ASN.1 Complete

John Larmouth

MK

Copyrighted material

ASN.1 Complete

by

Professor John Larmouth

Morgan
Kaufmann

ACADEMIC PRESS, A Harcourt Science and Technology Company

San Diego San Francisco New York Boston
London Sydney Tokyo

Copyrighted material

This book is printed on acid-free paper. ♾

Copyright © 2000 by OSS Nokalva

All rights reserved.
No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Requests for permission to make copies of any part of the work should be mailed to the following address: Permissions Department, Harcourt Inc., 6277 Sea Harbor Drive, Orlando, Florida 32887-6777.

Academic Press
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495
http://www.apnet.com

Academic Press
24-28 Oval Road, London NW1 7DX United Kingdom
http://www.hbuk.co.uk/ap/

Morgan Kaufmann Publishers
A Harcourt Science and Technology Company
340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205
http://www.mkp.com

Library of Congress Catalog Number: 99-66138
International Standard Book Number: 0-12-233435-3

Printed in the United States of America
99 00 01 02 03 IP 9 8 7 6 5 4 3 2 1

Copyrighted material

Contents

Foreword xxi

Introduction xxiii

1 The Global Communications Infrastructure xxiii
2 What Exactly Is ASN.1? xxiv
3 The Development Process with ASN.1 xxvi
4 Structure of the Text xxvii

SECTION I ASN.1 OVERVIEW 1

Chapter 1 Specification of Protocols 3

1.1 What Is a Protocol? 3
1.2 Protocol Specification: Some Basic Concepts 6
1.2.1 Layering and Protocol "Holes" 6
1.2.2 Early Developments of Layering 8
1.2.3 The Disadvantages of Layering: Keep It Simple! 9
1.2.4 Extensibility 9
1.2.5 Abstract and Transfer Syntax 12
1.2.6 Command Line or Statement-Based Approaches 13
1.2.7 Use of an Interface Definition Language 14
1.3 More on Abstract and Transfer Syntaxes 15
1.3.1 Abstract Values and Types 15
1.3.2 Encoding Abstract Values 16
1.4 Evaluative Discussion 18
1.4.1 There Are Many Ways of Skinning a Cat: Does It Matter? 18
1.4.2 Early Work with Multiple Transfer Syntaxes
1.4.3 Benefits 19
1.4.3.1 Efficient Use of Local Representations 19
1.4.3.2 Improved Representations over Time 20
1.4.3.3 Reuse of Encoding Schemes 20

Copyrighted material

1.4.3.4 Structuring of Code 21
1.4.3.5 Reuse of Code and Common Tools 22
1.4.3.6 Testing and Line Monitor Tools 23
1.4.3.7 Multiple Documents Require "Glue" 23
1.4.3.8 The "Tools" Business 23
1.5 Protocol Specification and Implementation: A Series of Case Studies 24
1.5.1 Octet Sequences and Fields within Octets 24
1.5.2 The TLV Approach 24
1.5.3 The EDIFACT Graphical Syntax 26
1.5.4 Use of BNF to Specify a Character-Based Syntax 27
1.5.5 Specification and Implementation Using ASN.1: Early 1980s 29
1.5.6 Specification and Implementation Using ASN.1: 1990s 31

Chapter 2 Introduction to ASN.1 34
2.1 Introduction 34
2.2 The Example 35
2.2.1 The Top-Level Type 36
2.2.2 Bold Is What Matters! 36
2.2.3 Names in Italics Are Used to Tie Things Together 37
2.2.4 Names in Normal Font Are the Names of Fields/Elements/Items 37
2.2.5 Back to the Example! 38
2.2.6 The BranchIdentification Type 41
2.2.7 Those Tags 43
2.3 Getting Rid of the Different Fonts 45
2.4 Tying Up Some Loose Ends 46
2.4.1 Summary of Type and Value Assignments 46
2.4.2 The Form of Names 47
2.4.3 Layout and Comment 47
2.5 So What Else Do You Need to Know? 49

Chapter 3 Structuring an ASN.1 Specification 50
3.1 An Example 51
3.2 Publication Style for ASN.1 Specifications 52
3.2.1 Use of Line Numbers 53
3.2.2 Duplicating the ASN.1 Text 54
3.2.3 Providing Machine-Readable Copy 55

Copyrighted material

3.3 Returning to the Module Header! 56
3.3.1 Syntactic Discussion 56
3.3.2 The Tagging Environment 58
3.3.2.1 An Environment of Explicit Tagging 60
3.3.2.2 An Environment of Implicit Tagging 60
3.3.2.3 An Environment of Automatic Tagging 61
3.3.3 The Extensibility Environment 61
3.4 Exports/Imports Statements 64
3.5 Refining Our Structure 67
3.6 Complete Specifications 71
3.7 Conclusion 72

Chapter 4 The Basic Data Types and Construction Mechanisms: Closure 73
4.1 Illustration by Example 74
4.2 Discussion of the Built-In Types 75
4.2.1 The BOOLEAN Type 75
4.2.2 The INTEGER Type 75
4.2.3 The ENUMERATED Type 78
4.2.4 The REAL Type 79
4.2.5 The BIT STRING Type 81
4.2.5.1 Formal/Advanced Discussion 83
4.2.6 The OCTET STRING Type 86
4.2.7 The NULL Type 86
4.2.8 Some Character String Types 87
4.2.9 The OBJECT IDENTIFIER Type 88
4.2.10 The ObjectDescriptor Type 89
4.2.11 The Two ASN.1 Date/Time Types 90
4.3 Additional Notational Constructs 93
4.3.1 The Selection-Type Notation 93
4.3.2 The COMPONENTS OF Notation 94
4.3.3 SEQUENCE or SET? 96
4.3.4 SEQUENCE, SET, and CHOICE (Etc.) Value-Notation 97
4.4 What Else is in X.680/ISO 8824-1? 98

Chapter 5 Reference to More Complex Areas 100
5.1 Object Identifiers 101
5.2 Character String Types 101
5.3 Subtyping 104

Copyrighted material

5.4 Tagging 106
5.5 Extensibility, Exceptions, and Version Brackets 107
5.6 Hole Types 108
5.7 Macros 109
5.8 Information Object Classes and Objects and Object Sets 110
5.9 Other Types of Constraint 111
5.10 Parameterization 112
5.11 The ASN.1 Semantic Model 112
5.12 Conclusion 113

Chapter 6 Using an ASN.1 Compiler 114
6.1 The Route to an Implementation 115
6.2 What is an ASN.1 Compiler? 115
6.3 The Overall Features of an ASN.1-Compiler-Tool 118
6.4 Use of a Simple Library of Encode/Decode Routines 118
6.4.1 Encoding 119
6.4.2 Decoding 121
6.5 Using an ASN.1-Compiler-Tool 121
6.5.1 Basic Considerations 121
6.5.2 What Do Tool Designers Have To Decide? 122
6.5.3 The Mapping to a Programming-Language Data Structure 123
6.5.4 Memory and CPU Tradeoffs at Run-Time 124
6.5.5 Control of a Tool 125
6.6 Use of the "OSS ASN.1 Tools" Product 126
6.7 What Makes One ASN.1-Compiler-Tool Better than Another? 128
6.8 Conclusion 130

Chapter 7 Management and Design Issues for ASN.1 Specification and Implementation 131
7.1 Global Issues for Management Decisions 132
7.1.1 Specification 132
7.1.1.1 To Use ASN.1 or Not! 132
7.1.1.2 To Copy or Not? 132
7.1.2 Implementation: Setting the Budget 133
7.1.2.1 Getting the Specs 133
7.1.2.2 Training Courses, Tutorials, and Consultants 135
7.1.3 Implementation Platform and Tools 135

Copyrighted material

7.2 Issues for Specifiers 136
7.2.1 Guiding Principles 136
7.2.2 Decisions on Style 137
7.2.3 Your Top-Level Type 138
7.2.4 Integer Sizes and Bounds 138
7.2.5 Extensibility Issues 140
7.2.6 Exception Handling 141
7.2.6.1 The Requirement 141
7.2.6.2 Common Forms of Exception Handling 141
7.2.6.2.1 SEQUENCE and SET 141
7.2.6.2.2 CHOICE 141
7.2.6.2.3 INTEGER and ENUMERATED 142
7.2.6.2.4 Extensible Strings 142
7.2.6.2.5 Extensible Bounds on SET OF and SEQUENCE OF 143
7.2.6.2.6 Use of Extensible Object Sets in Constraints 143
7.2.6.2.7 Summary 144
7.2.6.3 ASN.1-Specified Default Exception Handling 144
7.2.6.4 Use of the Formal Exception Specification Notation 145
7.2.7 Parameterization Issues 146
7.2.8 Unconstrained Open Types 147
7.2.9 Tagging Issues 147
7.2.10 Keeping It Simple 148
7.3 Issues for Implementors 149
7.3.1 Guiding Principles 149
7.3.2 Know Your Tool 150
7.3.3 Sizes of Integers 150
7.3.4 Ambiguities and Implementation-Dependencies in Specifications 151
7.3.5 Corrigenda 152
7.3.6 Extensibility and Exception Handling 152
7.3.7 Care with Hand Encodings 152
7.3.8 Mailing Lists 153
7.3.9 Good Engineering: Version 2 **Will** Come! 153
7.4 Conclusion 154

Copyrighted material

SECTION II FURTHER DETAILS 155

Chapter 8 The Object Identifier Type 157
8.1 Introduction 157
8.2 The Object Identifier Tree 158
8.3 Information Objects 160
8.4 Value Notation 162
8.5 Uses of the Object Identifier Type 163

Chapter 9 The Character String Types 165
9.1 Introduction 165
9.2 NumericString 166
9.3 PrintableString 167
9.4 VisibleString (ISO646String) 167
9.5 IA5String 168
9.6 TeletexString (T61String) 168
9.7 VideotexString 169
9.8 GraphicString 169
9.9 GeneralString 169
9.10 UniversalString 169
9.11 BMPString 170
9.12 UTF8String 170
9.13 Recommended Character String Types 171
9.14 Value Notation for Character String Types 172
9.15 The ASN.1-CHARACTER-MODULE 175
9.16 Conclusion 175

Chapter 10 Subtyping 177
10.1 Introduction 177
10.2 Basic Concepts and Set Arithmetic 178
10.3 Single Value Subtyping 181
10.4 Value Range Subtyping 181
10.5 Permitted Alphabet Constraints 182
10.6 Size Constraints 184
10.7 Contained Subtype Constraints 186
10.8 Inner Subtyping 187
10.8.1 Introduction 187
10.8.2 Subsetting Wineco-Protocol 189
10.8.3 Inner Subtyping of an Array 192
10.9 Conclusion 192

Copyrighted material

15.5 First Attempts at PER—Start with BER and Remove Redundant Octets 286
15.6 Some of the Principles of PER 288
15.6.1 Breaking Out of the BER Straightjacket 288
15.6.2 How to Cope with Other Problems that a "T" Solves? 289
15.6.3 Do We Still Need T and L for SEQUENCE and SET Headers? 291
15.6.4 Aligned and Unaligned PER 292
15.7 Extensibility—You Have to Have It! 293
15.8 What More Do You Need to Know About PER? 294
15.9 Experience with PER 295
15.10 Distinguished and Canonical Encoding Rules 297
15.11 Conclusion 298

Chapter 16 The Basic Encoding Rules 300
16.1 Introduction 300
16.2 General Issues 301
16.2.1 Notation for Bit Numbers and Diagrams 301
16.2.2 The Identifier Octets 302
16.2.3 The Length Octets 304
16.2.3.1 The Short Form 304
16.2.3.2 The Long Form 304
16.2.3.3 The Indefinite Form 306
16.2.3.4 Discussion of Length Variants 307
16.3 Encodings of the V Part of the Main Types 308
16.3.1 Encoding a NULL Value 308
16.3.2 Encoding a BOOLEAN Value 309
16.3.3 Encoding an INTEGER Value 309
16.3.4 Encoding an ENUMERATED Value 310
16.3.5 Encoding a REAL Value 310
16.3.5.1 Encoding Base 10 Values 311
16.3.5.2 Encoding Base 2 Values 312
16.3.5.3 Encoding the Special Real Values 315
16.3.6 Encoding an OCTET STRING Value 315
16.3.7 Encoding a BIT STRING Value 316
16.3.8 Encoding Values of Tagged Types 317
16.3.9 Encoding Values of CHOICE Types 318
16.3.10 Encoding SEQUENCE OF Values 319

Copyrighted material

16.3.11 Encoding SET OF Values 319
16.3.12 Encoding SEQUENCE and SET Values 320
16.3.13 Handling of OPTIONAL and DEFAULT Elements in Sequence and Set 321
16.3.14 Encoding OBJECT IDENTIFIER Values 322
16.3.15 Encoding Character String Values 325
16.3.16 Encoding Values of the Time Types 327
16.4 Encodings for More Complex Constructions 328
16.4.1 Open Types 328
16.4.2 The Embedded PDV Type and the External Type 328
16.4.3 The INSTANCE OF Type 329
16.4.4 The CHARACTER STRING Type 329
16.5 Conclusion 329

Chapter 17 The Packed Encoding Rules 330
17.1 Introduction 331
17.2 Structure of a PER Encoding 332
17.2.1 General Form 332
17.2.2 Partial Octet Alignment and PER Variants 333
17.2.3 Canonical Encodings 333
17.2.4 The Outer-Level Complete Encoding 334
17.3 Encoding Values of Extensible Types 335
17.4 PER-Visible Constraints 338
17.4.1 The Concept 338
17.4.2 The Effect of Variable Parameters 339
17.4.3 Character Strings with Variable Length Encodings 339
17.4.4 Now Let Us Get Complicated! 340
17.5 Encoding INTEGERs—Preparatory Discussion 342
17.6 Effective Size and Alphabet Constraints 344
17.6.1 Statement of the Problem 344
17.6.2 Effective Size Constraint 345
17.6.3 Effective Alphabet Constraint 345
17.7 Canonical Order of Tags 346
17.8 Encoding an Unbounded Count 347
17.8.1 The Three Forms of Length Encoding 347
17.8.2 Encoding "Normally Small" Values 349
17.8.3 Comments on Encodings of Unbounded Counts 350
17.9 Encoding the OPTIONAL Bit-Map and the CHOICE Index 351

Copyrighted material

17.9.1 The OPTIONAL Bit-Map 351
17.9.2 The CHOICE Index 352
17.10 Encoding NULL and BOOLEAN Values 353
17.11 Encoding INTEGER Values 353
17.11.1 Unconstrained Integer Types 353
17.11.2 Semiconstrained Integer Types 354
17.11.3 Constrained Integer Types 354
17.11.4 And if the Constraint on the Integer Is Extensible? 357
17.12 Encoding ENUMERATED Values 358
17.13 Encoding Length Determinants of Strings, etc. 359
17.14 Encoding Character String Values 361
17.14.1 Bits per Character 361
17.14.2 Padding Bits 364
17.14.3 Extensible Character String Types 364
17.15 Encoding SEQUENCE and SET Values 365
17.15.1 Encoding DEFAULT Values 366
17.15.2 Encoding Extension Additions 366
17.16 Encoding CHOICE Values 369
17.17 Encoding SEQUENCE OF and SET OF Values 370
17.18 Encoding REAL and OBJECT IDENTIFIER Values 370
17.19 Encoding an Open Type 371
17.20 Encoding of the Remaining Types 372
17.21 Conclusion 373

Chapter 18 Other ASN.1-Related Encoding Rules 375
18.1 Why Do People Suggest New Encoding Rules? 376
18.2 LWER—Light-Weight Encoding Rules 377
18.2.1 The LWER Approach 377
18.2.2 The Way to Proceed Was Agreed 378
18.2.3 Problems, Problems, Problems 378
18.2.4 The Demise of LWER 380
18.3 MBER—Minimum Bit Encoding Rules 381
18.4 OER—Octet Encoding Rules 381
18.5 Extended Mark-up Language (XML) Encoding Rules (XER) 383
18.6 Building Automation Committee Network (BACnet) Encoding Rules (BACnetER) 383
18.7 Encoding Control Specifications 384

Copyrighted material

SECTION IV HISTORY AND APPLICATIONS **385**

Chapter 19 The Development of ASN.1 **387**

19.1 People 388
19.2 Going Round in Circles? 389
19.3 Who Produces Standards? 391
19.4 The Numbers Game 393
19.5 The Early Years—X.409 and All That 394
19.5.1 Drafts Are Exchanged and the Name ASN.1 Is Assigned 394
19.5.2 Splitting BER from the Notation 395
19.5.3 When Are Changes Technical Changes? 396
19.5.4 The Near-Demise of ASN.1—OPERATION and ERROR 397
19.6 Organization and Reorganization! 399
19.7 The Tool Vendors 400
19.8 Object Identifiers 401
19.8.1 Long or Short, Human or Computer Friendly, That Is the Question 401
19.8.2 Where Should the Object Identifier Tree be Defined? 403
19.8.3 The Battle for Top-Level Arcs and the Introduction of RELATIVE-OIDs 404
19.9 The REAL Type 405
19.10 Character String Types: Let's Try to Keep it Short! 406
19.10.1 From the Beginning to ASCII 406
19.10.2 The Emergence of the International Register of Character Sets 407
19.10.3 The Development Of ISO 8859 409
19.10.4 The Emergence of ISO 10646 and Unicode 409
19.10.4.1 The Four-Dimensional Architecture 409
19.10.4.2 Enter Unicode 411
19.10.4.3 The Final Compromise 412
19.10.5 And the Impact of All this on ASN.1? 413
19.11 ANY, Macros, and Information Objects—Hard to Keep That Short (Even the Heading Has Gone to Two Lines)! 416
19.12 The ASN.1 (1990) Controversy 419
19.13 The Emergence of PER 420

Copyrighted material

19.13.1 The First Attempt—PER-2 420
19.13.2 The Second Attempt—PER-1 424
19.13.3 And Eventually We Get Real-Per 424
19.14 DER and CER 425
19.15 Semantic Models and All That—ASN.1 in the Late 1990s 427
19.16 What Got Away? 428

Chapter 20 Applications of ASN.1 430

20.1 Introduction 430
20.2 The Origins in X.400 432
20.3 The Move into Open Systems Interconnection (OSI) and ISO 433
20.4 Use within the Protocol Testing Community 434
20.5 Use within the Integrated Services Digital Network (ISDN) 435
20.6 Use in ITU-T and Multimedia Standards 435
20.7 Use in European and American Standardization Groups 436
20.8 Use for Managing Computer-Controlled Systems 438
20.9 Use in PKCS and PKIX and SET and Other Security-Related Protocols 439
20.10 Use in Other Internet Specifications 441
20.11 Use in Major Corporate Enterprises and Agencies 441
20.12 Conclusion 442

APPENDICES 443

1 The Wineco Protocol Scenario 445
2 The Full Protocol for Wineco 447
3 Compiler Output for C Support for the Wineco Protocol 451
4 Compiler Output for Java Support for the Wineco Protocol 453
5 ASN.1 Resources via the Web 465

Index 467

Copyrighted material

Section III (Encodings) and indeed of the actual ITU-T Recommendations/ISO Standards for ASN.1.

- **Students on courses covering protocol specification techniques**: Undergraduate and postgraduate courses aiming to give their students an understanding of the abstract syntax approach to protocol specification (and perhaps of ASN.1 itself) should place the early parts of Section I (ASN.1 Overview) and some of Section IV (History and Applications) on the reading list for the course.
- **The intellectually curious**: Perhaps this group will read the whole text from front to back and find it interesting and stimulating! Attempts have been made wherever possible to keep the text light and readable—go to it!

There is an electronic version of this text available, and a list of further ASN.1-related resources, at the URL given in Appendix 5. **And importantly, errata sheets will be provided at this site for downloading.**

The examples have all been verified using the "OSS ASN.1 Tools" package produced and marketed by Open Systems Solutions (OSS), a U.S. company that has (since 1986) developed and marketed tools to assist in the implementation of protocols defined using ASN.1. I am grateful to OSS for much support in the production of this book, and for the provision of their tool for this purpose. While OSS has given support and encouragement in many forms, and has provided a number of reviewers of the text who have made very valued comments on early drafts, the views expressed in this text are those of the author alone.

John Larmouth
(«hyperlink mailto:j.larmouth@salford.ac.uk »)
May 1999

Copyrighted material

Introduction

Summary

This introduction

- describes the problem ASN.1 addresses,
- briefly says what ASN.1 is, and
- explains why it is useful.

1 The Global Communications Infrastructure

We are in a period of rapid advance in the collaboration of computer systems to perform a wider range of activity than ever before. Traditional computer communications to support human-driven remote logon, e-mail, file-transfer, and latterly the World Wide Web (WWW) are being supplemented by new applications requiring increasingly complex exchanges of information both between computer systems and between appliances with embedded computer chips.

Some of these exchanges of information continue to be human-initiated, such as bidding at auctions, money wallet transfers, electronic transactions, voting support, or interactive video. Others are designed for automatic and autonomous computer-to-computer communication in support of such diverse activities as cellular telephones (and other telephony applications), meter reading, pollution recording, air traffic control, control of power distribution, and applications in the home for control of appliances.

In all cases there is a requirement for the detailed specification of the exchanges the computers are to perform, and for the implementation of software to support those exchanges.

The most basic support for many of these exchanges today is provided by the use of TCP/IP and the Internet, but other carrier protocols are still in use, particularly in the telecommunications area. However, the specification of the data formats for messages that are to be passed using TCP (or other carriers) requires the design and clear specification of **application protocols,** followed by (or in parallel with) implementation of those protocols.

Copyrighted material

For communication to be possible between applications and devices produced by different vendors, standards are needed for these application protocols. The standards may be produced by recognized international bodies such as the International Telecommunications Union Telecommunications Standards Sector (ITU-T), the International Standards Organization (ISO), or the Internet Engineering Task Force (IETF), or by industrial associations or collaborative groups and consortia such as the International Civil Aviation Organization (ICAO), the Open Management Group (OMG) or the Secure Electronic Transactions (SET) consortium, or by individual multinational organizations such as Reuters or IBM.

These different groups have various approaches to the task of specifying the communications standards, but in many cases ASN.1 plays a key role by enabling

- rapid and precise specification of computer exchanges by a standardization body, and
- easy and bug-free implementation of the resulting standard by those producing products to support the application.

In a number of industrial sectors, but particularly in the telecommunications sector, in security-related exchanges, and in multimedia exchanges, ASN.1 is the dominant means of specifying application protocols. (The only other major contender is the character-based approach often used by IETF, but which is less suitable for complex structures, and which usually produces a much less compact set of encodings.) A description of some of the applications where ASN.1 has been used as the specification language is given in Chapter 20.

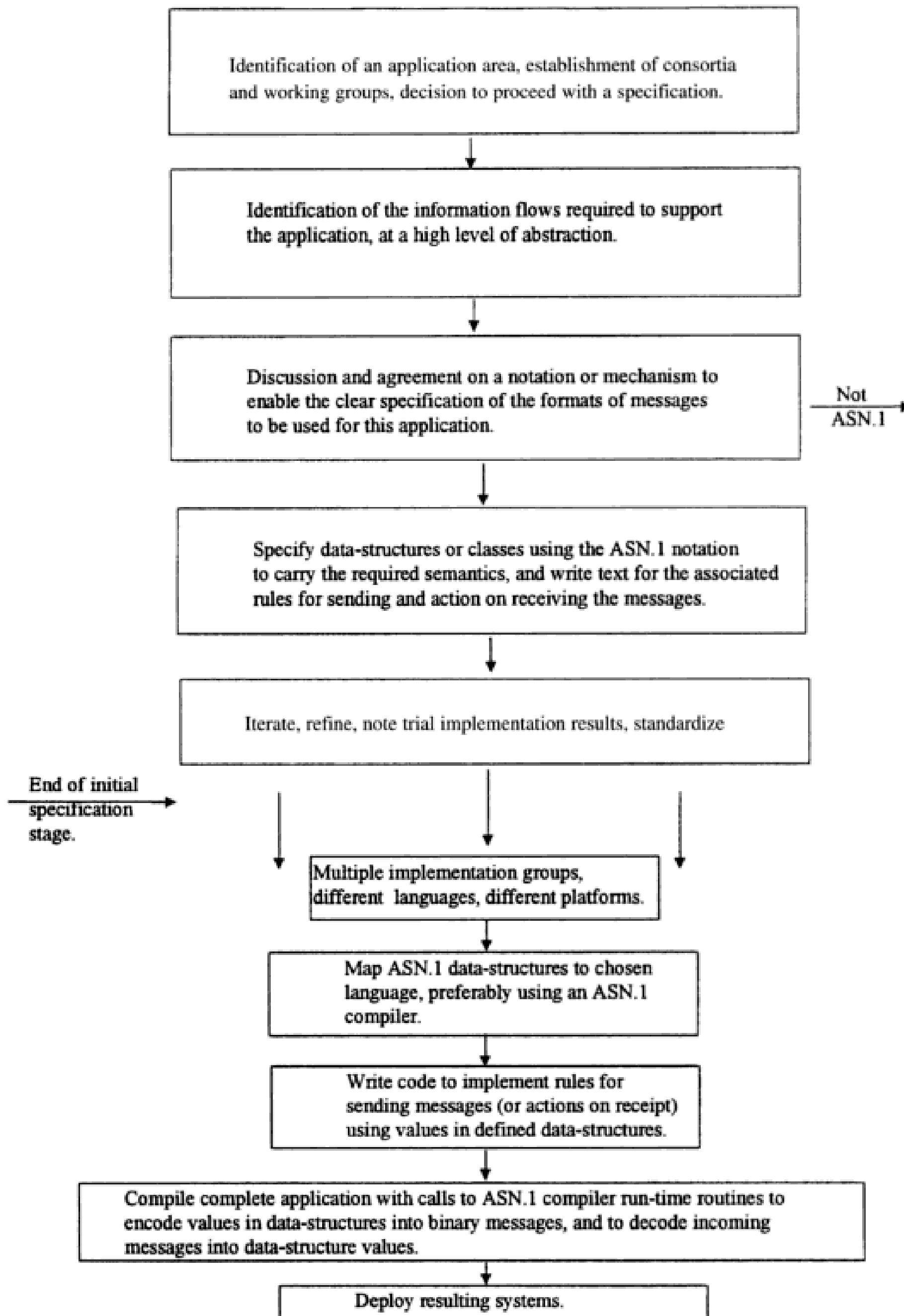
2 What Exactly Is ASN.1?

The term "TCP/IP" can be used to describe two protocol specifications (Transmission Control Protocol—TCP, and Internet Protocol—IP), or more broadly to describe the complete set of protocols and supporting software that are based around TCP/IP. Similarly, the term "ASN.1" can be used narrowly to describe a notation or language called "Abstract Syntax Notation One", or can be used more broadly to describe the notation, the associated encoding rules, and the software tools that assist in its use.

The things that make ASN.1 important and unique include the following:

- It is an internationally standardized, vendor-independent, platform-independent, and language-independent notation for specifying data-structures at a high level of abstraction. (The notation is described in Sections I and II.)

Copyrighted material



The development process with ASN.1.

Copyrighted material

- It is supported by rules that determine the precise bit-patterns (again platform-independent and language-independent) to represent values of these data-structures when they have to be transferred over a computer network, using encodings that are not unnecessarily verbose. (The encoding rules are described in Section III.)
- It is supported by tools available for most platforms and several programming languages that map the ASN.1 notation into data-structure definitions in a computer programming language of choice, and which support the automatic conversion between values of those data-structures in memory and the defined bit-patterns for transfer over a communications line. (The tools are described in Chapter 6.)

There are a number of other subtle features of ASN.1 that are important and are discussed later in this text. Some of these are

- It addresses the problem of, and provides support for, interworking between deployed "version-1" systems and "version-2" systems that are designed and deployed many years apart. (This is called "extensibility".)
- It provides mechanisms to enable partial or generic specification by one standards group, with other standards groups developing (perhaps in very different ways) specific specifications.
- It recognizes the potential for interworking problems between large systems capable of handling long strings, large integer values, large iterative structures, and small systems that may have a lesser capability.
- It provides a range of data-structures that is generally much richer than that of normal programming languages, such as the size of integers, naming structures, and character string types. This enables precision in the specification of the range of values that need to be transferred, and hence production of more optimal encodings.

3 The Development Process with ASN.1

The flow diagram on page xxv illustrates the development process from inception to deployment of initial systems.

(But it must be remembered that this process is frequently an iterative one, with both early revisions by the standardization group to "get it right" and with more substantial revisions some years later when a "version-2" standard is produced.)

Copyrighted material

SECTION I

ASN.1 Overview

Copyrighted material

Specification of Protocols

(Or: Simply Saying Simply What Has To Be Said!)

Summary

This chapter

- introduces the concept of a "protocol" and its specification,
- provides an early introduction to the concepts of
  - layering,
  - extensibility,
  - abstract and transfer syntaxes,
- discusses means of protocol specification, and
- describes common problems that arise in designing specification mechanisms and notations.

(Readers involved in protocol specification should be familiar with much of the early "concepts" material in this chapter, but may find that it provides a new and perhaps illuminating perspective on some of the things they have been trying to do.)

1.1 What Is a Protocol?

A computer protocol can be defined as

> A well-defined set of **messages** (bit-patterns or—increasingly today—octet strings), each of which carries a defined meaning (**semantics**), together with the **rules** governing when a particular message can be sent.

However, a protocol rarely stands alone. Rather, it is commonly part of a "protocol stack", in which several separate specifications work together to determine the complete message emitted by a sender, with some parts of that message destined for action by intermediate (switching) nodes, and some parts intended for the remote end system.

Copyrighted material

In this "layered" protocol technique

- One specification determines the form and meaning of the outer part of the message, with a "hole" in the middle. It provides a "carrier service" (or just "service") to convey any material that is placed in this "hole".

> **What Is a Protocol?**
>
> A well-defined set of messages, each of which carries a defined meaning, and,
>
> the rules governing when a particular message can be sent, and
>
> explicit assumptions about the nature of the service used to transfer the messages, which themselves either support a single-end application or provide a richer carrier service.

- A second specification defines the contents of the "hole", perhaps leaving a further hole for another layer of specification, and so on.

Figure 1.1 illustrates a TCP/IP stack, where real networks provide the basic carrier mechanism, with the IP protocol carried in the "hole" they provide, and with IP acting as a carrier for TCP (or the less well-known User Datagram Protocol—UDP), forming another protocol layer, and

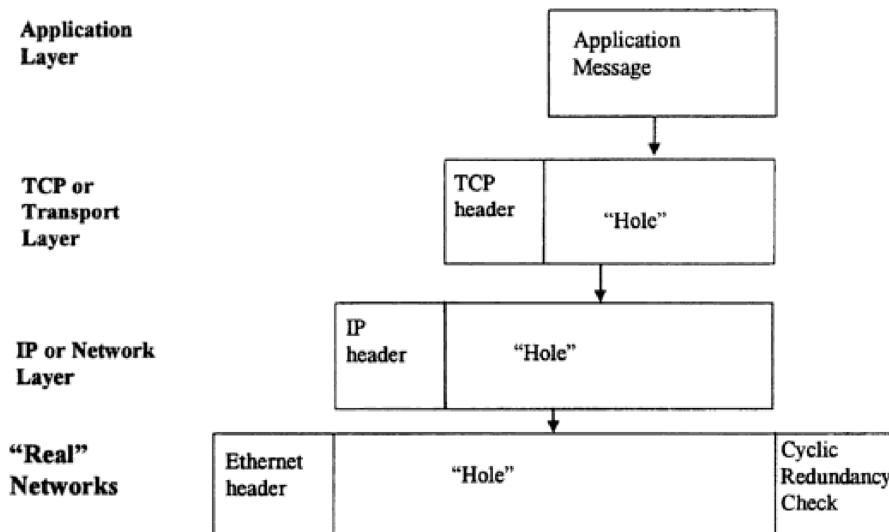


Figure 1.1 Sample protocol stack—TCP/IP.

Copyrighted material

with a (typically for TCP/IP) monolithic application layer—a single specification completing the final "hole".

The precise nature of the "service" provided by a lower layer—lossy, secure, reliable—and of any parameters controlling that service, need to be known before the next layer up can make appropriate use of that service.

We usually refer to each of these individual specification layers as "a protocol", and hence we can enhance our definition.

Note that in Figure 1.1, the "hole" provided by the IP carrier can contain either a TCP message or a UDP message—two very different protocols with different properties (and themselves providing a further carrier service). Thus one of the advantages of "layering" is in reusability of the carrier service to support a wide range of higher level protocols, many perhaps that were never thought of when the lower-layer protocols were developed.

When multiple different protocols can occupy a hole in the layer below (or provide carrier services for the layer above), this is frequently illustrated by the layering diagram shown in Figure 1.2.

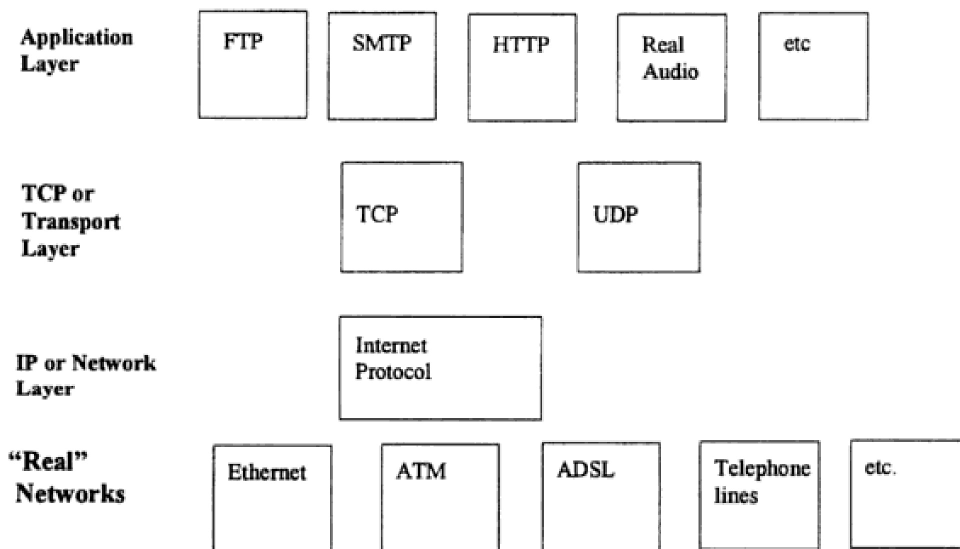


Figure 1.2 Layered protocols—TCP/IP.

Copyrighted material

1.2 Protocol Specification: Some Basic Concepts

Protocols can be (and historically have been) specified in many ways. One fundamental distinction is between *character-based specification* vs. *binary-based specification*.

Character-based specification	The "protocol" is defined as a series of lines of ASCII encoded text.
Binary-based specification	The "protocol" is defined as a string of octets or of bits.

For binary-based specification, approaches vary from various picture-based methods to use of a separately defined notation with associated application-independent encoding rules.

The latter is called the "**abstract syntax**" approach. This is the approach taken with ASN.1. It has the advantage that it enables designers to produce specifications without undue concern with the encoding issues, and also permits application-independent tools to be provided to support the easy implementation of protocols specified in this way. Moreover, because application-specific implementation code is independent of encoding code, it makes it easy to migrate to improved encodings as they are developed.

1.2.1 Layering and Protocol "Holes"

The layering concept is perhaps most commonly associated with the International Standards Organization (ISO) and International Telecommunications Union (ITU) "architecture" or "7-layer model" for Open Systems Interconnection; (OSI) shown in Figure 1.3.

While many of the protocols developed within this framework are not greatly used today, it remains an interesting academic study for approaches to protocol specification. In the original OSI concept in the late 1970s, there would be just 6 layers providing (progressively richer) carrier services, with a final "application layer" where each specification supported a single end-application, with no "holes".

However, over the next decade it became apparent that even in the "application layer" people wanted to leave "holes" in their specification for later extensions, or to provide a means of tailoring their protocol to specific needs. For example, one of the more recent and important protocols—Secure Electronic Transactions (SET)—contains a wealth of fully defined message semantics, but also provides for

Copyrighted material

1.2.2 Early Developments of Layering

The very earliest protocols operated over a single link (called, surprisingly, "LINK" protocols!) were specified in a single monolithic specification in which different physical signals (usually voltage or current) were used to signal specific events related to the application. (An example is the "off-hook" signal in early telephony systems.) If you wanted to run a different application, you redefined and rebuilt your electronics!

This illustrates the major advantage of "layering"—it enables reusability of carrier mechanisms to support a range of different higher-layer protocols or applications, as illustrated in Figure 1.2.

Nobody today would dream of providing a single monolithic specification similar to the old "LINK" protocols; perhaps the single most important step in computer communication technology was to agree that current, voltage, sound, and light signaling systems would do nothing more than transfer a two-item alphabet—a zero or a one—and that applications would build on that. Another important step was to provide another "layer" of protocol to turn this continuous flow of bits into delimited or "framed" messages with error detection, enabling higher layer protocols to talk about "sending a message" (which may get lost, may get through, but the unit of discussion is the message).

But this is far too low a level of discussion for a book on ASN.1! Between these electrical levels and the normal carriers that ASN.1 operates with we have layers of protocol concerned with both addressing and routing through the Internet or a telecoms network and with recovery from lost messages.

At the ASN.1 level, we assume that an application on one machine can "talk" to an application on another machine by reliably sending octet strings between themselves. (Note that all ASN.1-defined messages **are** an integral multiple of 8-bits—an octet string, not a general bit string.) This is illustrated in Figure 1.4.

Nonetheless, many ASN.1-defined applications are still specified by first specifying a basic "carrier" service, with additional specifications (perhaps provided differently by different groups) to fill in the holes. This is illustrated in Figure 1.5. As we will see later, there are many mechanisms in ASN.1 to support the use of "holes" or of "layering".

People have sometimes described the OSI 7-layer model as "layering gone mad". Layering **can** be an important tool in promoting reusability of specifications (and code), and in enabling parts of the total specification (a low or a high layer), to be later improved, extended (or just mended!) without affecting the other parts of the

Copyrighted material

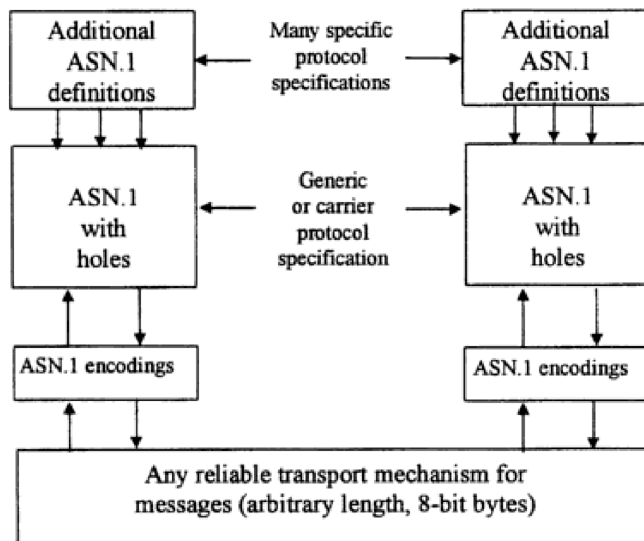


Figure 1.5 Generic and specific protocols with ASN.1.

of "later improvement" is a key phrase, and has importance beyond any discussion of layering. One of the important aspects of protocol specification recognized in the 1980s is that a protocol specification is rarely (probably never!) completed on date xyz, implemented, deployed, and left unchanged.

> **Extensibility Provision**
> Part of a version 1 specification designed to make it easy for future version 2 (extended) systems to interwork with deployed version 1 systems.

There is **always** a "version 2". And implementations of version 2 need to have a ready means of interworking with the already-deployed implementations of "version 1", preferably without having to include in version 2 systems a complete implementation of both version 1 and version 2 (sometimes called "dual-stacks"). Mechanisms enabling version 1 and version 2 exchanges are sometimes called a "migration" or "interworking strategy" between the new and the earlier versions. In the transition from IPv4 to IPv6 (the "IP" part of "TCP/IP"), it has perhaps taken as much work to solve migration problems as it took to design IPv6 itself! (An exaggeration of course, but the point is an important one—interworking with deployed version 1 systems matters.)

It turns out that provided you make plans for version 2 when you write your version 1 specification, you can make the task of "migration" or of defining an "interworking strategy" much easier.

Copyrighted material

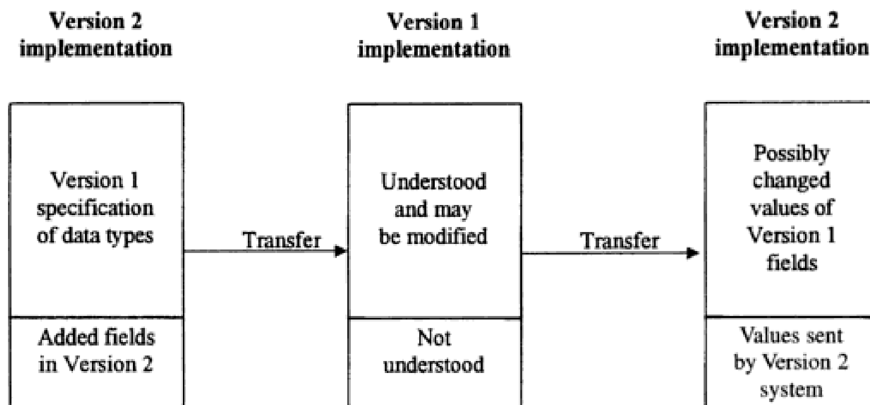


Figure 1.6 Version 1 and Version 2 interworking.

We can define **extensibility provision** as

- elements of a version 1 specification that allow the encapsulation of unknown material at certain points in the version 1 messages, and
- specification of the actions to be taken by the version 1 system if such material is present in a message.

Provision for extensibility in ASN.1 is an important aspect, which will be discussed further later in this book, and is illustrated in Figure 1.6.

Extensibility was present in early work in ITU-T and ISO by use of a very formalized means of transferring parameters in messages, a concept called "TLV"—Type, Length, Value, in which all pieces of information in a message are encoded with a type field identifying the nature of that piece of information, a length field delimiting the value, and then the value itself, an encoding that determines the information being sent. This is illustrated in Figure 1.7 for parameters and for groups of parameters. The approach is generalized in the ASN.1 Basic Encoding Rules (BER) to cover groups of groups, and so on, to any depth.

Note that the encoding used for the value only needs to unambiguously identify application information within the context of the parameter identified by the type field. This concept of distinct octet strings that identify information within the context of some explicit "class" or "type" identifier is an important one that will be returned to later.

By requiring in the version 1 specification that parameters that are "unrecognized"—added in version 2—should be silently ignored, the designers of version 2 have a

Copyrighted material

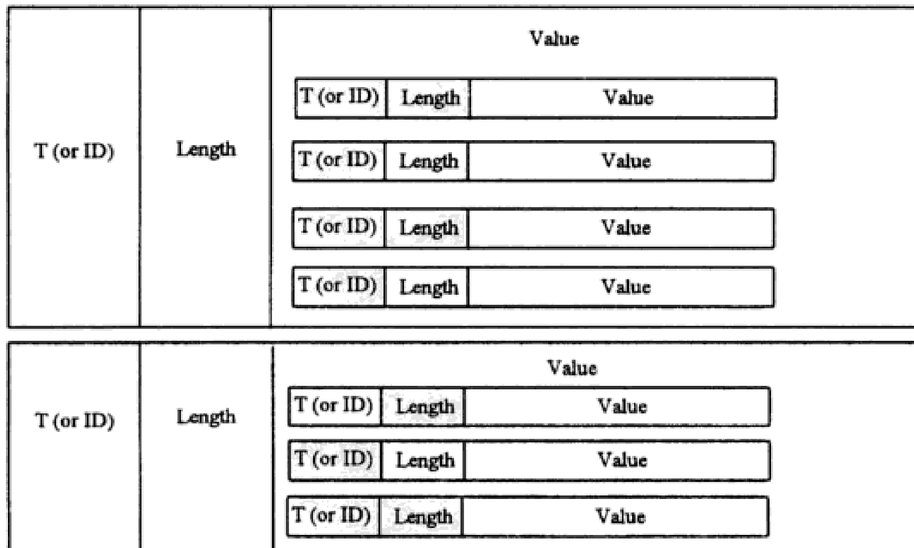


Figure 1.7 The "TLV" approach for parameters and groups.

predictable basis for interworking with deployed version 1 systems. Of course, any other well-specified behavior could be used, but "silently ignore" was a common specification. ASN.1 provides a notation for defining the form of messages, together with "encoding rules" that specify the actual bits on the line for any message that can be defined using the notation. The "TLV" described above was incorporated into the earliest ASN.1 encoding rules (the **Basic Encoding Rules**; or **BER**) and provides very good support for extensibility due to the presence in every element of the "T" and the "L", enabling "foreign" (version 2) material to be easily identified and skipped (or relayed). It does, however, suffer from encoding identification and length fields that are often unnecessary apart from their use in promoting extensibility. For a long time it was thought that this verbosity was an essential feature of extensibility, and it was a major achievement in encoding rule design when the ASN.1 **Packed Encoding Rules (PER)** provided good support for extensibility with little additional overhead on the line.

1.2.5 Abstract and Transfer Syntax

The terms abstract and transfer syntax were primarily developed within the OSI work, and are variously used in other related computer disciplines. The use of these terms in ASN.1 (and in this book) is almost identical to their use in OSI, but does not of course make ASN.1 in any way dependent on OSI.

Copyrighted material

The following steps are necessary when specifying the messages forming a protocol (see Figure 1.8):

- The determination of the information that needs to be transferred in each message; this is a "business-level" decision. We refer here to this as the **semantics** associated with the message.
- The design of some form of data-structure (at about the level of generality of a high-level programming language, and using a defined notation) capable of carrying the required semantics. The set of values of this data-structure are called the **abstract syntax** of the messages or application. We call the notation we use to define this data structure or set of values the **abstract syntax notation** for our messages; ASN.1 is just one of many possible abstract syntax notations, but is probably the one most commonly used.
- The crafting of a set of rules for encoding messages such that, given any message defined using the abstract syntax notation, the actual bits on the line to carry the semantics of that message are determined by an algorithm specified once and once only (independent of the application). We call such rules **encoding rules**, and we say that the result of applying them to the set of (abstract syntax) messages for a given application defines a **transfer syntax** for that application. A transfer syntax is the set of bit-patterns to be used to represent the abstract values in the abstract syntax, with each bit-pattern representing just one abstract value. (In ASN.1, the bit-patterns in a transfer syntax are always a multiple of 8 bits, for easy carriage in a wide range of carrier protocols.)

We saw that early LINK protocols did not clearly separate electrical signaling from application semantics, and similarly today, some protocol specifications do not clearly separate the specification of an abstract syntax from the specification of the bits on the line (the transfer syntax). It is still common to specify directly the bit-patterns to be used (the transfer syntax), and the semantics associated with each bit-pattern. However, as will become clear later, failure to clearly separate abstract from transfer syntax has important implications for reusability and for the use of common tools. With ASN.1 the separation is complete.

1.2.6 Command Line or Statement-Based Approaches

Another important approach to protocol design (not the approach taken in ASN.1) is to focus not on a general-purpose data-structure to hold the information to be

Copyrighted material

Interface Definition Language that enables the data-structures that are passed across each interface to be specified at a high-level of abstraction.

Probably the most important IDL today is the Common Object Request Broker Architecture (CORBA) IDL. In CORBA, the IDL is supported by a wealth of specifications and tools including encoding rules for the IDL, and means of transfer of messages to access interfaces across networks.

A detailed comparison of ASN.1 and CORBA goes beyond this text, and remarks made here should be taken as this author's perception in mid 1999. In essence, CORBA is a complete architecture and message passing specification in which the IDL and corresponding encodings form only a relatively small (but important) part. The CORBA IDL is simpler and less powerful than the ASN.1 notation, and as a result encodings are generally much more verbose than the Packed Encoding Rule (PER) encodings of ASN.1. ASN.1 is generally used in protocol specifications where very general and flexible exchange of messages is needed between communicating partners, whereas CORBA encourages a much more stylized "invocation and response" approach, and generally needs a much more substantial supporting infrastructure.

1.3 More on Abstract and Transfer Syntaxes

1.3.1 Abstract Values and Types

Most programming languages involve the concept of types or classes (and notation to define a more complex type by reference to built-in types and "construction mechanisms"), with the concept of a value of a type or class (and notation to specify values). ASN.1 is no different.

So, for example, in C we can define a new type "My-type" as:

```
typedef struct My-type {
        short        first-item;
        boolean       second-item} My-type;
```

The equivalent definition in ASN.1 appears below.

In ASN.1 we also have the concept of values of basic types or of more complex structures. These are often called **abstract values** (see Figure 1.8 again), to emphasize that we are considering them without any concern for how they might be represented in a computer or on a communications line. For convenience, these abstract values are grouped together into types. For example, we have the ASN.1 type notation

Copyrighted material

Now let us consider a designer who wants to specify the messages of a protocol using ASN.1. It would be possible to define a set of ASN.1 types (one for each different sort of message), and to say that the set of abstract values to be transmitted in protocol exchanges (and hence needing encoding) is the set of all the abstract values of all those ASN.1 types. The observant reader (some people will not like me saying that!) will have spotted that the preceding requirement on a correct set of encoding rules is not sufficient for unambiguous communication of the abstract values, because two abstract values in separate but similar ASN.1 types could have the same octet-string representation. (Both types might be a sequence of two integers, but they could carry very different semantics.)

Abstract Syntax

The set of abstract values of the top-level type for the application.

It is therefore an important requirement in designing protocols using ASN.1 to specify the total set of abstract values that will be used in an application as the set of abstract values **of a single ASN.1 type**. This set of abstract values is often referred to simply as **the abstract syntax of the application**, and the corresponding set of octet strings after applying some set of encoding rules is referred to as a possible **transfer syntax for that application.** Thus the application of the ASN.1 Basic Encoding Rules (as in Figure 1.8) to an ASN.1 type definition produces a transfer syntax (for the abstract syntax) which is a set of bit patterns that can be used to represent these abstract values unambiguously during transfer.

Transfer Syntax

A set of unambiguous octet strings used to represent a value from an abstract syntax during transfer.

Note that in some other areas, where the emphasis is on storage of data rather than its transfer over a network, the concept of abstract syntax is still used to represent the set of abstract values, but the term **concrete syntax** is sometimes employed for a particular bit-pattern representation of the material on a disk. Thus some authors will talk about "concrete transfer syntax" rather than just "transfer syntax", but this term is not used in this book.

We will see later how, if we have distinct ASN.1 types for different sorts of messages, we can easily combine them into a single ASN.1 type to use to define our abstract syntax (and hence our transfer syntax). There is specific notation in the post-1994 version of ASN.1 to identify this "top-level" type clearly. All other ASN.1 type definitions in the specification are there solely to give support to this top-level

Copyrighted material

type, and if they are not referenced by it (directly or indirectly), their definition is superfluous and a distracting irrelevance! Most people **don't** retain superfluous type definitions in published specifications, but sometimes for historical reasons (or through sloppy editing or both!) you may encounter such material.

In summary then—ASN.1 encoding rules provide unambiguous octet strings to represent the abstract values in any ASN.1 type; the set of abstract values in the top-level type for an application is called the abstract syntax for that application; and the corresponding octet-strings representing those abstract values unambiguously (by the use of any given set of encoding rules) is called a transfer syntax for that application.

Note that where there are several different encoding rule specifications available (as there are for ASN.1) there can in general be several different transfer syntaxes (with different verbosity and extensibility—etc.—properties) available for a particular application, as shown in Figure 1.8.

In the OSI world, it was considered appropriate to allow run-time negotiation of which transfer syntax to use. Today, we would more usually expect the application designer to make a selection based on the general nature and requirements of the application.

1.4 Evaluative Discussion

1.4.1 There Are Many Ways of Skinning a Cat: Does It Matter?

While the clear separation of abstract syntax specification (with associated semantics) from specification of a transfer syntax is clearly "clean" in a purist sort of way, does it matter? Is there value in having multiple transfer syntaxes for a given application? The ASN.1 approach to protocol design provides a common notation for defining the abstract syntax of any number of different applications, with common specification text and common implementation code for deriving the transfer syntax from this. Does this really provide advantages over the character line approach discussed earlier? Both approaches have certainly been employed with success. Different experts hold different views on this subject, and as with so much of protocol design, the approach you prefer is more likely to depend on the culture you are working within than on any rational arguments. Indeed, there are undoubted advantages and disadvantages to both approaches, so that a decision becomes more one based on which criteria you consider the most important, rather than on any absolute judgment. So here (as in a number of parts of this book) Figure 999: *Readers take warning* (modified—"Smoking"

Copyrighted material

Government
Health Warning
This discussion can damage your health!

Figure 999: Readers take warning.

replaced by "This discussion"—from text that appears on all UK cigarette packets!) applies. (I will refer back to Figure 999 whenever a remark appears in this book that may be somewhat contentious.)

1.4.2 Early Work with Multiple Transfer Syntaxes

Even before the concepts of abstract and transfer syntax were spelled out and the terms defined, protocol specifiers recognized the concepts and supplied multiple transfer syntaxes in their specifications.

Thus in the Computer Graphics Metafile (CGM) standard, the body of the standard defines the functionality represented by a CGM file (the abstract syntax), with three additional sections defining a "binary encoding", a "character encoding", and a "clear-text encoding". The "binary encoding" was the least verbose, was hard for a human to read (or debug), was not easy to produce with a simple program, and required a storage or transfer medium that was 8-bit transparent. The "character encoding" used two-character mnemonics for "commands" and parameters, and was in principle capable of being produced by a text editor. It was more human readable, but importantly mapped to octets via printing ASCII characters and hence was more robust in the storage and transfer media it could use (but was more verbose). The "clear-text" encoding was also ASCII-based, but was designed to be very human-readable, and very suitable for production by a human being using a suitable text editor, or for viewing by a human being for debugging purposes. It could be employed before any graphical interface tools for CGM became available, but was irrelevant thereafter.

These alternative encodings are appropriate in different circumstances, with the compactness of the "binary encoding" giving it the market edge as the technology matured and tools were developed.

1.4.3 Benefits

Some of the benefits that arise when a notation for abstract syntax definition is employed are identified here, with counterarguments where appropriate.

1.4.3.1 Efficient Use of Local Representations

Suppose you have an application using large quantities of material, which is stored on machine-type-A in a machine-specific format—say with the most significant

Copyrighted material

octet of each 16-bit integer at the lower address byte. On machine-type-B, however, because of differing hardware, the same abstract values are represented and stored with the most significant octet of each 16-bit integer at the higher address byte. (There are usually further differences in the machine-A/machine-B representations, but this so-called "big-endian/little-endian" representation of integers is often the most severe problem.)

When transferring between machine-type-A and machine-type-B, it is clearly necessary for one or both parties (and if we are to be even-handed it should be both!) to spend CPU cycles converting into and out of some agreed machine-independent transfer syntax. But if we are transferring between two separate machines both of machine-type-A, it clearly makes more sense to use a transfer syntax closely related to the storage format on those machines.

This issue is generally more important for applications involving the transfer of large quantities of highly structured information, rather than for small headers negotiating parameters for later bulk transfer. An example where it would be relevant is the Office Document Architecture (ODA) specification. This is an ISO Standard and ITU-T Recommendation for a large structure capable of representing a complete service manual for (for example) a Boeing aircraft, so the application data can be extremely large.

1.4.3.2 Improved Representations over Time

It is often the case that the early encodings produced for a protocol are inefficient, partly because of the desire to be "protective", or to have encodings that are easy to debug, in the early stages of deployment of the application, partly from simple time pressures. It can also be because insufficient effort is put into the "boring" task of determining a "good" set of "bits-on-the-line" for this application.

Once again, if the bulk of the protocol is small compared with some "bulk-data" that it is transferring, as is the case—for most messages—with the Internet's HyperText Transfer Protocol (HTTP) or File Transfer Protocol (FTP), then efficiency of the main protocol itself becomes relatively unimportant.

1.4.3.3 Reuse of Encoding Schemes

If we have a clear separation of the concept of abstract syntax definition from transfer syntax definition, and have available a notation for abstract syntax definition (such as ASN.1) that is independent of any application, then specification and implementation benefits immediately accrue. The task of generating "good" encoding rules for that notation can be done once, and these rules can be refer-

Copyrighted material

data-structures in the implementation language, this encourages (but of course does not require) a modular approach to implementation design in which the code responsible for performing the encodings of the data is kept clearly separate from the code responsible for the semantics of the application.

1.4.3.5 Reuse of Code and Common Tools

This is perhaps the major advantage that can be obtained from the separation of abstract and transfer syntax specification, which is characteristic of ASN.1.

By the use of so-called ASN.1 "compilers" (dealt with more fully in Chapter 7 and which are application-independent), any abstract syntax definition in ASN.1 can be mapped into the (abstract) data-structure model of any given programming language, through the textual representation of data-types in that language. Implementors can then provide code to support the application using that (abstract) data-structure model with which they are familiar, and can call an application-independent piece of code to produce encodings of values of that data-structure for transmission (and similarly to decode on reception).

It is very important at this point for the reader to understand why "(abstract)" was included in the preceding text. All programming languages (from C to Java) present to their users a "memory-model" by which users define, access, and manipulate structures. Such models are platform independent, and generally provide some level of portability of any associated code. However, in mapping through compilers and run-time libraries into real computer memory (concrete representation of the abstract data-structures), specific features of different platforms intrude, and the precise representation in memory differs from machine-type to machine-type (see the "big-endian/little-endian" discussion in Chapter 18).

A tool-vendor can provide (possibly platform-specific, but certainly application-independent) run-time routines to encode/decode values of the abstract data-structures used by the implementor, and the implementor can continue to be blissfully unaware of the detailed nature of the underlying hardware, but can still efficiently produce machine-independent transfer syntaxes from values stored in variables of the implementation language.

As with any discussion of code structure, reusability, and tools, real benefits arise only when there are multiple applications to be implemented. It is sometimes worthwhile building a general-purpose tool to support a single implementation, but more often than not it is not. Tools are of benefit if they can be used for multiple implementations, either by the same implementors or by a range of implementors.

Copyrighted material

1.5 Protocol Specification and Implementation: A Series of Case Studies

This section completes this chapter with discussion of a number of approaches to protocol specification and implementation, ending with a simple presentation of the approach that is adopted when ASN.1 is used.

1.5.1 Octet Sequences and Fields within Octets

Protocols for which all or much of the information can be expressed as fixed-length fields, which are all required to be present, have traditionally been specified by drawing diagrams such as the one shown in Figure 1.10—*Traditional approach.*

Figure 1.10 is part of the Internet Protocol Header (the Internet Protocol is the IP protocol of the TCP/IP stack illustrated in Figure 1.2). A similar picture is used in X.25 level 2 to define the header fields.

This approach was very popular in the early days, when implementations were performed using assembler language or languages such as BCPL or later C, allowing the implementor close contact with the raw byte array of a computer memory.

It was relatively easy for the implementor to read in octets from the communications line to a given place in memory, and then to hardwire into the implementation code access to the different fields (as shown in the diagram) as necessary (similarly for transmission). In this approach the terms "encoding" and "decoding" were not usually used.

The approach worked well in the mid 1970s, with the only spectacular failures arising (in one case) from a lack of clarity in the specification of which end of the octets (given in the diagram) was the most significant when interpreting the octet as a numerical value, and which end of the octets (given in the diagram) was to be transmitted first on a serial line. The need for a very clear specification of these bit-orders in binary-based protocol specification is well understood today, and in particular is handled within the ASN.1 specification, and can be ignored by a designer or implementor of an ASN.1-based specification.

1.5.2 The TLV Approach

Even the simplest protocols found the need for variable length "parameters" of messages, and for parameters that could be optionally omitted. This was briefly described earlier (see Figure 1.7) in 1.2.4.

Copyrighted material

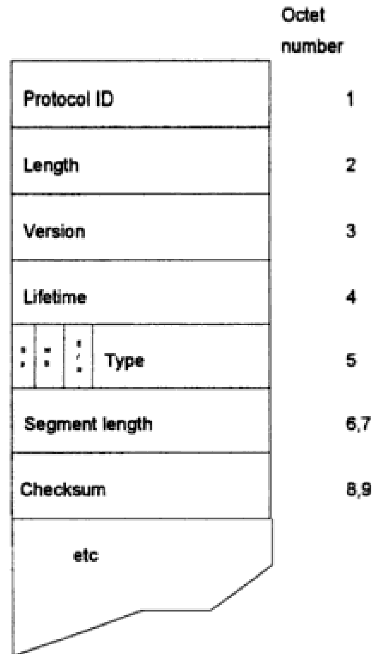


Figure 1.10 Traditional approach.

In this case, the specification would normally identify some fixed-length mandatory header fields, followed by a "parameter field" (often terminated by a length count). The "parameter field" would be a series of one or more parameters, each encoded with an identification field, a length field, and then the parameter value. The length field was always present, even for a fixed-length parameter, and the identification field even for a mandatory parameter. This ensured that the basic "TLV" structure was maintained, and enabled "extensibility" text to be written for version 1 systems to skip parameters they did not recognize.

An implementor would now write some fairly general-purpose code to scan the input stream and to place the parameters into a linked list of buffers in memory, with the application-specific code then processing the linked buffers. Note, however, that while this approach was quite common in several specifications, the precise details of length encoding (restricted to a count of 255 or unrestricted, for example), varied from specification to specification, so any code to handle these parameters tended to be application-specific and not easily reusable for other applications.

As protocols became more complicated, designers found the need to have complete groups of parameters that were either present or omitted, with all the parameters in

Copyrighted material

a given group collected together in the parameter field. This was the approach taken in the Teletex (and later the OSI Session Layer) specifications, and gave rise to a second level of TLV with an outer identifier for a parameter group, a length field pointing to the end of that group, and then the TLV for each parameter in the group (revisit Figure 1.7).

This approach was also very appropriate for information that required a variable number of repetitions of a given parameter value.

At the implementation level, the code to "parse" an input octet string is now a little more complex, and the resulting data-structure to be passed to the application-specific code becomes a two-level tree-structure rather than a simple linked list, with level 1 nodes being parameter groups, and level 2 nodes parameters.

This approach has been presented here in a very "pure" form, but in fact it was rarely so pure! The Teletex and Session Protocols actually mixed together at the top level parameter group TLVs and parameter TLVs!

Those who already have some familiarity with the ASN.1 Basic Encoding Rules (BER) (to be described in much more detail later), will recognize that this TLV approach was generalized to form the basic (application-independent) encoding used by BER. For BER, the entire message is wrapped up with an identifier (that distinguishes it from any other message type in the same abstract syntax) and a length field pointing to the end of the message. The body is then, in general, a sequence of further TLV triplets, with the "V" part of each triplet being either further TLV triplets (etc., to any depth), or a "primitive" field such as an integer or a character string. This gives complete support for the power of normal programming language data-structure definitions to define groupings of types and repetitions of types to any depth, as well as providing support at all levels for both optional elements and extensibility.

1.5.3 The EDIFACT Graphical Syntax

This approach comes closest to ASN.1, with a clear (graphical) notation for abstract syntax specification, and a separate encoding rule specification. An example of the Electronic Data Interchange For Administration, Commerce and Transport (EDIFACT) graphical syntax is given in Figure 1.11, *EDIFACT graphical syntax*. As with ASN.1, the definition of the total message can be done in conveniently sized chunks using reference names for the chunks, then those chunks are combined to define the complete message. So in Figure 1.11 we have the message fragment (defined earlier or later) "UNH", which is mandatorily present once, similarly "AAA", then "BBB", which is conditional and is present zero to ten times, then "CCC" similarly, then up to 200 repetitions of a composite structure consisting of one "DDD" followed by up to ten "EEE", etc.

Copyrighted material

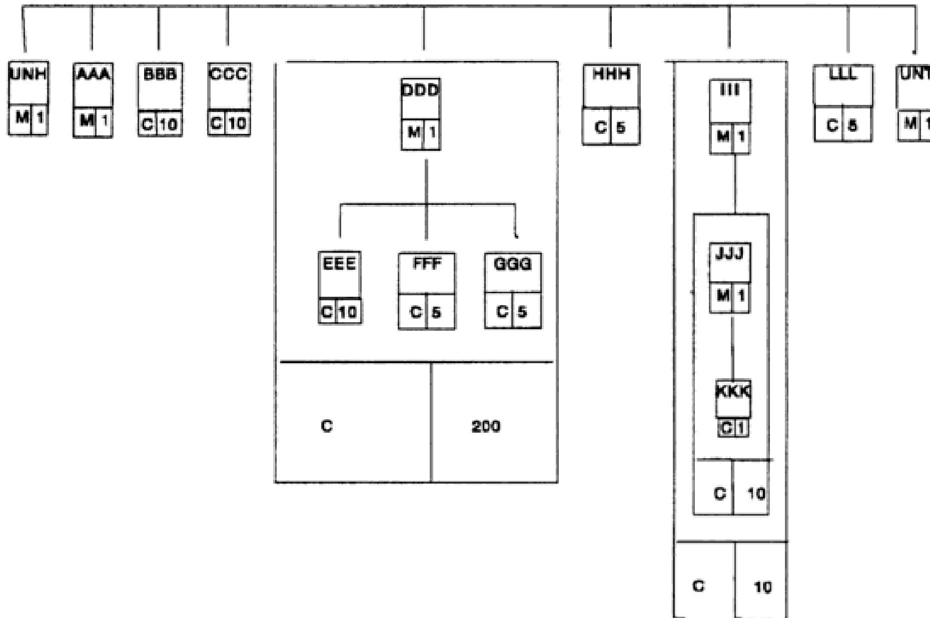


Figure 1.11 EDIFACT graphical syntax.

The actual encoding rules were, as with ASN.1, specified separately, but were based on character encoding of all fields. The graphical notation is less powerful than the ASN.1 notation, and the range of primitive types much smaller. The encoding rules also rely on the application designer to ensure that a type following a repeated sequence is distinct from the type in that repeated sequence, otherwise ambiguity occurs. This is a problem avoided in ASN.1, where any legal piece of ASN.1 produces unambiguous encodings.

At the implementation level, it would be possible to map the EDIFACT definition into a data-structure for the implementation language, but I am not aware of any tools that currently do this.

1.5.4 Use of BNF to Specify a Character-Based Syntax

This approach was briefly described earlier, and is common in many Internet protocols.

Where this character-based approach is employed, the precise set of lines of text permitted for each message has to be clearly specified. This specification is akin to the definition of an abstract syntax, but with more focus on the representation of the information on the line than would be present in an ASN.1 definition of an abstract syntax.

Copyrighted material

Identification fields for lines in the messages tend to be relatively long names, and "enumerations" also tend to use long lists of names, so the resulting protocol can be quite verbose. In these approaches, length fields are normally replaced by reserved-character delimiters, or by end-of-line, often with some form of escape or extension mechanism to allow continuation over several lines (again these mechanisms are not always the same for different fields or for different applications).

In recent years there has been an attempt to use exactly the same BNF notation to define the syntax for several Internet protocols, but variations still ensue.

At implementation-time, a sending implementation will typically hardwire the encoding as a series of "PRINT" statements to print the character information directly onto the line or into a buffer. On reception, a general-purpose tool would normally be employed that could be presented with the BNF specification and that would parse the input string into the main lexical items. Such tools are available without charge for Unix systems, making it easy for implementations of protocols defined in this way to be set as tasks for Computer Science students (particularly as the protocol specifications tend also to be available without charge!).

In summary then, this approach can work well if the information to be transferred fits naturally into a two-level structure (lines of text, with an identifier and a list of comma-separated text parameters on each line), but can become complex when a greater depth of nesting of variable numbers of iterated items becomes necessary, and when escape characters are needed to permit commas as part of a parameter. The approach also tends to produce a much more verbose encoding than the binary approach of ASN.1 BER, and a very much more verbose encoding than the ASN.1 Packed Encoding Rules (PER).

1.5.5 Specification and Implementation Using ASN.1: Early 1980s

ASN.1 was first developed to support the definition of the set of X.400 Message Handling Systems CCITT (the International Telegraph and Telephone Consultative Committee, later to be renamed ITU-T) Recommendations, although the basic ideas were taken from the Xerox Courier Specification.

X.400 was developed by people with a strong application interest in getting the semantics of the information flows for electronic messaging right, but with relatively little interest in worrying about the bit-level encoding of messages. It was clear that they needed more or less the power of data-structure definition in a high-level programming language to support their specification work, and ASN.1 was designed to provide this.

Copyrighted material

1.5.6 Specification and Implementation Using ASN.1: 1990s

It is of course still possible to produce an implementation of an ASN.1-based protocol without tools. What was done in the 1980s can still be done today. However, there is great pressure today to reduce the "time-to-market" for implementations, and to ensure that residual bugs are at a minimum. Use of tools can be very important in this respect.

Today there are two main families of ASN.1 encoding rules, the original (unchanged) BER, and the more recent (standardized 1994) Packed Encoding Rules (PER). The PER encoding rules specification is more complex than that of BER, but produces very much more compact encodings. (For example, the encoding of a boolean value in PER uses only a single bit, but the TLV structure of BER produces at least 24 bits!)

There seems to be a "conventional wisdom" emerging that while encoding/decoding without a tool for BER is an acceptable thing to do if you have the time to spare, it is likely to result in implementation bugs if PER is being employed. The reader should again refer to Figure 999: *Readers take warning!*. This author would contend that there are implementation strategies that make PER encoding/decoding without tools a very viable proposition. Certainly much more care at the design stage is needed to identify correctly the field-widths to be used to encode various elements, and when padding bits are to be added (this comment will be better understood after reading Chapter 17 on PER), but once that is done, hardwiring a PER encode/decode into application code is still (this author would contend) possible.

ASN.1 Allows

designers to concentrate on application semantics,

design without encoding-related bugs and with compact encodings available,

implementors to write minimum code to support the application—fast development, and

bug-free encode/decode with absence of interworking problems.

Nonetheless, today, good tools, called "ASN.1 compilers", **do** exist, and for any commercial development they provide good value for money and are widely used. How would you implement an ASN.1 specification using a tool? This is covered more fully (with examples based on the "OSS ASN.1 Tools" package) in Chapter 7. However, the basic outline is as follows (see Figure 1.12).

Copyrighted material

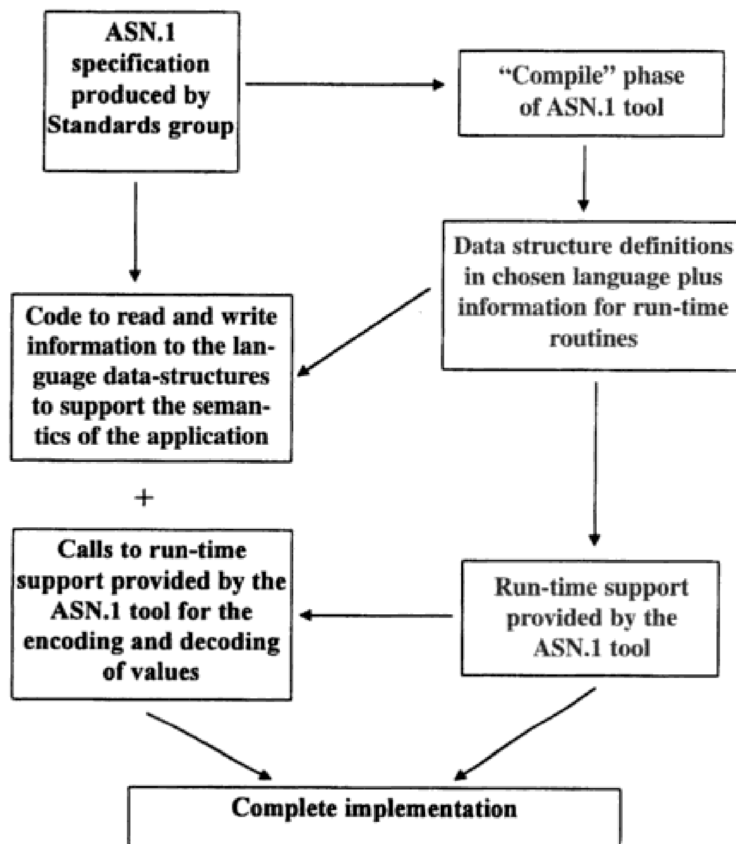


Figure 1.12 Use of an ASN.1 tool for implementation.

The ASN.1 produced by the application designer is fed into the "compile phase" of the tool. This maps the ASN.1 into a language data-structure definition in any one of a wide range of supported languages (and platforms), including C, C++, and Java. The application code is then written to read and write values from these data-structures, concentrating solely on the required semantics of the application.

When an encode is needed, a run-time routine is called that uses information provided by the compile phase about certain aspects of the ASN.1 definition, and which "understands" the way in which information is represented in memory on this platform. The run-time routine encodes the entire message, and returns the resulting octet string. A similar process is used for decoding. Any issues of big-endian or little-endian byte order (see 18.2.3), or most-significant bits of a byte, are completely hidden within the encode/decode routines, as are all other details of the encoding rule specifications.

Copyrighted material

Of course, without using a tool, a similar approach of mapping ASN.1 to a language data-structure and having separate code to encode and decode that data-structure is possible, but is likely to be more work (and more error prone) than the more "hardwired" approach outlined here. But with a tool to provide the mapping and the encode/decode routines, this is an extremely simple and fast means of producing an implementation of an ASN.1-based application.

In conclusion then, using a tool, ASN.1 today:

- Provides a powerful, clear and easy to use way for protocol designers to specify the information content of messages.
- Frees application designers from concerns over encoding, identification of optional elements, termination of lists, etc.
- Is supported by tools mapping the ASN.1 structures to those of the main computer languages in use today.
- Enables implementors to concentrate solely on the application semantics without any concern with encoding/decoding, using application-independent run-time encode/decode routines to produce bug-free encodings for all the ASN.1 encoding rules.

Copyrighted material

Introduction to ASN.1

(Or: Read Before You Write!)

Summary

The best way of learning any language or notation is to read some of it. This chapter presents a small example of ASN.1 type definitions and introduces the main concepts of

- built-in key-words,
- construction mechanisms,
- user-defined types with type-reference-names,
- identifiers or "field-names", and
- alternatives.

There is a reference to "tagging", which is discussed in more detail in Section II.

This chapter is intended for beginners in ASN.1, and can be skipped by those who have already been exposed to the notation.

2.1 Introduction

Look at Figure 2.1. The aim here is simply to make sense of the data-structure it is defining—the information that transmission of a value of this structure would convey.

Figure 2.1 is an "artificial" example designed to illustrate the features of ASN.1. It does not necessarily represent the best "business solution" to the problem it appears to be addressing, but the interested reader could try to invent a plausible rationale for some of its more curious features. For example, why have different "details" been used for "uk" and for "overseas" when the "overseas" case can hold any information the "uk" case can? Plausible answer, the "uk" case was in version 1, and the "overseas" was added later when the business expanded, and the designer wanted to keep the same bits-on-the-line for the "uk" case.

This example is built-on as this book proceeds, and the scenario for this "Wineco protocol" appears in Appendix 1 with the complete protocol in Appendix 2.

Copyrighted material

2.2.1 The Top-Level Type

There is nothing in the example (other than that it appears first) to tell the reader clearly that "Order-for-stock" is the top-level type, the type whose values form the abstract syntax, the type that when encoded provides the messages that are transmitted by this application. In a real ASN.1 specification, you would discover this from human-readable text associated with the specification, or in post-1994 ASN.1 by finding a statement:

```
my-abstract-syntax ABSTRACT-SYNTAX ::=
 {Order-for-stock IDENTIFIED BY
   {joint-iso-itu-t international-organization(23) set(42) set-vendors(9)
    wineco(43) abstract-syntax (1)}}
```

This simply says that we are naming the abstract syntax "my-abstract-syntax", that it consists of all the values of the type "Order-for-stock", and that if it were necessary to identify this abstract syntax in an instance of computer communication, the value given in the third line would be used. This is your first encounter with a piece of ASN.1 called "an OBJECT IDENTIFIER value" (which you will frequently find in ASN.1 specifications). The whole of that third line is actually just equivalent to writing a string of numbers:

> **Top-Level Type**
> All application specifications contain a (single) ASN.1 type that defines the messages for that application. It will often (but need not) appear first in the specification, and is a good place to start reading!

```
{2 23 42 9 43 1}
```

But for now, we will ignore the OBJECT IDENTIFIER value and go back to the main example in Figure 2.1.

2.2.2 Bold Is What Matters!

The parts in bold are the heart of the ASN.1 language. They are reserved words (note that they are mainly all uppercase—case **does** matter in ASN.1), and reference built-in types or construction mechanisms. A later chapter goes through each and every built-in type and construction mechanism!

Copyrighted material

It may be helpful initially to think of the normal font words as the names of fields of a record structure, with the following bold or italic word giving the type of that field. The correct ASN.1 terminology is to say that the normal font words are either

- naming elements of a sequence,
- naming elements of a set,
- naming alternatives of a choice, or
- (in one case only) naming enumerations.

If an ASN.1 tool is used to map the ASN.1 specification to a data-structure definition in a programming language, these normal font names are mapped to identifiers in the chosen language, and the application code can set or read values of the corresponding parts of the data-structure using these names.

The alert reader—again!—will immediately wonder about the length of these names, and the characters permitted in them, and ask about any corresponding problems in doing a mapping to a given programming language. These are good questions, but will be ignored for now, except to say that all ASN.1 names can be arbitrarily long, and are distinct even if they differ only in their hundredth character, or even their thousandth (or later)! Quite long names are fairly common in ASN.1 specifications.

2.2.5 Back to the Example!

So, . . . what information does a value of the type "*Order-for-stock*" carry when it is sent down the line?

"*Order-for-stock*" is a structure with a sequence of fields or "elements" (an ordered list of types whose values will be sent down the line, in the given order). The first field or element is called "order-no", and holds an integer value. The second is called "name-address" and is itself a fairly complex type to be defined later, with a lot of internal structure. The next top-level field is called "details", and is also a fairly complex structured field, but this time the designer, purely as a matter of style, has chosen to write out the type "in-line" rather than using another type-reference-name.

This field is a "**SEQUENCE OF**", that is to say, an arbitrary number of repetitions of what follows the "**SEQUENCE OF**" (could be zero). There is ASN.1 notation to require a minimum or maximum number of repetitions, but that is not often encountered and is left until later.

Copyrighted material

What follows is another "**SEQUENCE**", binding together an "**OBJECT IDENTIFIER**" field called "item" and an "**INTEGER**" field called "cases". (Remember, we are ordering stocks—cases—of wine!) So the whole of "details" is arbitrarily many repetitions of a pair of elements—an object identifier value and an integer value.

You already met object identifier values when we discussed identification of the abstract syntax for this application. Object identifiers are world-wide unambiguous names. Anybody can (fairly!) easily get a bit of the object identifier name space, and these identifiers are frequently used in ASN.1-based applications to name a whole variety of objects. In the case of this example, we use names of this form to identify an "item" (in this case, the "item" is probably some stock item—identification of a particular wine). We also see later that the application designer has chosen to use identifications of this same form in "BranchIdentification" to provide a "unique-id" for a branch.

Following the "details" top-level field, we have a field called "urgency", that is of the built-in type "**ENUMERATED**". Use of this type name **requires** that it be followed by a list of names for the enumerations (the possible values of the type). In ASN.1, but not in most programming languages, you will usually find the name followed by a number in round brackets, as in this example. These numbers were required to be present up to 1994, but can now be automatically assigned if the application-designer so desires. They provide the actual values that are transmitted down the line to identify each enumeration, so if the "urgency" is "deliver it tomorrow", what is sent down the line in this field position is a zero. (The reason for requiring the numbers to be assigned by the designer in the early ASN.1 specifications is discussed later, but basically has to do with trying to avoid interworking problems if a version 1 specification has an extra enumeration added in version 2—extensibility again!)

> Keyword DEFAULT: Identifies a default value for an element of a SEQUENCE or SET, to be assumed if a value for that element is not included.
>
> Keyword OPTIONAL: Identifies an element for which a value can be omitted. Omission carries different semantics from any normal value of the element.

Again, the "urgency" field has a feature not found in programming language data-structure definition. We see the keyword "**DEFAULT**". What this means for the Basic Encoding Rules (BER—the original ASN.1 Encoding Rules) is that, as a sender's

Copyrighted material

option, that field need not be transmitted if the intended value is the value following the word "**DEFAULT**"—in this case "week". This is an example in which there is more than one bit-pattern corresponding to a single abstract value—it is an encoder's option to choose whether or not to encode a "**DEFAULT**" value. For the later Packed Encoding Rules, the encoder is **required** to omit this simple field if the value is "week", and the decoder assumes that value. (If "urgency" had been a more complex data type the situation would be slightly different, but that is a matter for Section III.)

There is another ASN.1 keyword similar to "**DEFAULT**", namely, "**OPTIONAL**" (not included in the example in Figure 2.1). Again, the meaning is fairly obvious: the field can be omitted, but there is no presumption of any default value. The keyword might be associated, for example, with a field/element whose name was "additional-information".

Just to return briefly to the question of "What are the precise set of abstract values in the type?", the answer is that the presence of **DEFAULT** does not change the number of abstract values, it merely affects encoding options, but the presence of **OPTIONAL does** increase the number of abstract values—an abstract value with an optional field absent is distinct from any abstract value where it is present with some value, and can have different application semantics associated with it.

Finally, in "*Order-for-stock*", the last element is called "authenticator" and is of some (possibly quite complex) type called "*Security-Type*" defined by the application designer either before or after its use in "Order-for-stock". It is shown in Figure 2.1 as a "**SET**", with the contents not specified in the example (in a real specification, of course, the contents of the "**SET**" would be fully defined). "SET" is very similar to "**SEQUENCE**". In BER (the original ASN.1 encoding rules), it again signals a sender's (encoder's) option. The top-level elements (fields) of the **SET**, instead of being transmitted in the order given in the text (as they are for **SEQUENCE**) are transmitted in any order that is convenient for the sender/encoder. Today, it is recognized that encoder options are a "BAD THING" for both security reasons and for the extra cost they impose on receivers and particularly for exhaustive testing, and there are many who would argue that "**SET**" (and the corresponding "**SET OF**") should never be used by application designers, and should be withdrawn from ASN.1! But please refer to Figure 999 again!

Figure 2.1 shows "*Security-Type*" being defined later in the specification, but actually, this is precisely the sort of type that is more likely to be **imported** by an application designer from some more specialized ASN.1 specification that defines types (and their semantics) designed to support security features.

Copyrighted material

There are mechanisms in ASN.1 (discussed later) to enable a designer to reference definitions appearing in other specifications, and these mechanisms are often used. You will, however, also find that **some** application designers will **copy** definitions from other specifications, partly to make their own text complete without the need for an implementor to obtain (perhaps purchase!) additional texts, and partly to ensure control over and "ownership" of the definition. If you are using this book with a colleague or as part of some course, you can have an interesting debate over whether it is a good thing to do this or not!

2.2.6 The BranchIdentification Type

Now let us look briefly at the "*BranchIdentification*" type, which illustrates a few additional features of the ASN.1 notation. (For now, please completely ignore the numbers in square brackets in this definition. These are called "tags", and are discussed at the end of this chapter.)

This time it has been defined as a "SET", so in BER the elements are transmitted in any order, but we will take them in textual order.

As an aside (but an important aside), we have already mentioned in Chapter 1 that BER uses a TLV type of encoding for all elements. Clearly, if the sender is able to transmit the elements of a "**SET**" in any order, the value used for the "T" in the TLV of each element has to be different. (This would not be necessary for **SEQUENCE**, unless there are **OPTIONAL** or **DEFAULT** elements whose presence or absence had to be detected.) It is this requirement that gives rise to the "tag" concept to be introduced briefly below, and covered more fully later.

The first listed element is "unique-id", an "**OBJECT IDENTIFIER**" value, which has already been discussed. The only other element is "details." Notice that the name "details" was also used in "*Order-for-Stock*". This is quite normal and perfectly legal—the contexts are different.

> **Names of Elements and Alternatives**
> Should all be distinct within any given SEQUENCE, SET, or CHOICE (a requirement post-1994).

It is usual for application designers to use distinct names for top-level elements in a **SEQUENCE** or **SET**, but it was not actually a requirement prior to 1994. It **is** now a requirement to have distinct names for the elements of both "**SEQUENCE**" and "**SET**" (and for the alternatives of a "**CHOICE**"—see later text). The requirement was added partly because it made good sense, but mainly because

Copyrighted material

However, we have already noted that some restrictions were added in 1994 (names of elements of a "**SEQUENCE**", "**SET**" etc. were required to be distinct, for example). Suppose you can not be bothered to upgrade your (300 pages long!) specification to conform to 1994 or later, but still want to use **UTF8String** in a new version? Well, legally, you **can not.** ("Oh yeah?", you say, "What government has passed that law?" "Which enforcement agency will punish me if I break it?" I remain silent!) But as an implementor/reader, and if you see it happening, you will know what it means. Of course, as part of an application design team, you would make absolutely sure it did not happen in **your** specifications, wouldn't you?

Back to Figure 2.1. The third alternative in the "details" is "warehouse", and this itself is another "**CHOICE**", with just two alternatives—"northern" and "southern," each with a type "**NULL**". What is "**NULL**"? "**NULL**" formally is a type with just a single value (which is itself perhaps confusingly called "**NULL**"). It is used where we need to have a type, but where there is no additional information to include. It is sometimes called a "placeholder." Note that in the "warehouse" case, we could just as well have used a **BOOLEAN** to decide "northern" vs. "southern", or an **ENUMERATED**. Just as a matter of style (and to illustrate use of "**NULL**"!) we chose to do it as a choice of **NULLs**.

2.2.7 Those Tags

Now we will discuss the numbers in square brackets—the "tags". In post-1994 ASN.1, it is **never** necessary to include these numbers. If they would have been required pre-1994, you can (post-1994) ask for them to be automatically generated (called AUTOMATIC TAGGING), and need never actually include them. However, in existing published specifications, you will frequently encounter tags, and should have some understanding of them.

> **Tags**
>
> Numbers in square brackets, not needed post-1994, are there to ensure unambiguous encodings. They do not affect the information that can be carried by the values of an ASN.1 type.

In some of the very oldest ASN.1-based application specifications you will frequently find the keyword "IMPLICIT" following the tag, and occasionally today the opposite keyword "EXPLICIT". These qualify the meaning of the tag, and are fully described in Chapter 3.

Copyrighted material

2.3 Getting Rid of the Different Fonts

Suppose you have a normal ASN.1-based application specification using a single font. How do you apply fonts as in Figure 2.1?

First, in principle, you need to know what are the reserved words in the language, including the names of the character string and the date/time types, and you make sure these become bold! In practice, you can make a good guess that any name that is all uppercase goes to bold, but this is not a requirement. The "*Address*" type-reference-name in Figure 1.4 could have been "*ADDRESS*", and provided that change was made everywhere in the specification, the result is an identical and totally legal specification. But as a matter of style, all uppercase for type reference names is rarely used.

Any other name that begins with an initial uppercase letter you set to italics—it is a type-reference-name. Type-reference-names are **required** to begin with an uppercase letter. After that they can contain uppercase or lowercase interchangeably.

You will see in Figure 2.1 a mixture of two distinct styles. In one case a type-reference-name ("*Order-for-stock*") made up of three words separates the words by a hyphen. In another case a type-reference-name ("*OutletType*") uses another uppercase letter to separate the words, and does not use the hyphen. "*Security-Type*" uses both!

You normally don't see a mix of these three styles in a single specification, but all are perfectly legal. Hyphens (but not two in adjacent positions, to avoid ambiguity with comment—see later text) have been allowed in names right from the first approved ASN.1 specification, but were not allowed by drafts prior to that first approved specification, so early writers had no choice, and used the "*OutletType*" style. Of course, nobody ever reads the ASN.1 specification itself—they just copy what everybody else does! So that style is still the most common today. It is, however, just that—a matter of style, and an unimportant one at that—all three forms are legal and it is a personal preference which you think looks neater or clearer.

And finally, the normal font, most names starting with a lowercase letter are names of elements or alternatives ("order-no", "urgency", etc.), and again such names are **required** to start with an initial lowercase letter, but can thereafter contain either uppercase or lowercase.

Names beginning with lowercase are also required for the names of **values**. A simple example is the value "week" for the "urgency".

Copyrighted material

Application specifications can contain not only type assignment statements such as those appearing in Figure 2.1 (and which generally form the bulk of most application specifications), but can also contain statements assigning values to "value-reference-names". The general form of a value reference assignment is illustrated below:

```
my-default-cases INTEGER ::= 20
```

which is defining the value-reference-name "my-default-cases", of type "**INTEGER**" to reference the integer value "20". It could then be used in the "cases" element in Figure 2.1 as, for example:

```
cases INTEGER DEFAULT my-default-cases
```

2.4 Tying Up Some Loose Ends

2.4.1 Summary of Type and Value Assignments

First, let us summarize what we have seen so far. ASN.1 specifies a number of pieces of notation (type-notation) that define an ASN.1 type. Some are very simple, such as "BOOLEAN", others are more complex such as that used to define an enumerated type or a sequence type. **A type-reference-name is also a piece of type-notation that can be used wherever ASN.1 requires a piece of type-notation.**

> An application specification contains lots of type assignment statements and occasionally (but rarely) some value assignment statements.

Similarly, ASN.1 specifies a number of pieces of value-notation (any type you can write with ASN.1 has a defined value-notation for all of its values). Again, some notations for values are very simple, such as "20" for integer values; others are more complex, such as the notation for object identifier values that you saw at the start of this chapter, or the notation for values of sequence types. **Again, wherever ASN.1 requires value-notation, a value-reference-name can be used** (provided it has been assigned a value somewhere).

The general form of a type assignment is:

```
type-reference-name  ::=  type-notation
```

and of a value assignment is:

```
value-reference-name type-notation ::= value-notation
```

Copyrighted material

where the value-notation has to be the "correct" value-notation for the type identified by the type-notation. This is an important concept. Anywhere in ASN.1 where you can use type-notation (for example to define the type of an element of a "SET" or "SEQUENCE", you can use **any** legal type-notation. However, where value-notation is allowed (for example, in value assignments or after DEFAULT), there is always a corresponding type-notation called the **governor** (which might be a type-reference-name) that restricts the syntax of the value-notation to that which is permitted for the type identified by the type-notation.

So far, you have seen value notation used in the "IDENTIFIED BY" at the start of the chapter, and following the word DEFAULT. There are other uses that will be described later, but it remains the case that value-notation is used much less often than type-notation.

2.4.2 The Form of Names

All names in ASN.1 are mixed upper/lowercase letters and digits and hyphens (but not two adjacent or one at the end, to avoid confusion with comment), starting with either an uppercase letter or with a lowercase letter, depending on what the name is being used for. (As you will have guessed by now, they cannot contain the space character!) In every case of naming in ASN.1, the case of the first letter is fixed. If an uppercase letter is legal, a lowercase letter will not be, and vice versa. Names can be arbitrarily long, and are different names if they differ in either content or case at any position in the name.

Note that because names can contain only letters and digits and hyphens, a name that is followed by any other character (such as an opening curly bracket or a comma), can have the following character adjacent to it with no space or new-line, or as a matter of purely personal style, one or more spaces or new-lines can be inserted.

> **Names and Layout**
>
> Names contain letters, digits, or hyphens. They are arbitrarily long. Case is significant. Layout is free format. Comment starts with a pair of adjacent hyphens and ends with a pair of adjacent hyphens or a new-line.

2.4.3 Layout and Comment

Layout is "free-format"—anywhere you can put a space you can put a new-line. Anywhere you have a new-line you can remove it and just leave a space. So a complete application specification can

Copyrighted material

appear as a single line of text, and indeed that is basically the way a computer sees it.

As a matter of style, **everybody** puts a new line between each type or value assignment statement, and generally between each element of a set or sequence and the alternatives of a choice. The layout style shown in Figure 2.1 is that preferred by this author, as it makes the pairing of curly brackets very clear, but a perhaps slightly more common layout style is to include the opening curly bracket after "SEQUENCE" on the same line as the keyword "SEQUENCE", for example:

```
SEQUENCE {
  items   OBJECT IDENTIFIER,
  cases   INTEGER   }
```

Still other authors (less common) will put the closing curly bracket on a line of its own and align it vertically with its matching opening bracket. All pure (and utterly unimportant!) stylistic matters.

On a slightly more serious vein, there was pre-1994 value notation for the "CHOICE" type in the "BranchIdentification" that would allow:

```
details  warehouse  northern  value-ref
```

as a piece of value notation (where "value-ref" is a value reference name for the "NULL" value). Remember that ASN.1 allows names to be used before they are assigned in a type or value assignment, and a poor dumb computer can be hit at the start of the specification with something looking like:

```
joe Fred ::= jack jill joseph Mary ::= etc etc
```

In this case, it cannot determine where the first assignment ends—after "jack" or after "jill" or after "joseph"—it depends on the actual type of "Fred"—defined later). This can give a computer a hard time! Some of the early tool vendors could not cope with this (even though it probably never actually occurred!), and asked for the "semicolon" character to be used as a statement separator in ASN.1. To this day, if you use these tools, you will need to put in semicolons between all your type assignments. (The "OSS ASN.1 Tools" package does **not** impose this requirement.) The requirement to insert semicolons in ASN.1 specifications was resisted, but to assist tool vendors a "colon" was introduced into the value notation for "CHOICE", so that post-1994 the above value notation would be written:

Copyrighted material

Structuring an ASN.1 Specification

(Or: The Walls, Floors, Doorways and Elevators, with Some Environmental Considerations!)

Summary

ASN.1-based application specifications consist mainly of type definitions as illustrated in Chapter 2, but these are normally (and are formally **required to be**) grouped into collections called *modules*.

This chapter

- introduces the module structure,
- describes the form of module headers,
- shows how to identify modules, and
- describes how to export and import type definitions between modules.

The chapter also discusses

- some issues of publication format for a complete application specification, and
- the importance of making machine-readable copy of the ASN.1 parts available.

Part of the definition of a module is the establishment of

- a *tagging environment*, and
- an *extensibility environment*

for the type-notations appearing in that specification. The meaning and importance of these terms is discussed in this chapter, with final details in Section II.

Copyrighted material

This example forms what is called an *ASN.1 module* consisting of a six-line (in this—simple!—case) module header, a set of type (or value) assignment statements, and an "END" statement. This is the smallest legal piece of ASN.1 specification, and many early specifications were of this form—a single module. Today, it is more common for a complex protocol to be presented in a number of ASN.1 modules (usually within a single physical publication or set of Web pages). This is discussed further later.

Modules

All ASN.1 type and value assignments are required to appear within a `module`, starting with a module header and ending with "END".

It is very common in a real publication for the module header to appear at the start of a page, for there then to be up to 10 or more pages of type assignments (with the occasional value assignment perhaps), and then the END statement, which terminates the module. Normally there would be a page-break after the END statement in a printed specification, whether followed by another module or not.

However, Figure 3.1 is typical of early ASN.1 specifications, where the total protocol specification was probably only a few pages of ASN.1, and a single self-contained module was used for the entire specification.

Note that while the use of new-lines and indentation at the start of this example is what is commonly used, the normal ASN.1 rule that white-space and new-lines are interchangeable applies here too—the module header could be on a single line.

We will look in detail at the different elements of the module header later in this chapter, but first we discuss a little more about publication style.

3.2 Publication Style for ASN.1 Specifications

Over the years, different groups have taken different approaches to the presentation of their ASN.1 specifications in published documents. Problems and variations stem from conflicting desires:

NOTE

The use of three lines of four dots in Figures 2.1 and 3.1 is not legal ASN.1! It is used in this book out of sheer laziness! In a real specification there would be a complete list of named and fully specified (directly or by type-reference-names) elements. In Figure 3.1, it is assumed that no further type-reference-names are used in the body of these types—they use only the built-in types of the language such as INTEGER, BOOLEAN, VisibleString, etc.

Copyrighted material

a. A wish to introduce the various ASN.1 types that form the total specification gradually (often in a "bottom-up" fashion), within normal human-readable text that explains the semantics of the different types and fields.

> You may want to consider adding line-numbers to your ASN.1 to help references and cross-references . . . but these are not part of the language!

b. A wish to have in the specification a complete piece of ASN.1 that conforms to the ASN.1 syntax and is ready to feed into an ASN.1 tool, with the type definitions in either alphabetical order of type-reference-name, or in a "top-down" order.

c. The desire not to repeat text, in order to avoid unintended differences, and questions of which text takes precedence if differences remain in the final product.

There is no one perfect approach—application designers must make their own decisions in these areas, but the following two subsections discuss some common approaches.

3.2.1 Use of Line Numbers

One approach is to give line numbers sequentially to the entire ASN.1 specification, as partly shown in Figure 3.2 (again, lines of four dots are used to indicate pieces of the specification that have been left out).

It is important to note that if this specification is fed into an ASN.1 tool, the line numbers have to be removed—they are not part of the ASN.1 syntax, and the writer knows of no tool that provides a directive to ignore them.

If you have tools to assist in producing it (and they exist), this line-numbered approach also makes it possible to provide a cross-reference at the end of the specification that gives, for each type-reference-name, the line number of the type assignment where it is given a type, followed by all the line numbers where that reference is used. **For a large specification, this approach is VERY useful to readers.** If you don't do this, then you may wish to reorder your definitions into alphabetical order.

Once you decide to use line numbers, there are two main possibilities. You can

- put the ASN.1 in only one place, as a complete specification (usually at the end), and use the line-numbers to reference the ASN.1 text from within the normal human-readable text that specifies the semantics, or

Copyrighted material

```
001 Wineco-ordering-protocol
002  { joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
003   wineco(43) modules(2) ordering(1)}
004 DEFINITIONS
005         AUTOMATIC TAGS ::=
006 BEGIN
007
008         Order-for-stock ::=  SEQUENCE
009                  {order-no       INTEGER,
010                   name-address   BranchIdentification,

....

....

159                 digest    OCTET STRING}
160
161 END
```

Figure 3.2 Module with line numbers.

- break the line-numbered ASN.1 into a series of "figures" and embed them in the appropriate place in the human-readable text, again using the line-numbers for more specific references.

The latter approach only works well if the order in which you have the type definitions (in the total specification) is the same as the order in which you wish to introduce and discuss them in the main text.

3.2.2 Duplicating the ASN.1 Text

A number of specifications have chosen to duplicate the ASN.1 text (usually but not necessarily without using line numbers). In this case the types are introduced with fragments of ASN.1 embedded in the human-readable text, and the full module specification with the module header and the "END" are presented as either the last clause of the document, or in an Appendix.

> You may choose to repeat your ASN.1 text, fragmented in the body of your specification and complete in an annex—but be careful the texts are the same!

Note that where ASN.1 text is embedded in normal human-readable text, it is highly desirable for it to be given a distinctive font. This is particularly important where the individual names of ASN.1 types or sequence (or set) elements or choice

Copyrighted material

alternatives are embedded in a sentence. Where a distinctive font is not possible, then use of italics or of quotation marks is common for such cases. (Quotation marks are generally used in this text.)

If ASN.1 text appears in more than one place, then it used to be common to say that the collected text in the Appendix "took precedence if there were differences". Today it is more common to say that "if differences are found in the two texts, this is a bug in the specification and should be reported as such".

3.2.3 Providing Machine-Readable Copy

An annex collecting together the entire ASN.1 is clearly better than having it totally fragmented within many pages of printed text, no matter how implementation is to be tackled.

> If your implementors use tools, they will want machine-readable copy: consider how to provide this, and to tell them where it is!

Prior to the existence of ASN.1 tools, the ASN.1 specification was there to tell an implementor what to code up, and would rarely need to be fed into a computer, so printed text sufficed. With the coming of ASN.1 compilers, which enable a major part of the implementation to be automatically generated directly from a machine-readable version of the ASN.1 specification, some attention is needed to the provision of such material.

Even if the "published" specification is in electronic form, it may not be easy for a user to extract the formal ASN.1 definition because of the format used for publication, or because of the need to remove the line numbers discussed in earlier text, or to extract the material from "figures".

Wherever possible, the "published" specification should identify an authoritative source of machine-readable text for the complete specification. This should currently (1998) be ASCII encoded, with only spaces and new-lines as formatting characters, and using character names (see Chapter 9) for any non-ASCII characters in value notations. It is, however, likely that the so-called UTF8 encodings (again see Chapter 9), allowing direct representation of any character, will become increasingly acceptable, indeed, preferable.

It is unfortunate that many early ASN.1 specifications were published by ISO and ITU-T. These organizations had a history of making money from sales of hard-copy specifications and did not in the early days provide machine-readable material.

Copyrighted material

```
001 Wineco-ordering-protocol
002  {joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
003    wineco(43) modules(2) ordering(1)}
004 DEFINITIONS
005         AUTOMATIC TAGS ::=
006 BEGIN
007
008         Order-for-stock ::=  SEQUENCE
009                   {order-no       INTEGER,
010                    name-address   BranchIdentification,

....
```

Figure 3.3 The module header.

wards compatibility and partly to take account of those who had difficulty in obtaining (or were too lazy to try to obtain!) a bit of the object identifier name-space.

It is relatively easy today to get some object identifier name-space to enable you to give world-wide unambiguous names to any modules that you write, but we defer a discussion of how to go about this (and of the detailed form of an object identifier value) to Section II. Suffice it to say that the object identifier values used in this book are "legitimate", and are distinct from others (legally!) used to name any other ASN.1 module in the world. If name-space can be obtained for this relatively unimportant book. . . !

The fourth and the sixth lines are "boiler-plate". They say nothing, but have to be there! No alternative syntax is possible. (The same applies to the "END" statement at the end of the module.)

The fifth line is one of several possibilities, and determines the "environment" of the module that affects the detailed interpretation of the type-notation (but not of type-reference-names) textually appearing within the body of the module.

Designers please note—Not only is it illegal ASN.1 to write a specification without a module header and an "END" statement, it can also be very ambiguous because the "environment" of the type-notation has not been determined.

So, what aspects of the "environment" can be specified, and what syntax is possible in this fifth line?

There are two aspects to the "environment", called (in this book) "the tagging environment" and "the extensibility environment". The reader will note that these both contain terms that we have briefly mentioned before, but have never properly explained. Please do not be disappointed, but the explanation here is again going to be partial—for a full discussion of these concepts you will need to go to Section II.

Copyrighted material

```
''''

        overseas [1] SEQUENCE
                {name      UTF8String,
                 type      OutletType,
                 location Address},
        warehouse [2] CHOICE
                {northern  [0] NULL,
                 southern [1] NULL}

....
```

Figure 3.4 A fragment of Figure 2.1.

the default tag for NULL, and the fact that this TLV does actually represent a NULL (or in other cases an INTEGER or a BOOLEAN, etc.) is now only **implied** by the tag in the "T" part—you need to know the type definition to recognize that [0] is in this case referring to a NULL. We say that we have "implicitly tagged the NULL". Similarly, the "overseas" "SEQUENCE" was implicitly tagged with tag "[1]".

But what about the tag we have placed on the "warehouse" "CHOICE"? There is a superficial similarity between "CHOICE" and "SEQUENCE" (they have almost the same following syntax), but in fact they are **very** different in their BER encoding. With "SEQUENCE", following elements are wrapped up in an outer-level TLV wrapper as described earlier, but with "CHOICE", we merely take any one of the TLV encodings for one of the alternatives of the "CHOICE", and we use that as the entire encoding (the TLV) for the "CHOICE" itself.

Where does that leave the tagging of "warehouse"? Well, at first sight, it will override the tag of the TLV for the "CHOICE" (which is either "[0]" or "[1]" depending on which alternative was selected) with the tag "[2]". **Think for a bit, and then recognize that this would be a BUST specification!** The alternatives were specifically given (by tagging the NULLs) distinct tags precisely so as to be able to know which was being sent down the line in an instance of communication, but now we are overriding **both** with a common value ("[2]")! This cannot be allowed.

To cut a long story short—two forms of tagging are available in ASN.1.

- **Implicit tagging**—(this is what has been described so far), where the new tag overrides the old tag and type information which was carried by the old tag is now only **implicit** in the encoding; **this cannot be allowed for a "CHOICE" type**; and

- **Explicit tagging**—we add a new TLV wrapper specifically to carry the new tag in the "T" part of this wrapper, and carry the entire original TLV

Copyrighted material

(with the old tag) in the "V" part of this wrapper; clearly this is OK for "CHOICE".

While implicit tagging is forbidden for "CHOICE" types (it is an illegal ASN.1 specification to ask for it), both implicit and explicit tagging can be applied to any other type. However, while explicit tagging retains maximum type information, and might help a dumb line monitor to produce a sensible display, it is clearly more verbose than implicit tagging.

Implicit tagging: overrides the "T" part

Explicit tagging: adds an extra TLV wrapper

Now, what do the different tagging environments mean?

3.3.2.1 An Environment of Explicit Tagging

With an *environment of explicit tagging*, all tags produce explicit tagging unless the tag (number in square brackets) is immediately followed by the keyword "IMPLICIT."

An environment of explicit tagging was the only one available in the early ASN.1 specifications, so it was common to see the word "IMPLICIT" almost everywhere, reducing readability. Of course, it was—and is—illegal to put "IMPLICIT" on a tag that is applied to a "CHOICE" type-notation, **or to a type-reference-name for such notation**.

3.3.2.2 An Environment of Implicit Tagging

An environment of implicit tagging only produces implicit tagging where it is legal—there is no need to say "EXPLICIT" on a "CHOICE".

With an *environment of implicit tagging*, all tags are applied as implicit tagging unless one (or both) of the following apply:

- The tag is being applied to a "CHOICE" type-notation or to a type-reference-name for such notation; or
- The keyword "EXPLICIT" follows the tag notation.

In the preceding cases, tagging is still explicit tagging. In practice most specifications written between about 1986 and 1995 specified an environment of implicit tagging in their module headers, and it was unusual to see either the keyword

Copyrighted material

"IMPLICIT" or the keyword "EXPLICIT" after a tag. Occasionally, EXPLICIT was used for reinforcement, and occasionally (mainly in the security world to guarantee an extra TLV wrapper) on specific types within an environment of implicit tagging.

3.3.2.3 An Environment of Automatic Tagging

The rules about explicit and implicit tagging add to what is already a complicated set of rules on when tagging is needed, and in the 1994 specification, partly to simplify things for the application designer, and partly because the new Packed Encoding Rules (PER) were not TLV-based and made little use of tags, the ability to specify an *environment of automatic tagging* was added.

> **Automatic Tagging**
>
> Set up this environment and forget about tags!

In this case, tags are automatically added to all elements of each sequence (or set) and to each alternative of a choice, sequentially from "[0]" onwards (separately for each "SEQUENCE", "SET", or "CHOICE" construction). They are added in an environment of implicit tagging EXCEPT that if tag-notation is present on any one of the elements of a particular "SEQUENCE" (or "SET") element or "CHOICE" alternative, then it is assumed that the designer has taken control, and there will be NO automatic application of tags. (The tag-notation that **is** present is interpreted in an environment of implicit tagging in this case.)

It is generally recommended today that "AUTOMATIC TAGS" be placed in the module header, and the designer can then forget about tags altogether. However (refer back to Figure 999 please!), there is a counterargument that "AUTOMATIC TAGS" can be more verbose than necessary in BER, and can give more scope for errors of implementation if ASN.1 tools are not used. You take your choice; I know what mine would be!

> **The Extensibility Marker**
>
> An ellipsis (or a pair) that identifies an insertion point where version 2 material can be added without affecting a version 1 system's ability to decode version 2 encodings.

3.3.3 The Extensibility Environment

We have already discussed the power of a TLV-style of encoding to allow additions of elements in version 2, with version 1 specifications able to skip and to ignore such additional elements. (This extensibility concept actually generalizes to things other than sequences and sets, but these are sufficient for now.)

Copyrighted material

If we are to retain some extensibility capability in ASN.1 **and** we are to introduce encoding rules that are less verbose than the TLV of BER (such as the new PER), then a designer's requirements for extensibility in his application specification have to be made explicit.

We also need to make sure not only that encoding rules will allow a version 1 system to find the end of (and perhaps ignore) added version 2 material, but also that the application designer clearly specifies the actions expected of a version 1 system if it receives such material.

To make this possible, the 1994 specification introduced an *extensibility marker* into the ASN.1 notation. In the simplest use of this, the type-notation "Order-for-stock" could be written as in Figure 3.5.

Here we are identifying that we require encoding rules to permit the later addition of outer-level elements between "urgency" and "authenticator", and additional enumerations, in version 2, without ill-effect if they get sent to version 1 systems. (Full details are in Section II.) (Should we have been happy to add the version 2 elements at the end after "authenticator", then a single ellipsis would have sufficed.)

The place where the ellipses are placed, and where new version 2 material can be safely inserted without upsetting deployed version 1 systems is called (surprise, surprise!) the *insertion point*. You are only allowed to have one insertion point in any given sequence, set, choice, etc.

The alert reader (you should be getting used to that phrase by now, but it is probably still annoying—sorry!) will recognize that in addition to warning encoding rules to make provision, it is also necessary to tell the version 1 systems what to do with added

```
Order-for-stock ::= SEQUENCE
    {order-no       INTEGER,
     name-address   BranchIdentification,
     details        SEQUENCE OF
                     SEQUENCE
                   {item    OBJECT IDENTIFIER,
                    cases   INTEGER},
     urgency        ENUMERATED
                   {tomorrow(0),
                    three-day(1),
                    week(2), ... }   DEFAULT week,
                    ... ,
                    ... ,
     authenticator Security-Type}
```

Figure 3.5 Order-for-stock with extensibility markers.

Copyrighted material

- When producing the version 2 specification, you have to actually insert the ellipses explicitly before your added elements—and you might forget!
- There is no provision (when this environment is used) for the presence of an exception specification with the extension marker, so all rules for the required behavior of version 1 systems in the presence of version 2 elements or values have to be generic to the entire specification.

Concluding advice: Think carefully about where you want extension markers and about the handling you want version 1 systems to give to version 2 elements and values (using exception specifications to localize and make explicit those decisions), but do not attempt a blanket solution using an environment of implied extensibility.

3.4 Exports/Imports Statements

It has taken more text to describe the effects of a six-line header than is contained in the ASN.1 Standard/Recommendation! And we are not yet done!

Exports/Imports Statements

A pair of optional statements at the head of a module that specify the use of types defined in other modules (import), or that make available to other modules types defined in this module (export).

Following the sixth line ("BEGIN") and (only) before any type or value assignment statements, we can include an *exports statement* (first) and/or an *imports statement*. These are usually regarded as part of the module header.

At this point it is important to highlight what has been only hinted at earlier: There is more in the ASN.1 repertoire of things that have reference names than just types and values, although these are by far the most important (or at least, the most prolific!) in most specifications.

Pre-1994 (only) we add *macro names*, and post-1994 we add names of *information object classes*, *information objects*, and *information object sets*. These can all appear in an export or an import statement, but for now we concentrate only on type-reference-names and value-reference-names.

An exports statement is relatively simple, and is illustrated in Figure 3.6, where we have taken our type definitions for "OutletType" and "Address", put them into a module of commonly used types, and exported them, that is to say, made them available for use in another module.

Copyrighted material

```
Wineco-common-types
      { joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
        wineco(43) modules(2) common(3)}
   DEFINITIONS
        AUTOMATIC TAGS ::=
   BEGIN

   EXPORTS  OutletType, Address, Security-Type;

   IMPORTS Security-Type FROM
          SET-module
       {joint-iso-itu-t internationalRA(23) set(42) module(6) 0};

   OutletType ::=  SEQUENCE
        { ....
          ....
          .... }

   Address ::= SEQUENCE
        { ....
          ....
          .... }

   .....

   END
```

Figure 3.7 The common types module (enhanced).

(a totally separate publication), and will be used in our “Wineco-common-types” module but also in our other modules. We import this for use in the “Wineco-common-types” module, but also export it again to make the imports clauses of our other modules simpler (they merely need to import from “Wineco-common-types”). This “relaying” of type definitions is legal.

This changes Figure 3.6 to Figure 3.7.

As with EXPORTS, the text between “IMPORTS” and “FROM” is a comma separated list of reference names. We will see how to import from more than one other module in the next figure.

Note at this point that if a type is imported from a module with a particular tagging or extensibility environment into a module with a different tagging or extensibility environment, the type-notation for that imported type continues to be interpreted with the environment of the module in which it was originally defined. This may seem obvious from the way in which the environment concept was presented, but

Copyrighted material

it is worth reinforcing the point—what is being imported is in some sense the "abstract type" that the type-notation defines, **not** the text of the type-notation.

3.5 Refining Our Structure

Now we are going to make quite a few changes. We will add a second top-level message (and make provision for more) called "Return-of-sales" defined in another module, and we will now include the "ABSTRACT-SYNTAX" statement (mentioned in Chapter 2) to define our new top-level type in yet another module. We will put that module first.

> **The Final Example**
>
> We now use several modules; we have a CHOICE as our top-level type, and we clearly identify it as our top-level type. We use an object identifier value-reference-name, we use APPLICATION class tags, we handle invalid encodings, we have extensibility at the top-level with exception handling. We are getting quite sophisticated in our use of ASN.1!

We will do a few more cosmetic changes to this top-level module, to illustrate some slightly more advanced features. We will

- use "APPLICATION" class tags for our top-level messages. This is not necessary, but is often done (see later discussion of tag classes),
- assign the first part of our long object identifiers to the value-reference-name "wineco-OID" and use that as the start of our object identifiers, a commonly used feature of ASN.1, and
- add text to "ABSTRACT-SYNTAX" to make clear that if the decoder detects an invalid encoding of incoming material our text will specify exactly how the system is to behave.

The final result is shown in Figure 3.8, which is assumed to be followed by the text of Figure 3.7. Have a good look at Figure 3.8, and then read the following text that "talks you through it".

Lines 001 to 006 are nothing new. Note that in lines 10 and 13 we will use "wineco-OID" (defined in lines 015 and 016) to shorten our object identifier value, but we are not allowed to use this in the module header, as it is not yet within scope, and the object identifier value must be written out in full.

Line 007 simply says that nothing is available for reference from other modules.

Copyrighted material

```
001     Wineco-common-top-level
002        { joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
003          wineco(43) modules(2) top(0)}
004     DEFINITIONS
005          AUTOMATIC TAGS ::=
006     BEGIN
007     EXPORTS ;
008     IMPORTS Order-for-stock FROM
009          Wineco-ordering-protocol
010        {wineco-OID modules(2) ordering(1)}
011              Return-of-sales FROM
012        Wineco-returns-protocol
013        {wineco-OID modules(2) returns(2)};
014
015     wineco-OID OBJECT IDENTIFIER ::=
016          { joint-iso-itu-t internationalRA(23)
017                      set(42) set-vendors(9) wineco(43)}
018     wineco-abstract-syntax  ABSTRACT-SYNTAX ::=
019               {Wineco-Protocol IDENTIFIED BY
020                                 {wineco-OID abstract-syntax(1)}
021                                 HAS PROPERTY
022                                 {handles-invalid-encodings}
023                                 --See clause 45.6 --    }
024
025     Wineco-Protocol ::= CHOICE
026          {ordering  [APPLICATION 1] Order-for-stock,
027           sales     [APPLICATION 2] Return-of-sales,
028           ... ! PrintableString : "See clause 45.7"
029          }
030
031     END
—New page in published spec.
032    Wineco-ordering-protocol
033    { joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
034      wineco(43) modules(2) ordering(1)}
035     DEFINITIONS
036          AUTOMATIC TAGS ::=
037     BEGIN
038     EXPORTS Order-for-stock;
039     IMPORTS OutletType, Address, Security-Type FROM
040                    Wineco-common-types
041              {wineco-OID modules(2) common (3)};
042
043     wineco-OID OBJECT IDENTIFIER ::=
044          { joint-iso-itu-t internationalRA(23)
045                      set(42) set-vendors(9) wineco(43)}
046
047     Order-for-stock ::=  SEQUENCE
048              { ....
....              ....
....              .... }
....
```

Copyrighted material

```
070     BranchIdentification ::= SET
071               { ::::
....                ....
....                ....}
....
....    ....
....
101     END
--New page in published spec.
102     Wineco-returns-protocol
103     { joint-iso-itu-t internationalRA(23) set(42)
104       set-vendors(9) wineco(43) modules(2) returns(2)}
105     DEFINITIONS
106         AUTOMATIC TAGS ::=
107     BEGIN
108     EXPORTS Return-of-sales;
109     IMPORTS OutletType, Address, Security-Type FROM
110                   Wineco-common-types
111               {wineco-OID modules(2) common (3)};
112
113     wineco-OID OBJECT IDENTIFIER ::=
114        {iso identified-organization icd-wineco(10)}
115
116     Return-of-sales ::=  SEQUENCE
117               { ....
....                ....
....                .... }
....
....    ....
....
139     END
```

Figure 3.8 (Last figure of this Chapter.)

Lines 008 to 013 are the imports we were expecting from our other two modules. Note the syntax here: If we had more types being imported from the same module, there would be a comma separated list as in line 039, but when we import from two different modules lines 011 to 013 just run on from lines 008 and 010 with no separator.

Lines 015 and 017 provide our object identifier value-reference-name with a value assignment. It is a (very useful!) curiosity of the value notation for object identifiers that it can begin with an object identifier value-reference-name, which "expands" into the initial part of a full object identifier value, and is then added to, as we see in lines 010, 013, and 020. If you are interested and want to jump ahead, the OID tree is more fully described in Chapter 8.

Lines 018 to 023 are the "piece of magic" syntax that defines the top-level type, names the abstract syntax, and assigns an object identifier value to it—something

Copyrighted material

obvious "infinite recursion") reasons be allowed to use "wineco-OID" in the "FROM" for that import, so we would end up writing out as much text (and repeating it in each module where we wish to do the import) as we have written in lines 015 to 017 and 043 to 045. What we have is about as minimal as we can get.

Lines 102 to 139 are our third module, structurally the same as 032 to 101, and introducing nothing new. The whole specification then concludes with the text of Figure 3.7, giving our "common-type" module, which we have already discussed.

3.6 Complete Specifications

As was stated earlier, there is no concept in ASN.1 of a "complete specification", only of correct (complete) modules, some of which may include an "ABSTRACT-SYNTAX" statement to identify a top-level type (or which may contain a top-level type identified in human-readable text).

In many cases if a module imports a type from some other module, the two modules will be in the same publication (loosely, part of the same specification), but this is not a requirement. Types can be imported from any module anywhere.

Suppose we take a top-level type in some module, and follow the chain of all the type-reference-names it uses (directly or indirectly) within its own module, and through import and export links (again chained to any depth) to types in other modules. This will give us the complete set of types that form the "complete specification" for the application for which this is the top-level type, and the specifications of all these types have (of course) to be available to any implementor of that application **and to any ASN.1 compiler tool assisting in the implementation.** Purely for the purposes of the final part of this chapter of this book, this tree of type definitions will be called the *application-required types*.

It is important advice to any application designer to make it very clear early in the text of any application specification precisely which additional (physical) documents are required to obtain the definitions of all the application-required types.

But suppose we now consider the set of modules in which these application-required types were defined. (Again, purely for the next few paragraphs, we will call these the *application-required modules*).

In general, the module textually containing the top-level type **probably** does not contain any types other than those that are application-required types (although there is no requirement that this be so). But as soon as we start importing, particularly from modules in other publications that were perhaps produced to satisfy

Copyrighted material

The Basic Data Types and Construction Mechanisms: Closure

(Or: You Need Bricks of Various Shapes and Sizes!)

Summary

There are a number of types that are predefined in ASN.1, such as

- INTEGER,
- BOOLEAN, and
- UTF8String.

These are used to build more complex user-defined types with construction mechanisms such as

- SEQUENCE,
- SET,
- CHOICE,
- SEQUENCE OF,
- SET OF, and
- etc.

Many of these construction mechanisms have appeared in the examples and illustrations of earlier chapters.

This chapter completes the detailed presentation of all the basic ASN.1 types, giving in each case a clear description of

Copyrighted material

- the type-notation for the type,
- the set of abstract values in the type, and
- the value-notation for values of that type.

Additional pieces of type/value-related notation are also covered, largely completing the discussion of syntax commonly used in pre-1994 specifications.

The chapter ends with a list of additional concepts whose treatment is deferred to either the next chapter or to Section II.

4.1 Illustration by Example

> Figure 4.1 has been carefully constructed to complete your introduction to all the basic ASN.1 types—that's it folks!

In order to illustrate some of the type and value notations, we will define our Return-of-Sales message as in Figure 4.1. Figure 4.1 has been designed to include all the basic ASN.1 types apart from NULL, and provides the hook for further discussion of these types.

```
Return-of-sales ::= SEQUENCE
      {version         BIT STRING
                 {version1 (0), version2 (1)} DEFAULT {version1},
       no-of-days-reported-on   INTEGER
              {week(7), month (28), maximum (56)} (1..56) DEFAULT week,
       time-and-date-of-report  CHOICE
                   {two-digit-year  UTCTime,
                    four-digit-year GeneralizedTime},
                 -- If the system clock provides a four-digit year,
                 -- the second alternative shall be used.  With the
                 -- first alternative the time shall be interpreted
                 -- as a sliding window.
       reason-for-delay  ENUMERATED
              {computer-failure, network-failure, other} OPTIONAL,
                 -- Include this field if and only if the
                 -- no-of-days-reported-on exceeds seven.
       additional-information  SEQUENCE OF PrintableString OPTIONAL,
                 -- Include this field if and only if the
                 -- reason-for-delay is "other".
       sales-data  SET OF Report-item,
       ... ! PrintableString : "See wineco manual chapter 15"}
```

Figure 4.1 (Part 1) Illustration of the use of basic ASN.1 types.

Copyrighted material

```
Report-item ::= SEQUENCE
    {item                   OBJECT IDENTIFIER,
     item-description      ObjectDescriptor OPTIONAL,
         -- To be included for any newly-stocked item.
     bar-code-data          OCTET STRING,
         -- Represents the bar-code for the item as specified
         -- in the wineco manual chapter 29.
     ran-out-of-stock   BOOLEAN DEFAULT FALSE,
         -- Send TRUE if stock for item became exhausted at any
         -- time during the period reported on.
     min-stock-level        REAL,
     max-stock-level        REAL,
     average-stock-level   REAL
       -- Give minimum, maximum, and average levels during the
       -- period as a percentage of normal target stock-level-- }

wineco-items OBJECT IDENTIFIER ::=
     { joint-iso-itu-t internationalRA(23) set(42) set-vendors(9)
wineco(43) stock-items (0)}
```

Figure 4.1 (Part 2) Illustration of the use of basic ASN.1 types.

Have a good look at Figure 4.1. It should by now be fairly easy for you to understand its meaning. If you have no problems with it, you can probably skip the rest of this chapter, unless you want to understand ASN.1 well enough to write a book, or to deliver a course, on it! (We included wineco-items in Figure 4.1 to reduce the verbosity of the object identifier values in Figure 4.2 later.)

4.2 Discussion of the Built-In Types

4.2.1 The BOOLEAN Type

(See "ran-out-of-stock" in Figure 4.1). There is nothing to add here. A "BOOLEAN" type has the obvious two abstract values, true and false, but notice that the value-notation is the words "TRUE" or "FALSE" (**all in capital letters**). You can regard the use of capitals as either consistent with the fact that (almost) all the built-in names in ASN.1 are all uppercase, or as inconsistent with the fact that ASN.1 **requires** value-reference-names to begin with a lowercase letter! ASN.1 does not always obey its own rules!

4.2.2 The INTEGER Type

(See "number-of-days-reported-on" in Figure 4.1). This example is a little more complicated than the simple use of "INTEGER" that we saw in Figure 2.1. The

Copyrighted material

example here contains what are called *distinguished values*. In some early ASN.1 specifications (ENUMERATED was not added until around 1988) people would sometimes use the "INTEGER" type with a list of distinguished values, whereas today they would use "ENUMERATED". In fact, the syntax can look quite similar, so we can write the equivalent of the example in Figure 2.1 as:

```
urgency  INTEGER
        {tomorrow (0),
         three-day (1),
         week (2)}  DEFAULT week
```

It is, however, important here to notice some important differences. The presence of the list following "INTEGER" is entirely optional (for "ENUMERATED" it is required), and the presence of the list in no way affects the set of abstract values in the type.

The following two definitions are **almost** equivalent:

```
My-integer ::= INTEGER {tomorrow(0), three-day (1), week(2) }
```

and

```
My-integer ::= INTEGER
tomorrow My-integer ::= 0
three-day My-integer ::= 1
week My-integer ::= 2
```

The Integer Type

- Just the word INTEGER, nice and simple!; and/or,
- Add a distinguished value list; and/or,
- Add a range specification (subtyping); then,
- Put an extension marker and exception specification in the range specification. (Getting complicated again!)

The difference lies in ASN.1 scope rules. In the second example, the names "tomorrow" etc. are value-reference-names that can be assigned only once within the module, can be used anywhere within that module where an integer value is needed (even, in fact, as the number on an enumeration or in another distinguished value list or in a tag—but all these uses would be unusual!), and can appear in an EXPORTS statement at the head of the module. On the other hand, in the first example, the

Copyrighted material

range constraint and no clarifying text, it is usually a safe assumption that a four-octet integer value will be the largest you will receive.

One final point: The similarity of the syntax for defining distinguished values to that for defining enumerations can be confusing. As the definition of distinguished values does not change in any way the set of abstract values in the type or the way they are encoded, there is never any "extensibility" question in moving to version 2—if additional distinguished values are added, this is simply a notational convenience and does not affect the bits on the line. So the ellipsis extensibility marker (available for the list in the enumerated type), is neither needed nor allowed in the list of distinguished values (although it can appear in a range constraint, as we will see later).

Enumerated

Can have an extension marker.

Numbers for encodings needed pre–1994, optional post–1994.

4.2.3 The ENUMERATED Type

(See "urgency" in Figure 2.1 and "reason-for-delay" in Figure 4.1). There is little to add to our earlier discussions. The numbers in round brackets were required pre-1994, and are optional post-1994. The type consists precisely and only of values corresponding to each of the listed names.

The numbers were originally present to avoid extensibility problems; if version 2 added a new enumeration, it was important that this should not affect the values used (in encodings) to denote original enumerations, and the easiest way to ensure this was to let the application designer list the numbers to be used. Post-1994, extensibility is more explicit, and we might see:

```
Urgency-type ::= ENUMERATED
        {tomorrow,
         three-day,
         week,
         ...,
          -- Version 1 systems should assume any other value
          -- means "week".
         month}
```

Here "month" was added in version 2, although the requirement placed on version 1 systems when version 1 was first specified actually means that such

Copyrighted material

a simple means of finding the end of the notation). The mathematical value being identified by {mantissa x, base y, exponent z} is (x times (y to the power z)), but y is allowed to take **only** the values 2 and 10.

There are also explicitly included (and encoded specially) two values with the following value notation:

```
PLUS-INFINITY
MINUS-INFINITY
```

Again, all uppercase letters. When "REAL" was first introduced, there was discussion of adding additional special "values" such as "OVERFLOW", or even "PI" etc., but this never happened.

That is really all you need to know, as the "REAL" type is infrequently used in actual application specifications. The rest of the discussion of the "REAL" type is a bit academic, and you can omit it without any "real" damage. However, if you want to know which of v1 to v7 represent the same abstract value and which different ones, read on!

You might expect from the name that the abstract values are (mathematical) real numbers, but for those of a mathematical bent, only the rationals are included.

Formally, the type contains two sets of abstract values, one set comprising all the numbers with a finite representation using base 10, and the other set comprising all the numbers with a finite representation base 2. (Notice that from a purely mathematical point of view, the latter values are a strict subset of the former, but the former contains values that are not in the latter set). In all ASN.1 encoding rules, there are *binary encodings* for "REAL", and there are also *decimal encodings* as specified in the ISO standard 6093. This standard specifies a character string to represent the value, which is then encoded using ASCII. An example of these encodings is:

```
56.5E+3
```

but ISO 6093 contains many options!

It is possible (post-1994) to restrict the set of abstract values in "REAL" to be only the base 10 or only the base 2 set, effectively giving the application designer control over whether the binary or the decimal encoding is to be used. Where the type is unrestricted, it is theoretically possible to put different application semantics on a base 10 value from that on the mathematically equal base 2 value, but probably no one would do so! (Actually, "REAL" is not used much anyway in real protocols.)

Copyrighted material

To wrap this discussion up—looking at the forementioned values v1 to v7, we can observe that the value-reference-names listed on the same line below are value notation for the same abstract value, and those on different lines are names for different abstract values:

```
v1, v2
v3
v4
v5, v6
v7
```

(v5 equals v6 because v5 is defined to represent the base2 value zero.)

4.2.5 The BIT STRING Type

(See "version" in Figure 4.1). There are two main uses of the BIT STRING type. The first is that given for "version", where we have a list of *named bits* associated with the type. The second and simplest is the type-notation:

```
BIT STRING
```

> BIT STRING is often used with named bits to support a bit-map for version negotiation.

Note that, as we would expect, this is all uppercase, but as we might not expect, the name of the type (effectively a type-reference-name) contains a space. The space is not merely permitted, it is **required**! Again ASN.1 breaks its own rules!

We will return to Figure 4.1 in a moment. Let us take the simpler case where there is no list of named bits.

If a field of a sequence (say) is defined as simply "BIT STRING", then this can be a sign of an inadequately specified protocol, as semantics need to be applied to any field in a protocol. "BIT STRING" with no further explanation is one of several ways in which "holes" can legally be left in ASN.1 specifications, but to the detriment of the specification as a whole.

We will see later that where any "hole" is left, it is important to provide fields that will clearly identify the content of the hole in an instance of communication, and to either ensure that all communicating partners will understand all identifications (and the resulting contents of the hole), or will know what action to take on an unknown identifier. ASN.1 makes provision for such "holes" and the associated

Copyrighted material

identification and it is not a good idea to use "BIT STRING" to grow your own "holes" (but some people do)!

So, BIT STRING without named bits has a legitimate use to carry encodings produced by well-identified algorithms, and in particular to carry encryptions for either concealment or signature purposes. But even in this case, there is usually a need to clearly identify the security algorithm to be applied, and perhaps to indirectly reference specific keys that are in use. The BIT STRING data type is (legitimately) an important building block for those providing security enhancements to protocols, but further data is usually carried with it.

> BIT STRING without named bits is also frequently used as part of a more complex structure to carry encrypted information.

The use of BIT STRING with named bits as for "version" in Figure 4.1 is common. The names in curly brackets simply provide names for the bits of the BIT STRING and the associated bit number. It is important to note that the presence of a named bit list (as with distinguished values for integers), does not affect the type. The list in no way constrains the possible length of the BIT STRING, nor do bits have to be named in order.

ASN.1 talks about "the leading bit" as "bit zero", down to the "trailing bit". Encoding rules map the "leading bit" to the "trailing bit" of a bit-string type into octets when encoding.

(BER—arbitrarily, it could have chosen the opposite rule—specifies that the leading bit be placed in the most significant bit of the first octet of the encoding, and so on.)

How are these names of bits used? As usual, they can provide a handle for reference to specific bits by the human-readable text. They can also, however, be used in the value notation.

The obvious (and simplest) value notation for a BIT STRING is to specify the value in binary, for example:

```
'101100110001'B
```

If the value is a multiple of four bits, it is also permissible to use hexadecimal:

```
'B31'H
```

(Note that in ASN.1 hexadecimal notation, only uppercase letters are allowed.)

Copyrighted material

If, however, there are named bits available, then an additional value notation is available, which is a comma-separated list of bit-names within curly brackets (see, for example, the "DEFAULT" value of "version" in Figure 4.1). The value being defined is one in which **the bit for every listed bit-name is set to one, and all other bits are set to zero.**

The alert reader (I have done it again!) will spot that this statement is not sufficient to define a BIT STRING value, as it leaves undetermined how many (if any) trailing zero bits are present in the value. So, the use of such a "value-notation" if the length of the BIT STRING is not constrained does not really define a value at all—it defines a set of values! All those with the same one bits, but zero to infinity trailing zero bits.

The ASN.1 specifications post-1986 (or so) circumvent this problem with some "weasel" words (slightly changed in different versions): "If a named bit list is present, trailing zero bits shall have no semantic significance"; augmented later by "encoding rules are free to add (or remove) trailing zero bits to (or from) values that are being encoded!"

This issue is not a big one for normal BER, where it does not matter if there is doubt over whether some value exactly matches the "DEFAULT" value, but it matters much more in the canonical encoding rules to be described later.

The most common use for named bits is as a "version" map, as illustrated in Figure 4.1. Here an implementation would be instructed to set the bits corresponding to the versions that it is capable of supporting, and—typically—there would be some reply message in which the receiver would set precisely one bit (one of those set in the original message), or would send some sort of rejection message.

4.2.5.1 Formal/Advanced Discussion

NOTE — Most readers should skip this next bit! Go on to OCTET STRING, which has fewer problems! If you insist on reading on, **please** read Figure 999 again!

There have been many different texts in the ASN.1 specifications over the last 15 years associated with "BIT STRING" definitions with named bits. Most have been constrained by the desire:

a. not **really** to change what was being specified, or at least, not to break current deployed implementations; and

Copyrighted material

rather abstract(!) problem was first understood.) Existing BER implementations will frequently include trailing zero bits in the encoding of a value of a BIT STRING type with a named-bit list.

For canonical encoding rules, however, including PER, a single encoding is necessary, and at first sight saying that such encoding rules never have trailing bits in the encoding looks like a good solution.

But the choice of encoding (and indeed the selection of the precise abstract BIT STRING value—from the set of abstract values with the same semantics—that is to be used for encoding) is complicated if there are **length constraints** at the abstract level on the bit-string type.

The matter is further complicated because in BER-related encoding rules, length constraints are "not visible" that is, they do not affect the encoding. In PER, they may or may not be visible.

The upshot of all this is that in the canonical versions of BER **trailing zero bits are never transmitted in an encoding**, but the value delivered to the application is required to have sufficient zero bits added (the minimum necessary) to enable it to satisfy any length constraints that might have been applied. (Such constraints are assumed to be visible to the application and to the Application Program Interface (API) code, whether they are visible to—affect—the encoding rules or not.)

However, PER, where (some) length constraints are PER-visible, changes this slightly: What is transmitted is always consistent with PER-visible constraints, so (the minimum number of) trailing zero bits are present in transfer if they are needed to satisfy a length constraint. The encoding can thus be delivered to the application unchanged, provided there are no not-PER-visible constraints applied, otherwise the canonical BER rules would apply; the application gets a value that is permitted by the constraints and carries the same application semantics as that derived directly from the transmitted encoding.

And if you have read this far, I bet you wish you hadn't! It kind of all works, but it is **not** simple!

Issues like this do not affect the normal application designer—just do the obvious things and it will all work; nor do they affect the normal implementor who obeys the well-known rules: encode the obvious encoding; and be **liberal** in your decoding.

These issues **are**, however, of importance to tool vendors who provide an option for "strict diagnostics" if incoming material is perceived to be erroneous. In such cases a very precise statement of what is "erroneous" is required!

Copyrighted material

> For NULL, you know it all—a placeholder: no problems.

The normal use is very much as in Figure 2.1, where we need a type to provide a TLV (whose presence or absence carries some semantics), but where there is no additional information to be carried with the type. NULL is often referred to as a "placeholder" in ASN.1 courses.

4.2.8 Some Character String Types

(See "additional-information" in Figure 4.1 and "name" (twice) in Figure 2.1). In the examples so far, you have met "PrintableString" (present in the earliest ASN.1 drafts), "VisibleString" (deprecated synonym "ISO646String"), and "UTF8String" (added in 1998). There are several others.

Despite not being all-uppercase, these (and the other character string type names) have been reserved words (names you may not use for your own types) since about 1988/90. The early designers of ASN.1 felt (rightly!) that the character string types and their names were a bit ad hoc, and gave them a somewhat reduced status!

Actually, in the earliest ASN.1 specification, there was the concept of "Useful Types", that is, types that were defined using the ASN.1 notation rather than pure human language, and these all used mixed upper/lowercase. The character string types were originally included as "Useful types", and were defined as a tagged OCTET STRING. Today (since about 1990 when they became reserved words) they are regarded as fairly fundamental types with a status more or less equal to that of INTEGER or BOOLEAN.

The set of characters in "PrintableString" values is "hardwired" into ASN.1, and is roughly the old **telex** character set, plus lowercase letters. The BER encoding in the "V" part of the TLV is the ASCII encoding, so the reduced character set over "VisibleString" (following) is not really useful, although a number of application specifications do use "PrintableString".

The set of characters in "VisibleString" values is simply the printing ASCII characters plus "space". The BER encoding in the "V" part of the TLV is, of course, ASCII.

The set of characters in "UTF8String" is any character—from Egyptian hieroglyphs to things carved in wood in the deepest Amazon jungle to things that we will in due course find on Mars—that has been properly researched and documented (including the ASCII control characters). The BER (and PER if the type is not constrained to a reduced character set) encoding per character is variable length, and has the

Copyrighted material

"nice" property that for ASCII characters the encoding per character is one octet, stretching to three octets for all characters researched and documented so far, and going to at most seven octets per character once we have all the languages of the galaxy in there! Those who are "into" character set stuff may recognize the name "Unicode". UTF8 is an encoding scheme covering the whole of Unicode (and more) that is becoming (circa 1999) extremely popular for communication and storage of character information. (**Advice**: If you are designing a new protocol, use UTF8String for your character string fields unless you have a **very** good reason not to do so.)

4.2.9 The OBJECT IDENTIFIER Type

(See "item" and "wineco-items" in Figure 4.1, and module identifiers in Figure 3.8.) Values of the object identifier type have been used and introduced from the start of this book. But we are still going to postpone to a later chapter a detailed discussion of this type.

> OBJECT IDENTIFIER—Perhaps used more than any other basic ASN.1 type—you can get some name-space in lots of ways, but you don't really need it!

The OBJECT IDENTIFIER type may well lay claim to being the most used of all the ASN.1 types (excluding the constructors SEQUENCE, SET, and CHOICE, of course). Wherever world-wide unambiguous identification is needed in an ASN.1-based specification, the object identifier type is used.

Despite the apparent verbosity of the value-notation, the encoding of values of type object identifier is actually very compact (the human-readable names present in the value notation do not appear in the encoding). For the early components of an object identifier value, the mapping of names to integer values is "well-known", and for later components in any value-notation, the corresponding integer value is present (usually in round brackets).

The basic name space is a hierarchically allocated tree-structure, with global authorities responsible for allocation of top-level arcs, and progressively more local authorities responsible for the lower-level arcs.

For you (as an application designer) to be able to allocate values from the object identifier name space, you merely need to "get hung" from this tree. It really does not matter where you are "hung" from (although encodings of your values will be shorter the nearer you are to the top, and international organizations tend to be sensitive about where they are "hung"!).

Copyrighted material

For a standards-making group, or a private company, or even an individual, there is a range of mechanisms for getting some of this name-space, most of which require no administrative effort (you probably have an allocation already). These mechanisms are described later, although such is the proliferation of branches of the OID tree (as it is often described) that it is hard to describe all the finer parts.

It has been a criticism of ASN.1 that you need to get some OID space to be able to write: ASN.1 modules. This is actually not true—the module identifier is **not** required. However, most people producing ASN.1 modules **do** (successfully) try to get a piece of the OID space and **do** identify their modules with OID values. But, if this provides you with problems, it is **not** a requirement.

4.2.10 The ObjectDescriptor Type

(See "item-description" in Figure 4.1). The type-notation for the ObjectDescriptor type is:

```
ObjectDescriptor
```

without a space, and using mixed uppercase and lowercase! This is largely a historical accident. This type was formally defined as a tagged "GraphicString" (another character string type capable of carrying most of the world's languages, but regarded as obsolete today). Because its definition was by an ASN.1 type-assignment statement, it was deemed originally to be merely a "Useful Type", and was given a mixed upper/lowercase name with no space. Today, the term "Useful Type" is not used in the ASN.1 specification, and the use of mixed case for this built-in type is a bit of an anachronism.

> **ObjectDescriptor**
>
> Yes, mixed case! You will never see it in a specification, and you are unlikely to want to use it—ignore this text!

The existence of the type stems from arguments over the form of the OBJECT IDENTIFIER type. There were those who (successfully) argued for an identification mechanism that produced short, numerical identifiers when encoded on the line. There were others who argued (unsuccessfully) for an identification mechanism that was "human-friendly", and contained a lot of text (for example, something like a simple ASCII encoding of the value notation we have met earlier), and perhaps no numbers. As the debate developed, a sort of compromise was reached that involved the introduction of the "OBJECT IDENTIFIER" type—short, numerical, guaranteed to be unambiguous world-wide, but supplemented by an additional

Copyrighted material

type "ObjectDescriptor" that provided an indefinitely long (but usually around 80 characters) string of characters plus space to "describe" an object. The "ObjectDescriptor" value is not in any way guaranteed to be unambiguous worldwide (the string is arbitrarily chosen by each designer wishing to describe an object), but because of the length of the string, usually it **is** unambiguous.

There is a strong recommendation in the ASN.1 specification that whenever an object identifier value is allocated to identify an object, an object descriptor value should also be allocated to describe it. It is then left for application designers to include in their protocol (when referring to some object) either an "OBJECT IDENTIFIER" element only, or both an "OBJECT IDENTIFIER" and an "ObjectDescriptor", perhaps making the inclusion of the latter "OPTIONAL".

In practice (apart from the artificial example of Figure 4.1) you will never encounter an "ObjectDescriptor" in an application specification! Designers have chosen not to use it. Moreover, the rule that whenever an object identifier value is allocated for some object, an object descriptor value should also be assigned, is frequently broken.

Take the most visible use of object identifier values—in the header of an ASN.1 module: What is the corresponding object descriptor value? It is not explicitly stated, but most people would say that the **module name** appearing immediately before the object identifier in the header forms the corresponding object descriptor. Well—OK!

But there are other object identifier values originally assigned in the ASN.1 specification itself, such as:

```
{iso standard 8571}
```

This identifies the numbered standard (which is actually a multipart standard), and also gives object identifier name-space to those responsible for that standard. There is, however, no corresponding object descriptor value assigned.

4.2.11 The Two ASN.1 Date/Time Types

Yes, you did indeed interpret Figure 4.1 correctly—UTCTime is a date/time type that carries only a two-digit year!

You will also notice that both "UTCTime" and "GeneralizedTime" are again mixed upper/lowercase. Again this is a historical accident: They were defined using an ASN.1 type-assignment statement as a tagged "VisibleString", and were originally listed as "Useful Types".

Copyrighted material

to any that are more than 49 years in the future), this system clearly works, and allows two-digit years to be used indefinitely. A neat solution!

What does "UTC" stand for? It comes from the Consultative Committee on International Radio (CCIR) and stands for "Coordinated Universal Time" (the curious order of the initials comes from the name in other languages). In fact, despite the different name, "GeneralizedTime" also records Coordinated Universal Time. What is this time standard? Basically, it is Greenwich Mean Time, but for strict accuracy, Greenwich Mean Time is based on the stars and there is a separate time standard based on an atomic clock in Paris. Coordinated Universal Time has individual "ticks" based on the atomic clock, but from time to time it inserts a "leap-second" at the end of a year (or at the end of June), or removes a second, to ensure that time on a global basis remains aligned with the earth's position in its orbit around the sun. This is, however, unlikely to affect any ASN.1 protocol!

What is the exact set of values of UTCTime? The values of the type are character strings of the following form:

```
yymmddhhmmZ
yymmddhhmmssZ
yymmddhhmm+hhmm
yymmddhhmm-hhmm
yymmddhhmmss+hhmm
yymmddhhmmss-hhmm
```

"yymmdd" is year (00 to 99), month (01 to 12), day (01 to 31), and "hhmmss" is hours (00 to 23), minutes (00 to 59), seconds (00 to 59).

The "Z" is a commonly used suffix on time values to indicate "Greenwich Mean Time" (or UTC time), others being "A" for one hour ahead, "Y" for one hour behind, etc., but these are **NOT** used in ASN.1.

If the "+hhmm" or "-hhmm" forms are used (called a *time differential*), then the first part of the value expresses **local** time, with UTC time obtained by **subtracting** the "hhmm" for "+hhmm", and **adding** it for "-hhmm". The ASN.1 specification contains the following example (another example, added in 1994 shows a "yy" of "01" representing 2001!):

```
If local time is 7am on 2 January 1982 and coordinated universal time
is 12 noon on 2 January 1982, the value of UTCTime is either of
```

Copyrighted material

The ASN.1 specification talks about "The selection type", but the heading in this clause is more accurate—this is a piece of notation more akin to "IMPORTS" than to a type definition and it references an existing definition.

The SELECTION TYPE notation—you are unlikely ever to see this—forget it!

The selection-type notation takes the following form:

```
identifier-of-a-choice-alternative < Type-notation-for-a-CHOICE
```

For example, given:

```
Example-choice ::= CHOICE
  {alt1     Type1,
   alt2     Type2,
   alt3     Type3}
```

then the following type-notation can be used wherever type-notation is required within the scope (module) in which "Example-choice" is available:

```
              alt1 < Example-choice
or            alt2 < Example-choice
or            alt3 < Example-choice
```

This notation references the type defined as the named alternative of the identified choice type, and should be seen as another form of type-reference-name. Notice that if the selection-type notation is in a module different from that in which "Example-choice" was originally defined, any tagging or extensibility environment applied to the referenced type is that of the module containing the original definition of Example-choice, **not** that of the selection-type notation.

Value notation for "a selection type" is just the value notation for the selected type.

In other words, for the type-notation "alt3 < Example-choice", the value-notation is the value-notation for "Type3". (The identifier "alt3" does not appear in the value-notation for the "selection type", nor are there any colons present.)

4.3.2 The COMPONENTS OF Notation

This is another example of a rarely used piece of notation that references the inner part of a sequence or set. The only reason to use it is that you can avoid an extra TLV wrapper in BER. It is not illustrated in Figure 4.1.

Copyrighted material

What follows is described in relation to "SEQUENCE", but applies equally to "SET". However, a "COMPONENTS OF" in a "SEQUENCE" must be followed by type-notation for a sequence-type (which remember may, and usually will, be a type-reference-name), and similarly for SET.

> The COMPONENTS OF notation—you won't often see this either, so forget this too!

Suppose we have a collection of elements (identifiers and type-notation) that we want to include in quite a few of the sequence types in our application specification. Clearly we do not want to write them out several times, for all the obvious reasons. We could, of course, define a type:

```
Common-elements ::= SEQUENCE
    {element1   Type1,
     element2   Type2,
     ....
     element23   Type23}
```

and include that type as the first (or last) element of each of our "actual" sequences:

```
First-actual-sequence ::=  SEQUENCE
   {used-by-all  Common-elements,
    next-element Some-special-type,
    next-again   Special2,
    etc           The-last}
```

We do the same for all the sequences in which these common elements are needed. That is fine (and with PER it really **is** fine!). But with BER, if you recall the way it works, we get an outer-level TLV for "First-actual-sequence", and in the "V" part a TLV for each of its elements, and in particular a TLV for the "used-by-all" element. Within the "V" part of that we get the TLVs for the elements of "Common-elements". But if we had textually **copied** the body of "Common-elements" into "First-actual-sequence", there would be no TLV for "Common-elements"—we would have saved (with BER) two or three, perhaps even four, octets!

If we use "COMPONENTS OF", we can write:

```
First-actual-sequence ::=  SEQUENCE
   {              COMPONENTS OF Common-elements,
    next-element Some-special-type,
    next-again   Special2,
    etc           The-last}
```

Copyrighted material

The "COMPONENTS OF" notation provides for such copying without textually copying, it "unwraps" the sequence type it references.

Note that there is no identifier on the "COMPONENTS OF element". This is not optional—the "identifier" **must** be omitted. The "COMPONENTS OF" is not really an element of the SEQUENCE, it is a piece of notation that extracts or unwraps the elements. It is often referred to as "textual substitution", but that is not quite correct (alert reader!) because the tagging and extensibility environment for the extracted elements remains that of the module where they were originally defined.

There is some complexity if automatic tagging is applied and COMPONENTS OF is used. The reader has two choices: just forget it and note that it all works (unless you are a hand-coding implementor, in which case see the next option!), or as a good exercise (none are formally set in this book!) go to the ASN.1 specification and work out the answer!

An application designer can generally choose to use SEQUENCE or SET more or less arbitrarily. Read this text then use SEQUENCE always!

4.3.3 SEQUENCE or SET?

The type-notation for SEQUENCE, SET, SEQUENCE OF and SET OF has been well illustrated in earlier text and examples, together with the use of "DEFAULT" and "OPTIONAL". Remember that in BER (not CER/DER/PER), the default value is essentially advisory. An encoder is permitted to encode explicitly a default value, or to omit the corresponding TLV, entirely as an encoder's option.

We have already discussed briefly the differences between

```
SEQUENCE { .... }    and    SET { .... }
```

from an encoding point of view in BER (the TLVs are in textual order for SEQUENCE, in an order chosen by the encoder for SET), and also from the more theoretical standpoint that "order is not semantically significant" in SET.

The problem is that if we regard the abstract value as a collection of unordered information, and we want a single bit pattern to represent that in an encoding, we have to invent some more or less arbitrary criteria to order the collection in order to form a single bit-pattern encoding. This can make for expensive (in CPU and perhaps also in memory terms) encoding rules. In the case of SET { .... }, if we want to remove encoder options, it is possible to use either textual order (not really a good idea) or tag order (tags are required to be distinct among the elements in a SET) to

Copyrighted material

provide the ordering as a static decision. However, in the case of "SET OF", no one has found a way of providing a single bit pattern for a complete SET OF value without doing a run-time sort of the encodings of each element. This can be expensive!

We will return to this point when we discuss the canonical (CER) and distinguished (DER) encoding rules in Section III, but advice today (but see figure 999!) would be: Best to keep off "SET {", and avoid "SET OF" like the plague!

One very small detail to mention here: the default tag provided for "SET {" and for "SET OF" is **the same**. It is different from that provided for "SEQUENCE {" and for "SEQUENCE OF", but these are also the same. This only matters if you are carefully applying tags within CHOICEs and SETs etc. with the minimal application of tags. In this case you will have studied and be happy with later text on tagging, and will carefully check the ASN.1 specification to determine the default tag for all types! If you are a normal mortal, however, you will routinely apply tags to everything (pre-1994), or will use "AUTOMATIC TAGS" (post-1994), and the fact that the default tag for "SEQUENCE {" is the same as that for "SEQUENCE OF" will not worry you in either case!

4.3.4 SEQUENCE, SET, and CHOICE (Etc.) Value-Notation

We have used the type notation for these constructions almost from the first page of this book, but now we need to look at their value-notation. (Actually, you will never encounter this except in courses or an illustrative annex to the ASN.1 specification, but it reinforces the point that for any type you can define with ASN.1 there is a well-defined notation for all of its values.)

> **SEQUENCE, SET, CHOICE, Etc. Value-Notation**
>
> You won't ever need to write it, and will only ever read it in courses and ASN.1 tutorials and books like this, but here it is. It is good to complete your education!

To say it simply: value notation for "SET {" and "SEQUENCE {" is a pair of curly braces containing a comma-separated list. Each item in the list is the identifier for an element of the "SEQUENCE {" (taken in order) or "SET {" (in any order), followed by value-notation for a value of that element. Of course this rule is recursively applied if there are nested "SEQUENCE {" constructs.

For "SET OF" and "SEQUENCE OF" we again get a pair of curly braces containing a comma-separated list, with each item being the value notation for a value of the type-notation following the "OF".

Finally, for "CHOICE" it is NOT what you might expect, there are no curly braces! Instead you get the identifier of one of the alternatives, then a colon (:), then value

Copyrighted material

related subject, but do not appear in X.680; they are in X.682 and are also treated later.)

- **Tagging:** Touched on briefly already. This was important in the past, but with the introduction of automatic tagging in 1994 is much less important now.

- **The object identifier type:** This was fully covered in X.680/ISO 8824-1 pre-1998, but parts of the material are now split off into another Recommendation/Standard. Previous chapters of this book produced a lot of introductory material, but the discussion remains incomplete!

- **Hole types:** This term is used for the more formal ASN.1 terms EXTERNAL, EMBEDDED PDV, CHARACTER STRING, and "Open Types" (post-1994). And dare we mention ANY and ANY DEFINED BY (pre-1994)? If you have never heard of ANY or ANY DEFINED BY, that is a good thing. But you will have to be sullied by later text—sorry!

- **The character string types:** There are about a dozen different types for carrying strings of characters from various world-wide character sets. So far we have met PrintableString, VisibleString, GraphicString, and UTF8String, and discussed them briefly. There is a lot more to say!

- **Subtyping, or constrained types:** This is a big area, with treatment split between X.680/ISO 8824-1 and X.682/ISO 8824-3. We have already seen an example of it with the range constraint "(1..56)" on "no-of-days-reported-on" in Figure 4.1. This form is the one you will most commonly encounter or want to use, but there are many other powerful notations available if you have need of them.

- **Macros:** We have to end this chapter on an obscenity! Some reviewers said, "Don't dirty the book with this word!" But macros were **very** important (and valued) in ASN.1 up to the late 1980s, and will still be frequently encountered today. But I hope none of you will be driven to writing one! Sections I and II will not tell you much more about macros, but the historical material in Section IV discusses their introduction and development over the life of ASN.1. It is a fascinating story.

Additionally, there are a number of new concepts and notations that appear in X.681/ISO 8824-2, X.682/ISO 8824-3, and X.683/ISO 8824-4 (published in 1994). These are: information object classes (including information object definition and information object sets); and parameterization.

Where the preceding items have already been introduced (in this chapter or earlier), their detailed treatment is left to a chapter of Section II. Where they have not yet been discussed, a brief introduction appears in the following short chapter.

Copyrighted material

- to describe the concept and the problem that is being addressed,
- to illustrate where necessary key aspects of the notational support so that the presence of these features in a published protocol can be easily recognized, and
- to summarize the additional text available in Section II.

If further detail is needed on a particular topic (if something takes the reader's interest), then the appropriate chapter in Section II can be consulted. The chapter in Section II provides "closure" on all items mentioned in this chapter unless otherwise stated.

5.1 Object Identifiers

The OBJECT IDENTIFIER type was briefly introduced in Chapter 4 (4.2.9), where the broad purpose and use of this type was explained (with the type notation). Examples of its value notation have appeared throughout the text, although these have not completely illustrated all possible forms of this value notation.

> OBJECT IDENTIFIERs have a simple type notation, and a value notation that has already been seen. The "Further Details" chapter tells you about the form of the name-space and how to get some, and provides discussion of the value notation.

A more detailed discussion of the form of the object identifier tree (the name-space) is given in Chapter 8 together with a full treatment of the possible forms of OJECT IDENTIFIER value notation.

Earlier text has given enough for a normal understanding of this type and the ability to read existing specifications. It is only if you feel you need some object identifier name-space and do not know how to go about getting some that the "Further Details" material will be useful. This material also contains some discussion about the (legal) object identifier value notation that omits all names and uses numbers only, and about the (contentious) value notation where different names are associated with components, depending on where the value is being published and/or the nature of lower arcs.

5.2 Character String Types

The names of types whose values are strings of characters from some particular character repertoire have appeared throughout the earlier text, and Chapter 4 (4.2.8) discussed in some detail the type notations

Copyrighted material

```
PrintableString
 isibleString
ISO646String
UTF8String
```

although the treatment introduced terms such as "Unicode" that may be unfamiliar to some readers.

> There are many more character string types than you have met so far, and mechanisms for constructing custom types and types where the character repertoire is not defined until run-time. The value notation provides both a simple "quoted string" mechanism and a more complex mechanism to deal with "funny" characters.

There has also been little treatment so far of the value notation for these types, nor has the precise set of characters in each repertoire been identified fully.

Chapter 9 provides a full treatment of the value notation and provides references to the precise definitions of the character repertoires for all character string types. Chapter 9 describes the following additional character string types that you will encounter in published specifications (all the character string types are used in at least one published specification):

```
NumericString
IA5String
TeletexString
T61String
 ideotexString
GraphicString
GeneralString
UniversalString
BMPString
UTF8String
```

The simplest value notation for the character string types is simply the actual characters enclosed in quotation marks (the ASCII character QUOTATION MARK, usually represented as two vertical lines in the upper quartile of the character glyph).

Copyrighted material

For example,

```
"This is an example character string value"
```

The (alert—I hope we still have some!) reader will ask four questions:

- How do I express characters appearing in character string values that are not in the character set repertoire used to publish the ASN.1 specification? (Publication of ASN.1 specifications as ASCII text is common.)
- How do I include the ASCII QUOTATION MARK character (") in a character string value?
- Can I split long character string values across several lines in a published specification?
- How do I define precisely the white-space characters and control characters in a character string value?

These are topics addressed in the "Further Details" section.

In summary:

- A QUOTATION MARK character is included by the presence of adjacent quotation marks (a very common technique in programming languages).
- ASN.1 provides (by reference to character set standards), names for all the characters in the world (the names of these characters use only ASCII characters), and a value notation that allows the use of these names.
- Cell references are also available for ISO 646 and for ISO 10646 to provide precise specification of the different forms of white-space and of control characters appearing in ASCII.

An example of a more complex piece of character string value notation described in the "Further Details" is,

```
{ nul, {0,0,4,29}, cyrillicCapitalLetterIe, "ABC"}
```

go to "Further Details" if you want to know what that represents!

The preceding provision is, however, not the end of the story. If UniversalString or BMPString or UTF8String are used, then ASN.1 has built-in names (again defined by reference to character set standards) for about 80 so-called "collections" of characters. Here are the names of some of these collections:

Copyrighted material

```
BasicLatin
BasicGreek
Cyrillic
Katakana
IpaExtensions
MathematicalOperators
ControlPictures
Dingbats
```

Formally, these collections are subsets (subtypes—see 5.3) of the BMPString type, and it is possible to build custom character string types using combinations of these pre-defined types.

Chapter 9 provides full coverage of these features, but a more detailed discussion of the form and historical progression of character set standardization is presented in Section IV ("History and Applications"). Readers interested in gaining a full understanding of this area may wish to read the relevant chapter in Section IV before reading the Section II chapter.

Finally, ASN.1 also includes the type:

```
CHARACTER STRING
```

that can be included in a SEQUENCE or SET (for example) to denote a field that will contain a character string, but without (at this stage) determining either the character repertoire or the encoding.

This is an incomplete specification or "hole", and is covered in Chapter 14. If this character string type is used, both the repertoire and the encoding are determined by announcement (or if the OSI stack is in use, by negotiation) at run-time, but can be constrained by additional specification using "constraints" (see 5.9), either at primary specification time, or by "profiles" (additional specifications produced by some group that reduces options in a base standard).

5.3 Subtyping

There has been little discussion of this subject so far. We have seen an example of:

```
INTEGER (1..56)
```

Copyrighted material

A number of other forms of constraint were introduced into ASN.1 in 1994 related to constraining what can fill in a "hole", or to relating the contents of that "hole" to the value of some other field. These other forms of constraint are covered in Chapter 13.

5.4 Tagging

Earlier text has dipped in and out of tagging, but has never given a full treatment. The TLV concept (which underlies tagging) was introduced in 1.5.2 and further text on ASN.1 tagging appeared in 2.2.7 and 3.3.2, where tagging was described entirely in relation to the TLV encoding philosophy, and the concepts of "implicit tagging" and "explicit tagging" were introduced.

Some mention has also been made of different "classes" of tag, with syntactic constructs such as

```
[3] INTEGER
My-Useful-Type ::= [APPLICATION 4] SEQUENCE { .... }
[PRI ATE 4] INTEGER
[UNI ERSAL 25] GraphicString
```

Up to 1994, getting your tags right was fundamental to writing a correct specification. Post-1994, AUTOMATIC TAGS in the module header enables them to be forgotten. So details are relegated to Section II. If you want to read and understand a specification (or even to implement one), you already know enough about the tag concept, but if you want to take control of your tags (as you had to pre-1994), you will need the Section II material.

Chapter 11

- gives a full treatment of the different classes of tag,
- provides an abstract model of types and values that makes the concepts of explicit and implicit tagging meaningful, even if encoding rules are being employed that are not TLV-based,
- discusses matters of style in the choice of tag-class used in a specification, and
- gives the detailed rules on when tags on different elements of sets and sequences or alternatives of choices are required to be distinct.

Copyrighted material

image
not
available

```
SEQUENCE
     {field1  TypeA,
      field2  TypeB,
      ... ! PrintableString : "See clause 59",
      -- The following is handled by old systems
      -- as specified in clause 59.
      [[ v2-field1  Type2A,
         v2-field2  Type2B ]],
      [[ v3-field1  Type3A,
         v3-field2  Type3B ]],
      ... ,
      -- The following is version 1 material.
            field3  TypeC}
```

Figure 5.1 Illustration of extensibility markers and version brackets.

this material together with so-called "version brackets". This is illustrated in Figure 5.1, which is repeated and described more fully in Chapter 12.

Notice that it is not mandatory to include version brackets. If they are absent the effect is as if each element of the sequence had been added separately in a succession of versions.

Note also that if there is no further version 1 material ("field3 TypeC" in Figure 5.1 is not present), then the final ellipsis is not required, and will frequently be omitted.

5.6 Hole Types

Clause 2.2.1 introduced the concept of "holes": parts of a specification left undefined to allow other groups to "customize" the specification to their needs, or to provide a carrier mechanism for a wide variety of other types of material.

> You can leave a hole by using one of several ASN.1 types, but it may be better to use Information Object Classes instead!

In general, specifiers can insert in their protocols any ASN.1 type and leave the semantics to be associated with values of that type undefined. This would constitute a "hole". Thus "holes" can in principle be provided using INTEGER or PrintableString! But usually when specifiers leave a "hole", they want the container to be capable of carrying an arbitrary bit-pattern. Thus using OCTET STRING or BIT STRING to form a "hole" would be more common. This is generally not recommended, as there are specific ASN.1 types that are introduced to clearly identify the presence of a hole, and in some cases to provide an associated identification field that will identify the material in the "hole".

Copyrighted material

Provision for "holes" has been progressively enriched during the life of ASN.1, and some of the early mechanisms are disparaged now. The following are the types normally regarded as "hole" types, and are described fully in Chapter 14:

```
ANY (removed in 1994)
ANY DEFINED BY (removed in 1994)
EXTERNAL (deprecated)
EMBEDDED PDV
CHARACTER STRING
```

5.7 Macros

ASN.1 contained (from 1984 to 1994) a very complex piece of syntax called "the macro notation". It was removed in 1994, with equivalent (but much improved) facilities provided by the "Information Object Class" and related concepts (see below).

> There is much controversy surrounding macros. They were part of ASN.1 for its first decade, but produced many problems, and were replaced by Information Object Classes in 1994. You will not often see text defining a macro (and should certainly not write any today), but you may still see in older specifications text whose form depends on a macro definition imported into a module.

Many languages, graphics packages, and word processors have a macro facility. The name "macro" is very respectable. However, the use of this term in ASN.1 bears very little relationship to its use in these other packages. Section IV says a little more about what macros are all about. You are unlikely to meet the definition of a macro (use of the macro notation) in specifications that you read, but Figure 5.2 illustrates the general structure (the four dots representing further text whose form is defined by the macro notation specification).

```
MY-MACRO MACRO ::=
          BEGIN
                  TYPE NOTATION ::= ....
                  ....
                  VALUE NOTATION ::= ....
                  ....
         END
```

Figure 5.2 The structure of a macro definition.

Copyrighted material

This piece of syntax can appear anywhere in a module where a type reference assignment can occur, and the name of the macro (conventionally always in uppercase) can be (and usually is) exported from the module for use in other modules.

The macro notation is the only part of ASN.1 that is not covered fully in this book! Readers of this book should NEVER write macros! However, you will encounter modules that **import** a macro name and then have syntax that is an invocation of that macro. Again, a macro invocation can appear anywhere that a type definition can appear.

One standard that contains a lot of "holes" is called "Remote Operations Service Element (ROSE)." ROSE defines (and exports) a macro called the OPERATION macro to enable its users to provide sets of information to complete the ROSE protocol. A typical piece of syntax that uses the OPERATION macro would look like Figure 5.3 (but most real examples are much longer).

To fully understand this you need some knowledge of ROSE. A brief description of ROSE is given in Chapter 14, partly because of its widespread use, but mainly because it provides good illustrations of macro use, Information Object Class specification, and exception handling.

The OPERATION macro definition was replaced in the 1994 ROSE specification by specification of an OPERATION Information Object Class, and specifications including syntax like Figure 5.3 are gradually being changed to make use of the OPERATION Information Object Class instead.

5.8 Information Object Classes and Objects and Object Sets

When protocol specifiers leave "holes" in their specification, frequently there are several such holes, and the users of the specification need to provide information of a specified nature to fill in these holes. Most of the uses of the macro notation were to enable these users to have a notation to specify this additional information.

```
lookup  OPERATION
            ARGUMENT IA5String
            RESULT   OCTET STRING
            ERRORS {invalidName, nameNotFound}
                   ::= 1
```

Figure 5.3 An example of use of the ROSE OPERATION macro.

Copyrighted material

Computer Organization and Architecture / Programming Languages / Software Engineering

ASN.1 Complete

John Larmouth
University of Salford, Salford, England

ASN.1 Complete teaches you everything you need to know about ASN.1—whether you're specifying a new protocol or implementing an existing one in a software or hardware development project. Inside, the author begins with an overview of ASN.1's most commonly encountered features, detailing and illustrating standard techniques for using them. He then goes on to apply the same practice-oriented approach to all of the notation's other features, providing you with an easy-to-navigate, truly comprehensive tutorial.

The book also includes thorough documentation of both the Basic and the Packed Encoding Rules—indispensable coverage for anyone doing hand-encoding, and a valuable resource for anyone wanting a deeper understanding of how ASN.1 and ASN.1 tools work. The concluding section takes up the history of ASN.1, in terms of both the evolution of the notation itself and the role it has played in hundreds of protocols and thousands of applications developed since its inception.

Features

- Covers all the features—common and not so common—available to you when writing a protocol specification using ASN.1.
- Teaches you to read, understand, and implement a specification written using ASN.1.
- Explains how ASN.1 tools work and how to use them.
- Contains hundreds of detailed examples, all verified using OSS's ASN.1 Tools package.
- Considers ASN.1 in relation to other protocol specification standards.

About the Author

John Larmouth was the Founding Director of the Information Technology Institute at the University of Salford, where he has worked for over twenty years. A graduate of Cambridge University, he has been involved with ASN.1 since its introduction as an ISO standard in the early 1980s. He served as Editor of the Standard for the first ten years of ASN.1's existence and has been ISO Rapporteur for ASN.1 for the past decade.

Morgan Kaufmann is an imprint of Academic Press
A Harcourt Science and Technology Company

UPC

EAN

Copyrighted material