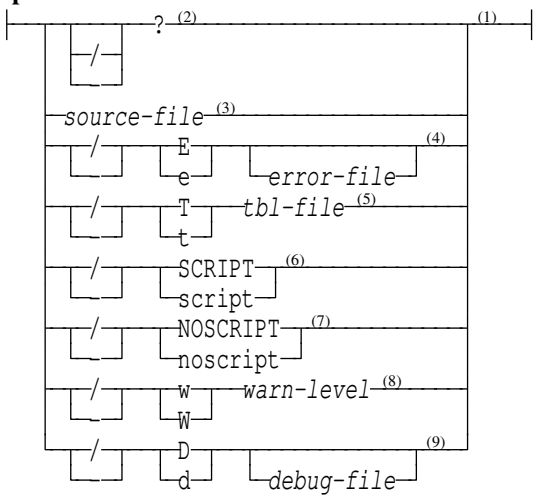
2.5 ASN1 - DOS and OS/2 Version

The syntax for invoking the ASN.1 compiler from an OS/2 or DOS command line is:

ASN1 - DOS and OS/2



option:



Notes:

1 The options may appear in any order. The characters in the option keywords are case insensitive. An option may be specified more than once, in which case the last specification overrides previous specifications.

2 The `?` option causes syntax help information to be displayed.

3 The *source-file* specifies the path and filename for the ASN.1 input file. The default is the *standard input* file, which is normally the console. If input is from the console, remember that you indicate the *end of file* condition by typing *control Z* See 2.1, "Source File" on page 2-1 for information about the source file format and contents.

4 The `/e` option specifies the output file for error messages. The *error-file* option, if specified, identifies the path and file name for the output file. The default error output is directed to the *standard output* file, which is normally the console. The *standard output* file is also indicated by the absence of *error-file*. See 2.2, "Error File" on page 2-2 for information about the error file format and contents.

5 The `/t` option specifies the output file for the metatable data in the *TBL* format. The *tbl-file* parameter indicates the path and filename for the output file. The default for this option is to produce no *TBL* definitions. See 2.3, "*TBL* file" on page 2-3 for information about the *TBL* file format and contents.

6 The `/script` option specifies that the input file contains Script or Bookmaster formatting controls. See 2.1, "Source File" on page 2-1 for more information about processing the input file.

7 The `/noscript` option specifies that the input file does not contain Script or Bookmaster formatting controls. See 2.1, "Source File" on page 2-1 for more information about processing the input file.

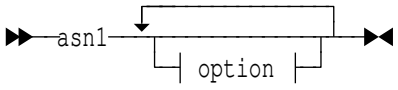
8 The `w` option sets the minimum message warning level. Values range from 1 to 4, and the default is 3. Level 4 messages are considered errors and cannot be suppressed.

9 The `/d` option turns on debugging information for the compiler. The output is sent to the *debug-file*, if it is specified. If *debug-file* is not specified, the output is directed to *standard output* (*stdout*).

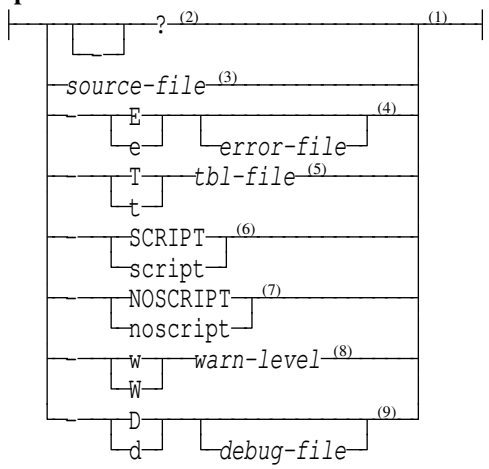
2.6 ASN1 - AIX Version

The syntax for invoking the ASN.1 compiler from an AIX command line is:

ASN1 - AIX



option:



Notes:

1 The options may appear in any order. The characters in the option keywords are case insensitive. An option may be specified more than once, in which case the last specification overrides previous specifications.

2 The `?` option causes syntax help information to be displayed.

3 The *source-file* specifies the path and filename for the ASN.1 input file. The default is the *standard input* file, which is normally the console. If input is from the console, remember that you indicate the *end of file* condition by typing *control D* See 2.1, “Source File” on page 2-1 for information about the source file format and contents.

4 The `-e` option specifies the output file for error messages. The *error-file* option, if specified, identifies the path and file name for the output file. The default error output is directed to the *standard output* file, which is normally the console. The *standard output* file is also indicated by the absence of *error-file*. See 2.2, “Error File” on page 2-2 for information about the error file format and contents.

5 The `-t` option specifies the output file for the metatable data in the *TBL* format. The *tbl-file* parameter indicates the path and filename for the output file. The default for this option is to produce no *TBL* definitions. See 2.3, “*TBL* file” on page 2-3 for information about the *TBL* file format and contents.

6 The `-script` option specifies that the input file contains Script or Bookmaster formatting controls. See 2.1, “Source File” on page 2-1 for more information about processing the input file.

7 The `-noscript` option specifies that the input file does not contain Script or Bookmaster formatting controls. See 2.1, “Source File” on page 2-1 for more information about processing the input file.

8 The `-w` option sets the minimum message warning level. Values range from 1 to 4, and the default is 3. Level 4 messages are considered errors and cannot be suppressed.

9 The `-d` option turns on debugging information for the compiler. The output is sent to the *debug-file*, if it is specified. If *debug-file* is not specified, the output is directed to *standard output* (*stdout*).

Chapter 3. Invoking the TBLXLATE Translator

The following sections detail how to invoke the TBLXLATE translator program. The TBLXLATE program translates a file in *TBL* format to another file in *DAT* format. The user application which performs the encoding and decoding function may use, but is not required to use, the *DAT* file. TBLXLATE checks the input file to ensure that it was created by a compatible version of the ASN1 compiler. If the input file was created by an incompatible version of ASN1, it will terminate with an error message. Similarly, the output *DAT* file contains the version number of the TBLXLATE program and the runtime library routines (specifically, the readdatfile function) check that the *DAT* file version is compatible with the runtime library version. Consequently, you should ensure that new versions of the ASN1, TBLXLATE, and runtime libraries are installed at the same time, and that you recompile your ASN.1 source files, etc.

3.1 *C* file

The *C* file contains C language data definitions which compactly represent the ASN.1 source file in compiled form. These data are called the *metatable*. The metatable represents the syntactic structure of the ASN.1 definitions, and the *symbol table*, which contains the names defined in the ASN.1 source. Each ASN.1 module in the source file has a corresponding metatable in the C file, and one symbol table contains all the names used in all the modules.

The runtime routines for encoding and decoding BER data use these structures. Unless the runtime routines are modified or replaced, the details of these structures are not relevant to the user application program.

The *C* file may be included with another C program, or separately compiled and linked with the user application and runtime library routines.

Several consequences result from using the *C* file:

- Changing the source syntax file requires recompiling and linking the application program.
- Storage for the metatable and symbol table are permanently allocated to the application program.

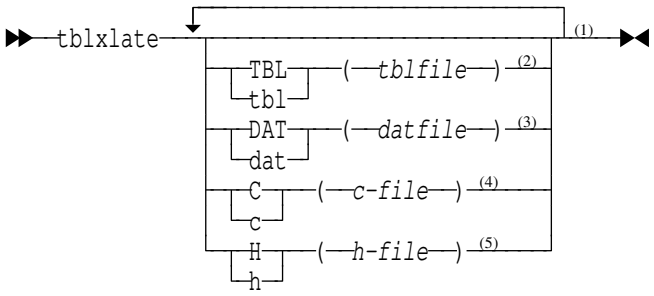
The *DAT* format offers an alternative to the C file format.

3.2 H File

The runtime library uses offsets into metatables to reference type and value definitions. The H file contains `#define` statements to symbolically define these offsets. The runtime library also has search routines to provide offsets given a symbolic name, but this approach is more time consuming than using the definitions in the H file. Note that this header file can be used with the runtime application program regardless whether the metatable is in the C language or the *DAT* format.

3.3 TBLXLATE - VM/CMS Version

TBLXLATE - VM/CMS Version



Notes:

1 The order of the options is not important. A option may be specified more than once, in which case the last specification overrides the previous ones. The keywords may be in any combination of uppercase and lowercase.

2 The *tblfile* file is in the *TBL* format and serves as input to TBLXLATE. The *tblfile* is one of the output files of the ASN1 compiler. If not specified, the default *tblfile* is `ASN1BER TBL`.

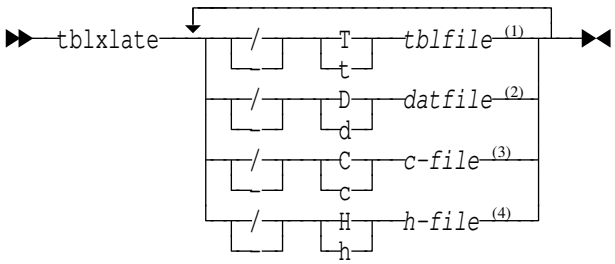
3 The *datfile* file is the output of TBLXLATE and it serves as input to the applications which perform the encoding/decoding functions. If it is not specified, the default *datfile* is `ASN1BER DAT`.

4 The `c` option specifies the output file for the C language metatable definitions. The *c-file* parameter indicates the filename, file type and file mode for the output file. The default for this option is to produce no output file. See 3.1, "*C* file" on page 3-1 for information about the C file format and contents.

5 The `h` option specifies the output file for the C language header definitions. The *h-file* parameter indicates the filename, file type, and file mode for the output file. The default for this option is to produce no output file. See 3.2, "H File" on page 3-1 for information about the header file format and contents.

3.4 TBLXLATE - DOS and OS/2 Version

TBLXLATE - DOS and OS/2



Notes:

1 The *tblfile* file is in the *TBL* format and serves as input to TBLXLATE. The *tblfile* is one of the output files of the ASN1 compiler.

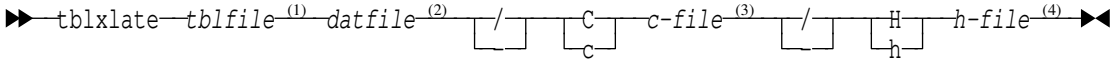
2 The *datfile* file is the output of TBLXLATE and it serves as input to the applications which perform the encoding/decoding functions.

3 The `/c` option specifies the output file for the C language metatable definitions. The *c-file* parameter indicates the path and filename for the output file. The default for this option is to produce no C language definitions. See 3.1, "*C* file" on page 3-1 for information about the C file format and contents.

4 The `/h` option specifies the output file for the C language header definitions. The *h-file* parameter indicates the path and filename for the output file. The default for this option is to produce no C language header definitions. See 3.2, "H File" on page 3-1 for information about the header file format and contents.

3.5 TBLXLATE - AIX Version

TBLXLATE - AIX



Notes:

1 The *tblfile* file is in the *TBL* format and serves as input to TBLXLATE. The *tblfile* is one of the output files of the ASN1 compiler.

2 The *datfile* file is the output of TBLXLATE and it serves as input to the applications which perform the encoding/decoding functions.

3 The /c option specifies the output file for the C language metatable definitions. The *c-file* parameter indicates the path and filename for the output file. The default for this option is to produce no C language definitions. See 3.1, "*C* file" on page 3-1 for information about the C file format and contents.

4 The /h option specifies the output file for the C language header definitions. The *h-file* parameter indicates the path and filename for the output file. The default for this option is to produce no C language header definitions. See 3.2, "H File" on page 3-1 for information about the header file format and contents.

Chapter 4. The ENCDEC Sample Program

The ASN1BER package includes a sample application program, ENCDEC, which illustrates use of the runtime library for encoding and decoding data according to user-provided ASN.1 definitions. It is also useful to see how a particular ASN.1 value would be translated into BER format.

All versions of ENCDEC work alike. They take a single invocation parameter, the name of a *DAT* format file. ENCDEC will attempt to read in the *DAT* file and, if successful, enter a loop to query the user for a data type to encode and decode. After entering the name of the datatype to encode, ENCDEC will start asking for values which comprise the datatype. When all the necessary values are entered, ENCDEC will build a BER-encoded PDU in storage. It will then decode that PDU and build internal data structures to represent the PDU, and it will print a formatted version of the PDU. This encode/decode combination is useful to check that the encoding and decoding runtime library routines are operating correctly, and the formatted output is useful to "see" what a BER version of the PDU looks like. ENCDEC then loops back to ask for another datatype name to encode and decode. The loop is terminated when the user types "stop".

The following example uses the file `test0.asn`. The contents of `test0.asn` are shown in Figure 4-1 on page 4-2.

Figure 4-2 shows an example compilation of this file on DOS or OS/2.

```
[C:\BASN1]asn1 test0.asn /ttest0.tbl
Copyright (c) Michael F. Gering, 1991, 1992, ASN1/BER version 1.13

Compiling test0.asn on Tue Feb 11 20:05:01 1992


Options:
          table: test0.dat
          error: standard output
  warning level: 3

Line Col Lv Num Message Text
---- --- -- --- -----------------
---- --- -- End of Messages -----

0 Warnings
0 Errors
Return code 0

[C:\BASN1]
```

Figure 4-2. Example ASN1 Invocation

Now run the `test0.tbl` file through the TBLXLATE program as shown in Figure 4-3 on page 4-3.

```
-- test0.asn


  Abc DEFINITIONS
      IMPLICIT TAGS
      ::= BEGIN
      EXPORTS B, D;
      IMPORTS Bitstr, Octstr FROM Xyz;

    A ::= [PRIVATE 12] EXPLICIT INTEGER (0..2 | a)
    B ::= [APPLICATION 42] IMPLICIT INTEGER (-10<..10 | 20..30)
    C ::= BOOLEAN
    D ::= NULL
    E ::= OCTET STRING (SIZE(50|42) | '01234'H)
    F ::= REAL
    G ::= CHOICE{ foo1 INTEGER (MIN..<MAX),
                  foo2 BOOLEAN,
                  n NULL,
                  os OCTET STRING }
    H ::= BIT STRING (SIZE(20..MAX))
    K ::= SEQUENCE SIZE(0..10) OF Bitstr
    L ::= SET SIZE(2..2) OF E
    a INTEGER ::= 42
    b INTEGER ::= 84
    M ::= SEQUENCE{ b1 BIT STRING,
                  b2 BIT STRING,
                  b3 BIT STRING OPTIONAL,
                  bool BOOLEAN OPTIONAL,
                  int INTEGER (a..b) OPTIONAL,
                  g [12] EXPLICIT G DEFAULT foo1 : 0
                  }
    N ::= SET{ i INTEGER, b BOOLEAN OPTIONAL, n NULL, b2 BIT STRING OPTIONAL}
    O ::= OBJECT IDENTIFIER
    P ::= ENUMERATED {foo(0), fee(10)}
    Q ::= ANY
    R ::= Octstr
    S ::= NumericString
    T ::= PrintableString
    V ::= TeletexString
    W ::= T61String
    X ::= VideotexString
    Y ::= VisibleString
    Z ::= ISO646String
    A1 ::= IA5String
    B1 ::= GraphicString
    C1 ::= GeneralString
    D1 ::= GeneralizedTime
    E1 ::= UTCTime
    F1 ::= EXTERNAL
    G1 ::= ObjectDescriptor
    H1 ::= INTEGER

  END

  Xyz DEFINITIONS
      IMPLICIT TAGS
      ::= BEGIN
    EXPORTS Bitstr, Octstr;
    IMPORTS B, D FROM Abc;

   Bitstr ::= BIT STRING
   Octstr ::= OCTET STRING
   X ::= Abc.B
   Y ::= Abc.D

  END
```

Figure 4-1. Example ASN.1 Source File

```
[C:\BASN1]tblxlate /ttest0.tbl /dtest0.dat
Copyright (c) Michael F. Gering, 1992, ASN1/BER version 1.13

Translating test0.tbl on Tue Feb 11 20:07:08 1992


Options:
          table: test0.tbl
            dat: test0.dat
              c: none
              h: none
          error: standard output

[C:\BASN1]
```

Figure 4-3. Example TBLXLATE Invocation

Now, we can invoke ENCDEC and encode/decode a few of the datatypes defined in test0. In the Figure 4-4, the user input is highlighted.

```
[C:\BASN1]encdec test0.dat

[Abc.]Type or Module.Type (or stop): A

module=Abc, symbol=A, base type=EXPLICIT TAG:

module=Abc, symbol=A, base type=INTEGER:
  numeric value: 123


Abc.A [PRIVATE 12] type=INTEGER lop=5 id.len='EC03'H
  Abc.INTEGER [UNIVERSAL 2] type=INTEGER lop=3 id.len='0201'H loc=1
    contents=123 ('7B'H)

[Abc.]Type or Module.Type (or stop): B

module=Abc, symbol=B, base type=INTEGER:
  numeric value: 123


Abc.B [APPLICATION 42] type=INTEGER lop=4 id.len='5F2A01'H loc=1
  contents=123 ('7B'H)

[Abc.]Type or Module.Type (or stop): stop

&.lbrkC:\BASN1]
```

Figure 4-4. Example ENCDEC Invocation - 1

In the above listing, take note of the following:

- The datatype name may be simplely an identifier, in this case it is **A**, and the default module name appears between the brackets, in this case it is `Abc`
- The decoded and formatted listing includes the following information:
  - The name of the module and symbol (`Abc.A`),
  - the tag information (`[PRIVATE 12]`),
  - the base datatype (`INTEGER`),
  - the length of the PDU (`lop=5`),
  - the hexadecimal representation for the id and length fields of the pdu (`id.len='EC03'H`
- The contents field is indented and printed on the next line.

- The ENCDEC program stops when **stop** is entered for the datatype name.

In another example, see Figure 4-5, we first encode an instance of **Xyz.Octstr** and then an instance of **Abc.G**. Again, the user input is highlighted.

```
[C:\BASN1]encdec test0.dat

[Abc.]Type or Module.Type (or stop): Xyz.Octstr

module=Xyz, symbol=Octstr, base type=OCTET STRING:
  enter hex digits, no spaces: 0123456789ABCDEF


Xyz.Octstr [UNIVERSAL 4] type=OCTET STRING lop=10 id.len='0408'H loc=8
  contents='0123456789ABCDEF'H

[Xyz.]Type or Module.Type (or stop): Abc.G

module=Abc, symbol=G, base type=CHOICE:

module=Abc, symbol=G, base type=INTEGER:
Ok, Skip, Error (o, s, e) ?s

module=Abc, symbol=G, base type=BOOLEAN:
Ok, Skip, Error (o, s, e) ?s

module=Abc, symbol=G, base type=NULL:
Ok, Skip, Error (o, s, e) ?o


Abc.G [UNIVERSAL 5] type=NULL lop=2 id.len='0500'H loc=0
  contents=NONE

[Abc.]Type or Module.Type (or stop): stop

[C:\BASN1]
```

Figure 4-5. Example ENCDEC Invocation - 2

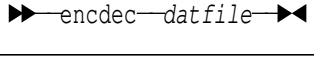
In the above listing, note:

- The ASN.1 external reference syntax is used to specify a datatype in a module other than the default one (in this case, **Xyz.Octstr** and **Abc.G**)
- To select the CHOICE alternative to be encoded, **s** is used to skip over the undesired alternatives, and **o** is used to select the "okay" alternative. The skip/okay/error query is used for other datatypes to signal whether an OPTIONAL element is to be encoded or whether a SET OF or SEQUENCE OF is to be terminated.

4.1 ENCDEC - VM/CMS Version

The syntax for invoking ENCDEC on VM/CMS is as follows:

ENCDEC - VM/CMS



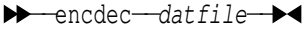
The *datfile* parameter should be enclosed in quotes with the filename, filetype and mode separated by periods (and not by spaces). The following are valid example invocations:

```
encdec "test0.dat"
encdec "test0.dat.a"
```

4.2 ENCDEC - DOS and OS/2 Version

The syntax for invoking ENCDEC on OS/2 or DOS is as follows:

ENCDEC - DOS and OS/2



The *datfile* parameter refers to the path and file for the input file.

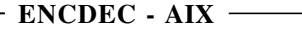
The following are valid example invocations:

```
encdec c:\basn1\test0.dat
encdec \test0.dat
encdec test0.dat
```

4.3 ENCDEC - AIX Version

The syntax for invoking ENCDEC on AIX is as follows:

ENCDEC - AIX



The *datfile* parameter refers to the path and file for the input file.

The following are valid example invocations:

```
encdec c:/basn1/test0.dat
encdec /test0.dat
encdec test0.dat
```

Chapter 5. ASN.1 Syntax

The following sections provide details of the ASN.1 syntax, including supported features and extensions to the syntax.

5.1 ASN.1 Character Set

The standard ASN.1 syntax uses the following characters.

- A to Z
- a to z
- 0 to 9
- :, =, ,, {, }, <, .
- (,), [,], -, ', "
- |

In addition to the standard characters above, the following characters are recognized by the ASN1 compiler (but only in specified contexts):

- %
- _
- &

The ASN.1 language makes distinctions between upper and lowercase letters. Hence, the following ASN.1 items are *not* equivalent:

```
FooBar
foobar
Foobar
```

Rendition aspects of the above characters are not specified or limited by ASN.1. Therefore, the font, size, and color of these characters are unimportant as far as ASN.1 is concerned.

The ASN1 compiler provides some extensions to the standard syntax. One of these extensions is to allow some of the above characters to be represented by Bookmaster/Script substitution symbols. See 2.1, “Source File” on page 2-1 for details of how these substitution symbols work.

5.2 General Rules

The following are general rules about the ASN.1 syntax.

Each ASN.1 item (token) must be contained on single line. In ASN.1, line breaks are significant; with only one exception, a line break is equivalent to a space character. The exception is the the comment item.

The space character is a delimiter, and cannot be used within an item, except for comment item. Items are separated (delimited) by one or more space characters or by line breaks.

Note: The ASN1 compiler does not require that every item be separated by a space or line break. For example, the item "{" need not be surrounded by blanks or line breaks.

ASN.1 places no restriction on the number of characters in a line.

The following example shows a line of ASN.1 and the items it contains.

```
DataTypeA ::= INTEGER { foo(0) } -- need a value for foo
   ↑       ↑     ↑    ↑  ↑↑↑↑ ↑ ↑

                Items
```

5.3 Reserved Words

Standard ASN.1 defines reserved words shown in Figure 5-1 on page 5-3.

In addition to the standard reserved words, the ASN.1 compiler reserves additional words as shown in Figure 5-2 on page 5-4. Notice that the additional reserved words contain the underscore character, which is also an extension to the standard set of ASN.1 characters. This convention prevents standard ASN.1 syntax which was developed without knowledge of these ASN1 extensions from accidentally using these reserved words. To prevent other ASN.1 compilers from complaining about these syntax extensions, a special construct called a *syntactic guard* is provided (see 5.4, “Syntactic Guards”).

5.4 Syntactic Guards

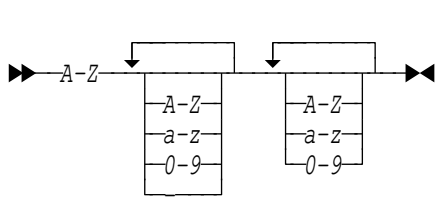
The ASN1 compiler admits extensions to the standard ASN.1 syntax. Since these extensions are unique to the ASN1 compiler, other ASN.1 compilers will generate error messages if they encounter the extensions in your ASN.1 source. To aid portability, the ASN1 compiler provides *syntactic guards* to "hide" these extensions from other compilers. To other compilers, these syntactic guards look like standard ASN.1 comment delimiters.

The starting syntactic guard consists of two hyphens and a percent character, all consecutive, i.e., "--%". The ending syntactic guard consists of a percent character and two consecutive hyphens, i.e., "&percent--". Because syntactic guards are a variation of the ASN.1 comment item, the ending syntactic guard may also be simply the end of the line. If the *script* option is used, the percent character can be represented as "&percent.". For the same reason, you should not use regular ASN.1 comments between a syntactic guard start and end. If you do, then other ASN.1 compilers will interpret your comment start item, "--", as the end of a comment you had started with the syntactic guard start, "--%". Also, you should not nest syntactic guard start/end pairs.

See Figure 5-3 on page 5-5 for examples of syntactic guard usage.

5.5 Type Reference Item

A type reference is a label assigned to an ASN.1 datatype and is used to refer to the datatype. A type reference consists of one or letters, digits, or hyphens in the following form:



```
ABSENT              MAX
ANY                 MIN
APPLICATION         MINUS-INFINITY
BEGIN               NULL
BIT                 NumericString
BOOLEAN             OBJECT
BY                  OCTET
CHOICE              OF
COMPONENT           OPTIONAL
COMPONENTS          ObjectDescriptor
DEFAULT             PLUS-INFINITY
DEFINED             PRESENT
DEFINITIONS         PRIVATE
END                 PrintableString
ENUMERATED          REAL
EXPLICIT            SEQUENCE
EXPORTS             SET
EXTERNAL            SIZE
FALSE               STRING
FROM                T61String
GeneralString       TAGS
GeneralizedTime     TRUE
GraphicString       TeletexString
IA5String           UNIVERSAL
IDENTIFIER          UTCTime
IMPLICIT            VideotexString
IMPORTS             VisibleString
INCLUDES            WITH
INTEGER
ISO646String
```

Figure 5-1. Standard ASN.1 Reserved Words

In the above syntax, note that the initial character is an uppercase letter, and the last character cannot be a hyphen. Also, because ASN.1 comments are delimited by two consecutive hyphens, a type reference cannot contain two consecutive hyphens. ASN.1 does not limit the number of characters in an item, but the ASN1 compiler does have a limit that depends on the implementation (currently, this limit is about 200 characters). A type reference cannot be one of the reserved words (see 5.3, “Reserved Words” on page 5-2).

Type references must be uniquely defined within a module, that is it must be assigned to exactly one datatype. The datatype assignment may be in another module, in which case the type reference is listed in the IMPORTS clause of the module definition (see 5.10, “Module Definition” on page 5-8).

These are valid type references:

```
ThisIsAVeryLongTypeName

A-Short-One

A-1
```

These are *illegal* examples:

```
startsWithLowercase

EndsIn-

HasTwo--In-it
```

```
ANY_TABLE
ANY_TABLE_REF
```

Figure 5-2. Additional Reserved Words

5.6 Any Table Reference Item

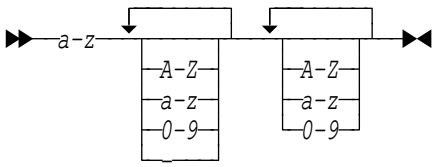
The ASN1 compiler provides an extension to standard ASN.1 syntax for a construct called *any table*. An any table is used to resolve the ANY (and its variant, ANY DEFINED BY) datatype. An any table is assigned a label called a *table reference*. The syntax for an any table reference is identical to that of the type reference (see 5.5, “Type Reference Item” on page 5-2).

Note: The any table is nonstandard ASN.1. To use it, you should use any table references only within syntactic guards (see 5.4, “Syntactic Guards” on page 5-2).

5.7 Value Reference Item

A value reference is a label assigned to a value of an ASN.1 datatype and is used to refer to the value. A value reference consists of one or letters, digits, or hyphens in the following form:

Value Reference



In the above syntax, note that the initial character is a lowercase letter, and the last character cannot be a hyphen. Also, because ASN.1 comments are delimited by two consecutive hyphens, a type reference cannot contain two consecutive hyphens. ASN.1 does not limit the number of characters in an item, but the ASN1 compiler does have a limit that depends on the implementation (currently, this limit is about 200 characters).

Value references must be uniquely defined within a module, that is it must be assigned to exactly one value definition. The value assignment may be in another module, in which case the value reference is listed in the IMPORTS clause of the module definition (see 5.10, “Module Definition” on page 5-8).

These are valid type references:

```
thisIsAVeryLongTypeName

a-Short-One

a-1
```

These are *illegal* examples:

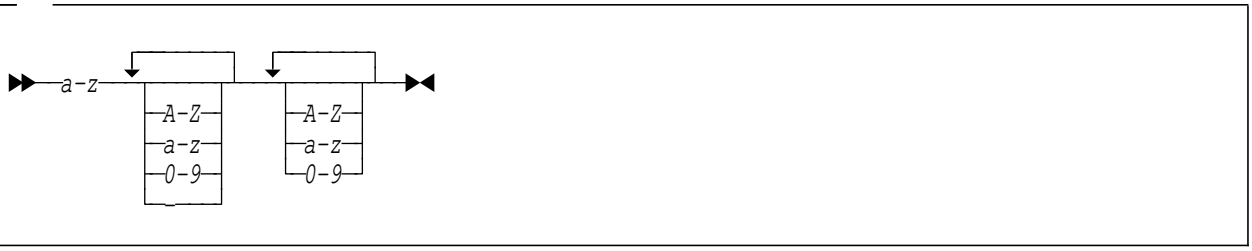
```
StartsWithUppercase

endsIn-

hasTwo--In-it
```

```
A ::= INTEGER  -- normal ASN.1 line and comment

--% Syntactic extensions go here %--

B ::= BOOLEAN  -- normal ASN.1 line and comment

--% Syntactic extensions go here.
--% End of line is also a syntactic guard end.

C ::= NULL     -- normal ASN.1 line and comment

-- The following will cause problems for other
-- compilers. ASN1 issues a warning message.

--% ... -- comment -- ... %--
```

Figure 5-3. Syntactic Guard Examples

5.8 Identifier Item

Identifiers are used to label parts of ASN.1 that are not type references or module references. An identifier consists of one or letters, digits, or hyphens in the following form:



In the above syntax, note that the initial character is a lowercase letter, and the last character cannot be a hyphen. Also, because ASN.1 comments are delimited by two consecutive hyphens, an identifier cannot contain two consecutive hyphens. ASN.1 does not limit the number of characters in an item, but the ASN1 compiler does have a limit that depends on the implementation (currently, this limit is about 200 characters).

Identifiers need only be unique within the context they are used. For example, identifiers are used to label different alternatives of a CHOICE datatype. Identifiers within the same CHOICE must be unique, but they may be reused in other CHOICE datatypes and in other datatypes which use identifiers.

Valid examples are:

```
anIdentifier

z1-2-3
```

Invalid examples are:

```
UppercaseStarter

hasBad)Char
```

5.9 Other Items

Other ASN.1 items are shown below.

5.9.1 Value Reference Item

An ASN.1 value may be assigned a label (see 5.12, “Value Assignment” on page 5-12). This label is called a value reference. The syntax for a value reference is just like that of an identifier (see 5.8, “Identifier Item” on page 5-5).

5.9.2 Module Reference Item

An ASN.1 module is assigned a label which is used to reference it. The module reference item has the same syntax as the type reference item (see 5.5, “Type Reference Item” on page 5-2).

5.9.3 Comment Item

ASN.1 allows commentary text to mixed in with the other syntax. There are no restrictions about where comments may appear, and there are no restrictions of which characters may appear within a comment.

A comment starts with two consecutive hyphens, "--", and is ended either by the end of the line or by another two consecutive hyphens.

Note: The ASN1 compiler admits extensions to the standard ASN.1 syntax. These extensions are bracketed by variations of the ASN.1 comment introducer, called *syntactic guards* To avoid accidently invoking these extensions, do not use the percent character immediately before or after the two hyphens unless you intend to use syntactic extensions. For more information on syntactic guards, see 5.4, “Syntactic Guards” on page 5-2.

5.9.4 Number Item

An ASN.1 number item consists of a sequence of decimal digits (0-9), not beginning with a "0" (except for zero). Note that where the ASN.1 syntax calls for *number* it means exactly this definition. You cannot use other items, like binary string and hexadecimal string, to represent an ASN.1 number. Note that an ASN.1 number is a non-negative integer. ASN.1 syntax which allows negative numbers will specifically include provisions for a sign character, "-". For specifying floating point values, you must use the REAL value notation (see 5.39, “Real Number” on page 5-31).

Valid examples are:

```
123

0
```

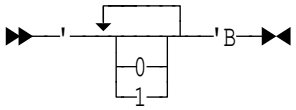
Invalid examples are:

```
0123    -- starts with a zero

-2      -- is a negative number

1.5     -- is a floating point number

'FF'H   -- is an hstring, not a number
```

5.9.5 Binary String Item

A binary string has the following syntax:

Binary String (bstring)



A binary string may consist of *zero* bits and it need not have an integral number of eight bits.

The binary string is padded on the right, as necessary, with zeros.

The single quote and the 'B' must be consecutive with no intervening spaces or line breaks.

A binary string may not be used where an ASN.1 *number* is required.

Valid examples are:

```
'01110'B

''B
```

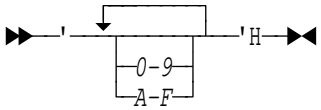
Invalid examples are:

```
'00 11'B  -- has an embedded space

'0011' B  -- space between ' and B
```

5.9.6 Hexadecimal String Item

A hexadecmial string has the following syntax:

Hexadecimal String (hstring)



A hexadecimal string may consist of *zero* semi-octets and it need not have an even number of semi-octets.

The hexadecimal string is padded on the right, as necessary, with zeros.

The single quote and the 'H' must be consecutive with no intervening spaces or line breaks.

A hexadecimal string may not be used where an ASN.1 *number* is required.

Valid examples are:

```
'0123456789ABCDEF'H

''H

'8'H
```

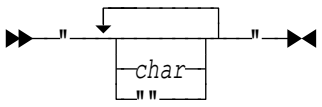
Invalid examples are:

```
'abcd'H  -- lowercase not allowed

'01 23'H -- embedded space not allowed

'AB' H   -- embedded space not allowed
```

5.9.7 Character String Item (cstring)

The syntax for a character string (cstring) item is:

Character String (cstring)



A character string item (cstring) may have *zero* or more characters. A double quote character in the string is represented by *two* consecutive double quote characters.

5.9.8 Assignment Item

The assignment item is used in the module assignment, type assignment and value assignment syntax. It consists of the three consecutive characters, "::=". Note that one or more of these characters may be substituted by a Bookmaster/Script symbol (see 2.1, “Source File” on page 2-1).

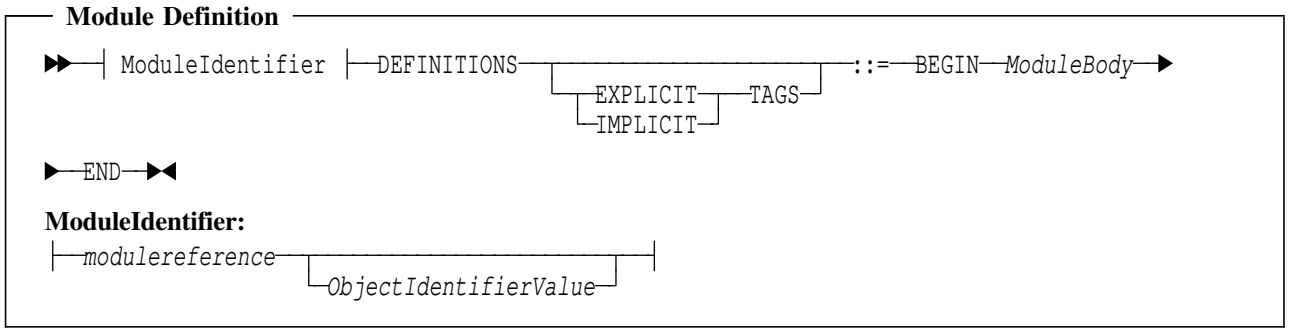
Valid examples are:

```
"This can be anything at all"

""  -- or nothing at all

"One embedded double quote: """
```

5.10 Module Definition

ASN.1 provides a way to group type and value definitions into packages called *modules*, much like programming languages provide procedures and functions to group together program statements.

The syntax for a module definition is:



5.10.1 Module Identification

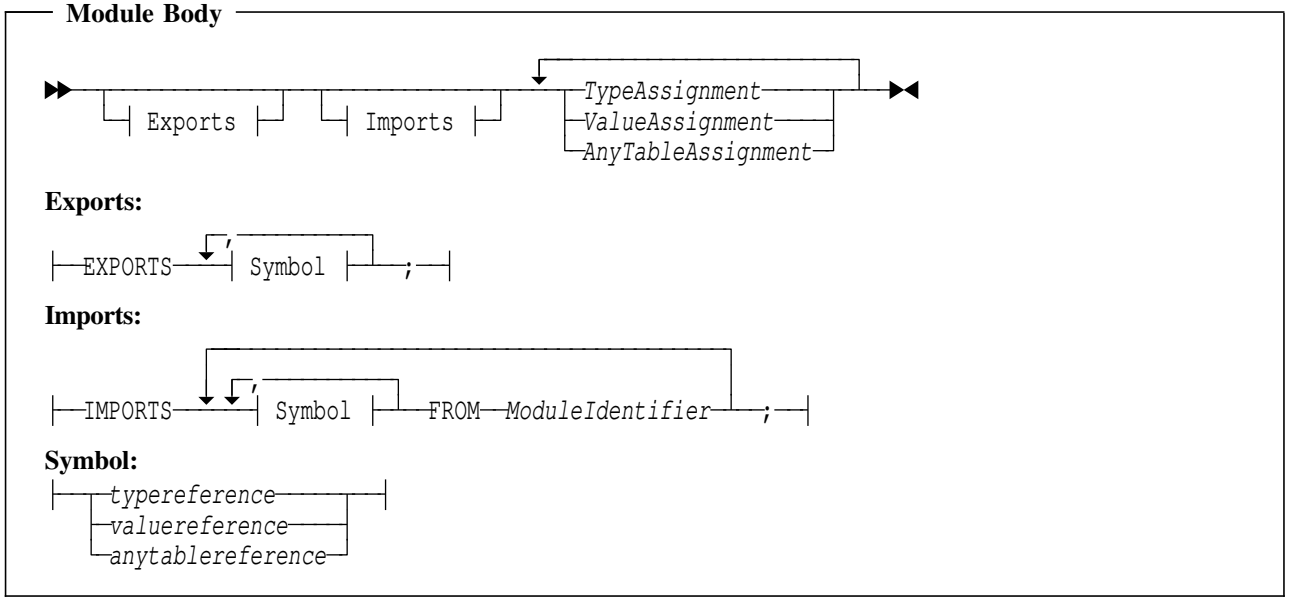
A module is always assigned a *module identifier* which consists of a mandatory *module name* (shown as *modulereference* in the syntax diagram) and an optional*assigned identifier*. The module name must begin with an uppercase letter and is like a type reference. The module name should be chosen so as to make it unambiguous. The optional assigned identifier is an object identifier value, which unambiguously identifies the module, i.e. the same object identifier value should never be assigned to more than one module definition. Because the assigned identifier is the only sure way to create an globally unique identifier for the module, it is recommended that each module be assigned an object identifier value.

5.10.2 Default Tagging

The module definition may specify how default tagging works. For a description of tagging, see 5.28, “Tagged Types” on page 5-23. Default tagging for types in the module can be IMPLICIT or EXPLICIT. If the default tagging is not specified, EXPLICIT is assumed. Default tagging affects only tags in this module; it does not affect tags imported or referenced from other modules.

5.10.3 Module Body Syntax

The module body contains assignment statements and clauses to specify which symbols are imported into the module and which symbols are exported from the module. The syntax for the body of a module is shown below:



Imported and Exported Symbols

Type and value references may be imported from other modules or exported to other modules. In addition to the standard ASN.1 syntax, the ASN1 compiler allows you to import any table references (see 5.6, “Any Table Reference Item” on page 5-4).

Symbols IMPORTED from a module need not be EXPORTED from that module, though that is the best practice. IMPORTED symbols are used with simple reference, i.e. the reference does not need to include the name of the module defining the imported symbol (modulename.symbolname).

A symbol defined in another module may be referenced without appearing in the IMPORT clause if explicit references (*modulename.symbolname*) are used. However, explicit references should be avoided; ASN.1 allows them for compatibility with an earlier version of the standard (CCITT Rec. X.409-1988).

Module Assignment List

The bulk of the module body consists of assignments for type, value and any table definitions. Each type, value and any table definition is given an label that is unique within the module. The scope of uniqueness includes symbols which are imported into the module. These labels are called *type references* (see 5.5, “Type Reference Item” on page 5-2) and *value references* (see 5.9, “Other Items” on page 5-6) and *any table references* (see 5.6, “Any Table Reference Item” on page 5-4).

5.10.4 Module Example

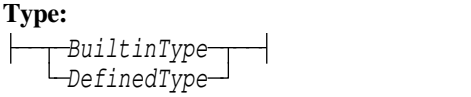
```
CMIP-Abbreviated

DEFINITIONS ::= BEGIN

   IMPORTS InvokeIdType, Operation, Error
   FROM REMOTE-OPERATIONS ;

NullType ::= NULL

ROIVapdu ::= [1] IMPLICIT SEQUENCE
{invokeID     InvokeIdType,
 linked-ID     [0] IMPLICIT InvokeIdType OPTIONAL,
 operation-value     Operation,
 argument     ANY DEFINED BY operation-value}

ROERapdu ::= [3] IMPLICIT SEQUENCE
{invokeID     InvokeIdType,
 error-value     Error,
 parameter     ANY DEFINED BY error-value OPTIONAL}

RORJapdu ::= [4] IMPLICIT SEQUENCE
{invokeID     CHOICE{InvokeIdType,NullType},
 problem     CHOICE
               {[0] IMPLICIT GeneralProblem,
                [1] IMPLICIT InvokeProblem,
                [2] IMPLICIT ReturnResultProblem,
                [3] IMPLICIT ReturnErrorProblem} }

InvokeIdType ::= INTEGER
Operation    ::= INTEGER
Error        ::= INTEGER
```

```
-- The following problems are detected by ROSE-providers:

GeneralProblem ::= INTEGER{
                     unrecognizedAPDU(0),
                     mistypedAPDU(1),
                     badlyStructuredAPDU(2) }

m-EventReport Operation ::= 0
m-EventReportConfirmed Operation ::= 1

ROIVeventReport ::= [1] IMPLICIT SEQUENCE
{invokeID InvokeIdType,
 operation-value Operation (m-EventReport|
                  m-EventReportConfirmed),
 argument SEQUENCE
   {managedObjectClass ObjectClass,
    managedObjectInstance ObjectInstance,
    eventTime [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType EventTypeId,
    eventData ANY DEFINED BY eventType OPTIONAL} }

RORSeventReportConfirmed ::= [2] IMPLICIT SEQUENCE
{invokeID InvokeIdType,
 resultOption SEQUENCE
   {operation-value Operation (m-EventReportConfirmed),
    result SEQUENCE
      {managedObjectClass ObjectClass OPTIONAL,
       managedObjectInstance ObjectInstance OPTIONAL,
       currentTime [5] IMPLICIT GeneralizedTime OPTIONAL,
       eventReply SEQUENCE
        {eventType EventTypeId,
         eventReplyInfo [8] ANY DEFINED BY eventType OPTIONAL
        } OPTIONAL
      }
   } OPTIONAL
}


END -- CMIP definitions
```

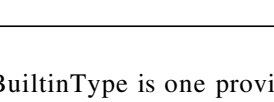
5.11 Type Assignment

The purpose of a type assignment is to assign a type reference to a type definition. The type reference may be used in other places to refer to the type in the definition. The syntax of a type assignment is:

Type Assignment



Type:



A BuiltinType is one provided in the ASN.1 standard (see 5.15, “Built-In Types” on page 5-14). A DefinedType is created by a type definition.

The following are examples of type definitions:

```
EmployeeId ::= SocialSecurityNumber

SocialSecurityNumber ::= INTEGER
```

In the above example, INTEGER is a BuiltInType. The DefinedType definitions are EmployeeId and SocialSecurityNumber. In general, type references may be logically replaced by their assigned type definitions; the above example is logically equivalent to:

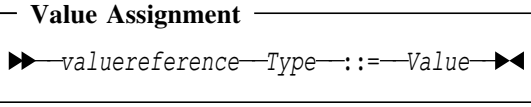
```
EmployeeId ::= INTEGER
```

5.12 Value Assignment

A value assignment assigns a label (the reference) to a value of a particular data type.

The syntax for a value assignment is:

Value Assignment



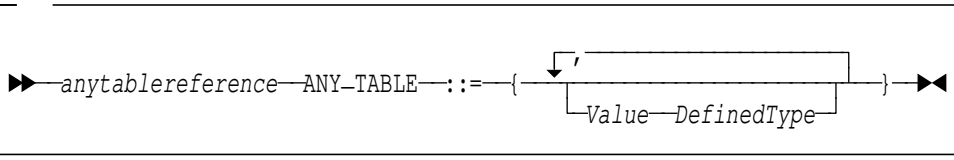
The syntax of *Value* depends on *Type* Example value assignments are:

```
firstEmployee EmployeeId ::= 1

unity INTEGER ::= 1

messageBits1 BIT STRING ::= '00101010111'B
```

5.13 Any Table Assignment

An any table is an extension to the standard ASN.1 syntax. Its purpose is to provide a formal notation for associating values to datatypes which in turn determine the datatype for resolving an ANY DEFINED BY datatype. An any table is a variable number of table entries with each entry consisting of a value and a defined type. The value part of each table entry must be either an INTEGER or an OBJECT IDENTIFIER.

The syntax of the any table is as follows:



Note that the *anytablereference* is a symbol that may be imported and exported between modules.

Note also that since this is a nonstandard extension to the ASN.1 standard, you should protect each part of the definition with syntactic guards (see 5.4, “Syntactic Guards” on page 5-2).

The following example shows a correct any table assignment:

```
--%  Tbl ANY_TABLE ::= {
--%      0    A,
--%      1    B,
--%      x    C }

     A ::= INTEGER
     B ::= OCTET STRING
     C ::= BOOLEAN
     x INTEGER ::= 2
```

See 5.14, “Referencing Types, Values and Any Table Definitions” for the syntax of DefinedType.

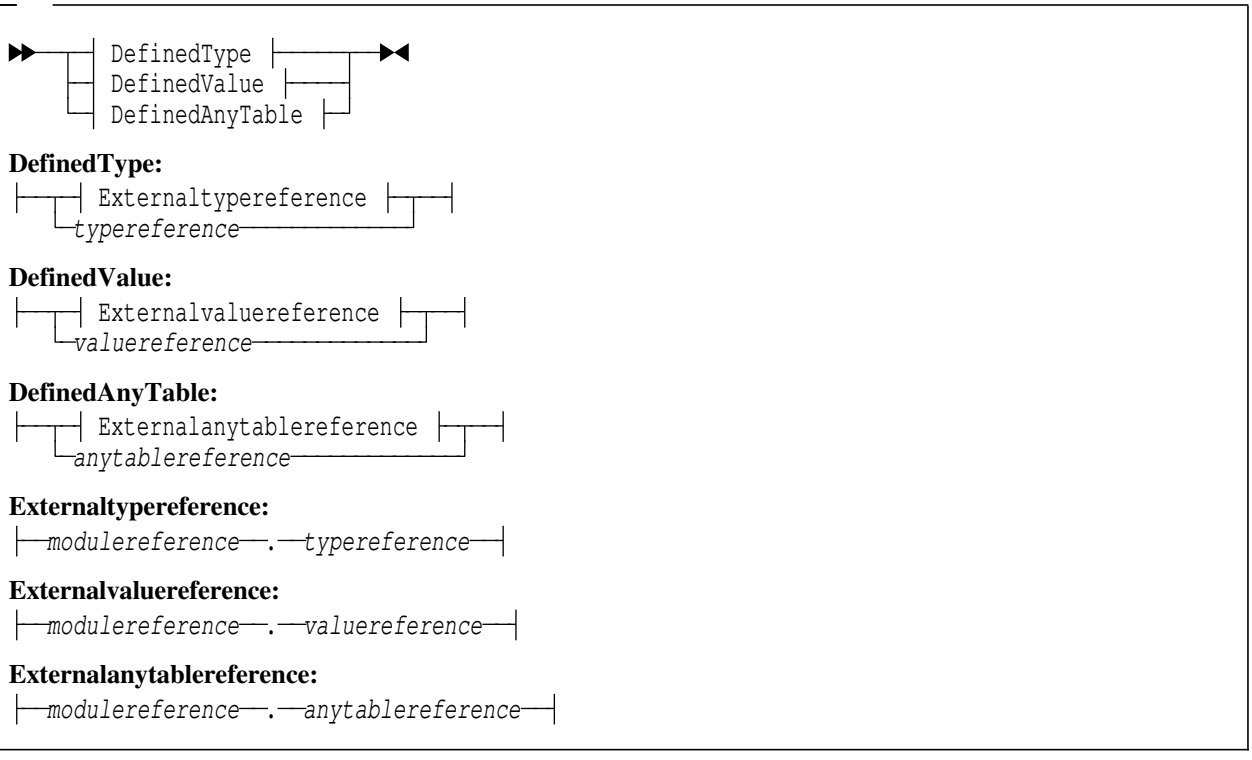
5.14 Referencing Types, Values and Any Table Definitions

A type, value or any table definition is referenced by its assigned reference name if the definition and reference occur in the same module or if the definition is imported into the module.

If the reference and definition are in different modules and the definition is not imported, then reference is must take the form of an external reference. An external reference is of the form: *modulename.symbolname*, where *modulename* refers to the module containing the definition.

The ASN.1 standard calls these kinds of references *DefinedType*, *DefinedValue*. In addition, the ASN1 compiler provides a reference for any tables called *DefinedAnyTable*.

The syntax for these defined references is:



5.15 Built-In Types

The ASN.1 standard provides a number of datatypes which are "built-in". The built-in types are the building blocks for all ASN.1 data types. Specifically, they are used to create defined datatypes (see 5.14, “Referencing Types, Values and Any Table Definitions”).

The following is a list of the built-in datatypes.

- BOOLEAN
- INTEGER
- BIT STRING
- OCTET STRING
- NULL
- SEQUENCE
- SEQUENCE OF
- SET
- SET OF
- CHOICE
- Selection Type
- Tagged
- ANY
- OBJECT IDENTIFIER
- Character string types
- Useful types
- ENUMERATED
- REAL

The subsequent sections provide details on each of these.

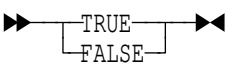
5.16 BOOLEAN

The boolean datatype is used to express values which connote truth or falseness. The syntax for a boolean datatype is:

Boolean Type

The two values for a boolean datatype are true and false:

Boolean Value



The tag for a boolean value is UNIVERSAL, number 1.

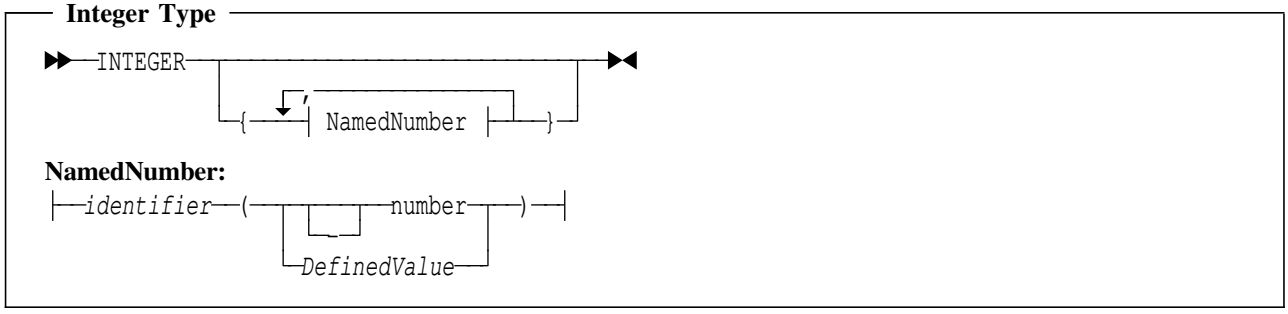
Example boolean values and datatypes are:

```
IsBreathing ::= BOOLEAN

probablyDead IsBreathing ::= FALSE

couldBeAlive IsBreathing ::= TRUE
```

5.17 INTEGER

The integer datatype is used to model integer or cardinal values. Integer values may be positive, negative, or zero. Integers have arithmetic semantics, that is they can be added, subtracted, etc. If the values you wish to model do not have arithmetic properties, you should consider using another datatype, such as boolean or enumerated.

The syntax for an integer datatype is:



Named numbers allow you to give names to specific values of the datatype. However, named numbers do **not** restrict the range of permissible values. That is, the allowable values for an integer with named numbers can be any negative or positive number, or zero. Use subtyping to restrict range of permissible values.

Named numbers must have unique identifiers and values within the context of the type definition. This means two different datatypes may use some of the same identifiers for named number, named bits, sequence and set components, and choice alternatives. The following examples illustrate this point.

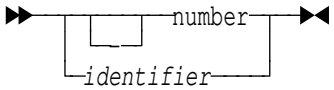
```
A ::= INTEGER{a(0), b(42)} -- legal

B ::= INTEGER{a(42), b(0)} -- legal
```

Only one identifier in a single named number list may be used to define the same integer value. The following example illustrates this point.

```
C ::= INTEGER{a(0), b(0)} -- illegal
```

The syntax for an integer value is:

Integer Value



The *identifier* in the value syntax refers to either a named number or to a defined integer value.

The tag for an integer value is UNIVERSAL, number 2.

Examples:

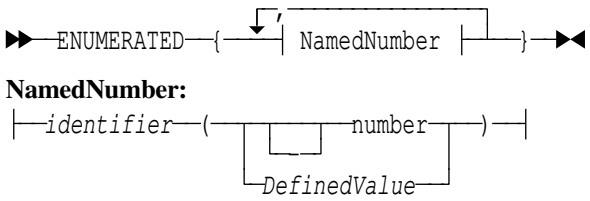
```
MessageType ::= INTEGER{ basic(0),
                         extended(1),
                         unsupported(foo)}

foo INTEGER ::= -2

ErrorCounter ::= INTEGER
```

5.18 ENUMERATED

The enumerated datatype is used to model distinct values by associating an identifier with an integer value. However, the integer values are not expected to have arithmetic semantics, e.g. you should not expect them to be added or subtracted.

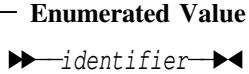
The type notation is:

Enumerated Type



The syntax for an enumerated value is:

Enumerated Value



The *identifier* in the value notation must refer to one of the named numbers in the type notation. Note that only identifiers are used in the value notation; you cannot use numbers (such as 42) for specifying enumerated values.

The tag for an enumerated value is UNIVERSAL, number 10.

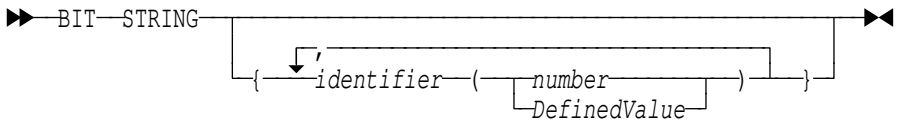
Examples:

```
Status ::= ENUMERATED{notThere(0),
           asleep(1), working(2)}

parm1 Status ::= asleep
```

5.19 BIT STRING

The bit string data type is used to model data which may contain an arbitray pattern of bits, not necessary octet-aligned. Individual bits in a bit string may be named. The notation for a bit string type is:

Bit String Type

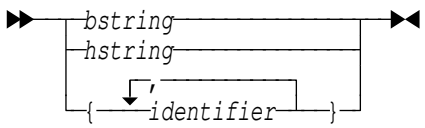


Bits are numbered from 0 for the first bit. The final bit in a bit string is called the *final* bit. This convention is used for the value notation and for the encoding rules.

Not every bit in a bit string needs to be named. The identifiers must be unique within a bit string type notation, but need not be unique outside a type notation, i.e., a name can only refer to one bit within a bit string, but the name can be re-used outside the bit string for other purposes. The numbers and defined values for the named bits within a type notation must be unique, i.e., you cannot give the same bit more than one name.

The syntax for the bit string value notation is:

Bit String Value



Each identifier in the bit string value corresponds to one of the named bits in the type notation. Do not use the {identifier1, identifier2, ... } notation if trailing zero bits are significant to the application. Using commentary, you should indicate whether trailing zero bits are significant.

The leftmost bit in the *bstring* notation corresponds to bit number zero.

Do not use the *hstring* notation if trailing zero bits are significant or if the value does not consist of a multiple of four bits. The most significant bit of a hexadecimal digit corresponds to the leftmost bit in the value notation.

The tag for a bit string is: UNIVERSAL, number 3

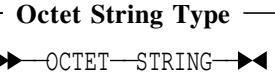
Examples:

```
MessageFlags ::= BIT STRING {posResp(0), negResp(1),
                 doNotForward(2)}
DigitizedVoiceMessage ::= BIT STRING
setting1 MessageFlags ::= '001'B
```

5.20 OCTET STRING

The octet string type is Used to model binary data whose length and format are unspecified, and whose length is a multiple of 8 bits. The syntax for an octet string type is:

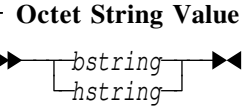
Octet String Type



Note that the keywords "OCTET" and "STRING" are separate; they must be separated by one or more blanks, lines, comments, etc.

The notatation for an octet string value is:

Octet String Value



Although an octet string value must consist of an integral multiple of eight bits, the binary and hexadecimal strings need not be. The binary or hexadecimal string is padded on the right with zeros until an integral multiple of eight bits is achieved.

The tag for an octet string is UNIVERSAL, number 4. Examples

```
ObjectCode ::= OCTET STRING
program1 ObjectCode ::= '08634737AF234EC'H

a OCTET STRING ::= '90A'H -- padded with 4 zero bits
b OCTET STRING ::= '01'B -- padded with 6 zero bits
```

5.21 NULL

The null data type is Used to show effective absence of sequence element, or as a placeholder for future use. The notation for a null type is:

Null Type

The value notation for a null type is:

Null Value

The null tag is UNIVERSAL, number 5.

Examples are:

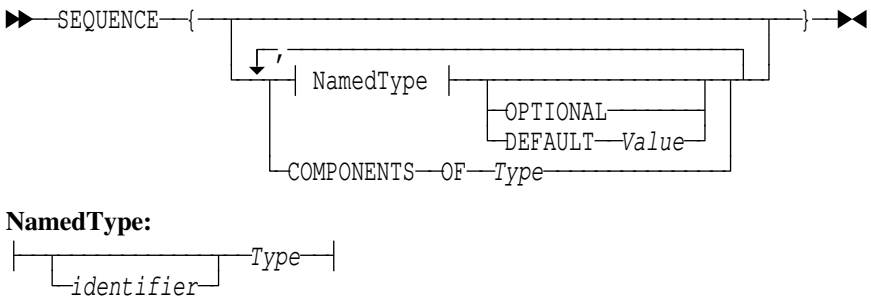
```
RoomNumber ::= CHOICE{ rnum INTEGER,
                       none NULL}

nullValue NULL ::= NULL

noRoom RoomNumber ::= none : NULL -- also a choice value
```

5.22 SEQUENCE

The sequence datatype is used to group, or structure, other datatypes. The components of this kind of sequence may be of different datatypes, and, with certain restrictions, may be optionally present in encoded values.

The type notation is as follows:

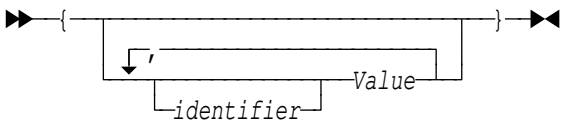
Sequence Type



NamedType:

The value notation for this datatype is:

Sequence Value



The tag for a sequence is universal, number 16.

A sequence is used to model an ordered number of components, each of which may be of a different types, and (with restrictions) may be omitted. This kind of sequence is sometimes referred to as an *enumerated*, *heterogeneous* sequence. It is enumerated because each of the possible elements is listed in the type notation. It is heterogeneous because not all of the components are necessarily of the same datatype. In most programming languages, the closest analog would be a structure (e.g., in C and C++), or a record (e.g., in Pascal).

In the above type notation, the component types appear between the curly braces. Each component consists of a *named type* and an optional *OPTIONAL* or an optional *DEFAULT* clause.

The named type is fairly self descriptive; it consists of an optional identifier and the type notation for the component. If present, the identifier in the named type may be referenced from other places in the abstract syntax. For example, the value notation may use the identifier to unambiguously refer the component for which a value is defined. The scope of uniqueness for a named type identifier is only within the sequence definition where it occurs. This means that an identifier, e.g. "foo", must be unique within the sequence definition, but may be used elsewhere in the abstract syntax, e.g. as a value reference or an identifier for a named type in some other sequence definition.

Although the current ASN.1 standard does not require each component to have an identifier in its named type, it is always good practice to supply one. Indeed, the new ASN.1 standard will make these identifiers mandatory. It's not just a good idea, it will be the law.

The *OPTIONAL* keyword is used to signify that the corresponding component in a sequence value may be absent. This is a very useful feature for reducing the amount of information that is encoded in sequence values, hence reducing the number of octets stored in memory, stored in DASD, or transmitted over a communications link. However, the components of a sequence which may be declared as optional are restricted to avoid ambiguity when decoding a sequence value. The rule is simple: the tag of an optional (or defaulted) component must be distinct from the tags of the following components up to, and including, the next mandatory (i.e, non-optional and non-defaulted) component. The reason for this rule should be apparent from the following example. Suppose we define the following type:

```
A ::= SEQUENCE{s INTEGER,
               t BOOLEAN OPTIONAL,
               u OCTET STRING OPTIONAL,
               v BOOLEAN OPTIONAL}
```

Since three of the four components are optional, a value of type A may have from one to four encoded elements. Suppose the following value is encoded:

```
a A ::= {s 42, t TRUE}
```

From the definition of *a*, we can see that components *u* and *v* have been omitted (which is okay since they are optional). The *abstract* definition of *a* is unambiguous. However, the *concrete*, or *encoded* representation of *a* is ambiguous. The source of the problem is that both *t* and *v* have the same tags, i.e. universal number 1. Unlike the absract syntax for *a*, the concrete syntax does not encode the identifiers *s* and *t*. Only the tags are encoded to resolve ambiguities. From a receiver's or decoder's point of view, the second element of *a* may be either for the *t* or the *v* component of *A*.

A *DEFAULT* clause is a special case for an optional component. Like an optional component, a default component my be omitted from a value specification for the sequence. Unlike an optional component, when a defaulted component is omitted, the receiver or decoder must interpret the sequence as though the defaulted component were actually present. Default components offer additional opportunities for conserving storage or bandwidth. Note that the restrictions for components of a sequence which may be defaulted are the same restrictions as those for optional components (see above).

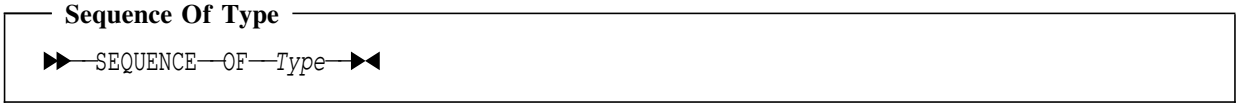
The *COMPONENTS OF* construction allows components to be included by reference to another sequence type. It is a shorthand technique to achieve the same result as if a text editor is used to copy and paste all the components from the referenced type into the place where the *COMPONENTS OF* clause appears. All of the same restrictions for uniqueness of named type identifiers and unique tags apply.

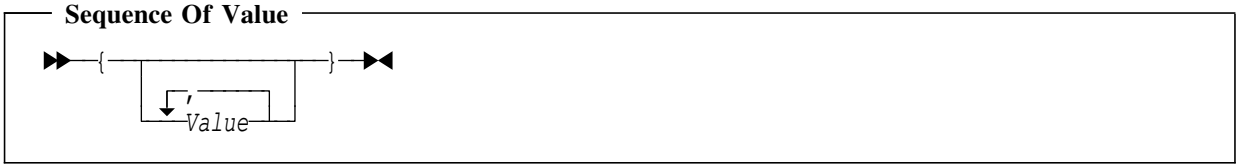
The following are examples:

```
Record ::= SEQUENCE{empNum INTEGER,
                    phone PrintableString OPTIONAL,
                    retired BOOLEAN DEFAULT FALSE}

PayRec ::= SEQUENCE{COMPONENTS OF Record,
                    bits BIT STRING}
employee1 Record ::= {1009, "543-0481", TRUE}
```

5.23 SEQUENCE OF

- Notation:





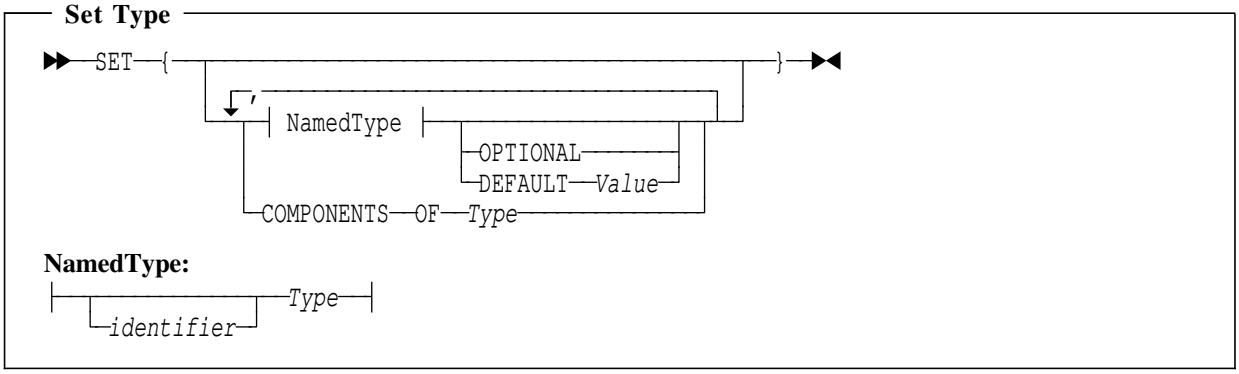
- Tag: UNIVERSAL, number 16 (same as SEQUENCE{ ... })
- Used for homongenous structures
- Semantics may be placed on order
- SEQUENCE (alone) as a shorthand for SEQUENCE OF ANY has been eliminated by a defect report
- Examples:

  File ::= SEQUENCE OF Record

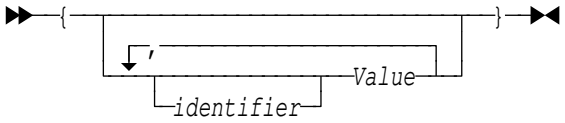
  MeterReadings ::= SEQUENCE OF INTEGER

5.24 SET

- Notation:



Set Value



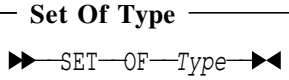
- Tag: UNIVERSAL, number 17
- All element tags must be unique
- No semantics associated with order of elements
- Example:

```
Message ::= SET{header OCTET STRING,
            body BIT STRING}
```

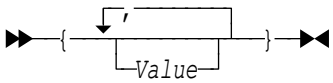
5.25 SET OF

- Notation:

Set Of Type



Set Of Value



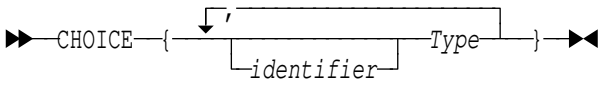
- Tag: UNIVERSAL, number 17 (same as SET{ ... })
- Order of elements unimportant
- Encoding rules don't require order to be preserved
- Example:

```
Keywords ::= SET OF VisibleString

keywordSet1 Keywords ::= {"INTEGER", "SET", "OF"}
```

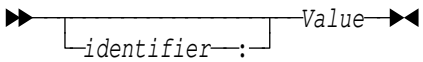
5.26 CHOICE

- Choice is used to allow a type to have a range of alternatives; a value of a CHOICE type must select one of the alternatives
- Notation:

Choice Type



Choice Value



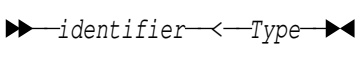
- Tag: same as selected choice
- CHOICE alternatives must have distinct tags
- Identifiers (which are optional) must be distinct
- Example:

```
A ::= CHOICE{w INTEGER, x BOOLEAN, y C}
C ::= CHOICE{z BIT STRING, s OCTET STRING}
b A ::= w : 10  -- b in an INTEGER type
```

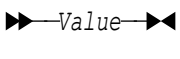
5.27 Selection Type

- Selection used to reference a CHOICE type alternative
- Notation:

Selection Type



Selection Value



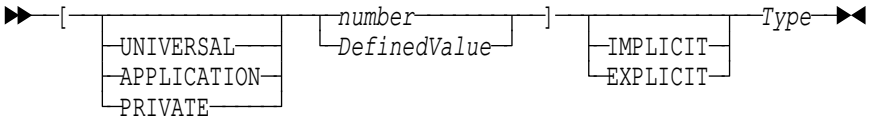
- Tag: same as named type
- Examples:

```
A ::= CHOICE{x INTEGER, y BOOLEAN}

B ::= x <  A   -- B is an INTEGER type

c B ::= 15    -- c is an INTEGER value
```

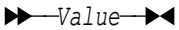
5.28 Tagged Types

- A tagged type is a new type which acts like the old type but which has a different tag. This is often needed to make distinct tags for the same base type.
- Notation:

Tagged Type



Tagged Value



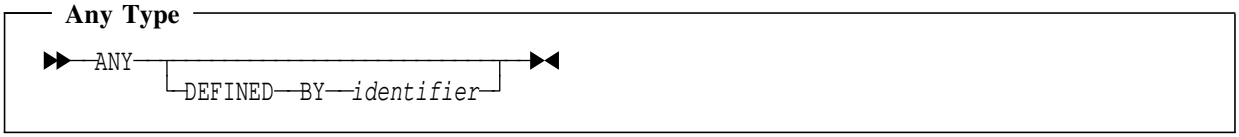
- Tag: defined by notation
- Explicit tagging retains old tag and semantics
- Implicit tagging replaces old tag and semantics
- Implicit tagging cannot be used with CHOICE or ANY
- UNIVERSAL reserved for ISO 8824/CCITT X.208
- APPLICATION reserved for other ISO/CCITT standards
- When class is not used, context-specific

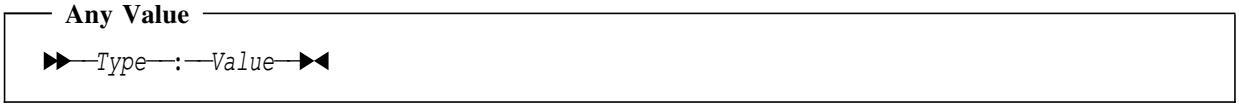
- Examples:

```
A ::= CHOICE{a [0] IMPLICIT INTEGER,
             b [1] IMPLICIT INTEGER,
             c [2] IMPLICIT INTEGER,
             d [3] IMPLICIT INTEGER,
             e [4] IMPLICIT INTEGER}

B ::= [APPLICATION 12] OCTET STRING

INTEGER ::= [UNIVERSAL 2] IMPLICIT OCTET STRING
```

5.29 ANY

- ANY acts as placeholder for further specification
- Notation:





- Tag: indeterminate
- ANY DEFINED BY
  - is used as a pointer to semantics for the unspecified type
  - "identifier" must name a non-OPTIONAL component of the containing SET or SEQUENCE
  - the referenced type must be a subtype of INTEGER or OBJECT IDENTIFIER
- Cannot be used where distinct tags are required

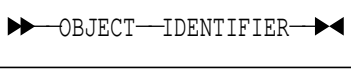
- Examples:

```
Message ::= SEQUENCE
        {msgType INTEGER,
         contents ANY DEFINED BY msgType}

msg1 Message ::= {3, OCTET STRING '01AF'H}
```

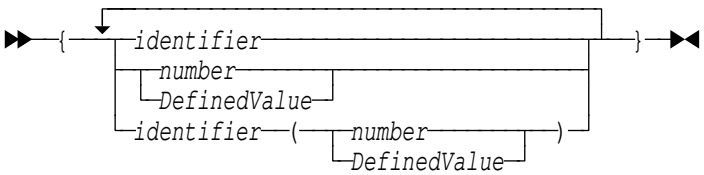
5.30 OBJECT IDENTIFIER

- Used to identify ***globally unique*** information objects
- Notation:

Object Identifier Type



Object Identifier Value



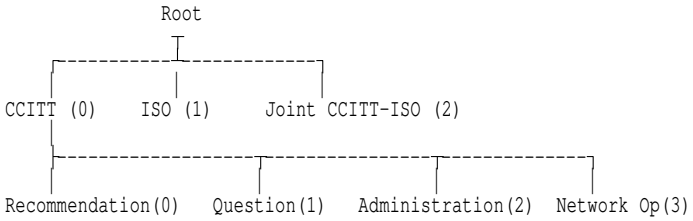
- Tag: UNIVERSAL, number 6

- Examples:

```
fTAM OBJECT IDENTIFER ::= {iso standard 8571 pci(1)}

-- the following is equivalent to
-- the above
fTAM OBJECT IDENTIFIER ::= {1 0 8571 1}
```

- Name Form reserved for annexes B to D
- Uniquely identifies information objects
- Object Id Tree: root ISO 8824, verticies are allocation authorities, each arc labelled with a number

- OBJECT IDENTIFIER value corresponds to path of labels from the root to a leaf



- ISO components:
  - 0 - standard (uses ISO std number)
  - 1 - registration authority (values TBD)
  - 2 - member body (3 digits, as defined by ISO 3166)
  - 3 - identified organization (uses ICD from ISO 6523)
- Joint components:
  - as agreed by CCITT and ISO

5.31 Character String Types

- Character string types are identified by assigning
  - a tag for the type
  - a name for the type
  - a specification or reference for a character set
- Values are specified using ***cstring***

5.32 NumericString

- Allowed characters: digits 0 to 9, space
- Tag: UNIVERSAL, number 18
- Examples:

```
zero NumericString ::= "0 "
phoneHelp NumericString ::= "  911"
PhoneNumber ::= NumericString -- spaces not
                              -- allowed
```

5.33 PrintableString

- Allowed characters: A to Z, a to z, 0 to 9, space, ', (,), + , comma, -, ., /, :, = , ?
- Tag: UNIVERSAL, number 19
- Examples:

```
warning PrintableString ::= "Do Not Enter"
Message ::= PrintableString
```

5.34 Other Character String Types

- TeletexString and T61String:
  - characters from CCITT T.61
  - Tag: UNIVERSAL, number 20
- VideotexString:
  - characters from CCITT T.100 and T.101
  - Tag: UNIVERSAL, number 21
- VisibleString and ISO646String:
  - characters from ISO 2375, reg 2 + SPACE
  - Tag: UNIVERSAL, number 26
- IA5String:
  - characters from ISO 2375, reg 1, 2 + DELETE
  - Tag: UNIVERSAL, number 22
- GraphicString:
  - characters from ISO 2375, all G sets + SPACE
  - Tag: UNIVERSAL, number 25

- GeneralString:
  - characters from ISO 2375, all G and C sets + SPACE + DELETE
  - Tag: UNIVERSAL, number 27

5.35 Generalized Time

- Definition:

```
GeneralizedTime ::=
    [UNIVERSAL 24] IMPLICIT VisibleString
```

- Form:
  - YYYYMMDDtt...t - local time (tt...t as per ISO 3307)
  - YYYYMMDDtt...tZ - UTC time
  - YYYYMMDDtt...t-xx...x - local differential (xx...x per ISO 4031)

5.36 Universal Time

- Definition:

```
UTCTime ::=
  [UNIVERSAL 23] IMPLICIT VisibleString
```

- Form:
  - YYMMDD
  - YYMMDDhhmm
  - YYMMDDhhmm
  - YYMMDDhhmmss
  - YYMMDDhhmmssZ
  - YYMMDDhhmmss+hhmm
  - YYMMDDhhmmss-hhmm

5.37 External

- EXTERNAL is used to allow any data value from a defined set of data values (abstract syntax)
- Definition:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE{
  direct-reference OBJECT IDENTIFIER OPTIONAL,
  indirect-reference INTEGER OPTIONAL,
  data-value-descriptor ObjectDescriptor OPTIONAL,
  encoding CHOICE {
    single-ASN1-type [0] ANY,
    octet-aligned [1] IMPLICIT OCTET STRING,
    arbitrary [2] IMPLICIT BIT STRING}}
```

- Direct reference OBJECT IDENTIFIER value, by registration, indicates abstract syntax and encoding

- Indirect reference used by OSI presentation layer for negotiating abstract syntaxes; the integer value refers to the presentation context identifier
- One or both references are mandatory

5.38 Object Descriptor

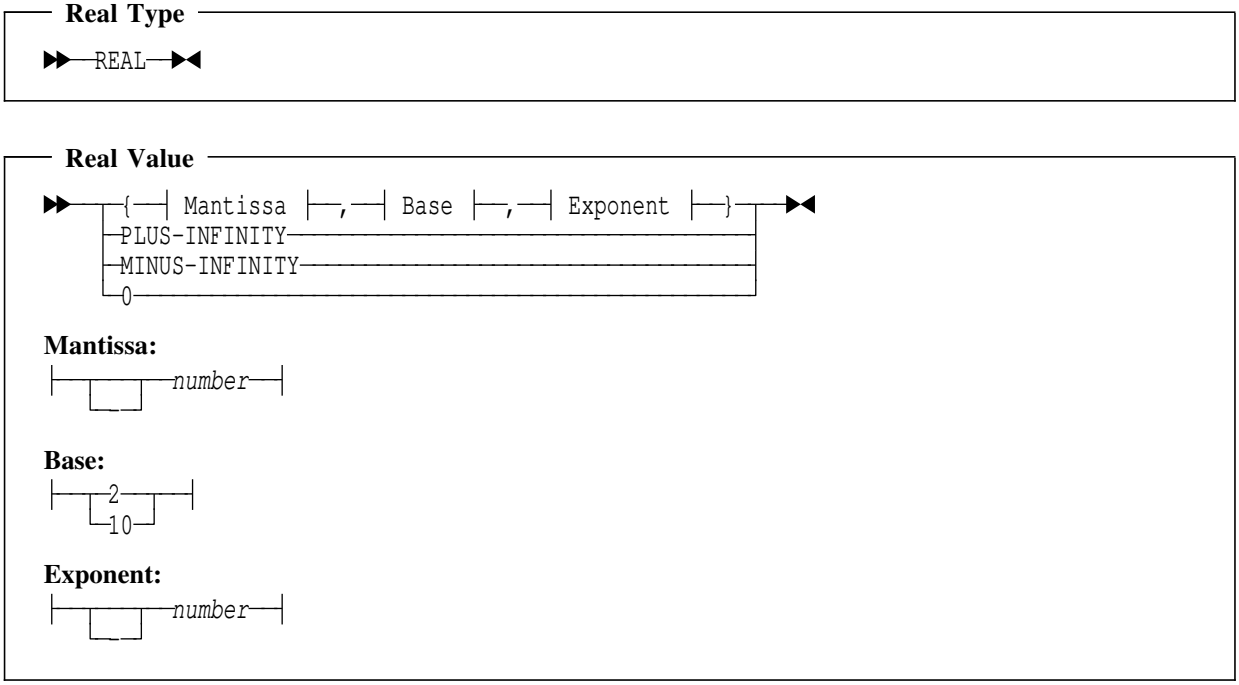
- Human-readable text to describe an information object
- Definition:

  ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString

- May be ambiguous
- Recommended that ObjectDescriptor values be assigned concurrently with OBJECT IDENTIFIER values

5.39 Real Number

- Form:



- Values defined by: Mantissa * Base ** Exponent
- Examples:

  pi REAL ::= {3141592653589, 10, -12}
  logOfZero REAL ::= MINUS-INFINITY

5.40 Subtypes

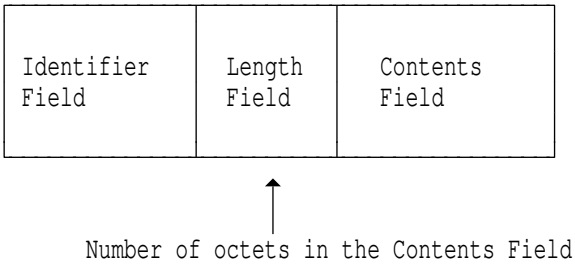
- Subtyping allows additional constraints to be placed on values allowed by the parent type
- Value Ranges for INTEGER and REAL types: lower..Upper
- Size constraints for BIT STRING, OCTET STRING, and character string types
- Presence/Absence constraints for OPTIONAL elements or CHOICE alternatives
- Examples:

```
AtomicNumber ::= INTEGER (1..104)
TouchToneString ::= IA5String (FROM ("0"|"1"|"2"|"3"|"4"|"5"|"6"|
                "7"|"8"|"9"|"*"|"#"))
ParameterList ::= SET SIZE (0..63) OF Parameter
SmallPrime ::= INTEGER(2|3|5|7|11|13|17|19|23|29)
NameField ::= PrintableString (SIZE(1..24))
FixedLenField ::= OCTET STRING (SIZE(10))
VeryStrange ::= OCTET STRING (SIZE(0 | 3 | (5..13)))
```

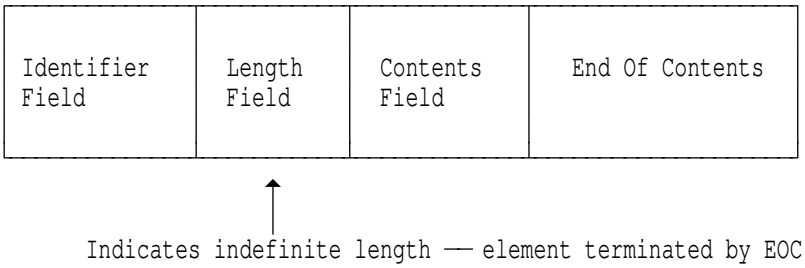
Chapter 6. Basic Encoding Rules

6.1 Basic Structure

- Definite Length Form

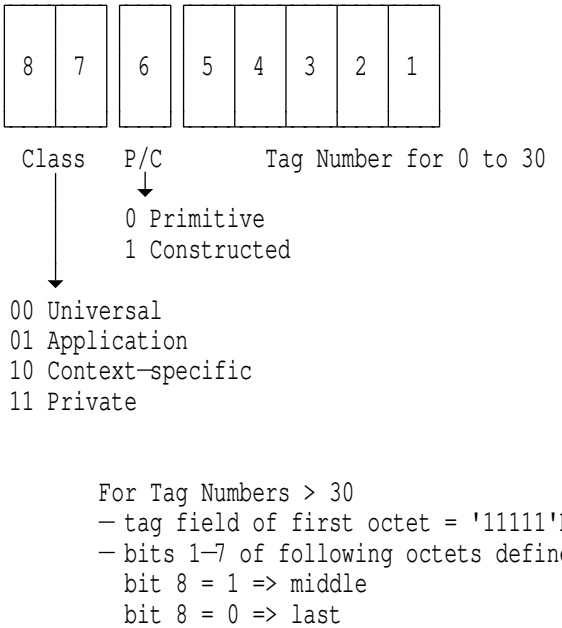


- Indefinite Length Form



- Each of the fields is variable length (except EOC)

6.2 Identifier Field



For Tag Numbers > 30
— tag field of first octet = '11111'B
— bits 1—7 of following octets define tag number
  bit 8 = 1 => middle
  bit 8 = 0 => last

6.3 Length Field

- Definite form
  - always used for primitive types
  - short form for 0 to 127 values - bit 8=0
  - long form for greater than 127 values
    - bit 8 = 1 (indicates long form)
    - bits 1-7 of first octet indicate number of subsequent length octets; x'FF' is reserved
    - bits 1-8 of subsequent octets form length of contents
    - Sender may choose long form for contents < 128 octets
    - Sender need ***not*** use fewest octets for encoding long form

- Indefinite form may be used for constructed types
  - single length field octet: x'80'
  - End of Contents value: x'0000'

6.4 Contents Field

- zero or more octets (as specified by length)
- encoding depends on data type

6.5 BOOLEAN Value

- primitive
- Length shall always be 1 octet
- TRUE - any non-zero value 1-255
- FALSE - zero

6.6 INTEGER And ENUMERATED Value

- primitive
- Smallest possible encoding required
- Two's complement
- Note: this requires, eg, 255 to be encoded as x'00FF'

6.7 BIT STRING Value

- Primitive or constructed, as determined by sender
- First octet indicates number of unused bits in the last octet
- Empty bit string = > length=1, contents ='00'H
- Always multiple of eight bits
- Constructed form:
  - used when entire bit string is not available when transfer of part of it is necessary
  - may use indefinite length form
  - embedded types are BIT STRING, may be primitive or constructed
  - No significance of component boundaries
  - only last embedded BIT STRING may have unused bits

6.8 OCTET STRING Value

- Primitive or constructed, as determined by sender
- Constructed form:
  - used when entire octet string is not available and transfer of part of it is necessary
  - may use indefinite length form
  - embedded types are OCTET STRING, may be primitive or constructed, and may have zero length
  - No significance of component boundaries

6.9 Null Value

- Primitive, zero-length

6.10 Sets and Sequence Values

- Constructed
- Contents contains concatenated component values
- Sender may choose any order of component values in the case of SETs
- Absence of a DEFAULTed component is interpreted as though the default value was actually present

6.11 Tagged Values

- IMPLICIT values replace the tagged type id
- EXPLICIT values are constructed, the contents contain the complete encoding (id, len, contents) of the tagged value.
- Examples:

```
foo1 [3] INTEGER ::= 18
              -- A3 03 02 01 12
              --             val=18
              --          len=1
              --       INTEGER
              --    len=3
              -- [3]
foo2 [3] IMPLICIT INTEGER ::= 18
              -- 83 01 12
              --       val=18
              --    len=1
              -- [3]
```

6.12 OBJECT IDENTIFIER Value

- Primitive encoding, contents are a ordered list of subidentifiers
- Each subidentifier encoded as 1 or more octets, HOB=0 for last octet
- First subidentifier = (X*40+Y) where X and Y are first two object identifier components
- Example:

```
id1 OBJECT IDENTIFIER ::= {2 100 3}
  -- 06 03 81 34 03
                   3
            40*2+100=180
         len=3
      OBJECT IDENTIFIER
```

Programmer's Guide

Chapter 7. Writing User Encoding Routines

Chapter 8. Runtime Library

The runtime library routines are used by applications to encode and decode ASN.1 data, to convert between transfer syntax and local forms, and to format data for printing or displaying, etc.

8.1 Global Variables

The runtime environment uses a few global variables to manage its affairs. These are described below.

8.1.1 modules

Source File

berglob.c

Syntax

```
struct moddef  *modules;
```

Purpose

The modules variable is the head of a list of the active ASN.1 modules in the runtime environment. For functions which modify this variable, see 8.3.13, “initmodules” on page 8-11, 8.3.10, “firstmodule” on page 8-9, 8.3.26, “unloadmodule” on page 8-18, and 8.3.1, “addmodule” on page 8-3.

8.1.2 freepdus

Source File

berglob.c

Syntax

```
struct pdu  *freepdus;
```

Purpose

The freepdus variable is the head of a list of pdu structures which are available for allocation. The runtime library acquires storage for the pdu structures from the heap, but it does not free the storage itself. Instead, the freepdu() routine places the freed structures on this list. The runtime routine for newpdu() will first look at this list to find an available pdu structure. If it finds none, it will allocate storage from the heap. See 8.3.17, “newpdu” on page 8-13 and 8.3.11, “freepdu” on page 8-10.

8.1.3 usrencfn

Source File

berglob.c

Syntax

```
int (*usrencfn) (struct encinh ia, struct encinhsyn *sa,
                 enum tagclass cls, long id, enum frm form,
                 char *encsubs) =  dfltusrenc;
```

Purpose

The usrencfn variable is a pointer to the user-assigned encoding function. It is initialized to a default encoding function contained in enc.c. The default encoding function always returns a syntax error. The user application uses the setusrencfn() routine to set this variable to something that makes more sense than the default function. See 8.3.24, “setusrencfn” on page 8-17.

8.1.4 version_string

Source File

version.c

Syntax

char *version_string="ASN1/BER version *vernum*";

Purpose

The version_string variable points to a string which indicates the version number of the ASN1BER package. This string is used by the TBLXLATE program and is embedded int the *DAT* files it produces. The runtime routine readdatfile(), see 8.3.21, “readdatfile” on page 8-15, compares this string with the corresonding one in the *DAT* files to ensure their versions are compatible. The ASN1 and SORTRES programs also use this string.

8.2 Routines by Category

The runtime library routines may be grouped into these categories:

- Initialization/Termination
- Searching/Accessing
- Data Conversion
- Encoding/Decoding
- Data Formatting

8.2.1 Initialization/Termination

addmodule	Adds an ASN.1 module to the list of modules in the runtime environment
freepdu	frees a PDU tree
initmodules	initializes runtime environment for ASN.1 modules
newpdu	creates and initializes a PDU tree node
readdatfile	reads a *DAT* file into the runtime environment
resolveallimports	resolves the imported symbols in each of the active modules in the runtime environment
resolveimports	resolves the imported symbols for one module
setusrencfn	sets the user encoding function address

unloadmodule unloads a module from memory

8.2.2 Searching/Accessing

basetype determines the base type of an ASN.1 type

findmodule finds a module by name among the active modules

findtype find a type definition in a module given its type name

findval find a value definition in a module given its value name

findtbl find a table definition in a module given its table name

firstmodule returns the first module in the list of active modules in the runtime environment

8.2.3 Data Conversion

lngs2OID convert an array of longs to an object identifier value

INTEGERtolong converts an ASN.1 INTEGER into a long int

longtoINTEGER converts a long int into an ASN.1 INTEGER

OIDtolngs converts an obect identifier to an array of long ints

8.2.4 Encoding/Decoding

decber decodes BER-formatted data into a PDU tree

decode decodes a PDU tree according to an ASN.1 type

encber encodes BER-formatted data from a PDU tree

encode encode a PDU tree according to an ASN.1 type

8.2.5 Formating

hexstr converts a binary hex string to a printable form

printpdu prints a formatted version of a PDU structure

PrintUnresolvedImports prints unresolved imported symbols

typestr converts an ASN.1 type to a printable string

8.3 Alphabetical Listing

The runtime library routines are described in the following sections.

8.3.1 addmodule

Include

modfns.h

Syntax

int addmodule(struct moddef * *m*)

Parameters

m address of the module definition to be added to the list of active modules in the runtime environment

Results

ADDMODULE_OK The module was successfully added

ADDMODULE_DUP_MODULE
Another module with the same name already exists in the list of active modules in the runtime environment.

Purpose

The addmodule routine is used to add an ASN.1 module to the list of active modules in the runtime environment. It does not resolve external references.

8.3.2 basetype

Include

basetype.h

Syntax

int basetype(struct moddef ***m*, tbldex **t*,
enum basetypes **bt*);

Parameters

m address of a pointer to a module structure. As an input parameter, this points to the module containing the type to be resolved. As an output pointer, it is updated to the module containing the base type.

t address of a metatable index. As an input parameter, it refers to the type to be resolved. As an output parameter, it is updated to refer to the base type specification in the metatable.

Results

BASETYPE_OK The base type was successfully found

BASETYPE_WRONG_TOKEN
In searching for the base type, a metatoken was encountered that shouln't have been there, indicating a structural problem with the metatable.

BASETYPE_UNRESOLVED_EXTERN_REF
In searching for the base type, an unresolved external reference was encountered.

Purpose

The basetype routine determines the base type of a given type according to these rules:

- base(tagged type) = base(type being tagged)
- base(typereference) = base(referenced type)
- base(selection type) = base(choice alternative selected)
- base(something else) = something else

8.3.3 decber

Include

decber.h

Syntax

int decber(struct pdu **p*, char **buff*);

Parameters

p	address of a pointer to the PDU tree created by decoding the data in the buffer; the pointer is set by this routine.
buff	address of a buffer containing the basic encoding rule-formated data

Results

DECBER_OK	The decoding process was successful
DECBER_SYNTAX_ERROR	A syntax error was encountered in the buffer.

Purpose

The decber routine decodes a pdu contained in the buffer pointed to by the *buf* parameter. The buffer is assumed to hold a PDU encoded accoring to the basic encoding rules. The result will be a PDU tree, a pointer to which is set in **p*.

8.3.4 decode

Include

dec.h

Syntax

int decode(struct moddef * *m*, tbldex *metaindex*, struct pdu * *p*);

Parameters

m	address of the module definition structure for the ASN.1 type to be decoded
metaindex	metatable index for the ASN.1 type to be decoded
p	address of the pdu structure for the value to be matched with the ASN.1 type

Results

A code indicating the success or failure of the operation:

DECODE_MATCH The PDU tree matches the ASN.1 type.

DECODE_NO_MATCH
The PDU tree does not match the ASN.1 type.

DECODE_SYNTAX_ERROR
The PDU tree does not match the ASN.1 type and furthermore, some syntax error has been detected, e.g. a constructed INTEGER value.

DECODE_INTERNAL_ERROR
A programming problem with the decode routine was encountered.

DECODE_UNRESOLVED_REFERENCE
An unresolved external reference was encountered.

Purpose

The decode routine attempts to match an ASN.1 type with a value. The value is represented by a PDU tree, and the type is represented by a reference into the metatable. The PDU tree is updated to indicate the features of the matched type, e.g. the name and base type.

8.3.5 encode

Include

enc.h

Syntax

int encode(struct moddef **m*, tbldex *metaindex*,
struct pdu ***p*, void * *userp*);

Parameters

m	address of the module structure containing the type to be encoded
metaindex	metatable index of type definition to be encoded
p	address of a pointer to the encoded pdu structure resulting from the encoding process. This pointer is set by the encode routine.
userp	address of a user-defined parameter which is passed to the user encode routine during the encoding process.

Results

ENCODE_MATCH The encode operation concluded successfully

ENCODE_NO_MATCH
The encoding operation terminated unsuccesfully due to a mismatch in the syntax of the value to be encoded and its related type.

ENCODE_SYNTAX_ERROR
The encoding operation terminated unsuccessfully due to a syntax error.

ENCODE_INTERNAL_ERROR
The encoding operation terminated unsuccessfully due to an internal programming error.

ENCODE_UNRESOLVED_REFERENCE
The encoding operation terminated unsuccessfully due to an unresolved external reference.

Purpose

The encode routine is used to create a PDU structure which represents a value of the type referenced by the *m* and *metaindex* parameters. The encoding process involves calling the user encode routine (see the setusrencfn routine) to supply individual values and to make various encoding decisions. The *userp* is one of the parameters passed to the user encode function. It is a user-defined parameter whose contents are not examined by the runtime library.

8.3.6 findmodule

Include

modfns.h

Syntax

struct moddef *findmodule(const char **mname*);

Parameters

mname null-terminated name of the module to search for

Results

The address of the module structure with the specified name, or NULL if no module is found with the specified name.

Purpose

The findmodule routine searches the active modules in the runtime environment for one with the name specified by the *mname* parameter.

8.3.7 findtbl

Include

lookup.h

Syntax

int findtbl(const struct moddef **mod*, const char **symname*, tbldex **fdex*);

Parameters

mod	address of the module structure containing the table to be found
symname	address of the string containing the name of the symbol to be found
fdex	address of a table index variable which will be set to the table definition in the metatable, if the find operation is successful. If the find operation is not successful, the variable is unchanged.

Results

FIND_TABLE_OK if the definition is found; FIND_TABLE_FAIL if no definition is found

Purpose

The findtbl routine is used to look up a definition in a metatable and return its index. If the definition is not found, FIND_TABLE_FAIL is returned.

8.3.8 findtype

Include

lookup.h

Syntax

int findtype(const struct moddef **mod*, const char **symname*, tbldex **fdex*);

Parameters

mod	address of the module structure containing the type to be found
symname	address of the string containing the name of the symbol to be found
fdex	address of a table index variable which will be set to the type definition in the metatable, if the find operation is successful. If the find operation is not successful, the variable is unchanged.

Results

FIND_TYPE_OK if the definition is found; FIND_TYPE_FAIL if no definition is found

Purpose

The findtype routine is used to look up a definition in a metatable and return its index. If the definition is not found, FIND_TYPE_FAIL is returned.

8.3.9 findval

Include

lookup.h

Syntax

int findval(const struct moddef **mod*, const char **symname*, tbldex **fdex*);

Parameters

mod	address of the module structure containing the value to be found
symname	address of the string containing the name of the symbol to be found
fdex	address of a table index variable which will be set to the value definition in the metatable, if the find operation is successful. If the find operation is not successful, the variable is unchanged.

Results

FIND_VALUE_OK if the definition is found; FIND_VALUE_FAIL if no definition is found

Purpose

The findval routine is used to look up a definition in a metatable and return its index. If the definition is not found, FIND_VALUE_FAIL is returned.

8.3.10 firstmodule

Include

modfns.h

Syntax

int firstmodule(struct moddef ***modptr*, char ***modname*);

Parameters

modptr	address of a pointer to the first module definition in the active module list. The pointer is set by the firstmodule routine. If the active list is empty, the pointer is set to NULL.
modname	address of a pointer to the first module name in the active module list. The pointer is set by the firstmodule routine. If the active list is empty, the pointer is set to NULL.

Results

FIRST_MOD_OK	The active module list is not empty and the *modptr* and the *modname* pointers are not NULL.
FIRST_MOD_FAIL	The active module list is emply and the *modptr* and the *modname* pointers are set to NULL.

Purpose

The firstmodule routine returns the first module of the list of active modules in the runtime environment. This function may be used in conjunction with the unloadmodule routine to unload all the modules in the active list.

8.3.11 freepdu

Include

pdu.h

Syntax

void freepdu(struct pdu *_p_);

Parameters

p	address of the PDU structure to be freed

Results

None

Purpose

The freepdu routine frees an entire tree addressed by the *p* parameter. The contents fields which are marked *allocd* are freed (returned to the heap). Note that the pdu structures which are freed are chained to the *freepdu* list rather than being returned to the heap.

8.3.12 hexstr

Include

hexstr.h

Syntax

void hexstr(char **outbuf*, int *outlen*,
char **inbuf*, long *inlen*);

Parameters

outbuf	address of the output buffer to contain the hex characters
outlen	length (in bytes) of the output buffer
inbuf	address of the input buffer containing the binary version of the hexadecimal string
inlen	length (in bytes) of the input buffer

Results

None

Purpose

The hexstr routine converts a binary version of a hexadecimal string into a printable format.

8.3.13 initmodules

Include

modfns.h

Syntax

void initmodules(void);

Parameters

None

Results

None

Purpose

The initmodules routine initializes the runtime environment for processing ASN.1 modules. This routine should be called only once because it resets internal variables, some of which refer to dynamically allocated storage which will not be freed by initmodules.

8.3.14 INTEGERtolong

Include

intg2lng.h

Syntax

int INTEGERtolong(char **buf*, long *loc*, long **l*);

Parameters

buf	address of the buffer containing the basic encoding rule-formatted INTEGER
loc	length (in bytes) of the INTEGER
l	address of the long int which will hold the converted value

Results

INTG2LNG_OK	The conversion was successful.
INTG2LNG_ERROR	The conversion was unsuccessful.

Purpose

The INTEGERtolong routine converts an ASN.1 INTEGER, formatted according to the basic encoding rules, into a long int. The *buf* parameter points to the contents of the INTEGER; it does not include the id or length octets.

8.3.15 longtoINTEGER

Include

lng2intg.h

Syntax

int longtoINTEGER(long *l*, unsigned char **buf*, long **loc*);

Parameters

l	the long integer which is to be converted
buf	the buffer into which the converted ASN.1 INTEGER is placed
loc	the length (in bytes) of the output buffer; it must be at least 4.

Results

LONG2INTEGER_OK
The conversion was successful.

Purpose

The longtoINTEGER routine converts a long int into an ASN.1 INTEGER value according to the basic encoding rules.

8.3.16 lngs2OID

Include

lngs2oid.h

Syntax

int lngs2OID(long **a*, unsigned char **buf*,
int *buflen*, int **loc*, int *compress*);

Parameters

a	address of an array of long integers, each element being one of the object identifier components. The array is terminated with a -1 value.
buf	address of a buffer to hold the encoded object identifier value
buflen	length of the buffer in characters
loc	address of the integer which will be set to the length of the contents of the object identifier value (number of bytes used to encode the object identifier value)
compress	a flag to indicate whether the routine is supposed to compress the first two components into one subidentifier. A non-zero value indicates compression shall be done.

Results

LNGS2OID_OK The conversion was successful.

LNGS2OID_ERROR
The conversion was unsuccessful. Reasons for failure include an inadequate output buffer or an invalid component.

Purpose

The lngs2OID routine converts an array of object identifier components into an object identifier value according to the basic encoding rules. The array is terminated with a -1 value. The *compress* parameter indicates whether the first two components are compressed into a single subidentifier (as the basic encoding rules require). The complete object identifier value may be encoded by repeated calls to this routine; the first call (to build the prefix) would have the *compress* flag set to a non-zero value, and subsequent calls (to build the suffixes) would indicate no compression with a zero value for *compress*.

8.3.17 newpdu

Include

pdu.h

Syntax

```
struct pdu  * newpdu(char *modname, char *symname,
  struct ietag *it, enum tagclass cls,
  long id, enum frm form, char *contents,
  long loc, enum stg contstg);
```

Parameters

modname	address of the module name for the type associated with this PDU node.
symname	address of the symbol name for the type associated with this PDU node.
it	implicit tagging information address or NULL
cls	class of the item being built into the PDU
id	identifier number of the item being built into the PDU
form	form of the item being built into the PDU
contents	address of the contents field for the PDU
loc	length of the contents field
contstg	storage class for the contents field

Results

A pointer to the new PDU structure

Purpose

The newpdu routine creates a new pdu structure and initializes most of its fields. If the *it* parameter is not NULL, the information addressed by it overides the *cls* and *id* parameters. The newpdu routine uses an existing structure on the free PDU list, if one is available; otherwise, it will allocate one from the heap. The *contstg* parameter indicates whether the storage associated with the contents field (pointed to by *contents*) should be freed when the PDU structure is freed.

8.3.18 OIDtolngs

Include

oid2lngs.h

Syntax

int OIDtolngs(char **buf*, int *loc*,
long **a*, int *asize*);

Parameters

buf	address of buffer containing the object identifier value contents
loc	length of the contents field
a	address of an array of long ints to hold the converted object identifier components
asize	maximum number of elements in the array

Results

OID2LNGS_OK	The conversion was successful.
OID2LNGS_ERROR	The conversion was unsuccessful.

Purpose

The OIDtolngs routine converts the contents part of an object identifier value into an array of long ints, one object identifier component per array element. The array is terminated by an element containing -1. The *asize* parameter must be large enough to include the terminating -1 element.

8.3.19 printpdu

Include

printpdu.h

Syntax

int printpdu(struct pdu **p*, int *indent*);

Parameters

p	address of the pdu tree to be printed
indent	the number of spaces to indent each nesting level of the embedded PDUs

Results

PRINT_PDU_OK The print operation was successful.

PRINT_PDU_ERROR
The print operation was unsuccessful.

Purpose

The printpdu routine is used to print a formatted representation of a PDU tree. The *indent* parameter is used to set the number of spaces with which to indent each nesting level of the embedded PDUs.

8.3.20 PrintUnresolvedImports

Include

prtunres.h

Syntax

int PrintUnresolvedImports(FILE **outfile*);

Parameters

outfile	file descriptor for the report

Results

PRTUNRES_OK The print operation was successful.

PRTUNRES_BAD_FILE
The print operation was unsuccessful because of a unusable file descriptor

Purpose

The *PrintUnresolvedImports* routine is used to print a formatted listing of the unresolved imported symbols in the current list of ASN.1 modules. Typically, you would use this routine to diagnose the cause of a bad return code from the *resolveallimports* or the *resolveimports* routines.

The *outfile* parameter is assumed to be a descriptor for a file which is already open. This routine neither opens or closes this file.

8.3.21 readdatfile

Include

modfns.h

Syntax

int readdatfile(char *_fn_);

Parameters

fn	A null terminated string for the filename of the *DAT* file.

Results

A return code with one of these values:

READ_DAT_OK The *DAT* file was successfully read into the runtime environment.

READ_DAT_OPEN An error occured in opening the file.

READ_DAT_VERSION
The *DAT* file has version number incompatible with the runtime library.

ADDMODULE_DUP_MODULE
A duplicate module was found already loaded into the runime environment.

Purpose

The module definitions in the referenced file are read into the runtime environment and added to the list of active ASN.1 modules. Note that external references are *not*resolved by the readdatfile routine.

8.3.22 resolveallimports

Include

modfns.h

Syntax

int resolveallimports(void);

Parameters

None

Results

RESOLVE_IMPORTS_OK
All imported symbols were resolved

UNRESOLVED_REFERENCE
One of the imported symbols could not be resolved amongst the active modules

Purpose

The resolveallimports routine resolves the imported symbols from each of the active modules in the runtime environment. This operation can be done after a number of modules have been added (with the addmodule or readdatfile routines), and it can be done subsequently after other modules have been added.

8.3.23 resolveimports

Include

modfns.h

Syntax

int resolveimports(struct moddef * *m*);

Parameters

m address of the module structure whose imported symbols are to be resolved against the list of active modules in the runtime environment

Results

RESOLVE_IMPORTS_OK
All imported symbols were resolved

UNRESOLVED_REFERENCE
One of the imported symbols could not be resolved amongst the active modules

Purpose

The resolveimports routine is used to resolve the imported symbols from the module addressed by the *m* parameter.

8.3.24 setusrencfn

Include

setusr.h

Syntax

void setusrencfn(int (*usrencfn) (struct encinh *ia*,
struct encinhsyn **sa*,
enum tagclass *cls*, long *id*,
enum frm *form*, char **encsubs*));

Parameters

usrencfn the address of a function with the prototype shown above

Results

None

Purpose

The setusrencfn routine initializes a variable to point to a user-defined encoding routine. This encoding routine is invoked during the encoding process to supply values and make various encoding decisions. The setusrencfn may be called anytime, even during an encode operation, to set or change the user encoding function.

8.3.25 typestr

Include

typestr.h

Syntax

char *typestr(enum basetypes *base*);

Parameters

base the base which is to be converted

Results

Returns a pointer to a character string which is a printable version of the value in *base*

Purpose

The typestr routine returns a pointer to a character string which is a printable version of the value in *base*.

8.3.26 unloadmodule

Include

modfns.h

Syntax

int unloadmodule(struct moddef **modptr*);

Parameters

modptr address of the module structure to be unloaded

Results

UNLOAD_MOD_OK
The module was successfully unloaded

UNLOAD_MOD_NOT_FOUND
The module was not found among the list of active modules

Purpose

The unloadmodule routine is used to remove a module from the active list of modules and free up dynamically allocated space associated with it. This routine should only be used for modules which are dynamically loaded by the readdatfile routine. Note that a single *DAT* file may contain many modules; to remove all of them, repeated calls the the unloadmodule function are needed. When the last module of a *DAT* file is unloaded, the dynamic space for its symbol table is also freed.

Chapter 9. The Metatable Structure

The structure of the metatable is directly related to the abstract syntax notation from which it is derived. Some notable differences are:

- The metatable structure is more compact than the abstract syntax.
  - Most keywords are reduced to a single byte in the metatable.
  - The metatable omits some structures present in the abstract syntax, e.g. comments.
  - Symbolic information is stored in tables; a single symbol only appears once in a table even though it may be referenced many places in the abstract syntax.
- The metatable structure is more amenable to computer processing by the encoder/decoder functions at application runtime.
  - The encoded information is reentrant and self-relocatable, meaning that it can be efficiently loaded/unloaded dynamically.
  - Information is encoded to efficiently find the end of a structure or a part of another structure, e.g. to identify the end of a *SEQUENCE*.
  - Optimizations are performed to directly encode a value when the abstract syntax may specify it through many indirections.

9.1 The Role of the ASN.1 Compiler

The role of the ASN.1 compiler is to ensure the correctness of the abstract syntax and to optionally produce a file, called a table, or *TBL* format file, which is intended to be a machine-independant representation of the abstract syntax as is possible. Machine independence is gained by only using character forms for all datatypes, including those for decimal numbers and some special types, such as for object identifier values.

By itself, the *TBL* file is not of much use to any application program. Although the *TBL* file is a fair representation of the original abstract syntax, it is not represented in a form that is the most efficient for application programs to perform encoding/decoding operations. The *TBL* format represents an intermediate form; it is more compact and has more embedded information than the original ASN.1 source, but it is still as machine-independent as is possible. This machine-independence allows the compiler to be run on one type of machine (e.g., a S/390), and the application program to run on another type of machine (e.g., a PS/2). The TBLXLATE program provides the bridge between the compilation environment for the ASN.1 syntax and the execution environment of the application program which must perform encoding and decoding operations per the ASN.1 specifications.

9.2 The Role of the TBLXLATE Program

The job of the TBLXLATE program is to translate the machine-independent representation of the metatable into a more compact and efficient form for the runtime application and the encode/decode routines.

The encode/decode routines depend on the C language data structures created by the TBLXLATE program. The TBLXLATE program is tailored to fit the execution environment of the application program. Because the data structures upon which it operates are based on the C programming language, the differences between the compilation environment (i.e., the input format) and the application environment (i.e., the output format), are typically minimal.

The following sections provide the details of both the TBL format and how they relate to the C language formats of the application environment.

9.3 Structure of a TBL File

The TBL file contains the following structures, in order:

1. Version identifier
2. Symbol Table length
3. Symbol Table name
4. Symbol Table
5. Module definitions

The version identifier is string, ended by a newline character, which is used to verify that the versions of the ASN1 program, which created the TBL file, and the TBLXLATE program are the same. An example verion identifier is:

```
ASN1/BER version 2.01
```

The symbol table length specifies the number of characters in the symbol table. Its format is a series of characters, ended by a newline character, of the following example form:

```
symbol table len: 495
```

The symbol table name is used in constructing a name for a data structure in the C file format. It is not used for the *DAT* file format. The ASN1 compiler uses one of the module reference identifiers for the symbol table name. Its format is a series of characters, ended by a newline character, of the following example form:

```
symbol table name: Abc
```

The symbol table contains each symbol needed by the module definitions which follow. Within the TBL file, each symbol is separated from its neighbors by a newline character. In the C and *DAT* file formats, the symbols are separated from one another by a null ('\0') character. The Figure 9-1 on page 9-3 is an example symbol table:

The module definitions appear after the symbol table and are discussed next.

9.3.1 Structure of Module Definitions

Each module definition in the TBL file is of this format:

1. Module identifier 1
2. Number of elements in the module
3. Type, value and table definitions
4. End of type, value and table definitions
5. Number of types, values, and table definitions
6. List of symbol table references for the types, values and table definitions
7. Number of imported symbols
8. Imported symbol information
9. Number of exported symbols

```
GLOBAL-MODULE
Abc
A
B
C
D
E
F
G
H
I
J
TBL1
TBL2
oid1
K
L
a
b
M
N
O
P
Q
R
S
T
V
W
X
Y
Z
A1
B1
C1
D1
E1
F1
G1
H1
Xyz
Bitstr
Octstr
INTEGER
EXPLICIT
IMPLICIT
BOOLEAN
NULL
OCTET STRING
REAL
foo1
foo2
foo3
foo4
BIT STRING
c
OBJECT IDENTIFIER
d
SET
```

Figure 9-1 (Part 1 of 2). Example Symbol Table

10. Exported symbol information
11. Module identifier 2
12. Symbol table index for the module name
13. Default tagging

The module identifier 1 is a string with this example format:

```
SEQUENCE OF
SET OF
b1
b2
b3
bool
int
fum
i
n
ENUMERATED
foo
fee
ANY
NumericString
PrintableString
TeletexString
T61String
VideotexString
VisibleString
ISO646String
IA5String
GraphicString
GeneralString
GeneralizedTime
UTCTime
EXTERNAL
ObjectDescriptor
```

Figure 9-1 (Part 2 of 2). Example Symbol Table

`module: Abc`

The number of elements in the module is specified with this example format:

`number of elements: 170`

An element corresponds to one metatoken structure as specified in the `tokstruc.h` file.

The type, value and table definitions occur next. Each type, value and table definition starts with a line consisting of "-2", an indicator of whether the definition is a type, value or table definition, and the name of the type, value, or table being defined. Then the elements for the definition follow, one line per element. For example, the following fragment shows two type definitions:

```
-2 type: A
2 18
1 137 3 12 6
3 129 3
26 6
27 0 0 2
32 42
```

Each element is specified as a series of numbers. The number of them and the meaning of each number in the element depends on the ASN.1 source used to define the corresponding type, value or table. Each element corresponds to one of the structures in the `tokstruc.h` file. The TBLXLATE program uses these to create the metatable parts of the C and *DAT* files. The first number is a token identifier as defined in the file `metatoks.h`.

The end of the type, value and table definitions is indicated by a single line consisting of "-1", for example:

```
-1
```

The number of type, value and table definitions follows next and is of this example format:

`types/values/tables: 38`

The list of symbol table references for the type, value and table definitions follow. This is an example list:

```
0
6
12
14
16
23
25
38
43
46
61
67
73
93
104
106
110
112
114
116
118
120
122
124
126
128
130
132
134
136
138
140
142
144
147
150
153
163
```

The number of imported symbols follows next and is of this example format:

```
imports: 2
```

Note: The number of imported symbols may not be exactly equal to those listed in the ASN.1 source IMPORTS clause. The ASN1 compiler includes external references (e.g., `Abc.X`), in the imported symbol list.

The imported symbol information follows next. Each imported symbol has one line in the TBL file consisting of two numbers. The first number is symbol table reference for the module name of the imported symbol. The second number is the symbol table reference for the imported identifier. The following example shows two imported symbols:

```
111 115
111 122
```

The number of exported symbols follows next and is of this example format:

```
exports: 2
```

The exported symbol information consists of zero or more lines, one line per exported symbol. Each line consists of two numbers. The first number is the symbol table reference for the name of the exported symbol. The second number is the element number of the corresponding definition in this module. This example shows two exported symbols:

```
20 6
24 14
```

The module identifier 2 follows next and looks like this:

```
Module: Abc
```

The symbol table reference for the module identifier follows next and looks like this:

```
14
```

The default tagging for the module follows next on a single line and consists of the characters, "implicit" or "explicit". For example:

```
implicit
```

9.4 Structure of the *DAT* File

The *DAT* file is created as one of the output files of the TBLXLATE program. Its structure is similar to the TBL structure. It consists of the following components, in order:

1. Version identifier (`char []` , terminated by '\n')
2. Symbol table length (`int`)
3. Symbol table (`char []`)
4. Module definitions

The version identifier is a string used to ensure the *DAT* file is compatible with the version of the runtime library used by the application. It is terminated by a newline character (not a null char).

The symbol table length is an `int` and its value is the number of characters in the symbol table.

The symbol table is contains all the symbols needed by the module definitions. Each symbol is terminated by a null ('\0') character.

The module definitions follow. Each module definition consists of the following information:

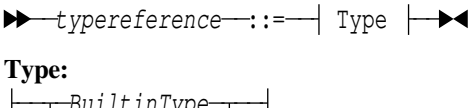
1. Number of bytes in the metatable (`int`)
2. Metatokens (mtok [])
3. Number of type, value and table definitions (`int`)
4. List of type, value and table definition references (`tbldex`)
5. Number of imported symbols (`int`)
6. List of imported symbol information (`struct berimp []`)
7. Number of exported symbols (`int`)
8. List of exported symbol information (`struct berexp []`)
9. Symbol table index for the module name (`symdex`)
10. Module tagging default (`enum tagging`)

The following sections provide information on how the ASN.1 definitions are mapped into the metatable structures. These metatable structures are mapped into the metatoken sections of the *DAT* files.

9.5 Structure of Type Assignments

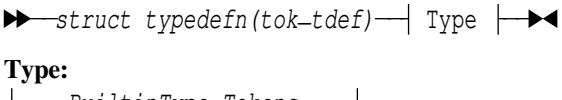
The structure of a type assignment is:

Type Assignment



The corresponding metatokens are:

Type Assignment Tokens



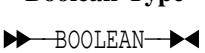
The specific type tokens are shown in the next section.

9.6 Structure of Types

9.6.1 BOOLEAN

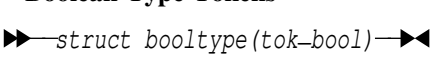
The syntax for a boolean datatype is:

Boolean Type



The corresponding metatokens are:

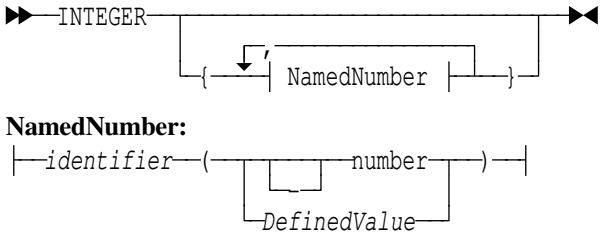
Boolean Type Tokens



9.6.2 INTEGER

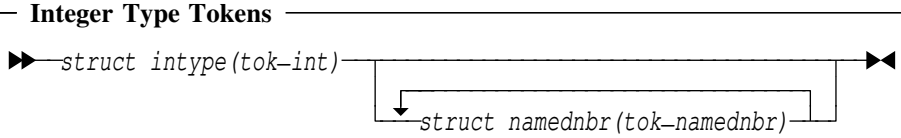
The syntax for an integer datatype is:

Integer Type



The corresponding metatokens are:

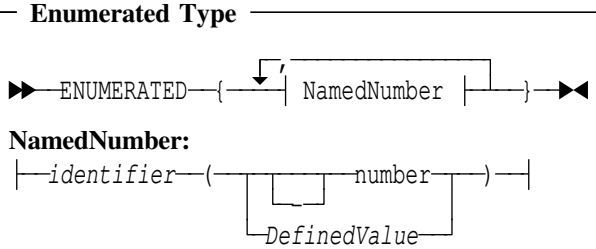
Integer Type Tokens



9.6.3 ENUMERATED

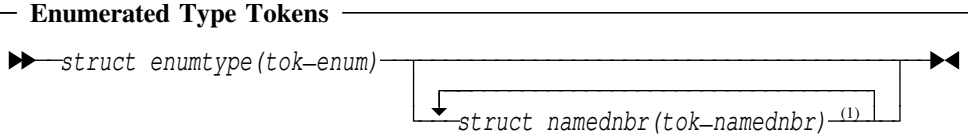
The type notation is:

Enumerated Type



The corresponding metatokens are:

Enumerated Type Tokens

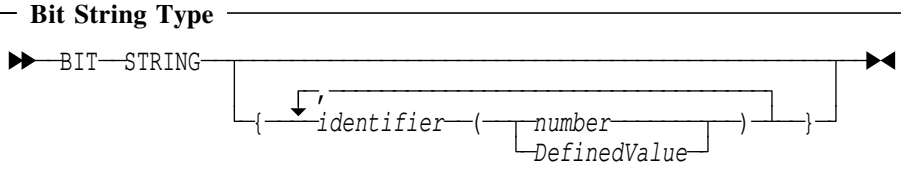


Note:
1 The compiler must be able to find the defined value for a named number in the source file for it to be set in the metatable.

9.6.4 BIT STRING

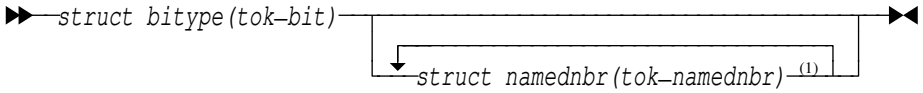
The notation for a bit string type is:

Bit String Type



The corresponding metatokens for a bit string type are:

Bit String Type Tokens

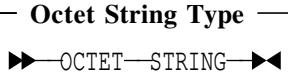


Note:
1 The compiler must be able to find the defined value for a named bit in the source file for it to be set in the metatable.

9.6.5 OCTET STRING

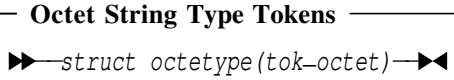
The syntax for an octet string type is:

Octet String Type



The corresponding metatokens for an octet string type are:

Octet String Type Tokens



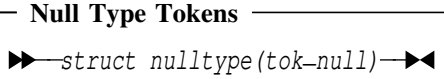
9.6.6 NULL

The notation for a null type is:

Null Type

The corresponding metatokens for a null type are:

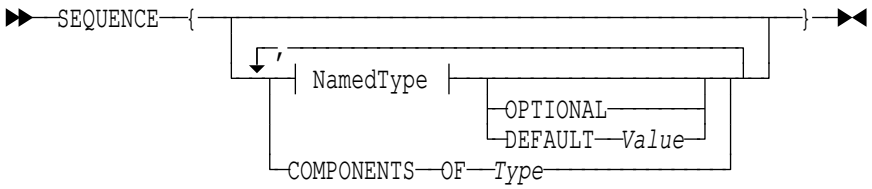
Null Type Tokens



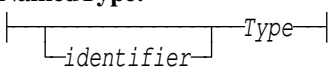
9.6.7 SEQUENCE

The type notation for a sequence is as follows:

Sequence Type

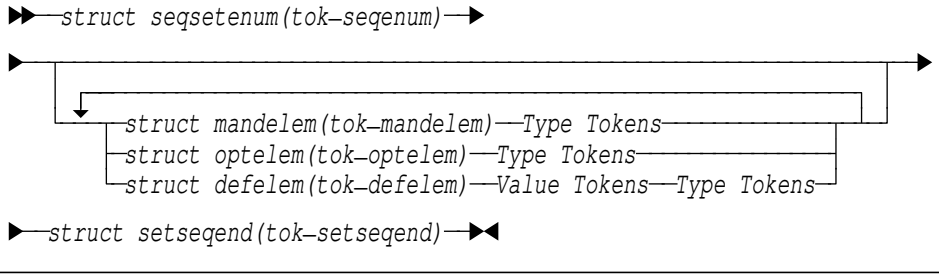


NamedType:



The corresponding metatokens are:

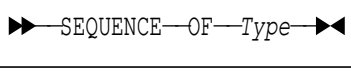
Sequence Type Tokens



9.6.8 SEQUENCE OF

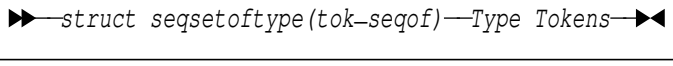
The type notation for sequence of is:

Sequence Of Type



The corresponding metatokens are:

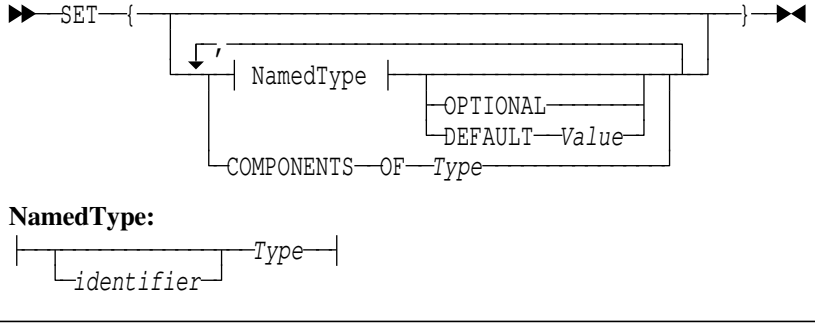
Sequence Of Type Tokens



9.6.9 SET

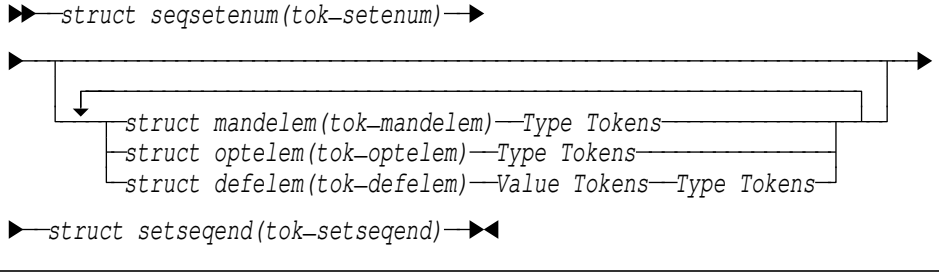
The type notation for a set is:

Set Type



The corresponding metatokens are:

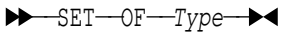
Set Type Tokens



9.6.10 SET OF

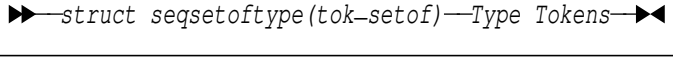
The notation for the type is:

Set Of Type



The corresponding metatokens are:

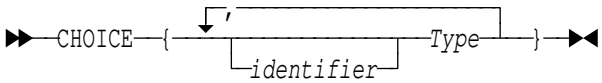
Set Of Type Tokens



9.6.11 CHOICE

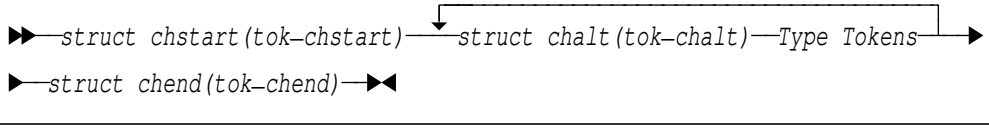
The type notation is:

Choice Type



The corresponding metatokens are:

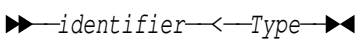
Choice Type Tokens



9.6.12 Selection Type

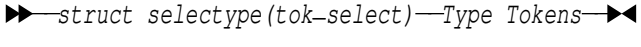
The type notation is:

Selection Type



The corresponding metatokens are:

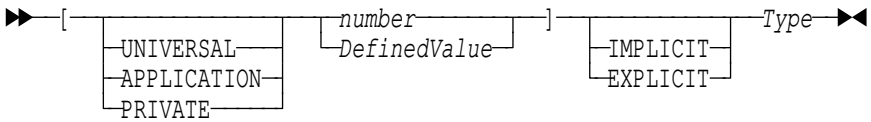
Selection Type Tokens



9.6.13 Tagged Type

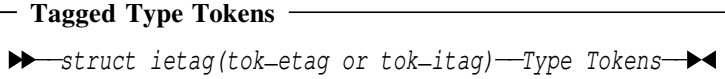
The type notation is:

Tagged Type



The corresponding metatokens are:

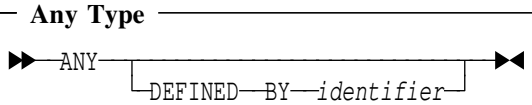
Tagged Type Tokens



9.6.14 Any Type

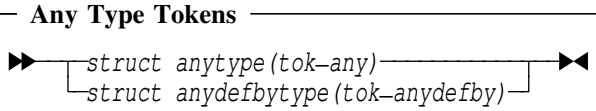
The type notation is:

Any Type



The corresponding metatokens are:

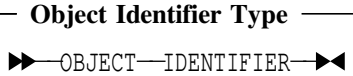
Any Type Tokens



9.6.15 OBJECT IDENTIFIER Type

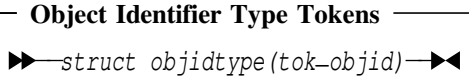
The type notation is:

Object Identifier Type



The corresponding metatokens are:

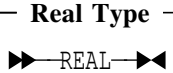
Object Identifier Type Tokens



9.6.16 REAL Type

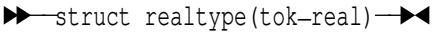
The type notation is:

Real Type



The corresponding metatokens are:

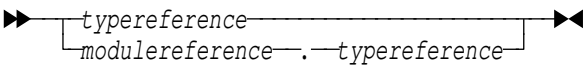
Real Type Tokens



9.6.17 Defined Type

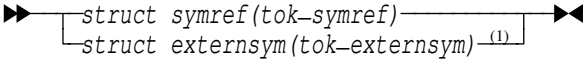
The notation for a defined type is:

Defined Type



The corresponding metatokens for a defined type are:

Defined Type Tokens



Note:
1 The *externsym* structure is also used to represent imported type references

9.6.18 Character String and Useful Types

The character string and useful types are all represented by these metatokens:

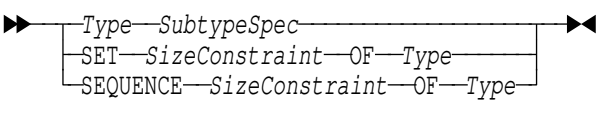
Character/Useful Type Tokens

9.7 Structure of Subtypes

The subtype notation is used in conjunction with the type notation. For all types except SET OF and SEQUENCE OF, the subtype specification follows the type specification. For SEQUENCE OF and SET OF, it occurs right before the OF keyword.[1]

The notation for subtypes and where they are used with types is:

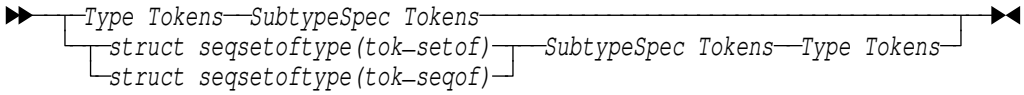
Subtype Uses



The corresponding metatokens are:

[1] This is necessary to avoid ambiguity. If the subtype specification were to occur after the SET OF or SEQUENCE OF type notation, it would be confused as being a subtype specification for the type *in* in the sequence or set.

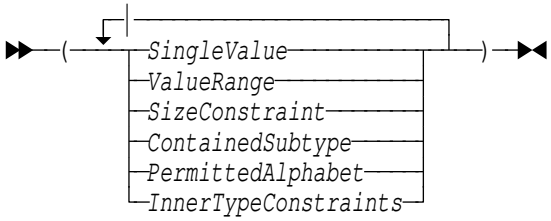
Subtype Tokens



9.7.1 SubtypeSpec

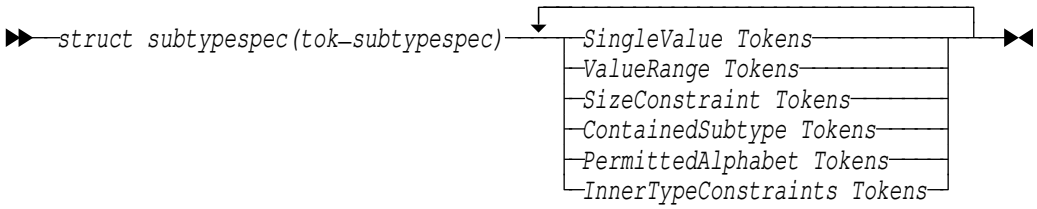
The notation for a subtype specification is:

Subtype Specification



The corresponding metatokens are:

Subtype Specification Tokens



The notation and tokens for the value set constraints are in the following sections.

9.7.2 Single Value Constraint

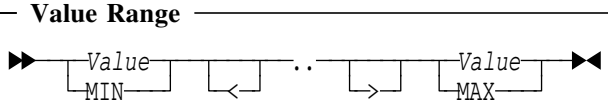
The notation for a single value constraint is the same as the value notation for the parent type.

The metatokens for a single value constraint are the same as those for the same value as the parent.

9.7.3 Value Range Constraint

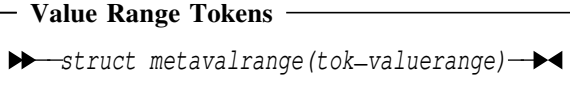
The notation for a value range constraint is:

Value Range



The corresponding metatokens are:

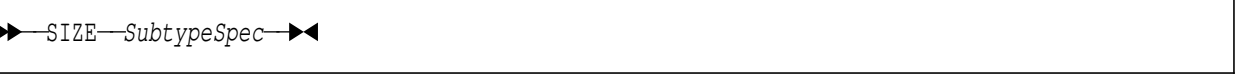
Value Range Tokens



9.7.4 Size Constraint

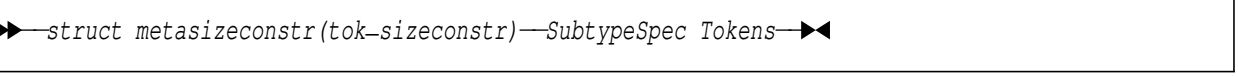
The notation for a size constraint is:

Size Constraint



The corresponding metatokens are:

Size Constraint Tokens



9.7.5 Contained Subtype

The contained subtype notation is supported by the compiler, but is not supported in the metatable.

9.7.6 Permitted Alphabet

The permitted alphabet notation is supported by the compiler, but is not supported in the metatable.

9.7.7 InnerType Constraints

The inner type constraint notation is supported by the compiler, but is not supported in the metatable.

9.7.8 Unimplemented Types

For some types which the compiler does not implement, the metatable will contain the *struct notimpl* structure with a token id of *tok_notimpl*.

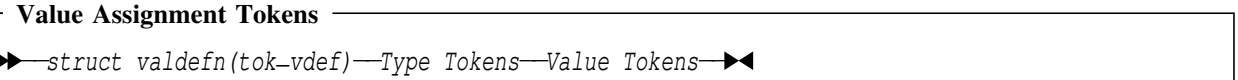
9.8 Structure of Value Assignments

The ASN.1 syntax for a value assignment is:

Value Assignment

The corresponding metatokens are:

Value Assignment Tokens



9.9 Structure of Values

The following sections describe the values.

9.9.1 Number

A number, signed or unsigned, is represented in the metatable by the *struct metavalint* structure with a token id of *tok_valnbr*.

9.9.2 OBJECT IDENTIFIER Value

An object identifier value is represented in the metatable by the *struct metavalobjid* structure with a token id of *tok_objidval*.

The BER representation of the object identifier value is used in the metatable.

9.9.3 BOOLEAN Value

A boolean value is represented in the metatable by the the *struct metavalbool* structure with a token id of *tok_valbool*.

9.9.4 NULL Value

A null value is represented in the metatable by the the *struct metavalnull* structure with a token id of *tok_valnull*.

9.9.5 Hex String Value

A hex string value is represented in the metatable by the the *struct metavalhstr* structure with a token id of *tok_valhstr*.

9.9.6 Binary String Value

A binary string value is represented in the metatable by the the *struct metavalbstr* structure with a token id of *tok_valbstr*.

9.9.7 Character String Value

A character string value is represented in the metatable by the the *struct metavalcstr* structure with a token id of *tok_valcstr*.

The characters in the string are stored in the symbol table and the *struct metavalcstr* structure has an index to them.

9.9.8 CHOICE Value

The notation for a choice value is:

Choice Value

The corresponding metatokens are:

Choice Value Tokens

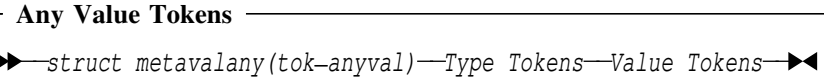
9.9.9 ANY Value

The notation for an ANY value is:

Any Value

The corresponding metatokens are:

Any Value Tokens



9.9.10 Named Number Values

Where a named number is used for a value, the compiler will attempt to replace the reference with the corresponding number value (see 9.9.1, “Number” on page 9-15).

The compiler cannot replace the reference with a number when the named number definition occurs in a different source file. In these cases, the metatable will contain a *struct namednbref* with a token id of *tok_namednbref*.

9.9.11 Unimplemented Values

For some values which the compiler does not implement, the metatable will contain the *struct notimpl* structure with a token id of *tok_notimpl*.

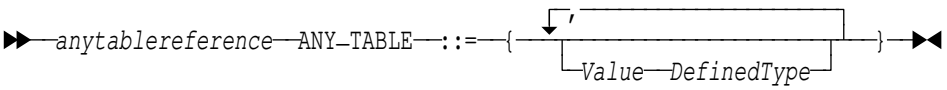
9.10 Structure of Table Assignments and References

The *nonstandard* extensions to ASN.1 for resolving ANY DEFINED BY constructs are called *any tables*. The following sections define the metatable structures to support the table references and assignments.

9.10.1 Any Table Assignment

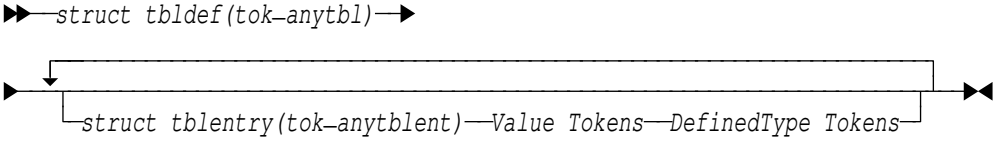
The notation for an *any table* assignment is:

Any Table Assignment



The corresponding metatokens are:

Any Table Assignment Tokens



9.10.2 Any Table References

The *any table* references occur for only two ASN.1 types: ANY and EXTERNAL. In both cases a table reference is represented just like a Defined Type, and the tokens follow the ANY or EXTERNAL tokens and precede any subtype tokens.

Chapter 10. SORTRES

Chapter 11. References

[**ASN1**] ISO/IEC 8824, *Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)* (April 1990)

[**BER**] ISO/IEC 8825, *Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)* (April 1990)

Appendix A. ASN1 Compiler Messages

401 **"*name*" has a duplicate definition at line *l*, column *c***

Severity: 4

Explanation: The symbol, *name*, has been used in another definition at the indicated line and column. ASN.1 allows names to be used more than once in a module, for example named numbers, named bits, choice alternatives, sequence and set components. Defined types and defined values, and imported symbols cannot be used more than once in any one module. Also, the compiler will complain if the same module reference is used to name more than one module.

402 **Internal error:** *info*

Severity: 4

Explanation: The compiler found a fatal internal inconsistency (a bug). The *info* is useful to the person who maintains the compiler for localizing and fixing the problem.

403 *SymbolKind* **"*Symbol*" referenced but not defined in module "*M*"**

Severity: 4

Explanation: A type, value, or table name has been referenced in module *M*, but it has not been defined or imported into the module. Symbols need not be defined before they can be referenced, but they must be defined or imported in the module in which they are referenced.

404 **Value range subtype not legal for parent type**

Severity: 4

Explanation: The value range may only be used for INTEGER and REAL datatypes. An attempt was made to use it with another datatype.

405 **Size constraint subtype not legal for parent type**

Severity: 4

Explanation: The size constraint may only be used for BIT STRING, OCTET STRING, CharacterString, SET OF, and SEQUENCE OF datatypes. An attempt was made to use it with some other datatype.

406 **ANY DEFINED BY reference "*identifier*" is undefined**

Severity: 4

Explanation: The *identifier* in the ANY DEFINED BY *identifier* must refer to a named type in the SEQUENCE or SET where the ANY DEFINED BY appears. The referenced named type may occur before or after the ANY DEFINED BY. Note that the *identifier* must begin with a lowercase letter and should not be confused with the identifier for the component's type. Here is a correctly defined example:

```
A ::= SET{ a ANY DEFINED BY b, b INTEGER}
```

See 5.29, “ANY” on page 5-25 for the syntax of ANY DEFINED BY.

---

407 Component "*comp*" referenced by ANY DEFINED BY must not be OPTIONAL

Severity: 4

Explanation: The referenced component, *comp*, in the SEQUENCE or SET is OPTIONAL. ASN.1 requires the component upon which the ANY DEFINED BY is referenced be mandatory or have a default value. See 5.29, “ANY” on page 5-25 for the syntax of ANY DEFINED BY.

---

408 Component "*comp*" referenced by ANY DEFINED BY must be a subtype of INTEGER or OBJECT IDENTIFIER

Severity: 4

Explanation: The component upon which the ANY DEFINED BY depends, *comp*, is not a subtype of INTEGER or OBJECT IDENTIFIER. See 5.29, “ANY” on page 5-25 for the syntax of ANY DEFINED BY.

---

409 ANY DEFINED BY may only be used in a SEQUENCE or SET

Severity: 4

Explanation: The ANY DEFINED BY construct may *not* be used within a CHOICE, even though the CHOICE is a component of a SET or SEQUENCE. It may not be used to define a type which is then referenced in a SET or SEQUENCE. It may not be tagged. The following are *illegal* examples:

```
A ::= SET{a INTEGER,
          b CHOICE{ [0] INTEGER,
             [1] ANY DEFINED BY a -- This is illegal --
                  }
          }

B ::= SET{a INTEGER, b C}

C ::= ANY DEFINED BY a -- This is illegal --
```

See 5.29, “ANY” on page 5-25 for the syntax of ANY DEFINED BY.

---

410 Illegal ANY DEFINED BY syntax

Severity: 4

Explanation: The illegal syntax for the ANY DEFINED BY construct is usually caused by misspelling one of the keywords or by failing to use an identifier starting with a lowercase letter after ANY DEFINED BY. The line and column number in the error message identify the offending source. See 5.29, “ANY” on page 5-25 for the syntax of ANY DEFINED BY.

411 Expected '-' or a digit

Severity: 4

Explanation: The lexical analyzer found a dash and expected to find another dash, signifying a comment, or a digit, signifying a signed number. The character at the line and column number identifies the offending source.

412 Identifier must not end in a dash - ignored

Severity: 4

Explanation: An ASN.1 identifier may include dashes, but they must:

- not occur at the beginning of the identifier
- not occur consecutively
- not occur as the last character in the identifier

The compiler ignores the trailing dash.

See 5.8, “Identifier Item” on page 5-5, 5.5, “Type Reference Item” on page 5-2.

413 Expected '*char*' for an assignment item, "::="

Severity: 4

Explanation: The compiler was expecting an assignment item, "::=" but found an illegal character at the indicated line and column number.

414 Expected '0' or '1' in binary string item

Severity: 4

Explanation: The lexical analyzer found an illegal character in a binary string item. The only allowed characters in a binary string are '0' and '1'. See 5.9, “Other Items” on page 5-6.

415 Expected hexadecimal digit in string item

Severity: 4

Explanation: The lexical analyzer found an illegal character in a hexadecimal string item. The only allowed characters in a hexadecimal string are '0' through '9' and 'A' through 'F'. Note that lowercase hexadecimal digits 'a' through 'f' are *not* allowed. See 5.9, “Other Items” on page 5-6.

416 Expected 'B' or 'H'

Severity: 4

Explanation: The lexical analyzer had found the terminating single quote for a hexadecimal or binary string item, but did not find the required 'B' (for binary) or 'H' (hexadecimal) letter following the last single quote. Note there must not be any spaces between the single quote and the letter ('B' or 'H') and that the letter must follow, not precede the quoted string. See 5.9, “Other Items” on page 5-6.

417 Too many characters in item, aborting compilation

Severity: 4

Explanation: The lexical analyzer filled up its internal buffer with characters before completing an ASN.1 item. This may be caused by a missing quotation mark.

418 Skipping unrecognized character '*char*'

Severity: 4

Explanation: The lexical analyzer found a character which is not in the ASN.1 repertoire. This may be caused by mismatched quotation marks.

419 "*identifier*" is a recursive *type/value* definition

Severity: 4

Explanation: The "*identifier*" has been used to define itself, perhaps through a chain of indirections. These are *illegal* definitions:

```
A ::= B  -- illegal recursion
B ::= C  -- illegal recursion
C ::= A  -- illegal recursion
```

Some forms of recursion are, however, allowed:

```
Tree ::= CHOICE{ i InternalNode, l LeafNode}
InternalNode ::= SEQUENCE{ leftSubtree Tree,
       rightSubtree Tree}
LeafNode ::= INTEGER
```

420 Script symbol "*sym*" is not recognized

Severity: 4

Explanation: The lexical analyzer either does not support the script substitution symbol *sym*, or it is misinterpreting the source as a containing a script substitution symbol. For this message to occur, the *script* option must be in effect, and the referenced symbol must occur within an ASN.1 section (i.e., between `.* asn.1 on` and `.* asn.1 off`), and it must occur in a context where an ASN.1 item is expected (i.e. not in a comment or character string). See 2.1, “Source File” on page 2-1 for the supported script symbols.

421 Expected "STRING" following "*Keyword*"

Severity: 4

Explanation: The compiler found the keyword "BIT" or "OCTET" and expected the next item to be the keyword "STRING". Note that "BIT STRING" is two separate words, as is "OCTET STRING". They may be on different lines, and even have comments separating them, but there must be no other keywords or punctuation between them.

422 Illegal module definition

Severity: 4

Explanation: The module definition is illegal. There are probably other error messages in the vicinity which indicate which clause(s) are incorrect. The compiler will attempt to recover at the next type or value assignment. An "END" item, if it exists, will probably generate an error message too, since the module definition lines are ignored.

Common problems in module definitions are:

- Failure to terminate an EXPORTS or IMPORTS clause with a semicolon.
- Putting the EXPORTS clause before the IMPORTS clause
- Misspelling keywords or using lowercase letters in "DEFINITIONS", "IMPLICIT", "TAGS", "BEGIN", "EXPORTS", "IMPORTS"

See 5.10, “Module Definition” on page 5-8 for the correct syntax.

423 Illegal *IMPORTS/EXPORTS* symbol list, skipping to ';'

Severity: 4

Explanation: The compiler found an error in an IMPORTS or EXPORTS clause of a module definition. See 5.10, “Module Definition” on page 5-8 for the correct syntax.

424 Expected an assignment

Severity: 4

Explanation: This message occurs when the compiler expects a type assignment or a value assignment but encounters something it cannot digest as one of these. This message is often accompanied by other messages which are in the same vicinity and are more specific about the syntax in error. See 5.11, “Type Assignment” on page 5-11 and 5.12, “Value Assignment” on page 5-12 for the correct syntax.

425 "*Identifier*" is an illegal *type/value* assignment

Severity: 4

Explanation: The parser found the beginning of a valid type or value assignment with the name *Identifier*, but encountered an error somewhere on the right-hand side of the "::=" item. Usually, other error messages will localize the problem.

426 Illegal *INTEGER/ENUMERATED* named number list

Severity: 4

Explanation: The compiler found the beginning of a named number list for an INTEGER or ENUMERATED type, but found an error within the list itself. There is usually another error message in the vicinity which more closely pinpoints the error. See 5.17, “INTEGER” on page 5-15 and 5.18, “ENUMERATED” on page 5-16 for the correct syntax of INTEGER and ENUMERATED.

427 "*identifier*" is an illegal named number

Severity: 4

Explanation: Within the named number list for an INTEGER or ENUMERATED, the compiler encountered a named number, *identifier*, which was illegal. See 5.17, “INTEGER” on page 5-15 or 5.18, “ENUMERATED” on page 5-16 for the correct syntax of a named number list.

428 "*identifier*" must begin with a lowercase letter

Severity: 4

Explanation: The *identifier* begins with an uppercase letter. Named bits (in a BIT STRING) and named numbers (in an INTEGER and ENUMERATED) must begin with lowercase letters.

429 Illegal named bit list

Severity: 4

Explanation: The compiler found the beginning of a named bit list for a BIT STRING type, but found an error within the list itself. There is usually another error message in the vicinity which more closely pinpoints the error. See 5.19, “BIT STRING” on page 5-17 for the correct syntax of BIT STRING.

430 Named bit value must be greater than or equal to zero

Severity: 4

Explanation: The named bit was given a negative value. Bits are numbered starting from zero at the first bit. See 5.19, “BIT STRING” on page 5-17 for the BIT STRING syntax.

431 "*identifier*" illegal named bit

Severity: 4

Explanation: The compiler found incorrect syntax following the *identifier* and it expected a named bit. See 5.19, “BIT STRING” on page 5-17 for the BIT STRING syntax.

432 Illegal element after comma

Severity: 4

Explanation: The compiler found an error in the SET or SEQUENCE component following the indicated comma. Usually other error messages will locate the error more closely. A frequent error is the omission of a comma between components or including the comma as part of a comment instead of placing it outside the comment.

433 Numeric REAL value base must be 2 or 10

Severity: 4

Explanation: ASN.1 requires that the REAL datatype be of base 2 or base 10 only. See 5.39, “Real Number” on page 5-31 for the REAL syntax.

434 Illegal subtype specification

Severity: 4

Explanation: The compiler had found what looks like the beginning of a subtype specification, but encountered an error somewhere inside the specification. Usually, other error messages in the vicinity will localize the problem.

435 Expected a '-' to terminate syntactic extension

Severity: 4

Explanation: The lexical analyzer found what looks like a syntactic extension termination, "%--", but found a wrong character in the indicated position. See -- Heading 'SYNEXT' unknown -- for a description of syntactic extensions.

436 Illegal ANY_TABLE_REF

Severity: 4

Explanation: The compiler found the keyword, "ANY_TABLE_REF", but did not find the correct syntax following it. See 5.6, “Any Table Reference Item” on page 5-4 for a description of the ANY_TABLE_REF syntax.

437 ANY_TABLE_REF, "*Identifier*", does not refer to an ANY_TABLE

Severity: 4

Explanation: The referenced *Identifier* was to a definition of other than an ANY_TABLE. See 5.6, “Any Table Reference Item” on page 5-4 for a description of the ANY_TABLE_REF syntax.

438 "*Identifier*" is an illegal ANY_TABLE assignment

Severity: 4

Explanation: The compiler found what looks like an ANY_TABLE assignment, but encountered syntax errors in the rest of the definition. Usually, other error messages in the vicinity will more closely pinpoint the problem. See -- Heading 'TBLDEF' unknown -- for a description of the ANY_TABLE syntax.

439 Illegal tag

Severity: 4

Explanation: A tag construction was begun with a '[', but was not followed by valid syntax. Common causes are misspelled keywords for UNIVERSAL, etc., and missing ']'. See 5.28, “Tagged Types” on page 5-23 for the correct syntax.

440 OBJECT IDENTIFIER value component "*identifier*" is not defined

Severity: 4

Explanation: The *identifier* must be defined or imported into the ASN.1 module somewhere. It must also be defined as an OBJECT IDENTIFIER value. See 5.30, “OBJECT IDENTIFIER” on page 5-26 for the correct syntax.

441 OBJECT IDENTIFIER value conversion error -- compiler limitation

Severity: 4

Explanation: The OBJECT IDENTIFIER has too many components in it for the compiler to handle. Contact the person maintaining the compiler.

442 OBJECT IDENTIFIER value reference "*identifier* is invalid

Severity: 4

Explanation: The referenced *identifier* is not a correctly defined OBJECT IDENTIFIER, causing a compilation error at this point. Other error messages should occur where the OBJECT IDENTIFIER *identifier* is defined.

443 Incompatible value for the associated type at line *l*, column *c*

Severity: 4

Explanation: The notations for the referenced type and value do not agree. For example, assigning TRUE to a value whose type is INTEGER causes this error. Check the allowed value notation for the datatype in question.

444 Implicitly tagging a CHOICE or ANY is illegal

Severity: 4

Explanation: ASN.1 prohibits implicitly tagging a CHOICE or ANY datatype. See 5.28, “Tagged Types” on page 5-23 for the restrictions and requirements for tagging.

445 "*Identifier*" is defined as a type, but it has an incompatible reference

Severity: 4

Explanation: The type *Identifier* is defined as a type but it is referenced as something else, probably as an ANY_TABLE.

---

446 "*identifier*" is already used in line *l* column *c*

Severity: 4

Explanation: The *identifier* was used more than once in a context where the identifiers must all be unique. Specifically, distinct identifiers must be used within a single instance of:

- CHOICE alternatives
- INTEGER named numbers
- ENUMERATED named numbers
- SEQUENCE named component types
- SET named component types
- BIT STRING named bits
- OBJECT IDENTIFIER value component names

The same identifier may be used in different instances of the above types.

---

447 The type referenced in the SelectionType is not a CHOICE

Severity: 4

Explanation: The selection type is used to select one of the CHOICE type alternatives. It cannot be used to select types from other datatypes. See 5.27, “Selection Type” on page 5-23 for the correct syntax.

---

448 CHOICE alternative "*identifier*" is not found

Severity: 4

Explanation: The *identifier* in the selection type does not reference a CHOICE alternative. See 5.27, “Selection Type” on page 5-23 for the correct syntax.

---

449 This tag must be distinct from the tag at line *l*, column *c*

Severity: 4

Explanation: ASN.1 requires that tags be distinct in certain cases. A tag is the combination of a class (UNIVERSAL, APPLICATION, PRIVATE, or context sensitive) and a number. The datatypes which have distinct tag requirements are:

- CHOICE
- SET
- SEQUENCE

For example, the following are all *illegal*:

```
A ::= CHOICE{ a INTEGER, b INTEGER}
B ::= SET{ a INTEGER, b INTEGER}
C ::= SEQUENCE{ a BOOLEAN, b INTEGER OPTIONAL,
                c INTEGER}
D ::= CHOICE{ a E, b F}
E ::= CHOICE{ e1 INTEGER, e2 BOOLEAN}
F ::= CHOICE{ e1 INTEGER, e2 NULL}
```

Since decoding a value requires matching the tags, any case where the same tag could be interpreted to refer to more than one type will cause a decoding failure.

---

450 ANY cannot be used where distinct tags are necessary

Severity: 4

Explanation: ASN.1 requires that tags be distinct in certain cases. A tag is the combination of a class (UNIVERSAL, APPLICATION, PRIVATE, or context sensitive) and a number. The ANY (and ANY DEFINED BY) datatype has an indeterminate tag and so must not be used where distinct tags are required. The datatypes which require distinct tags are:

- CHOICE
- SET
- SEQUENCE

For example, the following are all *illegal*:

```
A ::= CHOICE{ a ANY, b INTEGER}
B ::= SET{ a ANY, b INTEGER}
C ::= SEQUENCE{ a BOOLEAN, b INTEGER OPTIONAL,
                c ANY}
```

Since decoding a value requires matching the tags, any case where the same tag could be interpreted to refer to more than one type will cause a decoding failure.

---

451 Recursive CHOICE reference is illegal

Severity: 4

Explanation: A CHOICE definition includes alternatives which eventually refer back to the CHOICE (the recursion) without tagging. This type of recursion cannot be resolved and so is illegal. For example, this is *illegal*:

```
A ::= CHOICE{x B, y INTEGER}
B ::= CHOICE{s A, t NULL}
```

Not all recursions are illegal. For example, the following is legal:

```
A :&colon= CHOICE{x B, y INTEGER} -- legal
B :&colon= CHOICE{s [0] A, t NULL} -- legal
```

---

452 *YaccMessage*

Severity: 4

Explanation: The compiler encountered a syntax error, ran out of memory, or experienced some other problem which is indicated by *YaccMessage* In the case of a "parse error", the line and column number indicate exactly where the compiler detected an error. More explicit error messages in the same vicinity should explain the nature of the error.

453 This alternative is not found in the CHOICE type at line *l*, column *c*

Severity: 4

Explanation: A CHOICE value was specified that referenced a alternative in a CHOICE defined at line *l*, column *c.*, which was not found. This is usually caused by misspelling the CHOICE alternative name. See -- Heading 'CHOICE' unknown -- for the syntax of CHOICE values.

454 Missing or illegal tag before "*IMPLICIT/EXPLICIT*"

Severity: 4

Explanation: The keywords IMPLICIT and EXPLICIT must be preceded by a valid tag construction -- they cannot appear without the tag construction. A tag is constructed with square brackets with stuff between them. The following is an *illegal* example:

```
A ::= IMPLICIT INTEGER -- illegal
```

This is a *legal* example:

```
B ::= [0] IMPLICIT INTEGER -- legal
```

See 5.28, “Tagged Types” on page 5-23 for the correct syntax.

455 This defined value for the named *number/bit* must refer to a number

Severity: 4

Explanation: A named number or bit is defined by referencing another value which is not a number. For example, the following causes this error message:

```
A ::= INTEGER{a(x), b(3)}
x BOOLEAN ::= TRUE
```

456 Named *number/bit* "*identifier1*" has the same value as "*identifier2*"

Severity: 4

Explanation: The named numbers or bits associated with *identifier1* and *identifier2* have the same value. ASN.1 requires them to be unique. For example, the following syntax will cause this error message:

```
A ::= INTEGER{a(x), b(3)}
x INTEGER ::= 3
```

457 Imported symbol "*symbol*" must be exported from module "*module*"

Severity: 4

Explanation: The symbol is in an IMPORTS list, but it does not appear in the EXPORTS list of the referenced module.

Note that the absence of an EXPORTS clause in a module means *everything* in that module is considered to be exported. If a module has an EXPORTS clause with no symbols, that module is considered to export *nothing*.

458 Illegal assigned object identifier reference for module "*modulename*"

Severity: 4

Explanation: The object identifier value assigned to the module is incorrect. Another error message in the vicinity will more finely pinpoint the error(s).

459 Illegal numeric real value - extra component(s)

Severity: 4

Explanation: The REAL numeric value has too many components; exactly three numbers are required.

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

460 Illegal numeric real value - missing exponent

Severity: 4

Explanation: The REAL numeric value has only two components; exactly three numbers are required. The third component is the exponent.

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

461 Illegal numeric real value - missing base

Severity: 4

Explanation: The REAL numeric value has only one component; exactly three numbers are required. The second component is the base.

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

462 Illegal numeric real value - missing mantissa

Severity: 4

Explanation: The REAL numeric value has zero components; exactly three numbers are required. The first component is the mantissa.

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

463 Illegal numeric real value - the mantissa must be a signed number

Severity: 4

Explanation: All three components of the numeric real number must be numbers. A common mistake is to use a reference to a number rather than the number itself.

```
a REAL ::= {x, 2, 10} -- illegal reference

x INTEGER ::= 42
```

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

464 Illegal numeric real value - the exponent must be a signed number

Severity: 4

Explanation: All three components of the numeric real number must be numbers. A common mistake is to use a reference to a number rather than the number itself.

```
a REAL ::= {12, 2, x} -- illegal reference

x INTEGER ::= 42
```

See 5.39, “Real Number” on page 5-31 for a description of the correct syntax.

465 The EXPORT clause must precede the IMPORTS clause

Severity: 4

Explanation: The order of the IMPORT and EXPORT clauses is fixed by the ASN.1 standard. The IMPORTS clause, if it exists, comes *after* the EXPORTS clause.

See 5.10.3, “Module Body Syntax” on page 5-9 for a description of the correct syntax.

466 Only one *IMPORTS/EXPORTS* clause is allowed per module

Severity: 4

Explanation: The referenced IMPORTS or EXPORTS clause is not the only one of its kind in the module definition. All of the IMPORTED symbols must be collected into one IMPORTS clause, and likewise for EXPORTED symbols.

See 5.10.3, “Module Body Syntax” on page 5-9 for a description of the correct syntax.

467 No ".* asn.1 on" introducer was found

Severity: 4

Explanation: The SCRIPT option was specified and the compiler was not able to find a single ASN.1 section in the source file. Hence no ASN.1 source was parsed. If the file is indeed a mixture of Bookmaster/Script and ASN.1 source, then you must add the ".* asn.1 on" and ".* asn.1 off" lines in the appropriate places. If the file only contains ASN.1 statements, then you should probably not use the SCRIPT option. See 2.1, “Source File” on page 2-1 for more information on the SCRIPT option.

301 This alternative reference is to a CHOICE type defined in another module

Severity: 3

Explanation: It is not good practice to refer to a CHOICE alternative which is defined in another module not in this source file. Since the compiler cannot perform the necessary checks to ensure the reference is correct, you should do so.

302 External reference "*Identifier*" should be in IMPORTS list

Severity: 3

Explanation: It is good practice to identify the imported symbols in the IMPORTS list of the module definition. See 5.10, “Module Definition” on page 5-8 for the correct syntax.

303 Not in a syntactic extension - "%--" is out of context

Severity: 3

Explanation: A termination of a syntactic extension, "%--" was found outside of a syntactic extension, i.e. there is no matching "--%". Check to see that pairs of "--%" and "%--" match correctly. See -- Heading 'SYNEXT' unknown -- for the correct syntax.

304 This tag is made EXPLICIT because it tags *CHOICE/ANY* type

Severity: 3

Explanation: ASN.1 forces this tag to be EXPLICIT even though the default tagging for the module was IMPLICIT. If you intended this to be the case, better practice is to use the EXPLICIT keyword to make this clear. Note that ASN.1 prohibits the IMPLICIT keyword to be applied to CHOICE and ANY types. For example, these are *illegal* definitions:

```
A ::= [0] IMPLICIT CHOICE{ a INTEGER, b NULL }
B ::= [1] IMPLICIT ANY
```

305 A comment inside a syntactic extension is not portable

Severity: 3

Explanation: This compiler allows comments within a syntactic extension. However, since syntactic extensions are not standard ASN.1 notation, other ASN.1 compilers will interpret the comments incorrectly and will cause error messages. Specifically, standard ASN.1 compilers will interpret the syntactic introducer, "--%", as the start of a comment, and when they encounter a "--", they will terminate the comment. Then, your commentary will parsed as though it were not an ASN.1 comment. For example, the following is likely to cause other compilers to interpret Foo as a type reference, which it isn't:

```
--% -- Foo is a very useless name -- %--
```

306 Special character only allowed in syntactic extension

Severity: 3

Explanation: The character at the line and column number is not a standard ASN.1 character and should only be used within a syntactic extension, i.e. between "--%" and "%--". This is a portability warning since other ASN.1 compilers are likely to balk at this character.

307 A syntactic guard start, "--%", inside a syntactic extension is not portable

Severity: 3

Explanation: The syntactic guard start, "--%", was found when another, previous syntactic guard is still in effect. This usually indicates unmatched syntactic guard start/end pairs. In this case, the extraneous syntactic guard start is ignored.

308 This NamedType should have an identifier

Severity: 3

Explanation: The current version of ASN.1 does not require named types to have an identifier, but future versions will. It is good practice to always use identifiers for named types because the some value notations and type notations are made unambiguous if the identifier is used. A named type is a type specification preceded by an optional identifier (which must begin with a lowercase letter). In the following example, "foo INTEGER" is named type, "foo" is the identifier, and "INTEGER" is the type being named:

```
A :&colon= SET{ foo INTEGER, fum BOOLEAN}
```

310 External reference "*symbolname*" should be exported from module "*modulename*"

Severity: 3

Explanation: The external reference, *symbolname*, does not appear in an EXPORTS clause of the referenced module, *modulename*. Although this is not an error, better practice is to explicitly declare which symbols are imported and exported between modules.

201 *feature* **not supported at runtime**

Severity: 2

Explanation: The syntax is valid ASN.1. However, the information related to the *feature* is not carried over into the metatable output. This may or may not affect the way your encoding/decoding application works. You should check to ensure your program does not require this *feature* in this instance.

Appendix B. *H* File Example

The following is an example of an H file created from the `test0` source file (see Figure 4-1 on page 4-2).

```
/* Defined constants for the types and values in module Xyz */

#define Xyz_Y 0
#define Xyz_X 12
#define Xyz_Octstr 24
#define Xyz_Bitstr 32
/* Defined constants for the types and values in module Abc */

#define Abc_H1 0
#define Abc_G1 8
#define Abc_F1 18
#define Abc_E1 28
#define Abc_D1 38
#define Abc_C1 48
#define Abc_B1 58
#define Abc_A1 68
#define Abc_Z 78
#define Abc_Y 88
#define Abc_X 98
#define Abc_W 108
#define Abc_V 118
#define Abc_T 128
#define Abc_S 138
#define Abc_R 148
#define Abc_Q 160
#define Abc_P 168
#define Abc_O 176
#define Abc_N 184
#define Abc_M 227
#define Abc_b 326
#define Abc_a 344
#define Abc_L 362
#define Abc_K 392
#define Abc_H 424
#define Abc_G 450
#define Abc_F 507
#define Abc_E 515
#define Abc_C 548
#define Abc_A 556
#define Abc_D 596
#define Abc_B 604
```

Figure B-1. Example H File

Appendix C. *C* File Example

The following is an example C file output of the ASN1 compiler. It was produced from the `test0` source file (see Figure 4-1 on page 4-2) for OS/2.

```
char Xyz_s[] = {
 /*     0 */    'G','L','O','B','A','L','-','M','O','D','U','L','E','\0',
 /*    14 */    'A','b','c','\0',
 /*    18 */    'B','\0',
 /*    20 */    'D','\0',
 /*    22 */    'I','N','T','E','G','E','R','\0',
 /*    30 */    'a','\0',
 /*    32 */    'E','X','P','L','I','C','I','T','\0',
 /*    41 */    'A','\0',
 /*    43 */    'I','M','P','L','I','C','I','T','\0',
 /*    52 */    'B','O','O','L','E','A','N','\0',
 /*    60 */    'C','\0',
 /*    62 */    'N','U','L','L','\0',
 /*    67 */    'O','C','T','E','T',' ','S','T','R','I','N','G','\0',
 /*    80 */    'E','\0',
 /*    82 */    'R','E','A','L','\0',
 /*    87 */    'F','\0',
 /*    89 */    'f','o','o','1','\0',
 /*    94 */    'f','o','o','2','\0',
 /*    99 */    'n','\0',
 /*   101 */    'o','s','\0',
 /*   104 */    'C','H','O','I','C','E','\0',
 /*   111 */    'G','\0',
 /*   113 */    'B','I','T',' ','S','T','R','I','N','G','\0',
 /*   124 */    'H','\0',
 /*   126 */    'B','i','t','s','t','r','\0',
 /*   133 */    'X','y','z','\0',
 /*   137 */    'S','E','Q','U','E','N','C','E','\0',
 /*   146 */    'K','\0',
 /*   148 */    'S','E','T','\0',
 /*   152 */    'L','\0',
 /*   154 */    'b','\0',
 /*   156 */    'b','1','\0',
 /*   159 */    'b','2','\0',
 /*   162 */    'b','3','\0',
 /*   165 */    'b','o','o','l','\0',
 /*   170 */    'i','n','t','\0',
 /*   174 */    'g','\0',
 /*   176 */    'M','\0',
 /*   178 */    'i','\0',
 /*   180 */    'N','\0',
 /*   182 */    'O','B','J','E','C','T',' ','I','D','E','N','T','I','F','I','E','R','\0',
 /*   200 */    'O','\0',
 /*   202 */    'E','N','U','M','E','R','A','T','E','D','\0',
 /*   213 */    'P','\0',
 /*   215 */    'A','N','Y','\0',
 /*   219 */    'Q','\0',
 /*   221 */    'O','c','t','s','t','r','\0',
 /*   228 */    'R','\0',
 /*   230 */    'N','u','m','e','r','i','c','S','t','r','i','n','g','\0',
 /*   244 */    'S','\0',
 /*   246 */    'P','r','i','n','t','a','b','l','e','S','t','r','i','n','g','\0',
 /*   262 */    'T','\0',
 /*   264 */    'T','e','l','e','t','e','x','S','t','r','i','n','g','\0',
 /*   278 */    'V','\0',
 /*   280 */    'T','6','1','S','t','r','i','n','g','\0',
 /*   290 */    'W','\0',
```

Figure C-1 (Part 1 of 6). Example C File

```
 /*   292 */    'V','i','d','e','o','t','e','x','S','t','r','i','n','g','\0',
 /*   307 */    'X','\0',
 /*   309 */    'V','i','s','i','b','l','e','S','t','r','i','n','g','\0',
 /*   323 */    'Y','\0',
 /*   325 */    'I','S','O','6','4','6','S','t','r','i','n','g','\0',
 /*   338 */    'Z','\0',
 /*   340 */    'I','A','5','S','t','r','i','n','g','\0',
 /*   350 */    'A','1','\0',
 /*   353 */    'G','r','a','p','h','i','c','S','t','r','i','n','g','\0',
 /*   367 */    'B','1','\0',
 /*   370 */    'G','e','n','e','r','a','l','S','t','r','i','n','g','\0',
 /*   384 */    'C','1','\0',
 /*   387 */    'G','e','n','e','r','a','l','i','z','e','d','T','i','m','e','\0',
 /*   403 */    'D','1','\0',
 /*   406 */    'U','T','C','T','i','m','e','\0',
 /*   414 */    'E','1','\0',
 /*   417 */    'E','X','T','E','R','N','A','L','\0',
 /*   426 */    'F','1','\0',
 /*   429 */    'O','b','j','e','c','t','D','e','s','c','r','i','p','t','o','r','\0',
 /*   446 */    'G','1','\0',
 /*   449 */    'H','1','\0',
 /*   452 */    'A','b','c','.','B','\0',
 /*   458 */    'A','b','c','.','D',    '\0'};

mtok Xyz_m[] = {
/* type: Y */
  /* index:    0 */ 2, 0, 67, 1,
  /* index:    4 */ 24, 0, 202, 1, 14, 0, 20, 0,
/* type: X */
  /* index:   12 */ 2, 0, 51, 1,
  /* index:   16 */ 24, 0, 196, 1, 14, 0, 18, 0,
/* type: Octstr */
  /* index:   24 */ 2, 0, 221, 0,
  /* index:   28 */ 10, 0, 67, 0,
/* type: Bitstr */
  /* index:   32 */ 2, 0, 126, 0,
  /* index:   36 */ 9, 0, 113, 0,
  0};

tbldex Xyz_sym[] = {

  0,
  12,
  24,
  32};

struct berimp Xyz_imp[] = {
        {14, 20, NULL, 0}       /* Abc.D */,
        {14, 18, NULL, 0}       /* Abc.B */
        };

struct berexp Xyz_exp[] = {
        {221, 24}       /* Octstr */,
        {126, 32}       /* Bitstr */
        };

struct moddef Xyz_m = {
        &Xyz_s[133],    /* module name addr */
        NULL,   /* ptr to obj id for module */
        Xyz_s,  /* symbol table addr */
        implicit,       /* default tagging */
        Xyz_m,  /* metatable */
        4,      /* number of types and values */
        Xyz_sym,        /* addr of type/val table */
        2,      /* number of imports */
        Xyz_imp,        /* imports */
        2,      /* number of exports */
        Xyz_exp,        /* exports */
        NULL};  /* next moddef structure */
```

Figure C-1 (Part 2 of 6). Example C File

```
mtok Abc_m[] = {
/* type: H1 */
  /* index:    0 */ 2, 0, 193, 1,
  /* index:    4 */ 3, 0, 22, 0,
/* type: G1 */
  /* index:    8 */ 2, 0, 190, 1,
  /* index:   12 */ 25, 0, 173, 1, 7, 0,
/* type: F1 */
  /* index:   18 */ 2, 0, 170, 1,
  /* index:   22 */ 25, 0, 161, 1, 8, 0,
/* type: E1 */
  /* index:   28 */ 2, 0, 158, 1,
  /* index:   32 */ 25, 0, 150, 1, 23, 0,
/* type: D1 */
  /* index:   38 */ 2, 0, 147, 1,
  /* index:   42 */ 25, 0, 131, 1, 24, 0,
/* type: C1 */
  /* index:   48 */ 2, 0, 128, 1,
  /* index:   52 */ 25, 0, 114, 1, 27, 0,
/* type: B1 */
  /* index:   58 */ 2, 0, 111, 1,
  /* index:   62 */ 25, 0, 97, 1, 25, 0,
/* type: A1 */
  /* index:   68 */ 2, 0, 94, 1,
  /* index:   72 */ 25, 0, 84, 1, 22, 0,
/* type: Z */
  /* index:   78 */ 2, 0, 82, 1,
  /* index:   82 */ 25, 0, 69, 1, 26, 0,
/* type: Y */
  /* index:   88 */ 2, 0, 67, 1,
  /* index:   92 */ 25, 0, 53, 1, 26, 0,
/* type: X */
  /* index:   98 */ 2, 0, 51, 1,
  /* index:  102 */ 25, 0, 36, 1, 21, 0,
/* type: W */
  /* index:  108 */ 2, 0, 34, 1,
  /* index:  112 */ 25, 0, 24, 1, 20, 0,
/* type: V */
  /* index:  118 */ 2, 0, 22, 1,
  /* index:  122 */ 25, 0, 8, 1, 20, 0,
/* type: T */
  /* index:  128 */ 2, 0, 6, 1,
  /* index:  132 */ 25, 0, 246, 0, 19, 0,
/* type: S */
  /* index:  138 */ 2, 0, 244, 0,
  /* index:  142 */ 25, 0, 230, 0, 18, 0,
```

Figure C-1 (Part 3 of 6). Example C File

```
/* type: R */
  /* index:  148 */ 2, 0, 228, 0,
  /* index:  152 */ 24, 0, 221, 0, 133, 0, 221, 0,
/* type: Q */
  /* index:  160 */ 2, 0, 219, 0,
  /* index:  164 */ 19, 0, 215, 0,
/* type: P */
  /* index:  168 */ 2, 0, 213, 0,
  /* index:  172 */ 22, 0, 202, 0,
/* type: O */
  /* index:  176 */ 2, 0, 200, 0,
  /* index:  180 */ 20, 0, 182, 0,
/* type: N */
  /* index:  184 */ 2, 0, 180, 0,
  /* index:  188 */ 13, 0, 148, 0, 226, 0,
  /* index:  194 */ 15, 0, 202, 0,
  /* index:  198 */ 3, 0, 178, 0,
  /* index:  202 */ 16, 0, 210, 0,
  /* index:  206 */ 7, 0, 154, 0,
  /* index:  210 */ 15, 0, 218, 0,
  /* index:  214 */ 8, 0, 99, 0,
  /* index:  218 */ 16, 0, 226, 0,
  /* index:  222 */ 9, 0, 159, 0,
  /* index:  226 */ 18,
/* type: M */
  /* index:  227 */ 2, 0, 176, 0,
  /* index:  231 */ 11, 0, 137, 0, 69, 1,
  /* index:  237 */ 15, 0, 245, 0,
  /* index:  241 */ 9, 0, 156, 0,
  /* index:  245 */ 15, 0, 253, 0,
  /* index:  249 */ 9, 0, 159, 0,
  /* index:  253 */ 16, 0, 5, 1,
  /* index:  257 */ 9, 0, 162, 0,
  /* index:  261 */ 16, 0, 13, 1,
  /* index:  265 */ 7, 0, 165, 0,
  /* index:  269 */ 16, 0, 35, 1,
  /* index:  273 */ 3, 0, 170, 0,
  /* index:  277 */ 26, 0, 35, 1,
  /* index:  281 */ 27, 0, 42, 0, 0, 0, 84, 0, 0, 0,
  /* index:  291 */ 17, 0, 59, 1, 69, 1,
  /* index:  297 */ 1, 0, 174, 0, 0, 0, 12, 0, 0, 0, 59, 1,
  /* index:  309 */ 23, 0, 111, 0, 194, 1,
  /* index:  315 */ 42, 0, 208, 1,
  /* index:  319 */ 32, 0, 0, 0, 0, 0,
  /* index:  325 */ 18,
/* value: b */
  /* index:  326 */ 39, 0, 154, 0, 82, 1, 88, 1,
  /* index:  334 */ 3, 0, 22, 0,
  /* index:  338 */ 32, 0, 84, 0, 0, 0,
/* value: a */
  /* index:  344 */ 39, 0, 30, 0, 100, 1, 106, 1,
  /* index:  352 */ 3, 0, 22, 0,
  /* index:  356 */ 32, 0, 42, 0, 0, 0,
```

Figure C-1 (Part 4 of 6). Example C File

```
/* type: L */
  /* index:  362 */ 2, 0, 152, 0,
  /* index:  366 */ 14, 0, 148, 0, 136, 1,
  /* index:  372 */ 28, 0, 130, 1,
  /* index:  376 */ 27, 0, 2, 0, 0, 0, 2, 0, 0, 0,
  /* index:  386 */ 23, 0, 80, 0, 3, 2,
/* type: K */
  /* index:  392 */ 2, 0, 146, 0,
  /* index:  396 */ 12, 0, 137, 0, 168, 1,
  /* index:  402 */ 28, 0, 160, 1,
  /* index:  406 */ 27, 0, 0, 0, 0, 0, 10, 0, 0, 0,
  /* index:  416 */ 24, 0, 126, 0, 133, 0, 126, 0,
/* type: H */
  /* index:  424 */ 2, 0, 124, 0,
  /* index:  428 */ 9, 0, 113, 0,
  /* index:  432 */ 26, 0, 194, 1,
  /* index:  436 */ 28, 0, 194, 1,
  /* index:  440 */ 27, 8, 20, 0, 0, 0, 255, 255, 255, 127,
/* type: G */
  /* index:  450 */ 2, 0, 111, 0,
  /* index:  454 */ 4, 0, 104, 0, 250, 1,
  /* index:  460 */ 5, 0, 226, 1,
  /* index:  464 */ 3, 0, 89, 0,
  /* index:  468 */ 26, 0, 226, 1,
  /* index:  472 */ 27, 14, 1, 0, 0, 128, 255, 255, 255, 127,
  /* index:  482 */ 5, 0, 234, 1,
  /* index:  486 */ 7, 0, 94, 0,
  /* index:  490 */ 5, 0, 242, 1,
  /* index:  494 */ 8, 0, 99, 0,
  /* index:  498 */ 5, 0, 250, 1,
  /* index:  502 */ 10, 0, 101, 0,
  /* index:  506 */ 6,
/* type: F */
  /* index:  507 */ 2, 0, 87, 0,
  /* index:  511 */ 21, 0, 82, 0,
/* type: E */
  /* index:  515 */ 2, 0, 80, 0,
  /* index:  519 */ 10, 0, 67, 0,
  /* index:  523 */ 26, 0, 36, 2,
  /* index:  527 */ 28, 0, 31, 2,
  /* index:  531 */ 32, 0, 50, 0, 0, 0,
  /* index:  537 */ 32, 0, 42, 0, 0, 0,
  /* index:  543 */ 36, 3, 1, 35, 64,
/* type: C */
  /* index:  548 */ 2, 0, 60, 0,
  /* index:  552 */ 7, 0, 52, 0,
/* type: A */
  /* index:  556 */ 2, 0, 41, 0,
  /* index:  560 */ 1, 0, 32, 0, 3, 0, 12, 0, 0, 0, 84, 2,
  /* index:  572 */ 3, 0, 22, 0,
  /* index:  576 */ 26, 0, 84, 2,
  /* index:  580 */ 27, 0, 0, 0, 0, 0, 2, 0, 0, 0,
  /* index:  590 */ 23, 0, 30, 0, 88, 1,
/* type: D */
  /* index:  596 */ 2, 0, 20, 0,
  /* index:  600 */ 8, 0, 62, 0,
/* type: B */
  /* index:  604 */ 2, 0, 18, 0,
  /* index:  608 */ 0, 0, 43, 0, 2, 0, 42, 0, 0, 0, 136, 2,
  /* index:  620 */ 3, 0, 22, 0,
  /* index:  624 */ 26, 0, 136, 2,
  /* index:  628 */ 27, 1, 246, 255, 255, 255, 10, 0, 0, 0,
  /* index:  638 */ 27, 0, 20, 0, 0, 0, 30, 0, 0, 0,
  0};
```

Figure C-1 (Part 5 of 6). Example C File

```
tbldex Abc_sym[] = {

  0,
  8,
  18,
  28,
  38,
  48,
  58,
  68,
  78,
  88,
  98,
  108,
  118,
  128,
  138,
  148,
  160,
  168,
  176,
  184,
  227,
  326,
  344,
  362,
  392,
  424,
  450,
  507,
  515,
  548,
  556,
  596,
  604};

struct berimp Abc_imp[] = {
        {133, 221, NULL, 0}     /* Xyz.Octstr */,
        {133, 126, NULL, 0}     /* Xyz.Bitstr */
        };

struct berexp Abc_exp[] = {
        {20, 596}       /* D */,
        {18, 604}       /* B */
        };

struct moddef Abc_m = {
        &Xyz_s[14],     /* module name addr */
        NULL,   /* ptr to obj id for module */
        Xyz_s,  /* symbol table addr */
        implicit,       /* default tagging */
        Abc_m,  /* metatable */
        33,     /* number of types and values */
        Abc_sym,        /* addr of type/val table */
        2,      /* number of imports */
        Abc_imp,        /* imports */
        2,      /* number of exports */
        Abc_exp,        /* exports */
        NULL};  /* next moddef structure */
```

Figure C-1 (Part 6 of 6). Example C File

Appendix D. Universal Tag Assignments

The Table D-1 shows the assignments for universal datatypes.

Table D-1. Universal Tag Assignments

ASN.1 Type	**Tag Number**
BIT STRING	3
BOOLEAN	1
ENUMERATED	10
EXTERNAL	8
GeneralizedTime	24
GeneralString	27
GraphicString	25
IA5String	22
INTEGER	2
NULL	5
NumericString	18
OBJECT DESCRIPTOR	7
OBJECT IDENTI-FIER	6
OCTET STRING	4
PrintableString	19
REAL	9
SEQUENCE	16
SEQUENCE OF	16
SET	17
SET OF	17
T61String	20
TeletexString	20
UTCTime	23
VideotexString	21
VisibleString	26

List of Abbreviations

Term	Definition
ACSE	Association Control Service Element
ASN.1	Abstract Syntax Notation Number One

Index

LERS Definitions

id	File	Page	References
RUNLIB	RUNLIB	8-1	8-3
RUNLIBG	RUNLIB	8-1	8-1

Syntax Definitions

id	File	Page	References
ASN1SYN	ASN1BER SCRIPT	1	2-4, 2-5, 2-6, 3-2, 3-2, 3-2, 4-4, 4-5, 4-5, 5-2, 5-4, 5-5, 5-7, 5-7, 5-8, 5-8, 5-9, 5-11, 5-12, 5-12, 5-13, 5-14, 5-14, 5-15, 5-15, 5-16, 5-16, 5-17, 5-17, 5-18, 5-18, 5-18, 5-18, 5-19, 5-19, 5-21, 5-21, 5-21, 5-21, 5-22, 5-22, 5-22, 5-23, 5-23, 5-23, 5-23, 5-24, 5-25, 5-25, 5-26, 5-26, 5-31, 5-31, 9-7, 9-7, 9-7, 9-7, 9-7, 9-8, 9-8, 9-8, 9-8, 9-8, 9-9, 9-9, 9-9, 9-9, 9-9, 9-9, 9-10, 9-10, 9-10, 9-10, 9-11, 9-11, 9-11, 9-11, 9-11, 9-11, 9-11, 9-12, 9-12, 9-12, 9-12, 9-12, 9-12, 9-12, 9-13, 9-13, 9-13, 9-13, 9-13, 9-14, 9-14, 9-14, 9-14, 9-15, 9-15, 9-15, 9-15, 9-16, 9-16, 9-16, 9-16, 9-17, 9-17

Figures

id	File	Page	References
DFD1	ASN1BER SCRIPT	2	1-1 1
TEST0S	ASN1BER SCRIPT	4-2	4-1 4-1, B-1, C-1
T0COMP	ASN1BER SCRIPT	4-1	4-2 4-1
T0TBL	ASN1BER SCRIPT	4-3	4-3 4-1
T0ENC1	ASN1BER SCRIPT	4-3	4-4 4-3
T0ENC2	ASN1BER SCRIPT	4-4	4-5 4-4
RESWD	ASN1BER SCRIPT	5-3	5-1 5-2
RESWD2	ASN1BER SCRIPT	5-4	5-2 5-2
SYNGEX	ASN1BER SCRIPT	5-5	5-3 5-2
TSY1	ASN1BER SCRIPT	9-3	9-1 9-2
HFEX	ASN1BER SCRIPT	B-1	B-1
CEX1	ASN1BER SCRIPT	C-2	C-1

Headings

id	File	Page	References
SRCFL	ASN1BER SCRIPT	2-1	2.1, Source File 2-4, 2-4, 2-5, 2-5, 2-5, 2-6, 2-6, 2-6, 5-1, 5-8, A-4, A-14
SCRBKM	ASN1BER SCRIPT	2-1	2.1.1, Script and Bookmaster
ERRFL	ASN1BER SCRIPT	2-2	2.2, Error File 2-4, 2-5, 2-6
TFL	ASN1BER SCRIPT	2-3	2.3, *TBL* file 2-4, 2-5, 2-6
CFL	ASN1BER SCRIPT	3-1	3.1, *C* file 2-4, 3-2, 3-2, 3-3
HFL	ASN1BER SCRIPT	3-1	3.2, H File 3-2, 3-2, 3-3
ASNTAX	ASN1BER SCRIPT	5-1	Chapter 5, ASN.1 Syntax 2-2
CHARS	ASN1BER SCRIPT	5-1	5.1, ASN.1 Character Set
RESWRD	ASN1BER SCRIPT	5-2	5.3, Reserved Words 5-3
SYNGRD	ASN1BER SCRIPT	5-2	5.4, Syntactic Guards 5-2, 5-4, 5-6, 5-12
TYPREF	ASN1BER SCRIPT	5-2	5.5, Type Reference Item 5-4, 5-6, 5-10, A-3
TBLREF	ASN1BER SCRIPT	5-4	5.6, Any Table Reference Item 5-10, 5-10, A-7, A-7
VALREF	ASN1BER SCRIPT	5-4	5.7, Value Reference Item
IDENTF	ASN1BER SCRIPT	5-5	5.8, Identifier Item 5-6, A-3
OTHERIT	ASN1BER SCRIPT	5-6	5.9, Other Items 5-10, A-3, A-3, A-3
NBR	ASN1BER SCRIPT	5-6	5.9.4, Number Item
BSTR	ASN1BER SCRIPT	5-7	5.9.5, Binary String Item
HSTR	ASN1BER SCRIPT	5-7	5.9.6, Hexadecimal String Item
CSTR	ASN1BER SCRIPT	5-8	5.9.7, Character String Item (cstring)
MODULE	ASN1BER SCRIPT	5-8	5.10, Module Definition 5-3, 5-4, A-5, A-5, A-14
DEFTAG	ASN1BER SCRIPT	5-9	5.10.2, Default Tagging
MODBSYN	ASN1BER SCRIPT	5-9	5.10.3, Module Body Syntax A-13, A-13
TYPASS	ASN1BER SCRIPT	5-11	5.11, Type Assignment A-5
VALASS	ASN1BER SCRIPT	5-12	5.12, Value Assignment 5-6, A-5
TBLASS	ASN1BER SCRIPT	5-12	5.13, Any Table Assignment
REFS	ASN1BER SCRIPT	5-13	5.14, Referencing Types, Values and Any Table Definitions 5-13, 5-14
BLTIN	ASN1BER SCRIPT	5-14	5.15, Built-In Types 5-11
BOOL	ASN1BER SCRIPT	5-14	5.16, BOOLEAN
INT	ASN1BER SCRIPT	5-15	5.17, INTEGER A-5, A-6
ENUM	ASN1BER SCRIPT	5-16	5.18, ENUMERATED A-5, A-6
BITSTR	ASN1BER SCRIPT		

		5-17	5.19, BIT STRING A-6, A-6, A-6
SELTYP	ASN1BER SCRIPT	5-23	5.27, Selection Type A-9, A-9
TAG	ASN1BER SCRIPT	5-23	5.28, Tagged Types 5-9, A-7, A-8, A-11
ANY	ASN1BER SCRIPT	5-25	5.29, ANY A-2, A-2, A-2, A-2, A-2
OID	ASN1BER SCRIPT	5-26	5.30, OBJECT IDENTIFIER A-8
REAL	ASN1BER SCRIPT	5-31	5.39, Real Number 5-6, A-6, A-12, A-12, A-12, A-12, A-13, A-13
BER	ASN1BER SCRIPT	6-1	Chapter 6, Basic Encoding Rules
ADDMOD	RUNLIB	8-3	8.3.1, addmodule 8-1
FIRSTMD	RUNLIB	8-9	8.3.10, firstmodule 8-1
FREEPDU	RUNLIB	8-10	8.3.11, freepdu 8-1
INITMOD	RUNLIB	8-11	8.3.13, initmodules 8-1
NEWPDU	RUNLIB	8-13	8.3.17, newpdu 8-1
READDAT	RUNLIB	8-15	8.3.21, readdatfile 8-2
SETUSR	RUNLIB	8-17	8.3.24, setusrencfn 8-2
UNLOAD	RUNLIB	8-18	8.3.26, unloadmodule 8-1
NBRVAL	ASN1BER SCRIPT	9-15	9.9.1, Number 9-17
SYNEXT	?	?	? A-7, A-14
TBLDEF	?	?	? A-7
CHOICE	?	?	? A-11
HEX	ASN1BER SCRIPT	B-1	Appendix B, *H* File Example
CEX	ASN1BER SCRIPT	C-1	Appendix C, *C* File Example

Footnotes

id	File	Page	References
SBSQ1	ASN1BER SCRIPT	9-13	1 9-13

Tables

id	File	Page	References
TAGS	ASN1BER SCRIPT	D-1	D-1 D-1

Processing Options

Runtime values:

Document fileid	ASN1BER SCRIPT
Document type	USERDOC
Document style	DEFAULT
Profile	EDFPRF40
Service Level	0021
SCRIPT/VS Release	4.0.0
Date	94.01.17
Time	08:52:58
Device	3820A
Number of Passes	2
Index	YES
SYSVAR G	INLINE
SYSVAR S	1
SYSVAR X	XONLY

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	YES
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)
DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	YES
Figure/table number separation	YES
Folio-by-chapter	YES
Head 0 body text	(none)
Head 1 body text	Chapter
Head 1 appendix text	Appendix
Hyphenation	YES
Justification	NO
Language	ENGL
Keyboard	395
Layout	1
Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	.,
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	3
Table list on new page	YES
Title page (draft) alignment	CENTER
Write cross-reference file	(none)

Imbed Trace

DSMBEG323I STARTING PASS 2 OF 2.
+++EDF248W Page check: document requires more passes or extended cross-reference to reso
lve correctly. (Page A-16 File: ASN1BER SCRIPT)
DSMMOM397I '.EDFPGCK' WAS IMBEDDED AT LINE 320 OF '.EDFHEAD1'
DSMMOM397I '.EDFHEAD1' WAS IMBEDDED AT LINE 5831 OF 'ASN1BER'
+++EDF258W Cross references were not resolved. Check cross-reference listing to find proble
ms. (Page /XRL/5 File: ASN1BER SCRIPT)
DSMMOM397I '.EDF#END' WAS IMBEDDED AT LINE 187 OF 'BOOKPROF'