

Recommended Practices for Exploration and Production Data Digital Interchange

API RECOMMENDED PRACTICE 66, V2
SECOND EDITION, JUNE 1996



Recommended Practices for Exploration and Production Data Digital Interchange

API RECOMMENDED PRACTICE 66, V2
SECOND EDITION, JUNE 1996



SPECIAL NOTES

API publications necessarily address problems of a general nature. With respect to particular circumstances, local, state, and federal laws and regulations should be reviewed.

API is not undertaking to meet the duties of employers, manufacturers, or suppliers to warn and properly train and equip their employees, and others exposed, concerning health and safety risks and precautions, nor undertaking their obligations under local, state, or federal laws.

Information concerning safety and health risks and proper precautions with respect to particular materials and conditions should be obtained from the employer, the manufacturer or supplier of that material, or the material safety data sheet.

Nothing contained in any API publication is to be construed as granting any right, by implication or otherwise, for the manufacture, **sale**, or use of any method, apparatus, or product covered by letters patent. Neither should anything contained in the publication be construed as insuring anyone against liability for infringement of letters patent.

Generally, API standards are reviewed and revised, reaffirmed, or withdrawn at least every five years. Sometimes a one-time extension of up to two years will be added to this review cycle. This publication will no longer be in effect five years after its publication date as an operative API standard or, where an extension has been granted, upon republication. Status of the publication can be ascertained from the API Authoring Department [telephone (202) **682-8000**]. A catalog of API publications and materials is published annually and updated quarterly by API, 1220 L Street, N.W., Washington, D.C. 20005.

This document was produced under API standardization procedures that ensure appropriate notification and participation in the developmental process and is designated as an API standard. Questions concerning the interpretation of the content of this standard or comments and questions concerning the procedures under which this standard was developed should be directed in writing to the director of the Authoring Department (shown on the title page of this document), American Petroleum Institute, 1220 L Street, N.W., Washington, **D.C.** 20005. Requests for permission to reproduce or translate all or any part of the material published herein should also be addressed to the director.

API publications may be used by anyone desiring to do so. Every effort has been made by the Institute to assure the accuracy and reliability of the data contained in them; however, the Institute makes no representation, warranty or guarantee in connection with this publication and hereby expressly disclaims any liability or responsibility for loss or damage resulting from its use or for the violation of any federal, state, or municipal regulation with which this publication may conflict.

API standards are published to facilitate the broad availability of proven, sound engineering and operating practices. These standards are not intended to obviate the need for applying sound engineering judgment regarding when and where these standards should be utilized. The formulation and publication of API standards is not intended in any way to inhibit anyone from using any other practices.

Any manufacturer marking equipment or materials in conformance with the marking requirements of an API standard is **solely** responsible for complying with all the applicable requirements of that standard. API does not represent, warrant, or guarantee that such products do in fact conform to the applicable API standard.

*All rights reserved. No part of this work may be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission from the **publisher**. Contact the **Publisher**, API Publishing Services, 1220 L Street, N. W., Washington, D.C. 20005.*

Copyright © 1996 American Petroleum Institute

FOREWORD

This recommended practice was prepared by the Subcommittee on Standard Format for Digital Well Data. This standard is under the administration of the American Petroleum Institute Exploration and Production Department's Executive Committee on Drilling and Production Practices.

Background and Purpose

A wide variety of digital data is acquired, exchanged, and stored by petroleum industry businesses using an equally wide variety of data formats. Most of these formats have been around for many years and were designed for very specific needs. As a result, they reflect the limitations of early computing hardware and are typically oriented to one type of data. Although relatively simple to implement, many formats lack the flexibility or sufficient self-descriptive features to accommodate evolving data exchange needs.

Recommended Practice 66, Version 1 (RP66, VI) was introduced by the API in 1991 as an enhanced exchange format for well data. It incorporated the useful features of earlier formats and extended them by removing some of their historical limitations. The primary features of API Recommended Practice 66, Version 1 were machine independence, **self**-description, semantic extensibility, and efficient handling of bulk data. It has been implemented and is used widely for exchange of well data.

Following the introduction of API Recommended Practice 66 by the API, a number of efforts by other oil industry groups to develop data exchange formats began to mature and converge toward the adoption of API Recommended Practice 66, creating a major opportunity to have just one standard exchange format for all **oilfield** data. However, the strong bias of API Recommended Practice 66 toward well data was a stumbling block. In addition, introduction of new high-capacity storage devices created a need to expand the API Recommended Practice 66 physical binding mechanism, and **experience** in using API Recommended Practice 66 generated ideas for improvements. Working closely with other industry groups, the API Subcommittee on Recommended Format for Well Data developed a set of modifications to API Recommended Practice 66, which have resulted in the current API Recommended Practice 66, Version 2 (RP66, **V2**), which is described in this document.

API Recommended Practice 66, Version 2 retains the essential features of API Recommended Practice 66, Version 1 and is completely upward compatible. That is, any data recorded under API Recommended Practice 66, Version 1 can be translated into API Recommended Practice 66, Version 2 format.

The following list summarizes the differences between API Recommended Practice 66, Version 1 and API Recommended Practice 66, Version 2.

1. **Separate well data object types from neutral data object types.**
Public object types from API Recommended Practice 66, Version 1 are separated into two groups, both administered by the API Subcommittee on Recommended Format for Well Data. One group consists of basic object types having no well data bias. This is referred to as the basic schema, **described** in Part 6, and is administered under organization code 0 (zero). The second group consists of all the rest and is administered under organization code 66. This group, described in Part 8, is referred to as the DLIS schema.

2. **Add new attributes and symbolic codes.**

The following are added: DIMENSION and AXIS attributes to **CALIBRATION-COEFFICIENT**, DATE attribute to **CALIBRATION**, KIND attribute to **CHANNEL**, PARAMETER, and COMPUTATION, DESCRIPTION attribute to all object types, EXTENDED-ATTRIBUTES attribute to all object types except **FILE-HEADER** and **ORIGIN**, new codes for TYPE attribute of **EQUIPMENT**, and new codes for **PROPERTIES** attribute (many object types).

3. **Allow COORDINATES attribute of AXIS to reference a CHANNEL object.**
This permits dynamically changing coordinates.

4. **Add new representation codes and representation code classes.**

The following codes are new: **RNORM**, **RLONG**, **ISNORM**, **ISLONG**, **IUNORM**, **IULONG**, **IRNORM**, **IRLONG**, **TIDENT**, **TUNORM**, **TASCII**, **LOGICL**, **BINARY**, **FRATIO**, **DRATIO**. All representation codes are grouped into classes of related representations. A restriction in the definition of an attribute to any member of a class allows use of any other member of the class.

5. **Extend ASCII to include ISO 8859-1, and allow string padding.**

The set of allowable characters in an ASCII string is enlarged. In addition, character subfields in all representations may be padded by using a terminating null (zero) character. The string count may include characters past the null, but such characters are not considered part of the string data. This supports in-place editing of API Recommended Practice 66 data.

6. **Change copy number (in OBNAME) from USHORT to UVARI.**
A maximum of 256 instances of a named object was seen as potentially limiting for some applications.

7. **Extend unit expression syntax and add new unit symbols.**

The unit expression syntax supports additional types of units, for example units with fractional exponents, and use of multiple sets of parentheses to improve meaning. A more comprehensive set of SI unit symbols and a set of monetary unit symbols are added to the unit dictionary.

8. **Allow use of multiple unit models.**

In addition to the unit model described by API as part of API Recommended Practice 66, provision is made to identify and use other unit models having different sets of unit symbols.

9. **Uncontrol EQUIPMENT identifiers.**

These identifiers are no longer required to be in a dictionary. Attributes TYPE and TRADEMARK-NAME suffice to identify equipment.

10. **Require distinct object names in a logical file.**

This rule, that no two object names can match all three subfields (origin, copy number, identifier), makes an object name a unique reference. It is no longer necessary for a reader to know the object type to resolve the reference.

I I. Revise ORIGIN.

Well data attributes are removed and put into a new **DLIS-CONTEXT** object type in the **DLIS** schema. New attributes are added to identify schema and unit model.

12. Support use of multiple schemas.

The terms Private and Public are removed. All schemas are implemented equally with the only exception being mandatory (and exclusive) use of FILE-HEADER and ORIGIN object types from the basic schema. Mechanics to make this work include TIDENT representation of set types and attribute enumerations, and removal of logical record type numbers.

13. Remove rule restricting ordering of EFLRs and IFLRs.

An attribute in one set (EFLR) may reference an object in another set even if the two sets are separated by an IFLR.

14. Remove special constraints on representation of FILE-HEADER attributes.

The attributes of a FILE-HEADER object are no longer constrained regarding length and order.

15. Neutralize FRAME and CHANNEL and support multiple frames per record.

Attributes of these objects are revised to remove the well data bias. In addition, multiple frames per record are supported. This allows some applications to increase performance for reading and writing bulk data. In addition to dimensioned arrays, channel values may now also be described as aggregates (similar to “ragged arrays”).

16. Replace channel dimension updates with explicitly-sized channels.

Provisions are made to write explicitly-sized channels by optionally recording channel dimensions directly in frames. Correspondingly, the DIMENSION attribute of CHANNEL is no longer updatable.

17. Add new information to storage unit label.

The following fields are added: binding version, producer code, creation date, and serial number. The size of maximum record length is increased.

18. Expand visible record header and logical record segment header.

The maximum length of a visible record is increased from a 16-bit quantity to a 32-bit quantity, and similarly for lengths in the logical record segment. This allows binding onto very large tape blocks (megabyte or more). New fields file sequence number and file section number are added to the visible record header. The visible record now also has a trailing length. A new structure, **FIXREC**, is added to indicate fixed-length visible records.

19. Fix encryption mechanism to handle blind passthrough.

The logical record segment encryption mechanism is modified to allow readers to copy and re-segment data for which the encryption method is unknown.

20. Define a physical binding for files on random access disks.

A file on a random access disk is defined to be a storage unit if the file has a byte stream structure consisting of a storage unit label followed by a sequence of visible records.

Organization of API Recommended Practice 66, Version 2

API Recommended Practice 66, Version 2 consists of Parts 1 through 9, plus Appendix A. These parts are briefly described below:

PART 1: MODEL AND METHODOLOGY.

This part describes the data model upon which the format is based and the

methodology for specifying schemas—namely, object types, attributes, and the rules about them.

PART 2: LOGICAL FORMAT.

This part describes the logical organization and representation of data, including the definitions and uses of storage sets, logical files, visible records, logical records, sets, objects, attributes, and values. It also describes naming and referencing rules as well as bindings between logical records and visible records and includes specifications of all representation codes used in the format.

PART 3: PHYSICAL BINDINGS.

This part describes how storage unit labels and visible records are recorded on various common medium types, including 9-track magnetic tape and random access disk files. It also describes the use of filemarks and partitions where applicable.

PART 4: THE API-SI UNIT MODEL.

This part describes a unit model based on le **Système International d'Unités** (SI) from which a wide variety of units can be expressed. The unit model supports a general method for computing unit conversion coefficients for dimensionally-related units.

PART 5: THE API-SI UNIT SYMBOLS.

This part lists and defines the unit symbols recognized under the unit model described in Part 4.

PART 6: BASIC SCHEMA.

This part specifies the object types administered by the API Subcommittee on Recommended Format for Digital Well Data using organization code 0 (zero). This schema includes certain basic object types such as **FILE-HEADER** and **ORIGIN**, which are required by all implementations, as well as other object types such as **FRAME** and **CHANNEL**, which are standard mechanisms for describing the storage of dynamic (or bulk) data.

PART 7: BASIC SCHEMA DICTIONARY.

This part lists and describes reference values for attributes belonging to the basic schema.

PART 8: DLIS SCHEMA.

This part specifies the Digital Log Interchange Standard (DLIS) schema object types administered by the API Subcommittee On Recommended Format For Digital Well Data using organization code 66. This schema includes object types particularly oriented toward recording well log data.

PART 9: DLIS SCHEMA DICTIONARY.

This part lists and describes reference values for attributes belonging to the DLIS schema.

APPENDIX A: ORGANIZATION CODES.

This appendix lists currently-assigned organization codes.

This standard shall become effective on the date printed on the cover, but may be used voluntarily from the date of distribution.

CONTENTS

	Page
PART 1: MODEL AND METHODOLOGY	
0 Introduction	1-1
1 Scope	1-1
2 Definitions	1-1
3 Concepts	1-3
4 Audience	1-3
5 Schema Categories	1-3
6 Schema Administration	1-4
7 Data Model	1-4
8 Schema Specification	1-5
8.1 Preliminary Part	1-5
8.2 Normative Part	1-5
8.3 Supplementary Part	1-6
8.4 Object Type Specification	1-6
8.5 Attribute Specification	1-6
9 Notation: How to Refer to an Attribute	1-7
10 Dictionary Specification	1-7
10.1 Preliminary Part	1-7
10.2 Normative Part	1-7
10.3 Supplementary Part	1-8
PART 2: LOGICAL FORMAT	
0 Introduction	2-1
1 Scope	2-1
2 References	2-1
3 Definitions	2-2
4 Concepts	2-5
5 Representation Codes	2-6
6 Visible Record	2-7
6.1 Overview	2-7
6.2 Visible Record Header	2-7
6.3 Visible Record Body	2-8
6.4 Visible Record Trailer	2-8
7 Logical Record	2-8
7.1 Overview	2-8
7.2 Logical Record Segment Header	2-10
7.3 Logical Record Segment Encryption Packet	2-11
7.4 Logical Record Segment Body	2-12
7.5 Logical Record Segment Trailer	2-12
7.6 Checksum Algorithm	2-13
7.7 Logical Record Body	2-13
7.8 Logical Record Encryption Rules	2-14
8 Explicitly-Formatted Logical Record (EFLR)	2-14
8.1 Overview	2-14
8.2 Component	2-15
8.3 Sets	2-16
8.4 Templates	2-18
8.5 Objects	2-18
8.6 Attributes	2-20

	Page
PART 2: LOGICAL FORMAT (CONTINUED)	
9 Indirectly-Formatted Logical Record (IFLR)	2-22
9.1 Overview	2-22
9.2 Data Descriptor Reference (DDR)	2-22
9.3 IFLR Modifier	2-23
10 Logical File	2-23
10.1 Overview	2-23
10.2 Logical File Organization	2-24
11 Representation Code Descriptions	2-24
11.1 Introduction	2-24
11.2 Representation Code Summary	2-24
11.3 Descriptions of Simple Representation Codes	2-26
11.4 Descriptions of Compound Representation Codes	2-36
PART 3: PHYSICAL BINDINGS	
0 Introduction	3-1
1 Scope	3-1
2 Definitions	3-1
3 Concepts	3-2
4 Storage Unit Terminology and Requirements	3-3
4.1 Access Methods	3-3
4.2 Content	3-4
4.3 End of Storage Unit Indicator	3-4
5 Storage Set Terminology and Requirements	3-4
6 Storage Unit Label Contents	3-5
6.1 Storage Unit Structure Options	3-6
7 Binding to Standard Magnetic Tapes	3-6
7.1 Medium Characteristics	3-6
7.2 Binding Requirements	3-7
8 Binding to Random Access Files	3-7
8.1 Medium Characteristics	3-7
8.2 Binding Requirements	3-7
9 Binding to Peer-To-Peer Communication Streams	3-7
9.1 Medium Characteristics	3-7
PART 4: THE API-SI UNIT MODEL	
0 Introduction	4-1
1 Scope	4-1
2 References	4-1
3 Definitions	4-2
4 Concepts	4-2
5 Unit Expression Grammar	4-3
6 Unit Conversion	4-4
6.1 Derived Units	4-4
6.2 Unit Dimension and Unit Reduction	4-4
6.3 Unit Conversion Using Reduced Standard Forms	4-5
PART 5: THE API-SI UNIT SYMBOLS	
0 Introduction	5-1
1 Scope	5-1
2 References	5-1

	Page
PART 5: THE API-SI UNIT SYMBOLS (CONTINUED)	
3 Base Unit Symbols	5-2
3.1 SI Base Unit Symbols	5-2
3.2 API-SI Supplementary Base Units-Non-Currency	5-2
3.3 API-SI Currency Base Unit Symbols	5-2
4 Derived Unit Symbols	5-5
PART 6: BASIC SCHEMA	
0 Introduction	6-1
1 Scope	6-1
2 Definitions	6-1
3 Authority	6-4
4 Concepts	6-4
5 Unit Model	6-5
6 Dictionary-Controlled Identifiers	6-5
7 Required vs Optional Use of Attributes	6-5
8 Frequently-Used Attributes	6-5
8.1 Interpretation of a Dimensioned Array	6-6
9 Required Object Types	6-7
9.1 File-Header	6-7
9.2 Origin	6-8
10 Optional Object Types	6-10
10.1 Axis	6-10
10.2 Channel	6-11
10.3 Comment	6-13
10.4 Computation	6-13
10.5 Frame	6-14
10.6 Group	6-16
10.7 No-Format	6-16
10.8 Origin-Translation	6-17
10.9 Parameter	6-17
10.10 Process	6-18
10.11 Update	6-19
10.12 Zone	6-20
10.13 Updatable Attributes	6-20
PART 7: BASIC SCHEMA DICTIONARY	
0 Introduction	7-1
1 Scope	7-1
2 Authority	7-1
3 Concepts	7-1
4 Channel Reference Values	7-2
4.1 Direction	7-2
4.2 Flags	7-2
5 Frame Reference Values	7-2
5.1 Flags	7-2
6 Process Reference Values	7-3
6.1 Status	7-3

	Page
PART 8: DLIS SCHEMA	
0 Introduction	8-1
1 Scope	8-1
2 Definitions	8-1
3 Authority	8-4
4 Concepts	8-4
5 Unit Model	8-5
6 Dictionary-Controlled Identifiers	8-5
7 Required vs Optional Use of Attributes	8-5
8 Frequently-Used Attributes	8-5
9 DLIS Object Types	8-5
9.1 Calibration	8-5
9.2 Calibration-Coefficient	8-7
9.3 Calibration-Measurement	8-7
9.4 DLIS-Context	8-9
9.5 Equipment	8-9
9.6 Message	8-11
9.7 Path	8-11
9.8 Splice	8-13
9.9 Tool	8-14
9.10 Wellbore-Path-Datum	8-14
9.11 Updatable Attributes	8-15

PART 9: DLIS DICTIONARY	
0 Introduction	9-1
1 Scope	9-1
2 Authority	9-1
3 Concepts	9-1
4 Calibration-Measurement Reference Values	9-2
4.1 Phase	9-2
5 O/Channel Reference Values	9-2
5.1 Kind	9-2
5.2 Properties	9-2
6 O/Computation Reference Values	9-4
6.1 Properties	9-4
7 Equipment Reference Values	9-4
7.1 Location	9-4
7.2 Type	9-4
8 Message Reference Values	9-5
8.1 Type	9-5
9 O/Parameter Reference Values	9-5
9.1 Properties	9-5
10 O/Zone Reference Values	9-5
10.1 Domain	9-5

APPENDIX A: ORGANIZATION CODES	
0 Introduction	A-1
1 Scope	A-1
2 Assignment of Organization Codes	A-1

	Page
Figures	
2-1—Conceptual View of a Logical Format (Bound to a Physical Format)	2-5
2-2—Visible Record Structure	2-7
2-3—Logical Record Segmentation (Without Encryption)	2-10
2-4—Logical Record Segmentation (With Encryption)	2-10
2-5—Application-Level View of a Set (Type = FOO)	2-14
2-6—Component-Level View of a Set	2-15
2-7—Component Structure	2-16
2-8—Examples of Attribute Components	2-21
2-9—Structure of an IFLR Body and Relation to its Data Descriptor	2-22
6-1—Examples of Aggregates	6-6
6-2-A Frame Block	6-14
6-3—An Unbounded Grid of Frame Type Indexes	6-15
8-1—Spatial Coordinate System for a Wellbore Path	8-4
8-2—Illustration of Simple Two-point Linear Calibration	8-6
8-3—Tool String Configuration	8-12
Tables	
2-1—Visible Record Header Fields	2-8
2-2—Logical Record Segment Header Fields	2-11
2-3—Logical Record Segment Header Attributes	2-11
2-4—Logical Record Segment Encryption Packet Fields	2-12
2-5—Logical Record Segment Trailer Fields	2-13
2-6—Component Structure	2-15
2-7—Component Role	2-16
2-8—Normal, Redundant, and Replacement Set Components	2-18
2-9—Object Component	2-19
2-10—Attribute Components	2-20
2-11—IFLR Modifier Options	2-23
2-12—Representation Code Summary	2-25
2-13—Compound Representation Code Descriptions	2-37
3-1—Storage Unit Label Fields	3-5
3-2—Storage Unit Structure Options	3-6
5-1—SI Base Unit Symbols	5-2
5-2—API-SI Base Unit Symbols-Non-Currency	5-2
5-3—API-SI Currency Base Unit Symbols	5-2
5-4—API-SI Derived Unit Symbols	5-5
6-1—Frequently-Used Attributes	6-5
6-2—FILE-HEADER Attributes	6-7
6-3—ORIGIN Attributes	6-9
6-4—AXIS Attributes	6-10
6-5—CHANNEL Attributes	6-11
6-6—COMMENT Attributes	6-13
6-7—COMPUTATION Attributes	6-13
6-8—FRAME Attributes	6-16
6-9—GROUP Attributes	6-16
6-10—NO-FORMAT Attributes	6-17
6-11—ORIGIN-TRANSLATION Attributes	6-17
6-12—PARAMETER Attributes	6-18
6-13—PROCESS Attributes	6-18
6-14—UPDATE Attributes	6-19

	Page
Tables (Continued)	
6-15— ZONE Attributes	6-20
6-16— Updatable Attributes	6-20
7-1— CHANNEL:DIRECTION Reference Values	7-2
7-2— CHANNEL:FLAGS Reference Values	7-2
7-3— FRAME:FLAGS Reference Values	7-2
7-4— PROCESS:STATUS Reference Values	7-3
8- I - CALIBRATION Attributes	8-6
8-2— CALIBRATION-COEFFICIENT Attributes	8-7
8-3— CALIBRATION-MEASUREMENT Attributes	8-7
8-4— DLIS-CONTEXT Attributes	8-9
8-5— EQUIPMENT Attributes	8-10
8-6— MESSAGE Attributes	8-11
8-7— PATH Attributes	8-12
8-8— SPLICE Attributes	8-13
8-9— TOOL Attributes	8-14
8- I 0— WELLBORE-PATH-DATUM Attributes	8-15
8- I I - Updatable Attributes	8-15
9- I - CALIBRATION-MEASUREMENT:PHASE Reference Values	9-2
9-2— 0/CHANNEL:KIND Reference Values	9-2
9-3— 0/CHANNEL:PROPERTIES Reference Values	9-2
9-4— EQUIPMENT:LOCATION Reference Values	9-4
9-5— EQUIPMENT:TYPE Reference Values	9-4
9-6— MESSAGE:TYPE Reference Values	9-5
9-7— 0/ZONE:DOMAIN Reference Values	9-5
A- I - Organization Codes	A-2

Recommended Practices for Exploration and Production Data Digital Interchange

Part 1: Model and Methodology

Exploration and Production Department

API RECOMMENDED PRACTICE 66, V2
SECOND EDITION, JUNE 1996



CONTENTS

	Page
0 INTRODUCTION	1-1
1 SCOPE	1-1
2 DEFINITIONS	1-1
3 CONCEPTS	1-3
4 AUDIENCE	1-3
5 SCHEMA CATEGORIES	1-3
6 SCHEMA ADMINISTRATION	1-4
7 DATA MODEL	1-4
8 SCHEMA SPECIFICATION	1-5
8.1 Preliminary Part	1-5
8.2 Normative Part	1-5
8.3 Supplementary Part	1-6
8.4 Object Type Specification	1-6
8.5 Attribute Specification	1-6
9 NOTATION: HOW TO REFER TO AN ATTRIBUTE	1-7
10 DICTIONARY SPECIFICATION	1-7
10.1 Preliminary Part	1-7
10.2 Normative Part	1-7
10.3 Supplementary Part	1-8

Recommended Practice for Exploration and Production Data Digital Interchange Part 1: Model and Methodology

0 Introduction

0.1 This standard contains introductory, normative, and annotative information. Introductory information comprises everything that precedes Section 1, Scope. The introductory clauses describe how the publication is organized and provide commentary on the history and purpose of this standard. Normative information includes the actual requirements that must be satisfied by any data conforming to the standard. Annotative information is provided as rationale for and illustrations about the normative information and does not constitute part of the standard. The standard is completely stated even if all introductory and annotative information are removed. Annotative information may refer to terms introduced in later paragraphs or parts in order to tie together concepts, i.e., from time to time its value may increase after the reader has made a complete pass over the standard.

0.2 Different styles are used to distinguish between normative and annotative information.

0.3 All normative information is written using this same font, and is contained in either numbered paragraphs or numbered tables. Normative text is aligned with the left margin of the page.

0.4 All figures are annotative, and all annotative text is written in unnumbered paragraphs in Helvetica italic font, indented, as shown here:

This is a paragraph of annotative text. Its purpose is to include informal commentary on the normative information in immediately preceding or following paragraphs and may include references to or usage of normative information in other parts of the standard.

1 Scope

1.1 This part specifies a methodology for managing (generating and maintaining) a schema specification for data to be recorded under the API Recommended Practice 66, Version 2 format.

1.2 This document is intended to:

- a. Facilitate the development of exchange standards based on the API Recommended Practice 66, Version 2 format.
- b. Facilitate the development of schema-neutral software products and services, by promoting uniformity between API Recommended Practice 66, Version 2-based exchange standards.
- c. Promote compatibility between editions of a schema.

2 Definitions

2.1 attribute: A named item of information or data pertaining to an object type.

2.2 attribute count: The number of elements in an attribute value.

- 2.3 attribute label:** The name of an attribute.
- 2.4 attribute representation code:** A code that identifies the recorded representation of each element of an attribute value.
- 2.5 attribute units:** An expression that represents the units of measurement of each element of an attribute value.
- 2.6 attribute value:** The value of an attribute. It may be present or absent. When present, it consists of zero or more elements, each having the same units and the same representation.
- 2.7 characteristic:** A subitem of a component that contains one piece of information for a data item. Characteristics of an attribute component, for example, are its label, count, units, representation code, and value.
- 2.8 data model:** A description of a **specification** and representation paradigm for data.
- 2.9 dictionary:** A database in which identifiers and reference values used under API Recommended Practice 66, Version 2 are maintained and administered.
- 2.10 identifier:** One of the three parts of an object name. It is a character string used to distinguish the object from other objects of the same type. For some designated object types, the identifier conveys meaning of the nature of the object, and the identifier and its meaning are maintained in a dictionary.
- 2.11 logical file:** The main unit of data exchange. It consists of a sequence of one or more logical records, beginning with a record containing a single FILE-HEADER object.
- 2.12 logical model:** A conceptual organization of a domain of knowledge.
- 2.13 object:** A recorded instance of an object type.
- 2.14 object type:** A logical entity of a schema that has a unique type name and one or more defined attributes. Instances of an object type are written in explicitly formatted logical records.
- 2.15 organization code:** A number assigned by API to an organization that identifies the organization and represents **schemas** and dictionaries defined and administered by the organization. See Appendix A, "Organization Codes."
- 2.16 representation code:** A unique number that identifies a standard encoding for a value as a sequence of one or more contiguous bytes.
- 2.17 schema:** A formalized description of the encoding of information defined by a logical model, typically in terms of a data model.
- 2.18 schema code:** A numeric code found in Appendix A used to identify the organization responsible for defining and administering a schema.

3 Concepts

3.1 A logical model is the conceptual organization of a domain of knowledge: What classes of things are of interest, and what are their characteristics and the relationships between them which are of interest? A logical model is a framework for common understanding of the nature of classes of things, but is not sufficient for communication about individual members of the classes.

3.2 An implementation model, also called a schema, is a formalized description of the encoding of information defined by the logical model, typically in terms of a representational model. How are general representational constructs used to encapsulate information about the elements defined in the logical model? A schema must be sufficiently dynamic to meet new requirements, and yet must be sufficiently stable that change requirements are not unnecessarily imposed on the systems which use it.

3.3 A representational (or data) model is a description of the specification and representation paradigm to be used. What are the fundamental representational constructs and their organization? How is the schema specified? How is data organized, encoded, stored, and decoded?

3.4 The purpose of a schema specification for an exchange format is to enable the exchange of information in some domain of knowledge.

4 Audience

4.1 This document is intended for:

- a. Original authors and maintainers of schema specifications.
- b. Readers of schema specifications.
- c. Developers of schema-neutral API Recommended Practice 66, Version 2 software layers and systems.

4.2 It is assumed that the reader has a basic understanding of logical models, and has a particular logical model in mind. No particular modeling formalism is assumed; indeed, the schema specification may itself be the only formal expression of the logical model.

4.3 API Recommended Practice 66, Version 2 is a general-purpose data exchange mechanism which is not bound to a specific logical model. The purpose for using a standard methodology for schema specification is to enable the implementation of data exchange using model-driven software layers.

5 Schema Categories

5.1 Schemas may be categorized as managed, derived, and local.

5.2 A managed schema exists from the definition of its initial edition, through subsequent editions, until all data recorded under it are no longer of interest (i.e., indefinitely). During this life cycle each edition of a schema is expected to meet certain requirements for compatibility or continuity with previous editions. As the logical model and usage requirements change, new editions of the schema are developed, subject to these constraints.

5.3 A derived schema is one which is systematically derived or inferred from a formalized logical model according to a deterministic methodology. In this case a human-

readable schema specification might not be produced. Furthermore, **the** schema **edition** reflects the **version** of the derivation methodology and the versioning policy under which it is governed. The logical model may be governed under a different versioning policy. Organizations that define derivation methodologies should provide mechanisms to record and identify the author, name, and edition of the logical model from which the schema is derived.

5.4 A local schema is one that is neither managed nor derived but is used for local exchange or storage of data using the data model.

5.5 This document is normatively applicable to managed schemas only.

6 Schema Administration

6.1 An organization must have been issued an organization code to publish a managed schema or a schema derivation methodology. Code 999 is reserved for local schema. **An** organization code is assigned on request by the Exploration and Production Department, American Petroleum Institute, which also maintains the **most** current list of organization codes. Its address is listed at the end of the Foreword.

6.2 The holder of an organization code may publish (editions of) a schema **or** a schema derivation methodology under that code. An organization that wishes to manage multiple schemas may obtain multiple organization codes. No other constraints apply among schemas, but certain constraints apply between editions of a managed schema. If a schema element is defined under two editions, then it has the same meaning and use in both editions. Elements may be dropped from one edition to the next and new elements may be added, but elements may not be redefined.

6.3 Alternatively, an organization may manage multiple schemas by defining a derivation methodology under an organization code for mapping an appropriate class of logical models into derived schemas.

6.4 In general, it is expected that an organization that publishes a schema or a schema derivation methodology will also publish the specific versioning policy governing compatibility and constraints between editions. Derived schemas editions are determined indirectly by reflecting the logical model edition along with the name and edition of the derivation methodology applied to the logical model.

7 Data Model

7.1 API Recommended Practice 66, Version 2 schema specifications are defined in terms of object types, attributes, and the rules about them. These data model elements are described in detail in Part 2, "Logical Format."

7.2 The specification of an object type includes its name, its semantics, its attributes, and whether its instances have controlled names. No relation shall be assumed between object types in different schemas having the same type names.

7.3 The specification of an attribute includes its label, its semantics, and any restrictions on its count, representation code, units, or value. Unless explicitly stated, no relation shall be assumed between attributes in different object types having the same labels.

7.4 An instance of an object type is constrained by its object type specification. An object may have all, some, or none of the attributes defined for it, and shall not have attributes not defined for it or duplicates of the attributes defined for it.

7.5 An instance of an attribute is constrained by its specification in the object type of the object in which it appears. An attribute has exactly one occurrence of each type of characteristic: label, count, representation code, units, and value. The label identifies the attribute in terms of its definition in the object type. The value is composed of zero or more elements, or occurrences of the data representation identified by its representation code. It has a single count, which is the number of elements in its value. It has a single units value, which applies to each element of the value.

7.6 The logical file is the logical unit of data exchange, and is a collection of objects and other data described by its objects. The object is the unit of data which may be uniquely referenced within a logical file. An object is a collection of the attributes defined for its type, with assigned values. A logical file may contain objects from any number of schemas, and objects may refer to objects in different schemas in the same logical file.

8 Schema Specification

8.1 Preliminary Part

8.1.1 The preliminary part of a schema specification may include the following:

- a. Title page.
- b. Table of contents.
- c. Foreword.
- d. Introduction.

8.2 Normative Part

8.2.1 The normative part of a schema specification may include the following:

- a. Context.
 1. Title.
 2. scope.
 3. Normative references.
- b. Authority.
 1. Sanctioning organization.
 2. Document preparation, revision, and review committees.
 3. Proprietary considerations.
 4. Revision and versioning policies.
- c. Edition.
 1. Edition level.
 2. Summary of this and recent editions.
 3. Document status (draft or final).
- d. Definitions.
 1. Define terminology introduced.
 2. Normative references to other defining documents.
- e. Model.

Definition, presentation, or description of the logical model, or reference to equivalent documentation.
- f. Schema.
 1. Definition of object types and their attributes.
 2. Constraints on objects and attributes.

3. Relationships to corresponding elements of the logical model.
- g. Dictionary.
Tables of reference values, to which attributes they apply, and their meanings.
- h. Units.
 1. Specification of unit model(s) and unit dictionary(ies) used in the schema (if not **API-SI** units).
 2. Identification of units model in attribute restriction specifications.
- i. Normative appendices.

8.3 Supplementary Part

8.3.1 The supplementary part of a schema specification may include the following:

- a. Informative appendices.
- b. Footnotes.

8.4 Object Type Specification

8.4.1 An object type is specified by providing a unique type identifier, dictionary constraints on the identifier subfield of the name, a set of attributes, and the intended use of the object. An object type specification typically has the following parts:

- a. Type name.
- b. Context—a description of the semantics of the object type or reference to a logical model.
- c. Attribute specifications.

8.5 Attribute Specification

8.5.1 The characteristics of an attribute are its label, count, representation code, units, and value. The formal specification of an attribute includes its label, and may include constraints on the remaining characteristics.

8.5.2 The formal specification of the set of attributes defined for an object type is presented in a single table, followed immediately by a set of numbered notes. The columns of the attribute specification table are:

- a. Attribute Label: a valid **IDENT** value, unique in the object type,
- b. Restrictions: a set of restrictions on the count, representation code, units, or value characteristics of an attribute. Restriction specifications are of the form ' $q_1=r_1, q_2=r_2, \dots, q_n=r_n$ ', where ' q_n ' is the characteristic identifier and ' r_n ' is a restriction specification. The characteristic identifier is lower case. When no restrictions apply, the identifier is omitted. Restriction specifications are stated in the order ' $c= \dots r= \dots u= \dots v= \dots$ '.
 1. A count restriction is specified as ' $c=\{[min1:max1], [min2:max2], \dots, [minn:maxn]\}$ ', where mink and maxk represent bounds, or '?' if the interval lacks an upper or lower bound. If a single boundary set is specified, the curly braces ('{', '}') are omitted. If the minimum and the maximum are equal for any boundary range, the square brackets ('[', ']'), colon (':'), and second range limit are omitted.
 2. The representation code restriction is specified as ' $r=XXX I YYY I ZZZ$ ', where XXX, YYY, and ZZZ are alternative representation codes. If a single representation code is specified, the vertical bar 'I' is omitted. For each representation code, an alternative representation in the same representation code class may be used, but the value shall be representable in one of the specified representation codes.
 3. The units restriction is specified as ' $u=um:uexp$ ', where 'um' identifies the units model and its dictionary, and 'uexp' is a units expression in that units model. If the

'urn:' prefix is omitted, then a default unit model shall apply if declared in the general context part of the schema specification. The specification '**u=**' indicates the attribute value is unitless. When the unit model is **API-SI**, then restriction to a unit represents a typical or preferred use, although another unit having the same canonical form may be used. A unit having a different canonical form shall not be **used**.

4. The value restriction is specified as '**v=ref**', where 'ref' may refer to the note or to a table of reference values for the attribute. Reference values are dictionary-controlled values belonging to an enumerated set. A schema may declare a value to consist of reference values and defer actual definition of the reference values to other **schemas** that may **use** the attribute's object type. A schema may provide an actual table of reference values or a normative reference to them.

- c. Notes: Numbered references to notes that immediately follow the table. Notes are free-format descriptions of semantics and other rules about the attribute and its relation to other attributes.

9 Notation: How to Refer to an Attribute

9.1 The notation TYPE:LABEL is **used to** refer to an attribute in a schema, where TYPE is the object type and LABEL is the attribute label. TYPE **may be** omitted if understood from context. If the schema is not clear **from** the context of the referral, then the additional notation n/TYPE:ATTRIBUTE is used, where 'n' is the schema code of the object type.

10 Dictionary Specification

10.1 Preliminary Part

10.1.1 The preliminary part of the dictionary may include the following:

- a. Title page.
- b. Table of contents.
- c. Foreword.
- d. Introduction.

10.2 Normative Part

10.2.1 The normative part of the dictionary may include the following:

- a. Context.
 1. Title.
 2. Scope.
 3. Normative references.
- b. Schema: a reference to the schema (schema **code**, schema name, schema edition) and dictionary specification within the schema for which the dictionary is applicable.
- c. Authority.
 1. Sanctioning organization.
 2. Document preparation/revision/review committees.
 3. Proprietary considerations.
 4. Revision and versioning policies.
- d. Edition.
 1. Edition level.
 2. Summary of this and recent editions.
 3. Document status (draft or final).
- e. Definitions.
 1. Terminology.
 2. Normative references to other defining documents.

f. Description of **the** reference value tables.

g. Reference value tables.

1. Title which identifies the attribute restricted to the reference values,
2. Reference value column.
3. Description column.
4. **Other** columns **as** required by **the** schema.

10.3 Supplementary Part

10.3.1 The supplementary part of the dictionary may **include the following**:

- a. Informative appendices.
- b. Footnotes.

Recommended Practices for Exploration and Production Data Digital Interchange

Part 2: Logical Format

Exploration and Production Department

API RECOMMENDED PRACTICE 66, V2
SECOND EDITION, JUNE 1996



CONTENTS

	Page
0 INTRODUCTION	2-1
1 SCOPE	2-1
2 REFERENCES	2-1
3 DEFINITIONS	2-2
4 CONCEPTS	2-5
5 REPRESENTATION CODES	2-6
6 VISIBLE RECORD	2-7
6.1 Overview	2-7
6.2 Visible Record Header	2-7
6.3 Visible Record Body	2-8
6.4 Visible Record Trailer	2-8
7 LOGICAL RECORD	2-8
7.1 Overview	2-8
7.2 Logical Record Segment Header	2-10
7.3 Logical Record Segment Encryption Packet	2-11
7.4 Logical Record Segment Body	2-12
7.5 Logical Record Segment Trailer	2-12
7.6 Checksum Algorithm	2-13
7.7 Logical Record Body	2-13
7.8 Logical Record Encryption Rules	2-14
8 EXPLICITLY-FORMATTED LOGICAL RECORD (EFLR)	2-14
8.1 Overview	2-14
8.2 Component	2-15
8.3 Sets	2-16
8.4 Templates	2-18
8.5 Objects	2-18
8.6 Attributes	2-20
9 INDIRECTLY-FORMATTED LOGICAL RECORD (IFLR)	2-22
9.1 Overview	2-22
9.2 Data Descriptor Reference (DDR)	2-22
9.3 IFLR Modifier	2-23
10 LOGICAL FILE	2-23
10.1 Overview	2-23
10.2 Logical File Organization	2-24
11 REPRESENTATION CODE DESCRIPTIONS	2-24
11.1 Introduction	2-24
11.2 Representation Code Summary	2-24
11.3 Descriptions of Simple Representation Codes	2-26
11.4 Descriptions of Compound Representation Codes	2-36

	Page
Figures	
2-1—Conceptual View of a Logical Format (Bound to a Physical Format)	2-5
2-2—Visible Record Structure	2-7
2-3—Logical Record Segmentation (Without Encryption)	2-10
2-4—Logical Record Segmentation (With Encryption)	2-10
2-5—Application-Level View of a Set (Type = FOO)	2-14
2-6—Component-Level View of a Set	2-15
2-7—Component Structure	2-16
2-8—Examples of Attribute Components	2-21
2-9—Structure of an IFLR Body and Relation to its Data Descriptor ..	2-22
Tables	
2-1—Visible Record Header Fields	2-8
2-2—Logical Record Segment Header Fields	2-11
2-3—Logical Record Segment Header Attributes	2-11
2-4—Logical Record Segment Encryption Packet Fields	2-12
2-5—Logical Record Segment Trailer Fields	2-13
2-6—Component Structure	2-15
2-7—Component Role	2-16
2-8—Normal, Redundant, and Replacement Set Components	2-18
2-9—Object Component	2-19
2-10—Attribute Components	2-20
2-11—IFLR Modifier Options	2-23
2-12—Representation Code Summary	2-25
2-13—Compound Representation Code Descriptions	2-37

Recommended Practice for Exploration and Production Data Digital interchange Part 2: Logical Format

0 Introduction

0.1 This standard contains introductory, normative, and annotative information. Introductory information comprises everything that precedes Section 1, Scope. The introductory clauses describe how the publication is organized and provide commentary on the history and purpose of this standard. Normative information includes the actual requirements that must be satisfied by any data conforming to the standard. Annotative information is provided as rationale for and illustrations about the normative information and does not constitute part of the standard. The standard is completely stated even if all introductory and annotative information are removed. Annotative information may refer to terms introduced in later paragraphs or parts in order to tie together concepts, i.e., from time to time its value may increase after the reader has made a complete pass over the standard.

0.2 Different styles are used to distinguish between normative and annotative information.

0.3 All normative information is written using this same font, and is contained in either numbered paragraphs or numbered tables. Normative text is aligned with the left margin of the page.

0.4 All figures are annotative, and all annotative text is written in unnumbered paragraphs in Helvetica italic font, indented, as shown here:

This is a paragraph of annotative text. Its purpose is to include informal commentary on the normative information in immediately preceding or following paragraphs and may include references to or usage of normative information in other parts of the standard.

1 Scope

This part describes the logical format of data recorded under API Recommended Practice 66. The logical format is a sequential, media-independent organization of X-bit bytes used to represent data.

2 References

Unless otherwise specified, the most recent editions or revisions of the following standards, codes, and specifications shall, to the extent specified herein, form a part of this standard.

ANSI¹

STD 754-1985 IEEE Standard for Binary Floating Point Arithmetic
X3.4 American Standard Code for Information Interchange (ASCII)

ISO²

8859-1 Information processing—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet No. 1

¹ American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

² International Organization for Standardization. ISO publications are available from ANSI

3 Definitions

3.1 absent value: A placeholder that represents no value where a value is normally expected to be. For attributes, this may be an absent attribute component or a zero count. For channel samples, this is a zero count of the channel's dimension.

3.2 attribute: A named item of information or data pertaining to an object type.

3.3 attribute count: The number of elements in an attribute value.

3.4 attribute label: The name of an attribute.

3.5 attribute representation code: A code that identifies the recorded representation of each element of an attribute value.

3.6 attribute units: An expression that represents the units of measurement of each element of an attribute value.

3.7 attribute value: The value of an attribute. It may be present or absent. When present, it consists of zero or more elements, each having the same units and the same representation.

3.8 characteristic: A subitem of a component that contains one piece of information for a data item. Characteristics of an attribute component, for example, are its label, count, units, representation code, and value.

3.9 checksum: The integer result of a 16-bit cyclic redundancy computation on the bytes of a logical record segment, excluding the checksum result itself and any bytes (e.g., the logical record trailing length) that follow it. It is used to verify possible physical recording errors in the logical record segment.

3.10 component: A construct used to implement recording the parts of a set. It contains a descriptor that specifies which kind of data item the component represents and contains subitems called characteristics that describe and contain the value of the data item. A set consists of a sequence of components.

3.11 component descriptor: The part of a component that describes its role and which characteristics are present.

3.12 component format: A value in the component descriptor that indicates which characteristics are present.

3.13 component role: A value in the component descriptor that identifies what kind of data item the component represents, which include set, object, attribute, absent attribute, replacement set, redundant set.

3.14 compound representation code: A representation code that has two or more subfields.

3.15 consumer: The system or application program or company that reads or uses API Recommended Practice 66 information.

- 3.16 copy number:** One of three parts of an object name. It is used to distinguish two objects of the same type in the same logical file that have the same origin and identifier.
- 3.17 dictionary:** A database in which identifiers and reference values used under API Recommended Practice 66 are maintained and administered.
- 3.18 EFLR:** Explicitly formatted logical record. The content of an EFLR, which consists of a set of objects, is determined from an analysis of the record itself.
- 3.19 element:** One of a list of homogeneous quantities that make up the value of an attribute or channel. Every element of a value has the same units and representation.
- 3.20 explicitly formatted logical record:** See EFLR.
- 3.21 format version:** A two-byte field in the visible record header immediately following the length field that identifies the API Recommended Practice 66 version of the data in the record. The version is a one-byte unsigned integer, preceded by a byte containing the value FF₁₆.
- 3.22 identifier:** One of the three parts of an object name. It is a character string used to distinguish the object from other objects of the same type. For some designated object types, the identifier conveys meaning of the nature of the object, and the identifier and its meaning are maintained in a dictionary.
- 3.23 IFLR:** Indirectly formatted logical record. The content of an IFLR is determined from an analysis of related EFLRs.
- 3.24 indirectly formatted logical record:** See IFLR.
- 3.25 logical file:** The main unit of data exchange. It consists of a sequence of one or more logical records, beginning with a record containing a single FILE-HEADER object.
- 3.26 logical format:** A description of how to encode data in a media-independent sequence of 8-bit bytes. This is the view of API Recommended Practice 66 that is independent of any physical binding.
- 3.27 logical record:** An organization of data values into coherent, semantically-related packets of information. A logical record may have any length greater than sixteen bytes and is organized in one of two syntactic forms: explicitly-formatted logical record (EFLR) or indirectly-formatted logical record (IFLR).
- 3.28 logical record body:** An ordered sequence of bytes representing the primary data of a logical record.
- 3.29 logical record segment:** A construct that contains the structure necessary to describe and support the physical implementation of a logical record. A logical record is implemented as one or more logical record segments. A segment is wholly contained in a visible record, but two segments of the same logical record may be in different visible records.
- 3.30 logical record segment attributes:** The 16 bits of binary data that describe the attributes of a logical record segment.

- 3.31 logical record segment body:** One part of an ordered partition of a logical record body into disjoint parts. A logical record body can be reconstructed by concatenating all the segment bodies of the record.
- 3.32 logical record segment encryption packet:** The part of a logical record header that contains information required to decrypt the segment.
- 3.33 logical record segment header:** The part of a logical record segment that describes its length, attributes, and may contain an encryption packet.
- 3.34 logical record segment trailer:** The part of a logical record segment that contains optional padding, checksum, and segment trailing length.
- 3.35 logical record structure:** A logical record segment attribute. It specifies whether the logical record is an EFLR or IFLR.
- 3.36 object:** A recorded instance of an object type.
- 3.37 object component:** A component that delimits an object and contains its name.
- 3.38 object name:** A three-part unique reference to an object consisting of an origin, a copy number, and an identifier.
- 3.39 object type:** A logical entity of a schema that has a unique type name and one or more defined attributes. Instances of an object type are written in explicitly formatted logical records.
- 3.40 organization code:** A number assigned by API to an organization that identifies the organization and represents schemas and dictionaries defined and administered by the organization.
- 3.41 origin:** One of three parts of an object name. It is a number referring to a distinct ORIGIN object that contains context information for the objects that reference it.
- 3.42 pad count:** The part of the padding (written at the end) that specifies how many padding bytes there are.
- 3.43 padding:** Optional extra bytes following the logical record segment body used to ensure an even length for a segment and also to extend a segment when necessary to fill out a fixed-length visible record.
- 3.44 physical format:** The medium-specific organization of data bytes on a storage unit.
- 3.45 producer:** The system or application program or company that recorded information under API Recommended Practice 66.
- 3.46 redundant set:** A verbatim copy of a set written previously in the same logical file.
- 3.47 replacement set:** An updated copy of a set written previously in the same logical file. It includes any updates made to objects in the set since the original was written.
- 3.48 representation code:** A unique number that identifies a standard encoding for a value as a sequence of one or more contiguous bytes.

3.49 set: A collection of one or more objects of the same object type. A set is recorded in an EFLR, and each EFLR has exactly one set.

3.50 set type: The type of objects in a set.

3.51 subfield: A part of a datum for which the representation is described by a simple (not compound) representation code. For example, the subfields of a datum having representation code OBNAME are, in order, an integer (UVARI), another integer (USHORT), and a string (IDENT).

3.52 template: An ordered group of one or more attributes that represent a default or prototype object, written at the beginning of a set.

3.53 visible record: The interface between the logical format and a medium-specific physical format. A visible record has a header, a body, and a trailing length.

4 Concepts

4.1 The API Recommended Practice 66 logical format is a description of how to encode data in a media-independent sequence of 8-bit bytes. At the highest level, the logical format is a sequence of logical tiles. A logical file is subdivided into a sequence of related logical records. A logical record contains a set of related values and is subdivided into a sequence of logical record segments. A logical record segment is contained in a visible record. A visible record has a header and trailer and may contain one OR more segments. A visible record consists of a sequence of bytes mapped into some physical medium. The physical mapping of visible records into various media types is described in Part 3.

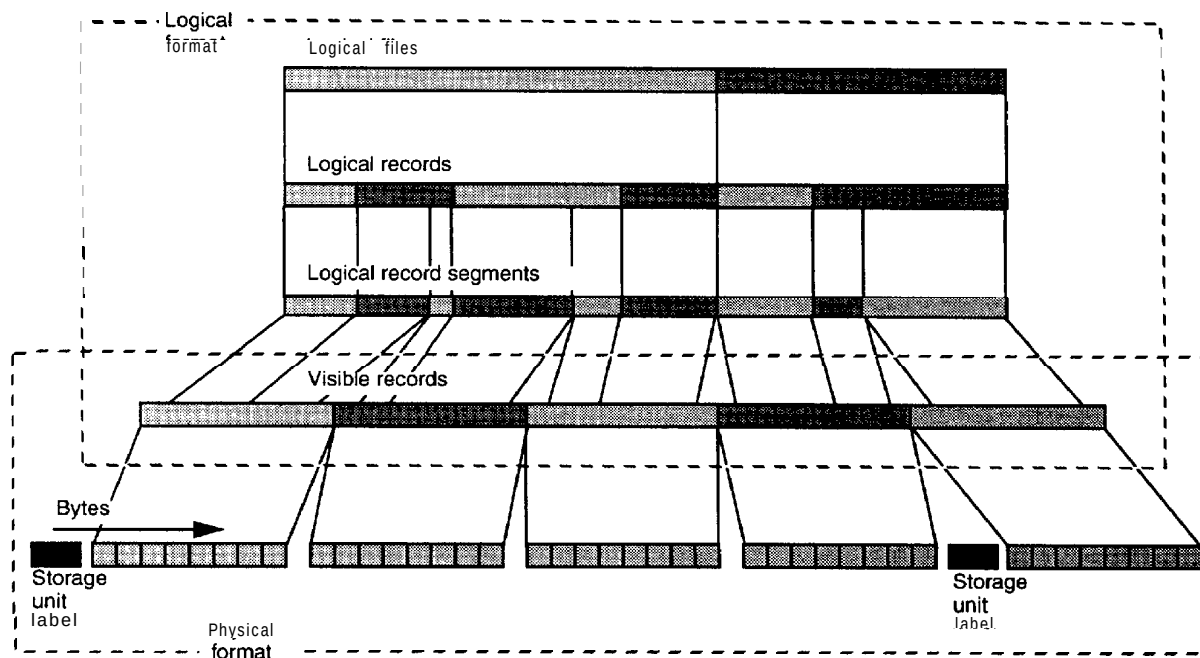


Figure 2-I-Conceptual View of a Logical Format (Bound to a Physical Format)

Figure 2-1 shows the transition from media-dependent organization of data (the physical format) to media-independent organization of data (the logical format). Notice that the two overlap at the visible record layer. Viewed from "above", the visible record layer is just a record-oriented organization of a stream of bytes, which looks the same to higher layers regardless of the medium used to record the bytes. Viewed from "below", the visible record layer is a mechanism that supports mapping logical constructs onto physical constructs such as device blocks and multiple storage units (e.g., tapes or disk files).

Logical record segmentation supports the flexible mapping of logical records (which may be of highly-varying sizes) into visible records. In particular, segments make it possible to begin a new logical record in the middle of a visible record. This results in two benefits. One, fixed-length visible records may be used. This provides a powerful recovery method in case corrupted data is encountered and also allows implementation of random access methods. Two, segmentation allows efficient packing of logical records into larger device blocks while also allowing spanning of large logical records across smaller device blocks.

4.2 A value, whether in a logical record or a visible record header or trailer, is encoded in one or more contiguous bytes according to one of a variety of standard representations. Each standard representation is identified by a unique integer code, called a representation code.

5 Representation Codes

5.1 A representation code is a unique number, specified in Table 2-12, that identifies a standard encoding for a value as a sequence of one or more contiguous bytes. Each representation code has a corresponding symbolic name used in this document to refer to the code. For example, the name UNORM refers to representation code 16, which identifies an encoding of an unsigned integer using two bytes. Only the encodings specified in this document may be used to encode **values** in API Recommended Practice 66. Only the representation codes specified here may be used to identify the encodings, and no representation codes may be redefined except by future editions of this document.

5.2 An encoding representation may be simple or compound. A simple representation is an encoding of a single atomic value such as an integer, decimal number, or character string. A compound representation is an encoding that is built up of previously-defined encodings and represents two or more simple values organized into a useful structure. The length of every representation is either fixed or may be determined from its content.

5.3 A subfield of a representation is any part of the representation that corresponds to a simple encoding (with an exception for representation code DTIME). Each subfield of a representation has a subfield name and subfield position, described in Table 2-13.

Representation code DTIME uses 4-bit fields to represent time zone and month.

Representation codes IDENT and ASCII include USHORT and UVARI values, respectively, to count number of characters in a string. However, there is no simple representation code corresponding to only the characters. Consequently, IDENT and ASCII are considered to be simple representations rather than compound.

5.4 The representation codes specified in Table 2-12 are divided into representation code classes. The codes in a representation code class are capable of representing the same kinds of values. Codes within a class are typically differentiated by amount of range or precision or alignment to a particular computing architecture.

For example, the codes **UNORM** (normal unsigned integer) and **FDOUBL** (double precision floating point) are in the same large class, since each is capable of representing a number. **FDOUBLE** has more range and precision than **UNORM**.

This organization of codes into classes is used when defining attributes for a schema (see Part 1). A representation code restriction may be specified for the attribute to indicate its possible range and precision, but any other code in the class may be used *that* supports the value being recorded.

6 Visible Record

6.1 Overview

6.1.1 Visible records are the interface between the logical format and a medium-specific physical format. They provide a structure that can be mapped into device structures and into which logical record segments may be packed.

6.1.2 A visible record is composed of three disjoint parts in the following order:

- a. Visible record header.
- b. Visible record body.
- c. Visible record trailer.

6.1.3 The parts of a visible record are described in detail in 6.2 to 6.4.

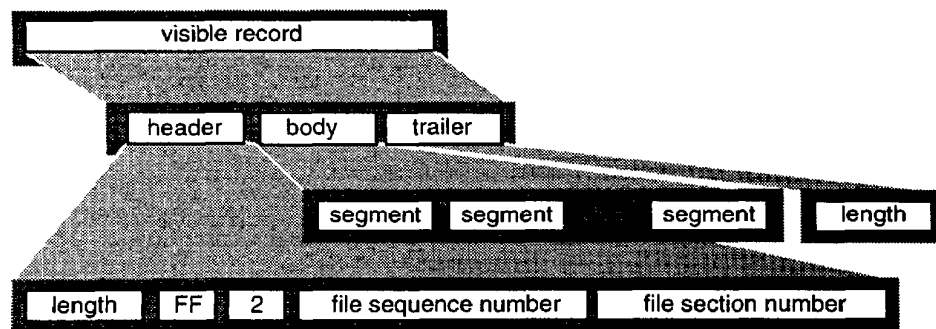


Figure 2-2—Visible Record Structure

6.2 Visible Record Header

6.2.1 The visible record header contains information about the length of the visible record, the format version being used, and the logical file in which the segments contained in the visible record belong.

6.2.2 Table 2-1 describes the fields of a visible record header in the order in which they appear.

Table 2-1 — Visible Record Header Fields

*Note	Field	Size in Bytes	Representation Code
1	Length	4	ULONG
2	The value FF ₁₆	1	USHORT
3	Format version	1	USHORT
4	File sequence number	4	ULONG
5	File section number	2	UNORM

***Notes:**

1. This field specifies the length in bytes of the visible record, including the header, body, and trailer. The maximum length of a visible record is limited by the largest even number having representation ULONG (see Table 2-12). This number is 4 294 967 294. The actual maximum length used for a visible record may depend on which medium the data is recorded (see Part 3). The minimum length of a visible record is the length of its header plus trailer plus the minimum length of a logical record segment (see Table 2-2).
2. This field contains the hexadecimal value FF. Its purpose is twofold: (1) to make the header length even, and (2) to provide a necessary (although not sufficient) identifier of a visible record.
3. The format version field contains the major API Recommended Practice 66 version of the data being recorded. For this edition of the standard, the format version value is 2.
4. This is the sequence number of the logical file to which the segments contained in the visible record body belong (see Part 6). Inclusion of this field implies the following requirement: All of the logical record segments contained in a visible record must belong to the same logical file.
5. This is the section number of the logical file to which the segments contained in the visible record body belong (see 10.1).

How visible records are used for various media types and various application purposes is described in Part 3. This includes how to specify variable-length or fixed-length visible records, for example.

By having a file sequence number in the visible record header it is possible to implement rapid file access methods based on scanning only visible record headers. If visible records are fixed-length, for example, an application may compute forward or backward a given number of visible record lengths, jump to that location and determine by looking at the visible record header whether the jump landed inside or outside the current logical file.

In API Recommended Practice 66, Version 1, the visible record length was arbitrarily limited to 16,384 even though its representation (UNORM) supported lengths up to 65,534. This was done to ensure that implementations on small-memory processors would not be prevented from reading any API Recommended Practice 66 data. This was found to be a burdensome limitation and one that should be administered by the various user groups. So, in this edition of the standard, no arbitrary length limitation is imposed beyond what is supported by the ULONG representation.

6.3 Visible Record Body

The visible record body consists of one or more complete logical record segments (see 7.1.4).

6.4 Visible Record Trailer

The visible record trailer consists of a copy of the length field from the visible record header. This is always present.

7 Logical Record

7.1 Overview

7.1.1 Logical records organize values into coherent, semantically-related packets of information. A logical record may have any length greater than or equal to sixteen bytes

and is organized in one of two syntactic forms: explicitly-formatted logical record (EFLR) or indirectly-formatted logical record (IFLR).

7.1.2 The content of an EFLR is determined from an analysis of the record itself. That is, an EFLR is self-descriptive in the sense that the type of, order of, and amount of information it contains may be determined by reading the record. Further information about EFLRs is given in 8.

7.1.3 The content of an IFLR is determined from an analysis of related EFLRs. The type of, order of, and amount of information in an IFLR is described by information in one or more EFLRs. A reference within each IFLR allows a reader to locate the data items within EFLRs that describe it. Many IFLRs may reference the same descriptor information. Further information about IFLRs is given in 9.

EFLRs and IFLRs are designed to handle different kinds of data, the differences resulting from either form or purpose. One of two views typically applies:

The first view is static vs dynamic. Static data has a fixed or static value within the context of a logical file and is recorded in EFLRs. Dynamic data has a dynamically-changing value in a logical file, each instance of the value occurring in a different IFLR. For example, the description of a sensor is static, whereas the values acquired by the sensor over time are dynamic.

The second view is parametric vs bulk. Parametric data is complex (typically tabular) in form and of known extent, whereas bulk data is sequential in form and has an indefinite extent, i.e., the extent may not be known when data recording begins. Parametric data, e.g., computation parameters, is recorded in 4EFLRs and bulk data, e.g., seismic traces, is recorded in IFLRs.

7.1.4 A logical record is implemented as one or more logical record segments. Segments contain the structure necessary to describe and support the physical implementation of a logical record. A segment is contained in a single visible record and is composed of up to four mutually-disjoint parts in the following order:

- a. Logical record segment header.
- b. Logical record segment encryption packet (only with encryption).
- c. Logical record segment body.
- d. Logical record segment trailer (optional).

7.1.5 The parts of a segment are described in detail in 7.2 through 7.8.

There is a very strict relation between segments and visible records. As stated earlier, a visible record body consists exactly of a whole number of logical record segments. Segments cannot span visible records, but logical records can and frequently do. It is also common for many logical records to fit into one large visible record, in which case each logical record normally has only one segment. A logical record will typically have more than one segment only when it cannot fit completely into the current visible record. The power of segments is that they allow complete separation of logical record and visible record alignment. The only exception to this alignment independence is with the first logical record of a logical file, which is described later.

Figure 2-3 and Figure 2-4 illustrate logical record segmentation.

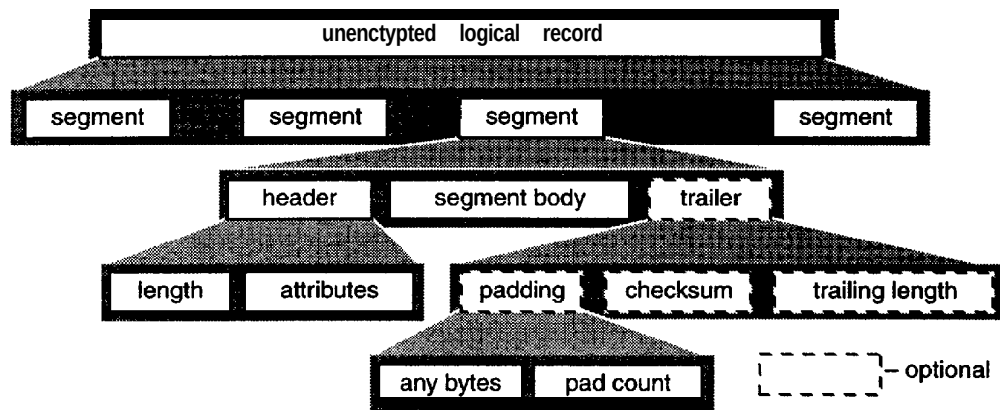


Figure 2-3—Logical Record Segmentation (Without Encryption)

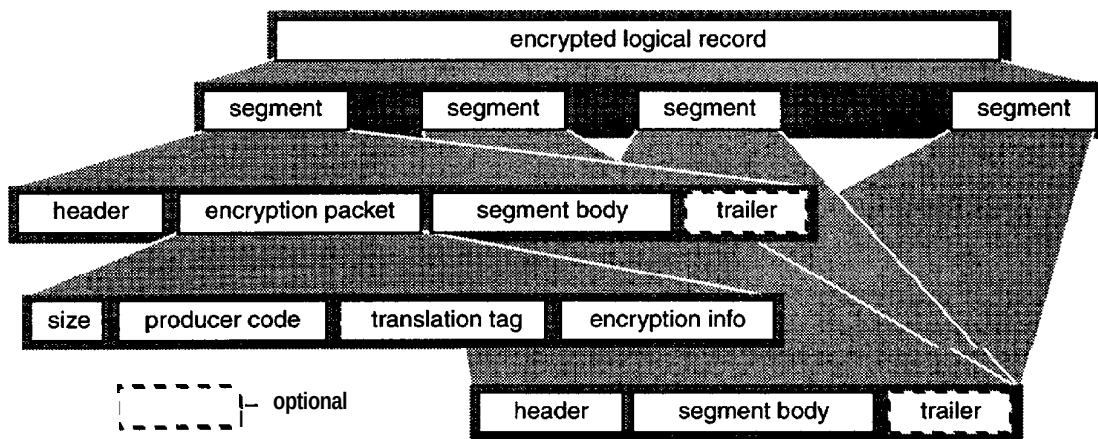


Figure 2-4—Logical Record Segmentation (With Encryption)

7.2 Logical Record Segment Header

7.2.1 The segment header describes the segment's length and attributes. The attributes indicate the kind of logical record, whether the segment is the first, middle, or last of the logical record, whether the segment is encrypted, and which optional parts of the segment are present.

7.2.2 Table 2-2 describes the fields of a logical record segment header in the order in which they appear.

Table 2-2 — Logical Record Segment Header Fields

*Note	Field	Size in Bytes	Representation Code
1	Length	4	ULONG
2	Attributes	2	(see note)

*Notes:

1. This field specifies the length in bytes of the segment, including the header, encryption packet, body, and trailer. A segment is required to have an even number of at least sixteen (16) or more bytes. The maximum length of a logical record segment is limited by the largest even number having representation ULONG (see Table 2-12). This number is 4 294 967 294.

2. Attributes are represented by bit values in a two-byte field, where bit 8 of each byte represents the highest-order bit and bit 1 represents the lowest-order bit (see Table 2-3).

The even length ensures that two-byte checksums can be computed. The minimum size, when combined with visible record overhead ensures a minimum physical block size required by some devices.

7.2.3 Table 2-3 describes the meanings of the bits in the logical record segment header attributes field. A bit is said to be set if its value = 1, and clear if its value = 0.

Table 2-3 — Logical Record Segment Header Attributes

*Note	Byte:Bit	Label	Value	Description
1	1:8	Logical record structure	0	Indirectly-formatted logical record.
			1	Explicitly-formatted logical record.
2	1:7	Predecessor	0	This is the first segment of the logical record.
			1	This is not the first segment of the logical record.
3	1:6	Successor	0	This is the last segment of the logical record.
			1	This is not the last segment of the logical record.
4	1:5	Encryption	0	The segment is not encrypted.
			1	The segment is encrypted.
5	1:4	Reserved	0	This bit is reserved for future use.
6	1:3	Checksum	0	The segment trailer does not have a checksum.
			1	The segment trailer has a checksum.
7	1:2	Trailing length	0	The segment trailer does not have a trailing length.
			1	The segment trailer has a trailing length.
8	1:1	Padding	0	The segment trailer does not have pad bytes.
			1	The segment trailer has pad bytes.
9	2:8-1	Reserved	0	These bits are reserved for future use.

*Notes:

1. All segments of a logical record must have the same value for the logical record structure bit.
2. When the predecessor bit is set, then the segment has a predecessor segment in the same logical record.
3. When the successor bit is set, then the segment has a successor segment in the same logical record.
4. All segments of a logical record must have the same value for the encryption bit. For rules on how to apply encryption, see 7.8.
5. This bit ~~was~~ previously used (in Version I) to indicate presence of the encryption packet. In the current version, presence of the encryption packet is determined by the predecessor and encryption bits.
6. See Table 2-5 and 7.6.
7. See Table 2-5.
8. See Table 2-5.
9. Bits 2: 1-8 must be clear. Meanings for these bits may be defined in later editions of this part.

7.3 Logical Record Segment Encryption Packet

7.3.1 The segment encryption packet immediately follows the segment header if and only if the predecessor bit is clear (no predecessor) and the encryption bit is set in the segment attributes. The packet includes its own length, who encrypted the segment, a translation tag, and may include information about the encryption method and decryption keys.

7.3.2 Table 2-4 describes the contents of an encryption packet.

Table 2-4 — Logical Record Segment Encryption Packet Fields

*Note	Field	Size in Bytes	Representation Code
1	Packet length	2	UNORM
2	Producer code	4	ULONG
3	Translation tag	V	OBNAME
4	Encryption information	V	(see note)

*Notes:

1. This is the length of the packet in bytes, including all fields. This field is required.
2. This is the organization code of the group responsible for the computer program that encrypted the record (see Appendix A). This field is required.
3. The translation tag is the name of an ORIGIN-TRANSLATION object. This field is required.
4. The encryption information is provided by the producer and has a representation known only to the producer (identified by the producer code field) of the logical record. It is used to assist producer-written computer programs in decrypting the logical record. It may consist of zero or more bytes.

The format of the encryption information is known only to the producer of the encrypted data. To the general reader, the encryption information field is just a group of bytes.

(The following discussion uses terminology from Part 6.) The purpose of the translation tag is to make it possible for "pass-through" copy-merge operations by organizations that encounter encrypted records they can't decrypt.

When data from two logical files is merged into a new logical file, some origin number collisions may occur (e.g., origin 3 is used in both input files). Such collisions are resolved by renaming the origin from one of the input files and all references to it in the output file. (This is called "origin translation".) Since origins in encrypted records can't be seen, they also cannot be translated. The translation tag in the encryption packet links this record with an optional ORIGIN-TRANSLATION object that has been recorded "in the clear" and that includes all the origins used in the encrypted record. When files are merged, the origins in the ORIGIN-TRANSLATION object, if provided, and the translation tag can be translated when copied. This makes it possible for later readers able to decrypt the merged data to figure out how to translate the origins in the decrypted record.

7.4 Logical Record Segment Body

The logical record segment body is an ordered sequence of zero or more bytes. The segment body immediately follows the encryption packet when the packet is present or immediately follows the segment header if the packet is not present. The length of the segment body is computed by subtracting the lengths of the encryption packet, the segment trailer, and the segment header from the segment length found in the header.

A zero-length segment body may be useful in the following situation:

A logical record is "edited" by replacing it with a revised version. If the replacement has less data than the original, then the replacement will require less space than the original. However, a replacement must cover the original space exactly. One way to handle this is to add padding. However, if multiple segments are required (e.g., the logical record spans visible records), then the replacement body data may be used up before writing the last segment. In this case, the last segment may have a zero-length segment body.

7.5 Logical Record Segment Trailer

7.5.1 The logical record segment trailer immediately follows the segment body. Its contents are optional and may include a segment trailing length, a checksum, and padding.

7.5.2 Table 2-5 describes the contents of the logical record segment trailer.

Table 2-5 — Logical Record Segment Trailer Fields

*Note	Field	Size in Bytes	Representation Code
1	Pad bytes	V	USHORT
1	Pad count	4	ULONG
2	Checksum	2	UNORM
3	Trailing length	4	ULONG

*Notes:

1. The pad count value is 4 (its own size) plus the number of pad bytes when padding is present (padding bit set in the segment attributes). There may be zero or more pad bytes in addition to the pad count. The pad count is present if and only if the padding bit is set in the segment attributes. Padding is used to ensure an even length for a segment and also to extend a segment when necessary to fill out a fixed-length visible record, and may be used for other reasons.
2. The checksum is a two-byte value computed according to the algorithm given in 7.6. It is present if and only if the checksum bit is set in the segment attributes. The computation applies to everything in the segment that precedes the checksum and does not include itself or the trailing length. If the segment is encrypted, the checksum is computed after encryption has been applied.
3. The trailing length is present if and only if the trailing length bit is set in the segment attributes. When present, it is a copy of the length field from the segment header. Trailing lengths must be used consistently for all segments in a logical file (see 10.2). That is, all segments in a logical file must have a trailing length, or all segments must have no trailing length.

The length of the trailer is $T + C + P$, where T is 4 if the **trailing** length is present and zero otherwise, C is 2 if **the** checksum is present and zero otherwise, and P is the value in the pad count if padding is present and zero otherwise.

When random access is supported by the device, and **trailing** lengths are present, it is possible for an implementation to read logical records from back to front.

7.6 Checksum Algorithm

The checksum, when present, is a 16-bit integer quantity computed using a cyclic-redundancy type checksum algorithm. This algorithm is described below. Note that it assumes that there are an even number of bytes in the data.

1) $c=0$	initialize 16-bit checksum to zero
2) loop $i=1, n, 2$	loop over the data two bytes at a time
3) $t=\text{byte}(i+1)*256+\text{byte}(i)$	compute a 16-bit addend by concatenating the next two bytes of data
4) $c=c+t$	add the addend to the checksum
5) ii carry $c=c+1$	add carry to checksum
6) $c=c*2$	left shift checksum
7) if carry $c=c+1$	add carry to checksum
8) endloop	

7.7 Logical Record Body

The logical record body is an ordered sequence of bytes representing the primary data of the logical record. The segment bodies comprise an ordered partition of the logical record body into disjoint parts. That is, the logical record body can be reconstructed by concatenating all the segment bodies of the record. The logical record body cannot be empty, i.e., at least one segment body must be non-empty. The format and rules for decoding information in the logical record body are described in 8 and 9.

7.8 Logical Record Encryption Rules

7.8.1 Encryption is applied at the logical record level. Data that would normally represent an unencrypted logical record body is encrypted by some method known by the data producer. This may involve expansion or compression of the data; that is, the number of encrypted bytes may be different from the number of unencrypted bytes. Logical record segmentation is then applied to the encrypted bytes, which constitute the recorded logical record body. No header, encryption packet, or trailer bytes (including padding) are encrypted.

7.8.2 When a logical record is encrypted, the encryption bit must be set in all segments. The first segment must have an encryption packet, and no other segments may have an encryption packet. Information required to decrypt the data (including use of any extraneous bytes to satisfy encryption blocking requirements) is placed in the encryption information field of the encryption packet.

Since encryption is applied without regard to segmentation, implementations may need to support encryption and decryption methods that require in-memory buffering of entire logical record bodies. Unencrypted records, on the other hand, can normally be accessed a segment at a time, and some implementations may wish to utilize this feature to conserve memory when processing very large logical records.

8 Explicitly-Formatted Logical Record (EFLR)

8.1 Overview

8.1.1 The data in an EFLR body (logical record body of an EFLR) is formatted using a general structure similar in nature to a table having a column heading, followed by any number of rows. The heading is called a template, and the rows are called objects. Within the template and within each object the column entries are called attributes. However, unlike a table, each object also has a name which is not considered to be the same as an attribute. The table (template plus objects) is called a set. Each EFLR body consists of exactly one set having at least one object, so the term set is a synonym for the term EFLR body.

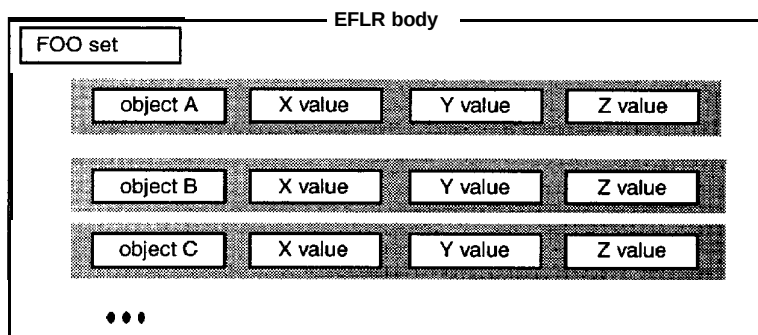


Figure 2-5—Application-Level View of a Set (Type = FOO)

API Recommended Practice 66 uses the terms set, template, object, and attribute instead of table, heading, row, and column entry to emphasize the object-oriented features of the data items in an EFLR. Although the structure

is "similar in nature to a table", there are enough differences from classical tables to warrant the newer terminology

8.1.2 Implementation of the component parts of a set is done using a construct called component. A component contains a descriptor that specifies which kind of data item the component represents and contains subitems called characteristics that describe and contain the value of the data item. A set, then, consists of a sequence of components.

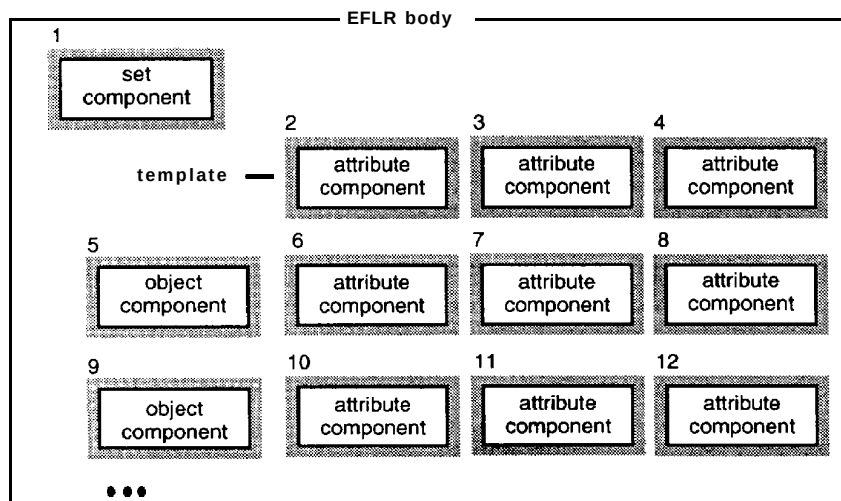


Figure 2-6—Component-Level View of a Set

8.1.3 The detailed description of set structure is given in 8.3.

8.2 Component

8.2.1 A component has a descriptor and zero or more characteristics. Table 2-6 describes the structure of a component. Bits are labeled 1 through 8, where 1 is the low-order bit.

Table 2-6 — Component Structure

*Note	Field	Size in Bytes
1	Descriptor	1
2	Characteristics	V

***Notes:**

1. The descriptor of a component is subdivided into two disjoint bit fields. The first field, bits 8-6, represents the component's role (see Table 2-7). The second field, bits 5-1, represents the component's format (see 8.2.3).
2. A component has zero or more characteristics, as specified by the format subfield of the descriptor. The characteristics of a component identify, describe, and evaluate the data item represented by the component.

8.2.2 The component's role indicates what kind of data item the component represents and also determines which characteristics are represented by the format bits. Currently-defined component roles are described in Table 2-7.

Table 2-7 — Component Role

*Note	Bits 8 - 6	Symbolic Name	Type of Component
1	000	ABSATR	absent attribute
2	001	ATTRIB	attribute
3	010		
4	011	OBJECT	object
5	100		
6	101	RDSET	redundant set
7	110	RSET	replacement set
8	111	SET	normal set

***Notes:**

1. See 8.6.

2. See 8.6.

3. This role is reserved and currently has no meaning. In API Recommended Practice 66, Version 1, this role represents an invariant attribute component. This component type is dropped in the current API Recommended Practice 66 version, since its use increases the complexity of implementations for very little benefit (at best one byte of storage savings per object when used).

4. See 8.5.

5. This role is reserved and currently has no meaning.

6. See 8.3.

7. See 8.3.

8. See 8.3.

8.2.3 The component's format specifies which of the characteristics for the given type of component are actually present in the component. Each component type (identified by its role) has a predefined group of characteristics that may occur in a predefined order. Each characteristic is represented by a bit in the format field of the descriptor. The characteristic is present in the component if and only if its bit is set. A global default value may be specified as part of the definition of a characteristic. This is a value that shall be assumed if the characteristic is not present. Characteristics immediately follow the descriptor in the same order as the format bits by which they are specified. There are no gaps for omitted characteristics.

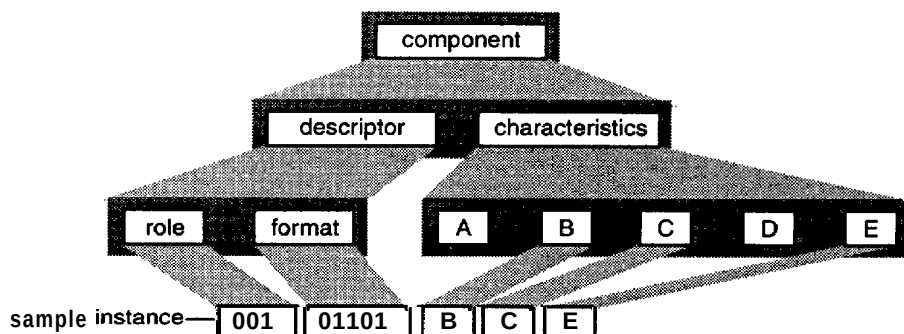


Figure 2-7—Component Structure

8.3 Sets

8.3.1 A set has a non-null type, a template, and one or more objects. It may also have a name and a count of its objects. The type identifies the type of objects in the set, and is also referred to as the object type of the set. The optional set name may be used to distinguish the set from other sets in the logical file when this is necessary and must be non-null when present.

8.3.2 Any number of sets in a logical file may have the same object type, but no two may have the same non-null set name.

8.3.3 There are three kinds of sets (see Table 2-7):

- a. Normal set
- b. Redundant set
- c. Replacement set.

8.3.4 A normal set is as described in 8.3.1. It is the predominant kind of set.

8.3.5 A redundant set is an exact copy of a normal set written previously in the same logical file. Its purpose is to provide redundancy of information as insurance against possible data corruption (e.g., from media errors) of the normal set. The link between a redundant set and the normal set of which it is a copy is by means of the set name or type. If there is a redundant set, it and the normal set of which it is a copy must have the same non-null set name unless the normal set is the only one in the logical file having a given type. In the latter case, set name is optional. Any number of redundant sets may be written for a given normal set.

8.3.6 There is a mechanism using UPDATE objects (see Part 6) for modifying the value of an attribute previously written in a logical file. A replacement set is a near copy of a normal set previously written in the same logical file, the difference being that the replacement set reflects application of all updates that have been made between the time the normal set was recorded and the time the replacement set is recorded. The replacement set has the same objects as the normal set and the same attributes, but some attributes may have updated values. The link between a replacement set and its corresponding normal set is by means of the set name or type as described for redundant sets in 8.3.5. Any number of replacement sets may be written for a given normal set.

The purpose of replacement sets is to provide data replication when a logical file spans multiple storage units. If a normal set is used to interpret the format of an IFLR sequence (or affects computations using IFLR data), having a replacement set on each of multiple storage units ensures that loss of the initial storage unit will not prevent the ability to read data on continuation storage units.

Note that replacement sets are needed in lieu of redundant sets only when updates have been applied to data in the normal set.

8.3.7 A set's type, name, and object count are characteristics of a normal, redundant, or replacement set component. A normal, redundant, or replacement set component is the first component in the set, i.e., in the EFLR body. A set has exactly one normal, redundant, or replacement set component. Table 2-8 describes the format and characteristics of such components.

Table 2-8 — Normal, Redundant, and Replacement Set Components

*Note	FormatBit	Symbol	Characteristic	Representation	Code	Global	Default	Value
1	5	t	type	TIDENT			(see note)	
2	4	n	name	I DENT			null	
3	3	c	count	ULONG			null	
4	2 -1							

*Notes:

1. The type characteristic identifies the object type of the objects in the set and references the schema (see Part 1) under which the object type (i.e., its list of available attributes and their meanings) is defined. The schema is obtained from the ORIGINSHEMA-CODE attribute (see Part 6) located using the tag subfield (see Table 2-13) of the characteristic. The object **type** is given by the identifier subfield of the characteristic. A non-null value (both subfields) of this characteristic must always be explicitly present. There is no global default value.
2. When present the name must be non-null. It is optional unless required to establish a link between redundant and normal sets (see 8.3.5) or between replacement and normal sets (see 8.3.6). No two normal sets in a logical file may have the same non-null name.
3. The count characteristic is optional. When present and non-null it specifies the number of objects in the set. A null value is valid and indicates a count has not been provided.
4. These bits are reserved. For the current format version, they are always clear (value = 0).

8.4 Templates

8.4.1 A **template** is an ordered group of one or more attributes **that** represent a default or prototype object. The template is recorded as one or more attribute components immediately following the set component. If a characteristic is omitted from an attribute component in the template, then a global default value is assumed. Table 2-10 specifies the global default values for attribute components.

8.4.2 The templates in two different sets having the same object type need not be the same. They may differ by the order in which attribute components are written and also may differ in the number and selection of attributes represented. The prototype object written in a template represents a view (or subset) of the attributes defined by a schema for a given object type. See Part 1 for a description of the methods used for defining a schema.

Whereas the object type is a name for the collection of attributes (i.e., column headings) that may apply to the objects of the set, the template contains the list of attributes actually used in the set, which may be a subset of the available attributes.

Note that a template may be empty, that is, a set may have objects with no attributes.

8.4.3 A template is terminated when the first object component is encountered. There must be at least one object component in any set.

8.5 Objects

8.5.1 An object has a name that uniquely identifies it within a logical file and has one or more attributes. The object name is a characteristic of an object component (see Table 2-9). The origin subfield of the object name identifies the schema (see Part 1) under which the object type is defined via the ORIGIN:SCHEMA-CODE attribute (see Part 6). It must identify **the** same schema identified by the tag subfield of the set type.

8.5.2 A schema may declare for each object type whether the identifier subfield of the object name is administered in a dictionary. If so, the ORIGIN:NAMESPACE-CODE and ORIGIN:NAMESPACE-NAME attributes identify the dictionary in which the identifier is included. The namespace assigns a permanent meaning to the identifier. When **an** object

type has a dictionary-controlled identifier, the two-character prefix 'U-' may be used to indicate an identifier not in the dictionary, in effect overriding the dictionary control on an instance by instance basis. No dictionary-administered object identifier may begin with these characters.

8.5.3 An object is written as one object component immediately followed by zero or more attribute and/or absent attribute components. The attributes of an object correspond in order to the attributes in the template, and any characteristic omitted from an attribute component assumes the value of the corresponding characteristic from the template, or global default value if also omitted from the template. Any characteristic explicitly present in an object's attribute component overrides what is in the template. An object may not have more attributes than the template.

Although an object may be followed by zero attribute components, it still has one or more attributes due to inheritance from the template, which must have at least one attribute. It is invalid for an object to have all absent attributes, since this would violate 8.5.1.

8.5.4 Use of an absent attribute component indicates that the attribute in that position has been deleted from the object, i.e., does not even have a default value.

8.5.5 An object may have fewer attribute components than its template. In this case, the attributes for which components have been omitted assume all of the template default characteristics for the object. Since attribute order in the object must match attribute order in the template, only trailing components may be omitted, i.e., if a component is omitted for any attribute, then components must also be omitted for all subsequent attributes for the current object.

To use all template defaults, an "interior" attribute could have a component with only a descriptor, i.e., with no explicit characteristics.

Note the difference between an omitted attribute component and an absent attribute component. In the first case, the attribute exists and has value for the object, whereas in the second case, the attribute has no existence and no value for the object.

8.5.6 An object is terminated when the end of the logical record is reached or when another object component is encountered, whichever occurs first.

8.5.7 Table 2-9 describes the characteristics of an object component.

Table 2-9 — Object Component

*Note	Format	Bit	Symbol	Characteristic	Representation	Code	Global	Default	Value
1	5		n	name	OBNAME			(see note)	
2	4 - 1								

*Notes:

1. The name characteristic uniquely identifies the object in a logical file and must be present. There is no global default value. To be unique in the logical file, the object name cannot match all three subfields—origin, copy number, identifier (see Table 2-13)—of any other object name in the logical file. The identifier subfield must be non-null.
2. These bits are reserved. For the current format version, they are always clear (value = 0).

8.6 Attributes

8.6.1 An attribute has five characteristics, a label, a count, a representation code, a unit (of measure), and a value. The first four characteristics identify and describe the value, which consists of count elements, each having the same unit and representation code. Attributes are represented by attribute components (see Table 2-10).

8.6.2 An absent attribute **has** no characteristics and is represented by an absent attribute component, which has all format bits clear. The purpose of an absent attribute component is to occupy a position for an attribute that is declared in the template but is logically omitted from its object. An absent attribute component may only appear in an object and must not appear in a template. When it appears in an object, it indicates that the attribute corresponding to its position (as declared in the template) is omitted from the object and has no **value**, no count, no unit, and no representation code.

8.6.3 Table 2-10 describes the characteristics of an attribute **component**.

Table 2-10 — Attribute Components

"Note	FormatBit	Symbol	Characteristic	RepresentationCode	Global Default Value
1	5	1	label	1 DENT	(see note)
2	4	C	count	UVARI	1
2	3	r	representation code	USHORT	IDENT
2	2	U	unit	UNITS	null
3	1	V	value	(see note)	null

***Notes:**

1. The label identifies an attribute. It is present if and only if the component is in the template. When present, the label must be non-null, and no two labels in the same template may be the same. There is no global default for the label.
2. The count, representation code, and unit describe the value. See note 3 for further details.
3. The value characteristic represents the value of the attribute. A value consists of zero or more ordered elements, where the number of elements is given by count. Every element has the same representation code and same unit, specified by representation code and unit. Inheritance and consistency rules for attribute characteristics are given in 8.6.4.

8.6.4 The primary consistency rule for attribute characteristics is that count and representation code must accurately describe the number and type of elements in value so that the component can be parsed correctly. When a characteristic is present in an attribute (its format bit is set), it has precedence over the corresponding characteristic in the template (or global default if the attribute is in the template). Consistency among characteristics is checked after inheritance, **that** is after template or global defaults are obtained for omitted characteristics. After inheritance, if count = 0, value must have zero elements. Count = 0 admits to two logical interpretations: If **value** is present, it is considered to be a present but empty list. If value is omitted, it is considered to be absent. After inheritance, if count > 0 value must have count elements. An inherited global default value is considered to match any count. An inherited template value must match the actual count after inheritance.

For example, if count = 3 and a global default value is inherited, then the attribute is considered to have 3 global default elements. However, it is inconsistent if 2 elements are inherited from the template when the attribute count is 3.