P1212
Draft Standard for a
Control and Status Registers (CSR) Architecture
for microcomputer buses

Sponsor

Microprocessor and Microcomputer Standards Committee
of the
IEEE Computer Society

Not yet approved by

IEEE-SA Standards Board

Not yet approved by

American National Standards Institute

Abstract: A common bus architecture (which includes functional components—modules, nodes and units—and their address space, transaction set, CSRs and configuration information) suitable for both parallel and serial buses. Bus bridges are enabled by the architecture, but their details are beyond its scope. Configuration information is self-administered by vendors and organizations based upon IEEE Registration Authority ID.
Keywords: address space, architecture, bus, computer, CSR, interconnect, microprocessor, register, transaction set

Copyright © 1999 by the Institute of Electrical And Electronics Engineers, Inc
345 East 47th Street,
New York, NY 10017, USA
All rights reserved

This is an unapproved IEEE Standards Project, subject to change. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities, including balloting and coordination. If this document is to be submitted to ISO or IEC, notification shall be given to the IEEE Copyrights Administrator. Permission is also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position. Other entities seeking permission to reproduce this document for these or other uses must contact the IEEE Standards Department for the appropriate license. Use of the information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyrights and Permissions
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1333, USA

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
PO Box 1331
Piscataway, NJ 08855-1331
USA

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Introduction

(This introduction is not part of the draft standard, IEEE P1212, Revised Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses)

The IEEE approved project P1212 on December 10, 1996, at the request of Dr. David James, initial Chair of the working group. Participants met informally for almost a year, during which time contact with other industry groups revealed new work (beyond simple reaffirmation of the CSR architecture) whose most natural home would be the P1212 working group. Two industry initiatives, HAVi and the Printer Working Group's (PWG) 1394 task force, were designing enhancements to the configuration ROM facilities of the CSR architecture. The former had been concerned with "self-describing devices" (SDD) and the latter with device discovery and "function discovery services" (FDS). They jointly proposed a new project to the IEEE Microcomputer and Microprocessor Standards Committee who in turn suggested that the work be subsumed within IEEE P1212. As a consequence, the first official meeting of the P1212 working group was convened November 11- 12, 1997 in San Jose, CA, with an expanded scope:

- reexamination and revision of the CSR architecture in light of the experience gained with FutureBus, Scaleable Coherent Interface (SCI) and High Performance Serial Bus—all buses compliant with the CSR architecture;
- coordination of the revised standard with other standards, notably ANSI NCITS 325-1998, which had exploited ambiguities in the original CSR architecture in order to incorporate desirable features; and
- enhancement of configuration ROM facilities to support device discovery independent of a device's software architecture, to permit the accurate characterization of physical instances of particular functions within a device and to enable self-administration of these features by organizations and vendors.

The working group met once again in December of the same year and continued to meet the next year. At the October, 1998 meeting in Ka'anapali, HI, there was a change in the Chair; Brian Batchelder indicated his willingness to serve and was elected without dissent. In subsequent meetings, the working draft was assembled from separate document submissions and refined to its present state. At the working group's final meeting, October 13 in Rennes, France, this draft was approved for initial sponsor ballot.

Significant changes between the prior standard, ISO/IEC 13213:1994, and this document are summarized below:

- The extended address model described in the original standard has never been implemented and is no longer considered useful. It has been deleted from this standard and only the 64-bit fixed address model remains.
- The transaction set (read, write and lock requests and their responses) is fundamentally unchanged from the prior standard, except that the requirements for various data lengths (byte, quadlet, octlet, *etc.*) have been trimmed to match implementation experiences. A new lock function, allocate, is defined to permit efficient management of shared resources.
- The majority of CSRs previously defined have been deleted from the architecture. Experience with actual implementations, both SCI and Serial Bus, demonstrated that either the registers were bus-dependent or else unused. The smaller set of fundamental CSRs that remains is the minimum common to different bus standards. With respect a pair of registers, MESSAGE_REQUEST and MESSAGE_RESPONSE, a common message format has been specified; prior to this change, multiple users were unable to share the message passing facility.
- Although the CSR architecture retains unit architectures as a fundamental building block, the descriptive material about interrupts, message passing, globally synchronized clocks and memory has been removed. These topics are application-dependent and their inclusion in the CSR architecture adds little value.
- Most of the changes affect configuration ROM. Key values unused in contemporary implementations were deleted in order to make them available for future standardization. New fields are defined for the bus information block, one to advertise dynamic changes in configuration ROM (the *generation* field) and the other to describe support for read requests larger than a quadlet (the *max_ROM* field). Textual descriptors are

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

enhanced to provide support for multiple character sets and languages and a new type of descriptor was created to describe graphic images (icons). Both of these descriptors may be modifiable (a seeming oxymoron in the context of read-only memory!); this permits users to attach nicknames to devices. A variety of new configuration ROM entries has been defined in response to the needs of applications under development for Serial Bus.

– The single most significant enhancement to configuration ROM was the introduction of a hierarchical structure of instance and feature directories. These are complementary to the existing unit directories, which characterize the software protocol used to access a device. The instance directories describe physical instantiations of a particular device function; the capability to separately describe physical instances and software protocols is particularly important when one instance is controllable by more than one protocol. Feature directories assist device discovery by permitting functional capabilities to be specified independently of the software protocol.

Accurate and unambiguous specification of the expanded configuration ROM facilities in normative sections of the standard is not sufficient, by itself, to adequately educate designers and promote interoperability. An informative annex is included with examples of good practice in the design of configuration ROM, but this, too, may fall short. The reason is that practices are continually evolving. The working group hopes that trade associations or similar forums will continue the ongoing effort necessary to maintain the value of the foundations laid down by this standard.

Patent notice

NOTE – Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying all patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

The patent holder has, however, filed a statement of assurance that it will grant a license under these rights without compensation or under reasonable rates and nondiscriminatory, reasonable terms and conditions to all applicants desiring to obtain such a license. The IEEE makes no representation as to the reasonableness of rates and/or terms and conditions of the license agreement offered by the patent holder. Contact information may be obtained from the IEEE Standards Department.

Committee membership

The following is a list of active participants in the IEEE P1212 working group (those who attended two or more meetings from inception to the time of publication).

Brian Batchelder, *Chair*
Peter Johansson, *Editor*
Lee Farrell, *Secretary*

Helmut Bürklin
Navin Chandler
Firooz Farhoomand
John Nels Fuller
David James
Greg LeClair
Daniel Meirsman
Atsushi Nakamura
Fritz Nordby
Earl Rydell
Hisato Shima
David Smith
Frank Zhao

The following people were members of the balloting group:

The following people served on the ballot response committee:

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Working group points of contact

Chair:

Brian Batchelder
Hewlett-Packard Company
1115 SE 164th Avenue
Vancouver, WA 98684

(360) 212-4107
(360) 212-4227
batchelder@ieee.org

Editor:

Peter Johansson
Congruent Software, Inc.
98 Colorado Avenue
Berkeley, CA 94707

(510) 527-3926
(510) 527-3856 FAX
pjohansson@aol.com

Secretary:

Lee Farrell
Canon Information Systems
110 Innovation Drive
Irvine, CA 92612

(949) 856-7163
(949) 856-7510 FAX
lee_farrell@cissc.canon.com

Home page: http://www.zayante.com/p1212r

Reflector: P1212r@Zayante.com
Majordomo@Zayante.com (to subscribe)

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Revision history

Draft 0.0 (May 23, 1999)

First release of working draft, based upon efforts to date in working group documents CSR-R02, ROM-R07 and 99-015r1.

Draft 0.1 (July 15, 1999)

Miscellaneous editorial and typographical corrections made throughout the draft.

An extended key specifier ID is required for the use of extended keys; it is no longer possible to default to the ID of the directory specifier for extended key definitions.

Although the Node_Unique_ID entry in the root directory is obsolete (because it is redundant with the EUI-64 in the bus information block), a description of *key* value D_{16}, EUI_64, was added for use in other directories. ANSI NCITS 325-1998 already makes use of this directory entry as Unit_Unique_ID.

The Unit_Location entry (specified by ISO/IEC 13213:1994) was restored with a modified description to reflect only the 64-bit fixed address model.

The *key_ID* value for modifiable descriptors was corrected to $1F_{16}$.

With the above mentioned changes incorporated, the working group voted unanimously to stabilize section 7, "Configuration ROM". Technical changes in the future to that section require a two-thirds affirmative vote of the working group.

An informative annex, "Configuration ROM examples", was added to illustrate typical usage contemplated by the working group.

Draft 0.2 (July 28, 1999)

Text that describes the draft standard's scope and purpose was added the document.

During working group review in Briarcliff Manor, NY, miscellaneous clarifications were made in sections 4, 5 and 6. One of these changes is significant: recipients of read requests addressed to the first two kilobytes of register space are required to generate a response.

The minimum transaction set specified by ISO/IEC 13213:1994 had been incorrectly specified by earlier drafts of P1212. This is rectified: compliant bus standards shall support one, two, four, eight, 16 and 64-byte read and write transactions.

Compliant bus standards shall specify the data format (the relative significance of the data bits or bytes) for lock functions that perform arithmetic operations. This is a necessary complement to the CSR architecture’s description of the targeted data at the addressed location. In addition, lock functions that perform arithmetic operations shall use unsigned operands.

A new clause on split transactions has been added to section 4. Related information, on the immediate effects of CSR architecture registers, in particular RESET_START and NODE_IDS, has been clarified.

New configuration ROM examples on the usage of extended keys have been added.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Draft 0.3 (August 27, 1999)

An introduction was added to the draft, as required by IEEE editorial standards.

A new configuration ROM entry, Directory_ID, was unanimously approved by the working group for addition to section 7.

The working group voted unanimously to stabilize section 4, "Architectural framework", section 5, "Transaction set" and section 6, "CSR definitions", as modified by the editorial changes reflected in this draft. Technical changes in the future to these sections require a two-thirds affirmative vote of the working group.

Minor editorial changes were made in Annex A to clarify the examples.

Draft 0.4 (October 18, 1999)

Miscellaneous editorial and typographical errors throughout the document have been corrected. Textual ambiguities have been clarified, in some cases by the addition of additional figures.

References for the ISO/IEC character set standards have been corrected to accurate, current citations.

The reference to the IANA registry of character set names was revised to explicitly mention the named variable, MIBenum, defined by IANA.

Throughout 7.6, "Required and optional usage", the tables of entries found in CSR architecture directories are not exhaustive lists of all entries permitted in the directory but only an enumeration of common entries.

The Hardware_Version entry is no longer restricted to the root directory.

Draft 1.0 (October 18, 1999)

Identical to Draft 0.4 except for the removal of the change bars.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Contents

1 Overview....1
1.1 Scope....1
1.2 Purpose....2

2 References....3

3 Definitions and notation....5
3.1 Definitions....5
3.2 Notation....8

4 Architectural framework....9
4.1 Modules, nodes and units....9
4.2 Addressing....10

5 Transaction set....13
5.1 Read and write transactions....13
5.2 Lock transactions....14
5.3 Bus-dependent transactions....15
5.4 Split transactions....15
5.5 Completion status....16

6 CSR definitions....19
6.1 STATE_CLEAR / STATE_SET registers....21
6.2 NODE_IDS register....22
6.3 RESET_START register....22
6.4 SPLIT_TIMEOUT register....23
6.5 MESSAGE_REQUEST / MESSAGE_RESPONSE registers....23

7 Configuration ROM....25
7.1 IEEE Registration Authority ID....25
7.2 ROM formats....26
7.3 CRC calculation....29
7.4 Minimal ASCII....30
7.5 Data structures....31
7.6 Required and optional usage....44
7.7 Directory entries....49

Annexes

Annex A (informative) Configuration ROM examples....59

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Draft Standard for a Control and Status Registers (CSR) Architecture for microcomputer buses

1 Overview

1.1 Scope

This is a full-use standard, a revision of ISO/IEC 13213:1994; its scope reflects accumulated experience with the CSR architecture since it was first promulgated as a standard in 1991. In the intervening years, two bus standards, Scaleable Coherent Interface (SCI), IEEE Std 1596-199x, and Serial Bus, IEEE Std 1394-1995, have been the source of most practical implementation experience. The revised scope of the CSR architecture is given below:

a) The overall architectural framework partitions the total available address space into equal spaces available to individual nodes. A node's address space is in turn partitioned into regions which have different usage models, *e.g.*, memory space, private space for vendor uses, configuration ROM and an I/O space (units space) where transactions may have side effects;

b) A minimal transaction set (read, write and lock requests and their associated completion responses) required for compliant bus standards. Bus bridges compliant with this architecture, whether in a homogeneous or heterogeneous environment, are also expected to transport this transaction set;

c) Fundamental control and status registers (CSRs) are defined to provide a common infrastructure for all compliant buses. In some cases the details of the registers are entirely bus-dependent but the function is common to all compliant buses;

d) Message request and response CSRs are specified to enable directed delivery or broadcast of messages to multiple nodes. The message format permits organizations or vendors to define the meaning of the data payload without the need for a centralized registry of all possible formats; and

e) Configuration ROM provides self-descriptive data structures that permit nodes to uniformly characterize the device services available. This is critical for buses that permit live insertion and removal of nodes; each newly inserted node contains sufficient information for it to be uniquely identified and for the requisite device drivers to be loaded.

Although the original CSR architecture anticipated widespread development of bridges between heterogeneous bus standards and a diversity of addressing modes, both fixed and variable, no such implementations have been made. As a consequence, the most significant changes in scope between the earlier CSR architecture and this standard are the adoption of a single, fixed addressing model and the removal of tutorial material pertaining to the design of bridges.

Copyright © 1999 IEEE. All rights reserved

This is an unapproved IEEE Standards Draft, subject to change.

1.2 Purpose

The cardinal purpose of any revised standard is to correct or resolve ambiguities or deficiencies in the prior standard that have been revealed by applications (or attempted applications) of the earlier standard. This standard also preserves the following objectives of the original standard:

a) *Scalability.* The CSR architecture is intended to be applicable to a wide range of bus standards and furthermore to a wide range of device capabilities implemented by particular products;

b) *Extensibility.* The architecture supports contemporary processor and bus designs and, to the extent that one can anticipate the future, should be a useful framework for similar new designs. In particular, the addressing model is well suited to 32- and 64-bit processors. The CSR architecture creates no obstacles for the designers of adapters that bridge between common system memory and I/O buses;

c) *Open systems interoperability.* The configuration ROM data structures, as well as the fundamental CSRs, provide tools which vendors may use to guarantee that their devices will work with each other. The segregation of one unit architecture from another tends to restrict possible adverse affects by one module, node or unit upon another;

d) *Manageability.* The CSR architecture permits the design of systems that require little or no administration and are simple to support. Automatic configuration based upon configuration ROM is a key feature; indeed, it is essential when the applicable bus standard supports live insertion and removal of devices; and

e) *Parallelism.* Although not directly addressed by this standard, the architecture contains no features that might preclude its use in a multiprocessor configuration.

The above is a smaller set of objectives than described in ISO/IEC 13213:1994. The working group responsible for the development of this revised standard believes that its usefulness has been enhanced by restricting some of the unrealized goals of the earlier work.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

2 References

The standards named in this section contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision; parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid international standards.

The following approved international standards may be obtained from ISO Central Secretariat, 1 rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse or from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036-8002, USA.

ISO/IEC 639:1998, Code for the representation of names of languages

ISO/IEC 639-2:1998, Codes for the representation of names of languages—Part 2: Alpha-3 code

ISO/IEC 646:1991, Information Technology—ISO 7-bit coded character set for information interchange

ISO/IEC 10646-1:1993, Information Technology—Universal multiple-octet coded character set (UCS)—Part 1: Architecture and basic multilingual plane

ISO/IEC 9899:1990, Programming Languages—C

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

3 Definitions and notation

For the purposes of this standard, the following definitions, terms and notational conventions apply. IEEE Std 100-1992, The New IEEE Standard Dictionary of Electrical and Electronics Terms, should be consulted for terms not defined in this section.

3.1 Definitions

3.1.1 Conformance

Several keywords are used to differentiate levels of requirements and optionality, as follows:

3.1.1.1 bus-dependent: A keyword used to describe objects—bits, bytes, quadlets, octlets and fields—or the code values assigned to these objects in cases where either the object or the code value is set aside for definition by the applicable bus standard. In cases where the bus standard does not define an object or code value it shall be considered reserved.

3.1.1.2 expected: A keyword used to describe the behavior of the hardware or software in the design models assumed by this standard. Other hardware and software design models may also be implemented.

3.1.1.3 ignored: A keyword that describes bits, bytes, quadlets, octlets or fields whose values are not checked by the recipient.

3.1.1.4 may: A keyword that indicates flexibility of choice with no implied preference.

3.1.1.5 reserved: A keyword used to describe objects—bits, bytes, quadlets, octlets and fields—or the code values assigned to these objects in cases where either the object or the code value is set aside for future standardization. Usage and interpretation may be specified by future extensions to this or other standards. A reserved object shall be zeroed or, upon development of a future standard, set to a value specified by such a standard. The recipient of a reserved object shall not check its value. The recipient of an object whose code values are defined by this standard shall check its value and reject reserved code values.

3.1.1.6 shall: A keyword that indicates a mandatory requirement. Designers are required to implement all such mandatory requirements to assure interoperability with other products conforming to this standard.

3.1.1.7 should: A keyword that denotes flexibility of choice with a strongly preferred alternative. Equivalent to the phrase "is recommended."

3.1.2 Technical glossary

The following terms are used in this standard:

3.1.2.1 bridge: An active component that forwards request and response subactions between buses.

3.1.2.2 broadcast request: A request that is distributed to all nodes on a bus.

3.1.2.3 bus ID: The most significant bits of the 16-bit node ID, which uniquely identify a particular bus within a group of interconnected buses. The number of bits used to represent bus ID shall be specified by the applicable bus standard; the remaining less significant bits of node ID constitute the local ID.

3.1.2.4 byte: Eight bits of data.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

3.1.2.5 command reset: An initialization event that is initiated by a write to the RESET_START register.

3.1.2.6 directory: A contiguous collection of one or more entries, which is contained within the node's configuration ROM.

3.1.2.7 doublet: Two bytes, or 16 bits, of data.

3.1.2.8 gigabyte: A quantity of data equal to 2^{30} bytes.

3.1.2.9 immediate effect: An effect of a transaction that appears to occur between the time the request subaction is accepted and the response subaction is returned. If a bus standard allows CSR transactions to be split, and sufficient time is allowed between the acceptance of a request subaction and the return of a response subaction, an immediate effect can be emulated by a processor on the node.

3.1.2.10 kilobyte: A quantity of data equal to 2^{10} bytes.

3.1.2.11 local ID: The least significant bits of the 16-bit node ID, which uniquely identify a particular node on a bus. The number of bits used to represent local ID shall be specified by the applicable bus standard; the remaining more significant bits of node ID constitute the bus ID.

3.1.2.12 leaf: A contiguous information field that is pointed to by a configuration ROM directory entry. A leaf contains a header (length and CRC specification) and other information fields.

3.1.2.13 megabyte: A quantity of data equal to 2^{20} bytes.

3.1.2.14 memory space: A portion of a node's address space, which may provide access to a memory controller unit. Other types of units may also be mapped to non-overlapping portions of the memory space.

3.1.2.15 module: A physical package (printed circuit board or enclosure for multiple boards) that consists of one or more nodes.

3.1.2.16 node: An addressable device attached to a bus. Although multiple nodes may be present within the same physical enclosure (module), each has its own bus interface and address space and may be reset independently of the others.

3.1.2.17 node ID: A 16-bit number that uniquely differentiates a node from all other nodes within a group of interconnected buses. Although the structure of the node ID is bus-dependent, it usually consists of a bus ID portion and a local ID portion. The most significant bits of node ID are the same for all nodes on the same bus; this is the bus ID. The least-significant bits of node ID are unique for each node on the same bus; this is called the local ID. The local ID is assigned as a consequence of bus initialization.

3.1.2.18 node space: The 256 terabytes of address space that may be available to each node. Addresses within node space are 48 bits and are based at zero. Node space includes memory space, private space, register space and units space (a subset of register space).

3.1.2.19 octlet: Eight bytes, or 64 bits, of data.

3.1.2.20 power reset: An initialization event triggered by the restoration of primary power.

3.1.2.21 private space: A 256 megabyte portion of node space, with a base address of FFFF E000 0000_{16}, allocated for vendor-dependent uses local to the node.

3.1.2.22 quadlet: Four bytes, or 32 bits, of data.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

3.1.2.23 register: A term used to describe quadlet-aligned addresses that may be read or written by bus transactions. In the context of this standard, the use of the term register does not imply a specific hardware implementation. For example, in the case of split transactions that permit sufficient time between the request and response subactions, the behavior of the register may be emulated by a processor.

3.1.2.24 register space: A 256 megabyte portion of node space with a base address of FFFF F000 0000_{16}. Registers defined by this standard are located within register space as are bus-dependent registers defined by the applicable bus standard.

3.1.2.25 request subaction: A subaction that initiates an action by the recipient. A request subaction contains an address, request type and sometimes data. For example, a read request specifies an address and a length but no data (it is returned in the response subaction) while a write request specifies an address and data.

3.1.2.26 requester: A node that initiates a transaction request by means of a request subaction addressed to one or more nodes.

3.1.2.27 responder: A node that completes a transaction by returning a response subaction.

3.1.2.28 response subaction: A subaction returned by a responder to complete a transaction initiated by a requester. A response subaction contains completion status and sometimes data.

3.1.2.29 root directory: A term used to describe the directory at the top level of the hierarchical configuration ROM directory structure.

3.1.2.30 split transaction: A transaction that consists of a request subaction followed by a separate response subaction. Subactions are considered separate if ownership of the bus is relinquished between the two.

3.1.2.31 subaction: One of the two components in a transaction; a transaction consists of request and response subactions.

3.1.2.32 terabyte: A quantity of data equal to 2^{40} bytes.

3.1.2.33 transaction: A request subaction and the corresponding response subaction. The request subaction transmits a transaction code (such as read, write or lock) and an address; some request subactions include data as well as transaction codes. The response subaction is null for transactions with broadcast destination addresses or broadcast request codes; otherwise it returns completion status and possibly data.

3.1.2.34 unified transaction: A transaction for which the request subaction and its corresponding subaction are completed as an indivisible sequence. Transactions with null response subactions are by definition unified.

3.1.2.35 unit: A component of a node that provides processing, memory, I/O or some other functionality. A node may have multiple units, which normally operate independently of each other.

3.1.2.36 unit architecture: The specification of the interface to and the services provided by a unit implemented within a node.

3.1.2.37 units space: A portion of node space with a base address of FFFF F000 0800_{16}; units space is a subset of register space. The CSRs and other facilities defined by unit architectures are expected to lie within this space.

3.1.3 Abbreviations

The following abbreviations are used in this standard:

CSR Control and status register

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

CRC Cyclical redundancy checksum

EUI-64 Extended Unique Identifier, 64-bits

RID Registration Authority ID

ROM Read-only memory, in particular configuration ROM

3.2 Notation

The following conventions should be understood by the reader in order to comprehend this standard.

3.2.1 Numeric values

Decimal and hexadecimal numbers are used within this standard. By editorial convention, decimal numbers are most frequently used to represent quantities or counts. Addresses are uniformly represented by hexadecimal numbers, which are also used when the value represented has an underlying structure that is more apparent in a hexadecimal format than in a decimal format.

Decimal numbers are represented by Arabic numerals without subscripts or by their English names. Hexadecimal numbers are represented by digits from the character set 0 – 9 and A – F followed by the subscript 16. When the subscript is unnecessary to disambiguate the base of the number it may be omitted. For the sake of legibility, hexadecimal numbers are separated into groups of four digits separated by spaces.

As an example, 42 and $2A_{16}$ both represent the same numeric value.

3.2.2 Bit, byte and quadlet ordering

CSRs and configuration ROM structures defined by this standard use the ordering conventions specified in this clause, which defines the order and significance of bytes within quadlets and quadlets within octlets in terms of their relative addresses—not their physical position on either a parallel or serial bus.

Within a quadlet, the most significant byte is that which has the smallest address and the least significant byte is that which has the largest address, as shown below.

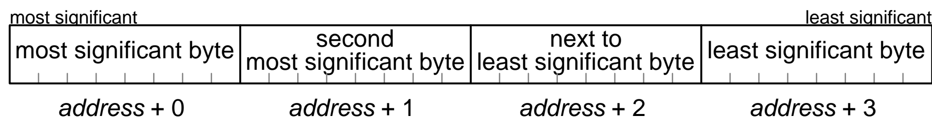

Figure 1 – Byte ordering within a quadlet

Within an octlet the most significant quadlet is that which has the smallest address and the least significant quadlet is that which has the largest address, as the figure below indicates.

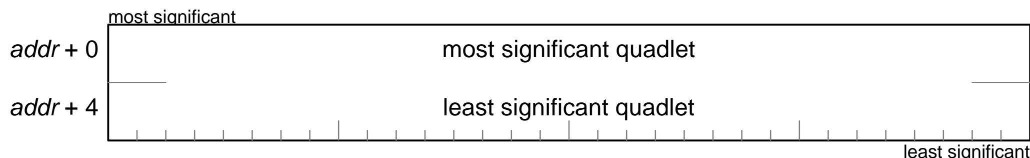

Figure 2 – Quadlet ordering within an octlet

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

4 Architectural framework

This document describes a generic architecture useful to other standards that specify memory-mapped microcomputer buses. This section establishes the basic components of the architecture (modules, nodes and units) as well as their common address model. The architecture enables groups of buses interconnected by bridges (a net), but the details necessary to use such a net are beyond the scope of this standard.

4.1 Modules, nodes and units

The most fundamental component of the CSR architecture is a node, an addressable entity with its own bus interface. Modules and units are other components of the CSR architecture; they have a hierarchical relationship with a node. More than one node may reside in the same physical enclosure (module) and more than one unit may be incorporated within a node, as illustrated by Figure 3.

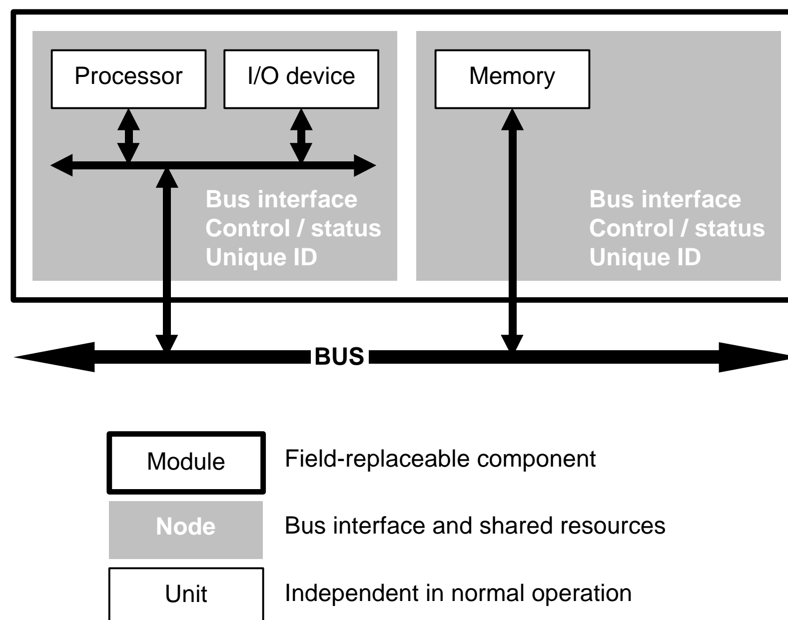

Figure 3 – Component architecture

Each module consists of one or more nodes, which may be independently initialized and configured. A module is a physical packaging concept while a node is a logical addressing concept. When a module contains more than a single node, the physical affinity of its nodes may be described by unique ID information contained within each node. For example, if one node is in some sense "primary", the other nodes in the module may establish their affinity by a reference to the primary node's unique ID. Single-node modules have no need for this additional information.

Each node is independently addressable and contains a dedicated bus interface and other resources common to the node. Nodes operate independently and may be inserted or removed without necessarily affecting the operations of other nodes. A node may contain one or more independently controllable units, each of which may provide processing, memory, I/O or some other functionality. The means by which software may control a unit are specified by a unit architecture, a document that specifies the programming interface for the device. A unit may be further subdivided into multiple logical units or subunits; their definition is beyond the scope of this document.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

4.2 Addressing

This standard utilizes a 64-bit fixed addressing scheme, where the most significant 16 bits of each address are the node ID and the least significant 48 bits are a byte offset into a particular node's address space. This provides address space for up to 65536 nodes, each of which has a generous address space of 256 terabytes. Some buses compliant with this standard reserve portions of the address space for broadcast. Compliant buses may also subdivide the node ID into a bus ID portion and a local ID portion. The hierarchy of the address space is illustrated by Figure 4.

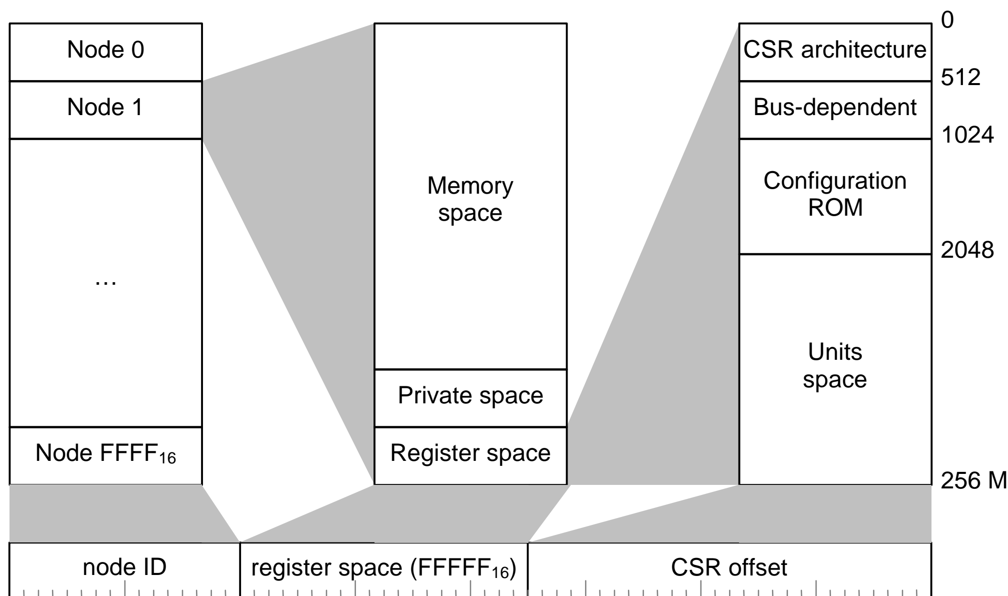

Figure 4 – 64-bit fixed addressing hierarchy

The most significant 16 bits of the address, the node ID, selects a particular node. Within that node's 256 terabyte address space, the next most significant 20 bits select an address space: memory, private or register. The register space is further subdivided into regions for standard CSR architecture registers, bus-dependent registers, configuration ROM and units space. The base address (relative to the node's address space) and size of these address spaces are summarized in the table below.

Table 1 – Address spaces

Name	Base address	Size (bytes)	Description
Memory space	zero	256 T - 512 M	The name reflects an historical expectation that node resources located here exhibit behaviors usually associated with memory. That is, read and write transactions do not have side effects. In fact, any node resources may be located in this address space—even registers with associated side effects.
Private space	FFFF E000 0000_{16}	256 M	Allocated for vendor-dependent uses local to the node. Some implementations may preclude the reuse of an internal address as an externally accessible bus address.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Name	Base address	Size (bytes)	Description
Register space	FFFF F000 0000_{16}	256 M	Register space is allocated for the definition of registers that may have side effects when accessed; it includes standard CSR architecture registers, bus-dependent registers, configuration ROM and units space.
CSR architecture registers	FFFF F000 0000_{16}	512	Registers defined by this or future CSR architecture standards.
Bus-dependent registers	FFFF F000 0200_{16}	512	Allocated for definition by the applicable bus standard.
Configuration ROM	FFFF F000 0400_{16}	1024	Descriptive information for the node (which includes a 64-bit unique identifier) and its units.
Units space	FFFF F000 0800_{16}	256 M - 2048	Allocated for cooperative, shared definition by the applicable bus standard or the organizations or vendors responsible for unit architectures.

The recipient of a read request addressed to the first two kilobytes of register space, FFFF F000 0000_{16} to FFFF F000 $07FF_{16}$ inclusive, shall transmit a response; the request shall have no side effects.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

5 Transaction set

Buses compliant with this standard shall define a transaction set that permits the exchange of data between a requester (initiator of a transaction request) and a responder. A common example of a transaction is a read request initiated by a processor and subsequently satisfied by a response from a memory controller.

This standard defines common facilities for transaction sets to be specified by bus standards and additionally mandates implementation of a minimum transaction set.

The underlying model is one of transactions in which the request and response may be separable from each other. A transaction consists of one or more subactions: indivisible operations unaffected by other subactions on the bus. A request subaction conveys the address, request type and sometimes data from the requester to one or more recipients. When there is a single recipient, a response subaction returns request completion status and sometimes data to the requester.

Transactions may be split or unified. In a split transaction, other subactions may intervene between a request subaction and its corresponding response subaction. The request and response subactions of a unified transaction complete indivisibly; other subactions are inhibited between the initiation of the request subaction and the completion of its associated response subaction. A transaction that consists solely of a request subaction is by definition unified.

Buses compliant with this standard may support split or unified transactions—or both. If a bus implements split transactions it shall support some method to label transactions in order that each response may be matched with its corresponding request.

5.1 Read and write transactions

All buses compliant with this standard shall implement read and write transactions that transfer the data payloads specified by the table below from or to addresses that are a multiple of the size of the data payload.

<table>
<tr><th>Data payload (bytes)</th><th>Comment</th></tr>
<tr><td>1</td><td>Provides accessibility to the smallest unit of data at an arbitrarily aligned address</td></tr>
<tr><td>4</td><td>Commonly referred to as quadlet reads and writes</td></tr>
<tr><td>8</td><td rowspan="2">Permits locations targeted by lock operations to be read with a single transaction</td></tr>
<tr><td>16</td></tr>
<tr><td>64</td><td>Required for the sake of efficiency. This is the size of the MESSAGE_REQUEST and MESSAGE_RESPONSE registers and also the largest payload guaranteed to be forwarded by bridges.</td></tr>
</table>

Bus standards may also define read and write transactions that transfer arbitrary amounts of data from or to arbitrarily aligned addresses—and these transactions may be used to satisfy the above requirements. These are hereafter referred to as block read and block write transactions.

A bus that specifies broadcast address capabilities may permit both quadlet write and block write requests to be broadcast. Although the lack of a transaction response prevents delivery confirmation of the broadcast write request, the expected effect is equivalent to the simultaneous receipt of individual addressed write requests.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Read and write requests shall be processed indivisibly whether completed as a split or unified transaction. For example, after the successful completion of two write requests addressed to the same location, the resultant contents of the location shall be equal to the data contained in one of the two requests (as opposed to a mixture of data present in both). This shall apply to quadlet and block transactions equally.

5.2 Lock transactions

Some operations, *e.g.*, test and set, have been implemented atomically when a bus permits the examination of a data value and its subsequent conditional update while the bus remains locked for other access. Because buses compliant with this standard may implement split transactions, another method is necessary to provide equivalent, atomic behavior. For this purpose, a set of *lock* transactions is defined. Locks are transactions that communicate the intended operation to the responder; the responder performs the lock operation indivisibly and returns the result.

Lock requests consist of an address, an optional *arg* value, a *data* value and a *lock function* that specifies the operation to be performed. Lock responses consist of a response status and the *old* value. The relationship between the parameters supplied by the lock request, the current value of the addressed location (in particular the way in which it is both returned as an argument of the lock response and provides input to the lock operation) and the updated value of the addressed location are illustrated by Figure 5.

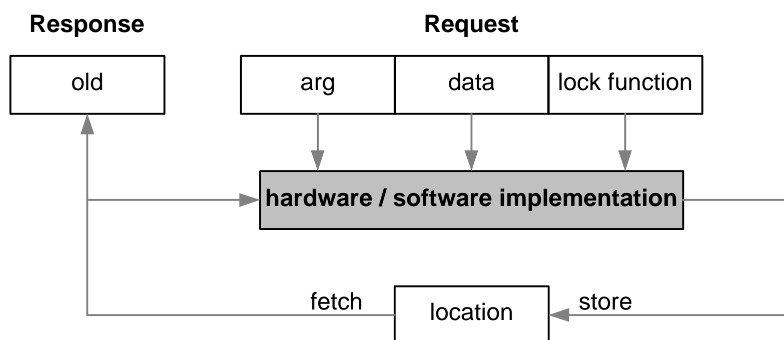

Figure 5 – Simplified lock model

The three operands, *arg*, *data* and *old*, are all the same size and are either four or eight bytes. The lock function, specified by Table 2, selects the operation. The current value at the addressed location is fetched and eventually returned as *old* in the lock response. In the meantime, it serves as an input to the lock operation. The *arg* value, if present, may be compared with the *old* value to determine conditional outcome of the lock operation. At the end, the result of the operation may also be masked and the *new* value is stored at the original location.

Table 2 – Lock operations

<table>
<tr><th>Value</th><th>Name</th><th>Definition</th></tr>
<tr><td>0</td><td></td><td>Reserved; not to be used</td></tr>
<tr><td>1</td><td>mask_swap</td><td><code>new = (data & arg) | (old & ~arg);</code></td></tr>
<tr><td>2</td><td>compare_swap</td><td><code>if (old == arg) new = data;</code></td></tr>
<tr><td>3</td><td>add_big</td><td rowspan="2"><code>new = old + data;</code></td></tr>
<tr><td>4</td><td>add_little</td></tr>
<tr><td>5</td><td>bounded_add_big</td><td><code>if (old != arg) new = old + data;</code></td></tr>
<tr><td>6</td><td>wrap_add_big</td><td><code>new = (old != arg) ? old + data : data;</code></td></tr>
</table>

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Value	Name	Definition
7	Bus-dependent	Allocated for definition by the applicable bus standard
8	*allocate_big*	`if (old >= arg) new = old - data;`
9	*allocate_little*	
A_{16} – F_{16}		Reserved for definition by future CSR architecture standards

NOTE – A swap command is not explicitly defined, since a *mask_swap* operation that specifies an *arg* value of all ones provides equivalent functionality.

Buses compliant with this standard shall define an encoding for the lock functions specified by Table 2 and sufficient to accommodate 16 values; the specification of other lock functions numbered 10_{16} and higher is a bus-dependent option.

Some of the lock operations perform arithmetic operations; their operands are unsigned integers. Correct implementation of these operations requires knowledge of the relative significance of the bytes within the addressed data; these are denoted by the suffixes *_big* and *_little*. For operations suffixed with *_big*, the byte with the smallest offset within the addressed location is deemed the most significant while for operations suffixed with *_little* it is the least significant. Correct implementation of arithmetic operations also requires that the format of the operands transferred (with respect to the relative significance of the data bits) over the bus be specified; this is not the responsibility of the CSR architecture but of the applicable bus standard.

When a lock transaction addresses a location that has one or more reserved bits or fields, the results are not necessarily obvious. Unless otherwise specified by the applicable bus standard, the behavior of a particular lock function shall be determined by applying rules for reserved fields in order, as follows:

a) The addressed location's *old* value shall be obtained as if *via* a read request and shall be returned in the lock response; reserved fields are read as zeros;

b) An intermediate value shall be calculated according to the C code in Table 2 (this is not explicitly shown but is the right-hand part of each of the assignment statements in the table); and

c) The intermediate value shall be stored in the addressed location as if *via* a write request; reserved fields shall be ignored and remain zero in the location. The contents of the location after this operation are the *new* value.

5.3 Bus-dependent transactions

A bus standard may define additional transactions whose behavior is beyond the scope of this standard. Examples include (but are not limited to) read and write transactions that selectively update bytes within a quadlet, large block transfers, either aligned or unaligned, that promote efficient use of the bus, move transactions (unacknowledged writes) or bus-dependent operations pertaining to initialization, clock synchronization and cache coherency.

5.4 Split transactions

When split transactions are supported, a node may accept more than one request subaction before any are processed. Similarly, several request subactions may be processed before any of the associated response subactions are transmitted. Implementation-dependent request and response queues mediate the flow of these subactions between the node and the bus, as illustrated by Figure 6.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

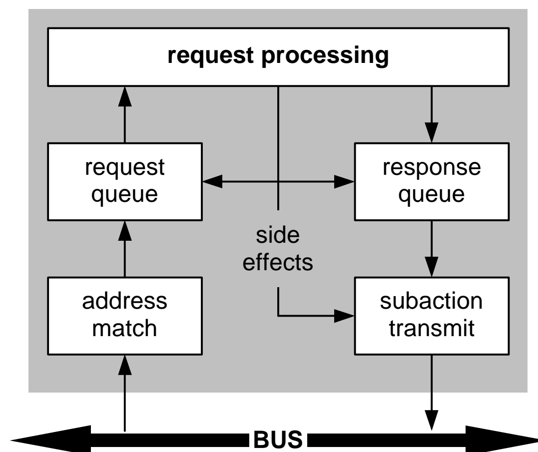

Figure 6 – Request processing model

In the figure above, a request subaction is accepted by a node if queue space is available; the request processing may be deferred. If a response is generated as a result of request processing, the response subaction may be temporarily queued before it is returned to the original requester. The CSR architecture imposes no ordering constraints, either within the individual queues or in relationship to each other.

Although the relative order of transactions is independent with respect to each other, all transaction effects of an individual request shall occur simultaneously after the request subaction has been dequeued (the creation of a response subaction is one of these effects).

Write requests addressed to certain CSRs, *e.g.*, the RESET_START register, may cause queued request or response subactions to be discarded; see clause 6 for detailed information. Other write requests may change the addresses recognized by a node. The latter may have undesirable side effects if the request or response queues are not empty.

5.5 Completion status

Buses compliant with this standard shall define a method for transaction responses to specify the completion status of the request. The minimum set for response status is specified by Table 3.

Table 3 – Transaction response status

Name	Description
complete	The responder has successfully completed the transaction request.
type_error	The transaction request specified an invalid or unsupported operand (*e.g.*, a write to a read-only address).
address_error	The address specified by the transaction request is not accessible or not implemented by the responder.
conflict_error	A lack of resources (*e.g.*, queue space for the receipt of the transaction request or a transient busy condition of some resource needed to service the request) prevents completion of the request. A subsequent retry may succeed.
timeout	The requester has received no completion status for the transaction request.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

When resources are temporarily unavailable to complete a transaction, there may be some bus-dependent information that hints at the duration of the condition. For fleeting busy conditions, bus standards may design retry protocols below the transaction layer. Presumably there would be less overhead to hardware automated retry than to the end-to-end retry engendered by the return of *conflict_error* status.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

6 CSR definitions

This section specifies the format and usage of fundamental control and status registers (CSRs); these CSRs provide a common point of departure for the definition of additional registers by compliant bus standards or unit architectures. The necessity for standardization within these CSRs is motivated by:

– uniform software access for essential features such as configuration of the bus and node address space;

– common definition of facilities generally useful to a wide range of buses; and

– consistent use of well-known addresses, in particular where different applications or protocols might use the same address for message passing.

CSRs are conventionally located with a node's register space, the range of addresses at or above FFFF F000 0000_{16}., although there are no prohibitions that forbid the definition of CSRs at lower addresses. Within register space, address ranges are reserved to different standards documents, organizations or vendors for definition as specified by Table 1.

The table below locates the CSRs defined by this standard in terms of a byte offset from the start of register space, FFFF F000 0000_{16}; all unallocated addresses within the first 512 bytes of register space are reserved for definition by future CSR architecture standards.

Offset	Size (bytes)	Register name	Required	Description
0	4	STATE_CLEAR		Bus-dependent state and control information
4	4	STATE_SET		Sets STATE_CLEAR bits
8	4	NODE_IDS	Y	Contains the 16-bit *node_ID* value used to address the node
$0C_{16}$	4	RESET_START	Y	Used to initiate a command reset
18_{16}	8	SPLIT_TIMEOUT		Time limit for split transactions
80_{16} $C0_{16}$	64	MESSAGE_REQUEST MESSAGE_RESPONSE		Well-known addresses for receipt and protocol demultiplex of 64-byte messages in a standard format

Except as noted in the individual register definitions that follow, all of the above CSRs shall support quadlet read and quadlet write requests.

In the clauses that follow, the conventions illustrated by Figure 7 are used to define CSR formats, initial values and the effects of read, write or other transactions. Bus standards compliant with the CSR architecture should use the same conventions.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

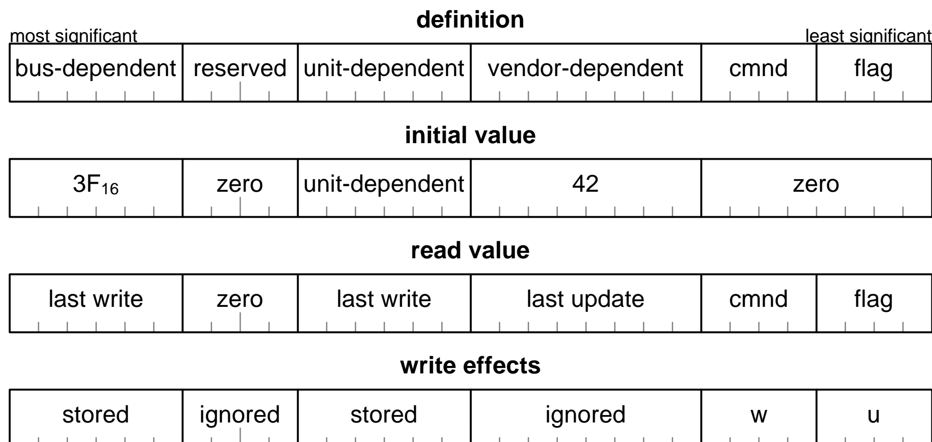

Figure 7 – CSR specification example

The register definition contains the names of register fields. The names are intended to be descriptive, but the fields are defined in the text; their function should not be inferred solely from their names. However, the following field names have defined meanings.

Name	Abbreviation	Definition
bus-dependent	bus-depend	The meaning of the field is defined by the applicable bus standard
reserved	r	The field is reserved for future standardization (see 3.1.1)
unit-dependent	unit-depend	The meaning of the field shall be defined by the organization or vendor responsible for the unit architecture
vendor-dependent	vendor-depend (or v)	The meaning of the field shall be defined by the node’s vendor

CSRs shall assume initial values upon the restoration of power (a power reset), upon a bus reset (if the applicable bus standard defines such a reset) or upon a write to the node’s RESET_START register (a command reset). If the power reset values differ from the command reset values, they are separately and explicitly defined. Initial values for register fields may be described as numeric constants or with one of the terms defined for the register definition. Values for register fields subsequent to a reset may be described in the same terms or as defined below.

Name	Abbreviation	Definition
unchanged	x	The field retains whatever value it had just prior to the power reset, bus reset or command reset.

In addition to numeric values for constant fields, the read values returned in response to a quadlet read request may be specified by the terms below.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Name	Abbreviation	Definition
last write	w	The value of the field shall be either the initial value or, if a write or lock transaction addressed to the register has successfully completed, the value most recently stored in the field.[1]
last update	u	The value of the field shall be that most recently updated by the node hardware or software. An updated field value may be the result of a write effect to the same register address, a different register address or some other change of condition within the node.

The effects of data written to the register are specified by the terms below.

Name	Abbreviation	Definition
effect	e	The value of the data written to the field may have an effect on the node's state, but the effect may not be immediately visible by a read of the same register. The effect may be visible in another register or may not be visible at all.
ignored	I	The value of the data written to the field shall be ignored; it shall have no effect on the node's state.
stored	s	The value of the data written to the field shall be immediately visible by a read of the same register; it may also have other effects on the node's state.

Reserved fields within a register shall be explicitly described with respect to initial values, read values and write effects. Initial values and read values shall be zero while write effects shall be ignored. CSRs that are not implemented, either because they are optional or they fall within a reserved address space, shall abide by these same conventions if a response of *complete* is returned for a read, write or lock request.

6.1 STATE_CLEAR / STATE_SET registers

The STATE_CLEAR and STATE_SET registers are bus-dependent but if implemented shall exhibit behaviors mandated by this standard. When a one is written to a modifiable bit in the STATE_CLEAR register the bit shall be zeroed; contrariwise, when a one is written to a modifiable bit in the STATE_SET register the bit shall be set to one. The format of both registers is specified by Figure 8.

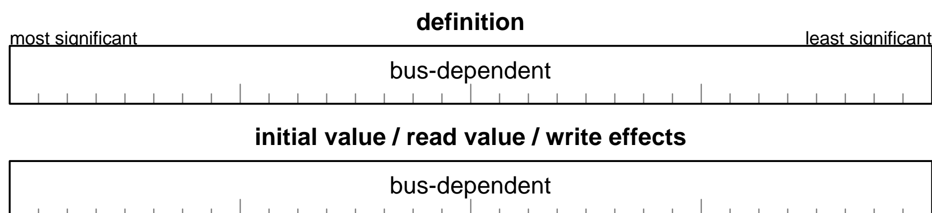

Figure 8 – STATE_CLEAR / STATE_SET format

[1] For clarity, read values for a field in a register that accepts lock transactions may be described as *last successful lock* rather than *last write*. However, the abbreviation in both cases remains *w*. Similar liberties may be taken with the use of *conditionally stored* in place of *stored* when the action occurs as the result of a lock transaction, but the corresponding one-letter abbreviation, *s*, is also unchanged.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

6.2 NODE_IDS register

The NODE_IDS register reports and permits modification of a node's address. The format of the register is specified by Figure 9.

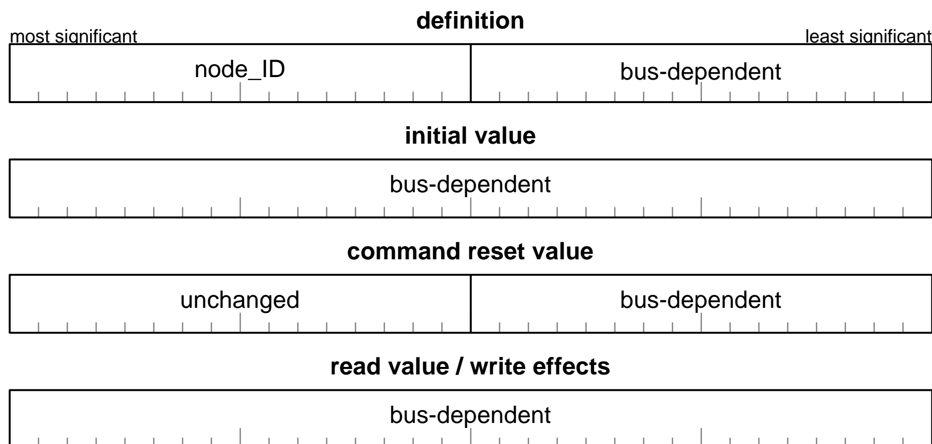

Figure 9 – NODE_IDS format

The *node_ID* field forms the most significant 16 bits of a 64-bit address used to reference a location within the node's address space. Bus standards may subdivide the *node_ID* field into a bus ID and a local ID. The bus ID provides software with a method to reconfigure the bus address portion of the node's address space while the local ID is typically assigned as a consequence of bus initialization (usually by hardware).

The value of *node_ID* shall be stable across a command reset; it may change as a result of a power reset or a bus reset (if such a reset is defined by the applicable bus standard).

If the applicable bus standard permits all or part of *node_ID* to be modified, the affected node shall delay the return of a transaction response until all visible effects of the write are complete. In the case of the NODE_IDS register, the return of a *complete* status in response to a write request shall indicate that the node recognizes subsequent requests that specify the newly assigned *node_ID* in their destination address. Bus standards that include the responder's *node_ID* in the response information shall specify whether the prior or updated value from the NODE_IDS register is returned.

6.3 RESET_START register

A quadlet write request to the RESET_START register shall cause an immediate command reset without regard for the data value of the request. Command reset shall initialize the node's bus interface, which may cause any pending request or untransmitted response subactions to be discarded, but shall not inhibit the node's ability to respond to the write request that caused the reset. Command reset may also invoke an initialization process. The format of this write-only register is specified by Figure 10.

Reset and initialization of a node's bus interface is an essentially instantaneous event, complete upon the transmission of the response subaction for the write request addressed to RESET_START. All of a node's fundamental CSRs (those specified in this section) shall be accessible to all supported transactions as soon as the response subaction is transmitted. Access to other node facilities, *e.g.*, configuration ROM or unit registers, may be delayed. Except in the case of configuration ROM, bus-, node- or unit-dependent means shall be used to determine when access is restored. This standard defines a uniform method to determine whether not configuration ROM is accessible (see 7.2).

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

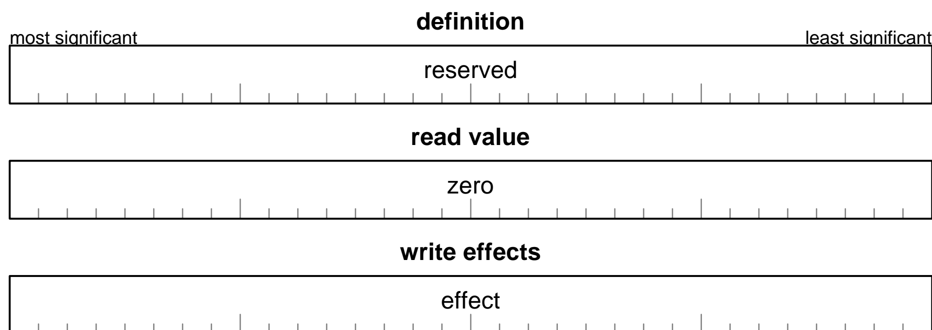

Figure 10 – RESET_START format

6.4 SPLIT_TIMEOUT register

The SPLIT_TIMEOUT register sets the time-out value for the detection of split-transaction errors. When a request subaction is originated, the requester shall commence a timer. If the corresponding response subaction is not received within the time-out period established by SPLIT_TIMEOUT, the requester shall terminate the transaction with a completion status of *timeout*. The time-out value may optionally specify the amount of time permitted a responder to attempt successful transmission of a response subaction. If this bus-dependent usage of SPLIT_TIMEOUT is elected, all nodes on a bus should share the same time-out value; the bus standard shall define complementary behaviors for the requester and responder. Figure 11 specifies the format of the SPLIT_TIMEOUT register.

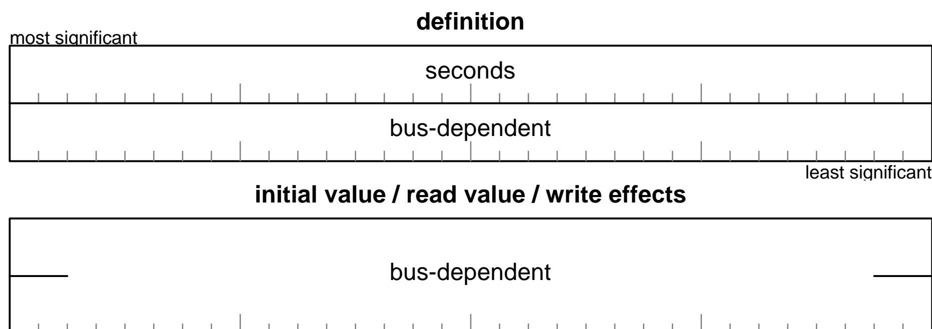

Figure 11 – SPLIT_TIMEOUT format

The *seconds* field in the most significant portion of the SPLIT_TIMEOUT register shall specify the time-out value in units of seconds. Although the least significant portion of the register shall be defined by the applicable bus standard, it should in represent a fractional second whose denominator is approximately 2^{32}. If the bus standard defines a system clock, the fractional portion of SPLIT_TIMEOUT may be represented in the same units as that clock.

When SPLIT_TIMEOUT is zero, split-transaction error detection shall be disabled; the time-out period is effectively infinite.

Updates of a node's SPLIT_TIMEOUT register while split transactions are active may have undesirable side effects.

6.5 MESSAGE_REQUEST / MESSAGE_RESPONSE registers

Two 64-byte registers, both aligned on 64-byte boundaries, provide well-known addresses for the exchange of messages (requests and responses) between nodes. The registers are write-only and shall support only 64-byte write

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

requests. Because multiple protocols may use these registers, the first two quadlets of message data are standardized, as specified by Figure 12.

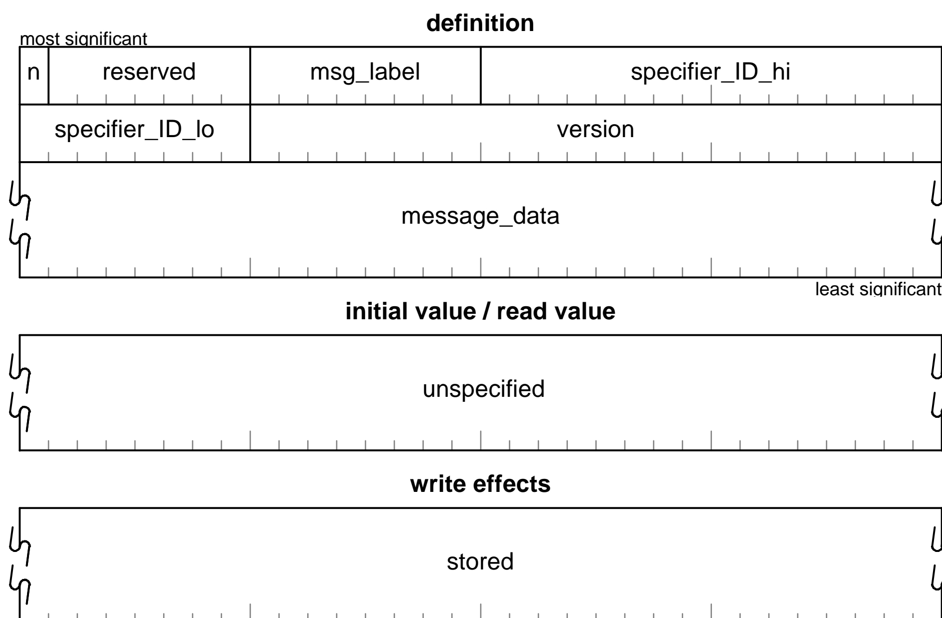

Figure 12 – MESSAGE_REQUEST / MESSAGE_RESPONSE format

The usage of the *notify* bit (abbreviated as *n* in Figure 12) depends upon its context. When a message is written to the MESSAGE_REQUEST register and *notify* is one, a response should be written to the requester's MESSAGE_RESPONSE register upon disposition of the request. Otherwise, when *notify* is zero, the requester indicates that no response shall be returned. When a message is written to the MESSAGE_RESPONSE register and the *notify* bit is one, the request has been refused by the recipient because it is unrecognized, in which case the data written to MESSAGE_RESPONSE shall be identical to that received at MESSAGE_REQUEST. When a response is received with a zero *notify* bit, the request has been processed by an application and the success or failure of the request should be indicated by information in the *message_data*.

The *msg_label* field permits the requester to correlate a response written to MESSAGE_RESPONSE with an outstanding request. Despite the presence of the *msg_label* field, the message passing facilities of these two registers are, for all practical purposes, unconfirmed. Transaction time-outs between request and response, as well as retry protocols, are beyond the scope of this standard.

The *specifier_ID* field shall contain a 24-bit RID (see 7.1). The organization or vendor identified by *specifier_ID* shall be responsible to define the meaning and usage of the remainder of the message requests or response.

The meaning and usage of the *version* and *message_data* fields shall be defined by the organization or vendor identified by *specifier_ID*. The 48-bit number formed by appending the value of *version* to the value of *specifier_ID* shall reference document(s) that specify the format of *message_data*.

When a message is written to the MESSAGE_RESPONSE register, the value of the *msg_label*, *specifier_ID* and *version* fields shall be identical to their values written to the MESSAGE_REQUEST register when the request was received.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

7 Configuration ROM

Configuration ROM is that part of a node's addressable space that lies between FFFF F000 0400_{16} and FFFF F000 $07FF_{16}$, inclusive, plus any data structures (directories or leaves) indirectly addressed from this address range. Quadlet read requests shall be supported over the entire address range; read requests that transfer 64 or more bytes per transaction should be supported. The effects of write requests and lock requests are unspecified. The standard facilities of configuration ROM are based upon the following criteria:

– Unique ID. Some buses assign part of a node's address dynamically whenever nodes are inserted or removed. Configuration ROM shall provide for an identifier unique among all nodes and stable across these bus reconfiguration events;

– Bus-dependent information. Space is allocated at a fixed location to identify the pertinent bus standard and to permit it to define configuration ROM structures necessary to describe its own characteristics;

– Rapid search. Compactness of data structures and the use of keywords permit bus enumerators to quickly abandon searches of nodes that do not provide desired functions;

– Device driver identification. A scheme based upon unique 24-bit numbers administered by the IEEE permits standards organizations, vendors and others to characterize the software interface (unit architecture) for a particular function;

– Physical instance differentiation. Because one function may be controllable via several software interfaces, configuration ROM implements an instance hierarchy that identifies which unit architectures correspond to the same physical element in the device;

– Unambiguous state. During initialization or device self-test, the first quadlet of configuration ROM provides an indication that the rest of configuration ROM is not accessible;

– Extensibility. Optional ROM data structures support large numbers of bus-, unit- or vendor-dependent parameters. This standard provides a uniform framework for others to express specialized features; and

– Minimal format. Although its use is deprecated, a minimal format of configuration ROM is available to vendors who wish to define their own data structures.

7.1 IEEE Registration Authority ID

The sources of the information expressed in configuration ROM may be organized into three groups: *a)* this standard and future CSR architecture standards, *b)* the bus standard utilized by the device and *c)* other organizations and vendors. The format and usage of both the CSR architecture and bus-dependent information are specified by this standard and the pertinent bus standard; each of these sources is a singular case that requires no registration scheme. The third category, on the other hand, may include a multiplicity of organizations or vendors, each of which shall be uniquely recognizable.

The IEEE Registration Authority maintains a database of 24-bit numbers uniquely assigned to organizations and vendors. Throughout the rest of this document such a 24-bit number is referred to as a registration ID (RID).[2] Any organization or vendor that wishes to define the format and usage of configuration ROM information not already specified by either the CSR architecture or a bus standard shall obtain a RID from:

[2] This 24-bit number is variously referred to in ISO/IEC 13213:1994 as an organizationally unique ID (OUI), company ID or vendor ID. Because of historical confusion about byte ordering, this standard renames the number to avoid any reference to earlier, conflicting definitions.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Institute of Electrical and Electronic Engineers, Inc.
IEEE Registration Authority
c/o IEEE Standards Association
445 Hoes Lane
Piscataway, NJ 08855-1331

The usage of a RID and the contexts in which it is meaningful are explained in subsequent clauses.

7.2 ROM formats

The format of the first kilobyte of configuration ROM is identified by the quadlet at FFFF F000 0400_{16}. During node initialization and self-test, access to configuration ROM shall be supported for only this quadlet. Until the node enables access to other addresses within configuration ROM this quadlet shall have a value of zero.

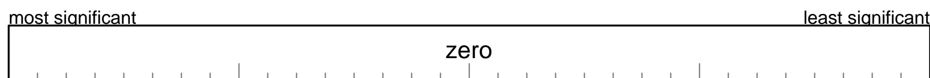

Figure 13 – First ROM quadlet (during initialization)

Upon completion of initialization and self-test, quadlet read requests addressed to any location within the first kilobyte of configuration ROM shall not complete with an *address_error* response. Some of the locations within the first kilobyte of configuration ROM may be unused in the sense that they are not part of the navigable hierarchy of configuration ROM; these locations shall return response data values of zero.

A deprecated minimal ROM format is provided for vendors who wish to design proprietary configuration ROM data structures. Minimal ROM format is identified by the first quadlet, as illustrated by Figure 14. Since minimal ROM format inhibits open systems interoperability between devices manufactured by different vendors, its use is deprecated by this standard. Implementers should include vendor-dependent information within the general format configuration ROM described later in this clause.

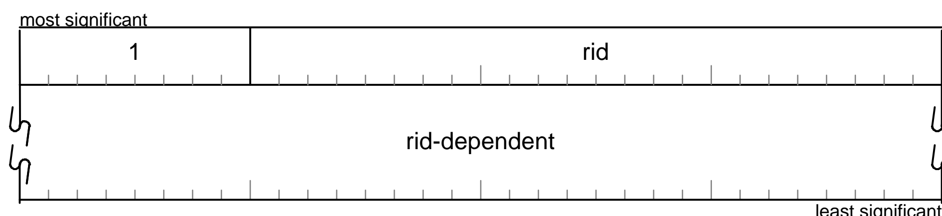

Figure 14 – Minimal ROM format

The value of the most significant byte of the first quadlet (one) indicates minimal ROM format. The 24-bit *rid* field uniquely identifies the organization or vendor responsible for the definition of the rest of configuration ROM; its value shall be assigned by the IEEE RA (see 7.1).

The rest of this clause describes the data structures and facilities of general format configuration ROM. This format provides additional information in a hierarchical structure that includes a bus information block, a root directory and optional subsidiary directories or leaves. Each entry within the root directory may provide information or may contain a pointer to either another directory or a leaf. Both directories and leaves provide information; the difference is that directories share a common structure (defined by this standard) which consists of entries, while the structure of a leaf may be specified by the organization that defined the pointer entry that referenced the leaf. The hierarchical organization of general format configuration ROM is illustrated by Figure 15.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

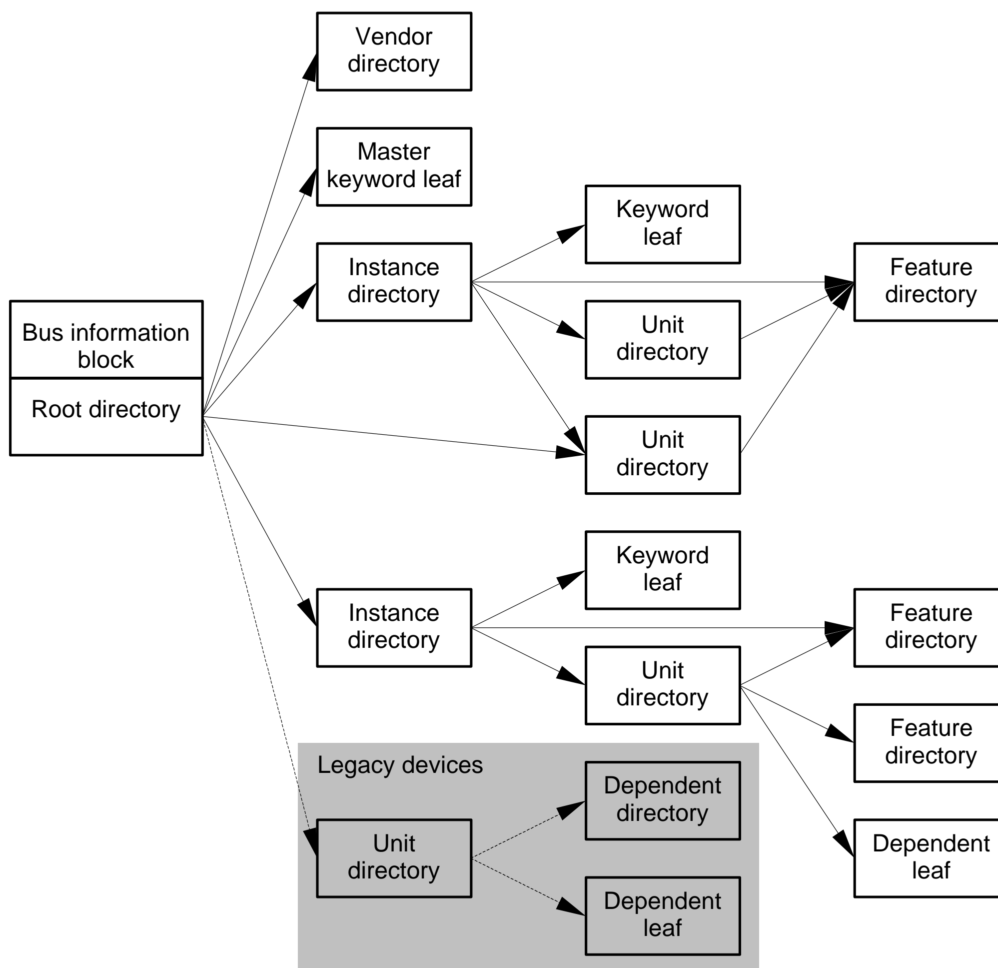

Figure 15 – Example configuration ROM hierarchy

One of the primary uses of configuration ROM is the efficient discovery of device capabilities expressed within the node and, for devices of interest, a way to select device driver(s). To this end, instance directories provide high-level information about particular instantiations of functions while unit directories identify the software interface (*i.e.*, device driver) used to access each separately controllable function. The instance directories are intended to present "thumbnails" of the devices by means of subsidiary keyword leaves—descriptions, such as "PRINTER", "VCR", "DISK" or "DISPLAY" that have cognitive resonance for the human user. The unit directories are complementary to the instance directories; there may be more than one unit directory for a particular device instance, each of which specifies a different software interface for the same device.

ISO/IEC 13213:1994 does not define instance directories; as a consequence, unit directories in legacy devices are always direct offspring of the root directory and are not accessible *via* instance directories (as shown in the shaded area). Devices manufactured since the development of this revised standard should not use this earlier style; each unit directory should be the child of an instance directory. Even when this is the case, configuration ROM compliant with this revised standard may address unit directories directly from the root (in addition to accessibility from intermediate instance directories), but this style is discouraged except as necessary to accommodate legacy device discovery software.

The first quadlet of general format configuration ROM, the bus information block and the root directory are all at predetermined offsets from $\text{FFFF F000 0400}_{16}$ while all other data structures—directories, leaves and vendor-dependent information—may occur at any address after the fixed elements. The structure of general format configuration ROM is shown by Figure 16.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

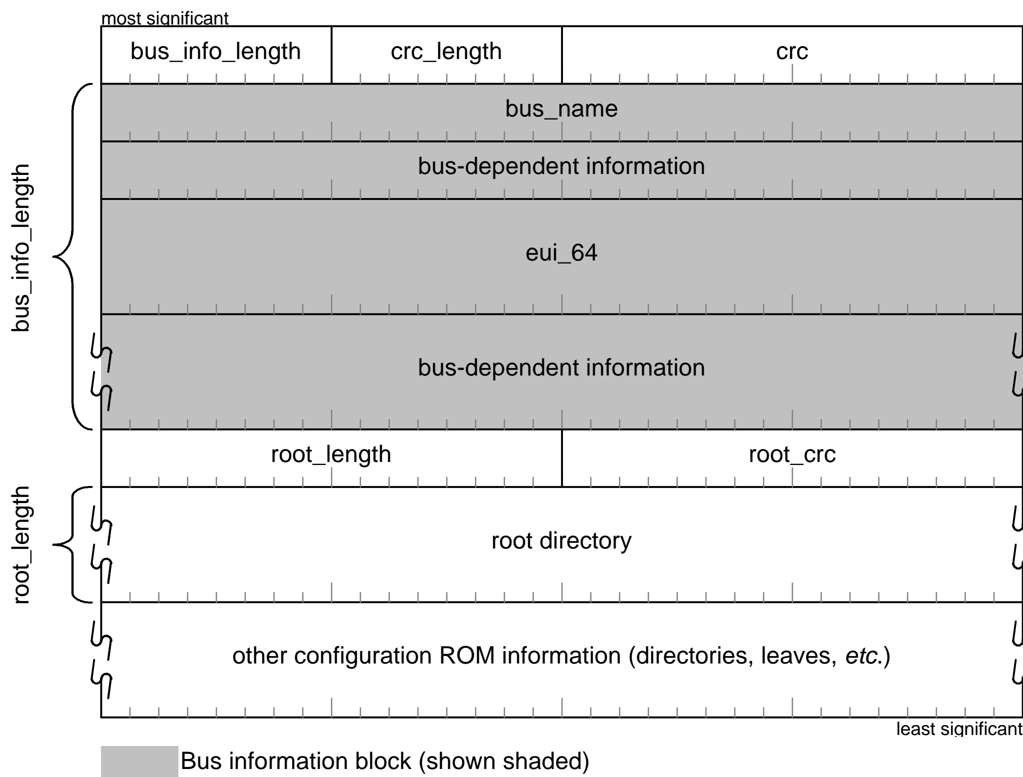

Figure 16 – General ROM format

The *bus_info_length* field shall have a minimum value of two and shall specify the size, in quadlets, of the bus information block that immediately follows the first quadlet of configuration ROM. This area identifies the applicable bus standard and is intended to contain bus-dependent information. The first quadlet of the bus information block, the *bus_name* field, is defined by this standard. With the exception of the *eui_64* field, the format and meaning of the remaining *bus_info_length* - 1 quadlets of the bus information block shall be specified by the relevant bus standard. If the bus has been standardized as an IEEE project, *bus_name* shall contain the ASCII representation of the project number assigned by the IEEE. For example, Figure 17 illustrates the *bus_name* field for IEEE Std 1394-1995.

Figure 17 – Example *bus_name* (for IEEE Std 1394-1995)

Buses compliant with this standard that utilize general format configuration ROM but that are not themselves IEEE standards shall set the value of *bus_name* to ASCII "0000".

The *eui_64* field shall contain an Extended Unique Identifier, 64 bits, whose value shall be unique among all EUI-64 values worldwide. The most significant 24 bits of the EUI-64 shall be a RID (see 7.1) and the remaining 40 least significant bits shall be administered by the owner of the RID so as to guarantee the uniqueness of the EUI-64.

NOTE – For historical reasons, the second and third quadlets of the bus information block should contain the *eui_64* field. This permits uniform access to this important unique identifier without recourse to bus-dependent format information.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Although parts of the bus information block other than the *bus_name* and *eui_64* fields shall be specified by the relevant bus standard, the inclusion of a *max_ROM* field is strongly recommended. If the bus standard defines *max_ROM*, the field shall specify the size and alignment of read requests supported by configuration ROM within the first kilobyte. Suggested encoding for the *max_ROM* field is given by the table below.

max_ROM	**Description**
0	Quadlet read requests are supported. This encoding is suggested for legacy devices and should not be reported by devices compliant with this standard.
1	Quadlet read requests and read requests aligned on 64-byte addresses with a data length of 64 bytes are supported.
2	Quadlet read requests and read requests aligned on quadlet addresses with a data length greater than four but less than or equal to 1024 bytes are supported.

NOTE – Devices that report a *max_ROM* value of zero should support read requests capable of returning the entire contents of the bus information block in one transaction even if similar read requests are not supported for any other portion of configuration ROM.

The *crc_length* field shall specify the number of following quadlets that are included in the calculation of the *crc* value (the number of covered bytes is four times the value of *crc_length*). The value of *crc_length* shall be greater than or equal to the value of *bus_info_length*.

An application shall not rely upon the accurate value of the EUI-64 unless all quadlets included in the calculation of *crc* are read and the CRC is verified. In order to minimize the number of quadlets read, it is strongly recommended that *crc_length* be equal to *bus_info_length*. Other data structures in configuration ROM, both directories and leaves, have their own calculated CRC; thus it is not necessary to include them in the quadlets covered by the *crc* value.

The value of the *crc* field shall be calculated by the method described in 7.3.

The root directory occurs at the offset FFFF F000 0400_{16} + 4 (*bus_info_length* + 1), immediately following the bus information block. The format of the root directory is the same as all other directories; the contents of the root directory are described in more detail in 7.6.1.

The contiguous block of configuration ROM that starts immediately after the end of the root directory is available for directories, leaves and vendor-dependent information. These data structures may appear in any order; they form a hierarchical tree accessed by means of relative address pointers as described in 7.7.18. If more than one kilobyte is required for configuration ROM entries, additional directories and leaves may be located at unreserved addresses within units space.

7.3 CRC calculation

All data structures defined by this standard provide a CRC that may be used to protect against data errors when configuration ROM is read. The CRC calculation is based on the ITU-T CRC-16 code (ITU-T Recommendation V.41 [bibliography reference here]). CRC generation is an iterative process that operates on two 16-bit values: the *crc* value that resulted from the preceding iteration and the next doublet to be included in the calculation. The first calculation commences with an initial *crc* value of zero. Calculation proceeds from the most to the least significant bits in each step and from the most to the least significant doublets over the range of quadlets to be covered by the CRC.

The algorithm specified by Table 4 shall be used to calculate a new *crc* doublet (C00–C15) based on the values of the previous *crc* doublet (c00–c15) and the data doublet (d00–d15) to be covered by the CRC. The numbering

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

convention used for the bits labels the most significant bit zero and the least significant bit 15. The *crc* value for a quadlet shall be calculated by applying the algorithm to both doublets, commencing with the most significant doublet. The resultant *crc* value after the last covered doublet has been included in the calculation is the CRC for the covered data.

Table 4 – CRC-16 calculation algorithm

C00 =	e04 ⊕ e05 ⊕ e08 ⊕ e12;
C01 =	e05 ⊕ e06 ⊕ e09 ⊕ e13;
C02 =	e06 ⊕ e07 ⊕ e10 ⊕ e14;
C03 =	e00 ⊕ e07 ⊕ e08 ⊕ e11 ⊕ e15;
C04 =	e00 ⊕ e01 ⊕ e04 ⊕ e05 ⊕ e09;
C05 =	e01 ⊕ e02 ⊕ e05 ⊕ e06 ⊕ e10;
C06 =	e00 ⊕ e02 ⊕ e03 ⊕ e06 ⊕ e07 ⊕ e11;
C07 =	e00 ⊕ e01 ⊕ e03 ⊕ e04 ⊕ e07 ⊕ e08 ⊕ e12;
C08 =	e00 ⊕ e01 ⊕ e02 ⊕ e04 ⊕ e05 ⊕ e08 ⊕ e09 ⊕ e13;
C09 =	e01 ⊕ e02 ⊕ e03 ⊕ e05 ⊕ e06 ⊕ e09 ⊕ e10 ⊕ e14;
C10 =	e02 ⊕ e03 ⊕ e04 ⊕ e06 ⊕ e07 ⊕ e10 ⊕ e11 ⊕ e15;
C11 =	e00 ⊕ e03 ⊕ e07 ⊕ e11;
C12 =	e00 ⊕ e01 ⊕ e04 ⊕ e08 ⊕ e12;
C13 =	e01 ⊕ e02 ⊕ e05 ⊕ e09 ⊕ e13;
C14 =	e02 ⊕ e03 ⊕ e06 ⊕ e10 ⊕ e14;
C15 =	e03 ⊕ e04 ⊕ e07 ⊕ e11 ⊕ e15;

where:

C00–C15	are the most through least significant bits of the new *crc* value
e00–e15	are the most through least significant bits of an intermediate value: e15 = c15 ⊕ d15; e14 = c14 ⊕ d14; … e00 = c00 ⊕ d00;
d00–d15	are the most through least significant bits of the data value
c00–c15	are the most through least significant bits of the prior *crc* value (or zero if there is no prior value)
⊕	Exclusive OR (in the C programming language, this operator is represented by ^)

The same CRC value may be efficiently calculated in a nibble-sequential fashion, as exemplified by the C language function in Table 5. The *calculateCRC()* function is initially called with a *crc* value of zero; *data* is the first data quadlet to be included in the calculation. The function returns a new CRC value on each call for use in the next iteration.

Table 5 – CRC-16 function (informative)

```
DOUBLET calculateCRC(QUADLET crc, QUADLET data) {
   int shift;
   QUADLET sum;

   for (shift = 28; shift >= 0; shift -= 4) {
      sum = ((crc >> 12) ^ (data >> shift)) & 0x0000000F;
      crc = (crc << 4) ^ (sum << 12) ^ (sum << 5) ^ sum;
   }
   return((DOUBLET) crc);
}
```

7.4 Minimal ASCII

Except where explicitly specified by this or other standards, textual information in configuration ROM shall be limited to the characters of the minimal ASCII subset defined by Table 6. This subset is derived from ISO/IEC 646:1991 and is compliant with that standard except for the omission of the following characters:

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

- Clause 4.1.1, Transmission control characters;
- Clause 4.1.2, The format effectors BS, HT and VT (backspace, horizontal and vertical tabulation);
- Clause 4.1.3, Code extension control characters;
- Clause 4.1.4, Device control characters;
- Clause 4.1.5, Information separators;
- Clause 4.1.6, Other control characters;
- Clause 4.3.2, Alternative graphics character allocations; and
- Clause 6.4, IRV graphics character allocations.

Each of the 7-bit ASCII codes is represented below as both a hexadecimal number and its equivalent graphic representation or, in the case of control actions, mnemonic acronym. Consult ISO/IEC 646:1991 for the definition of the control actions and characters. All codes not defined are reserved.

Table 6 – Minimal ASCII subset

							07 BEL
		0A LF		0C FF	0D CR		
20 SP	21 !	22 "			25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z					5F _
	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z					

7.5 Data structures

The hierarchical organization of configuration ROM is based upon elemental data structures: directories, leaves and descriptors. This clause specifies the format of these building blocks; their required usage is described in 7.5.4.3.

7.5.1 Directory format

All directories (including the root directory) shall have the format specified by Figure 18.

Copyright © 1999 IEEE. All rights reserved

This is an unapproved IEEE Standards Draft, subject to change.

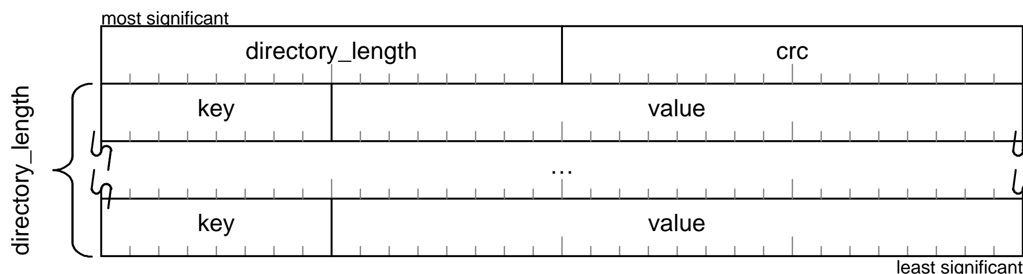

Figure 18 – Directory format

The *directory_length* field shall specify the number of following quadlets in the directory; the total size of the directory, in bytes, is 4 * (*directory_length* + 1).

The *crc* field shall be calculated as specified in 7.3 and shall include all *directory_length* following quadlets in its calculation.

The remainder of the directory consists of quadlet entries, each of which has a *key* field in the most significant byte and 24 bits of data *value* in the remaining bytes. All quadlets in a directory shall conform to this format. The meaning of directory entries may be position and context-dependent, as specified by this or other standards.

The *key* field in each directory entry is further subdivided into *type* and *key_ID* fields, as shown by Figure 19.

Figure 19 – Directory entry format

The *type* field shall specify the type of directory entry and the interpretation of *value*, as defined in Table 7.

Table 7 – Directory entry types

type	**Name**	**Usage of *value***
0	Immediate	Immediate value
1	CSR offset	Unsigned offset of an address within units space (relative to FFFF F000 0000_{16})
2	Leaf	Unsigned offset of a leaf within configuration ROM (relative to the current directory entry)
3	Directory	Unsigned offset of a directory within configuration ROM (relative to the current directory entry)

For CSR offset entries, the address within units space shall be calculated as FFFF F000 0000_{16} + 4 * *value*. For directory entries that specify a relative offset within configuration ROM, the directory or leaf address within units space shall be calculated by adding 4 * *value* to the address of the entry itself.

The *key_ID* field shall identify the meaning of the *value* field, whose usage is specified by *type*. For example, the *key_ID* named "Vendor" when combined with an immediate *type* specifies a 24-bit unique ID but when combined with a directory *type* specifies the address of a vendor directory. Both *key_ID* and *type*, collectively called the *key* field, are necessary to decode the directory entry.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Ranges of values for *key_ID* are allocated to different organizations or vendors for definition, as specified by Table 8.

Table 8 – Key ID allocations

<table>
<tr><th rowspan="2">key_ID</th><th colspan="3">Location of directory entry</th></tr>
<tr><th>CSR architecture directory</th><th>Bus-dependent directory</th><th>Dependent directory</th></tr>
<tr><td>0 – $2F_{16}$</td><td colspan="3">Allocated for definition by this standard or future CSR architecture standards.</td></tr>
<tr><td>30_{16} – 37_{16}</td><td>Allocated for definition by the bus standard identified in the bus information block.</td><td rowspan="2">Allocated for definition by the bus standard identified in the bus information block.</td><td rowspan="2">Allocated for definition by the organization or vendor identified by the directory specifier.</td></tr>
<tr><td>38_{16} – $3F_{16}$</td><td>Allocated for definition by the organization or vendor identified by the directory's Specifier_ID entry.</td></tr>
</table>

The interpretation of *key_ID* depends upon both its value and the type of directory that contains the entry. There are three different directory types within the CSR architecture:

- CSR architecture directories are those defined by this standard or future CSR architecture standards. These include the root, feature, instance, descriptor, module, unit and vendor directories;
- Bus-dependent directories are those defined by the bus standard identified in the bus information block. These include any directory addressed by an entry whose *key_ID* value is either two, Bus_Dependent_Info, or in the range 30_{16} to 37_{16} inclusive. A bus-dependent directory may also be addressed by an entry whose *key_ID* value is in the range 38_{16} to $3F_{16}$ inclusive, so long as the entry itself is in a bus-dependent directory; and
- Dependent directories are those defined by the organization or vendor identified by the directory's Specifier_ID entry. These include any directory addressed by an entry whose *key_ID* value is either 14_{16}, Dependent_Info, $1E_{16}$, Extended_Data, or in the range 38_{16} to $3F_{16}$ inclusive. A dependent directory may also be addressed by an entry whose *key_ID* value is in the range 30_{16} to 37_{16} inclusive, so long as the entry itself is in a dependent directory.

The only exceptions to the preceding exist in the case of a directory addressed by either a Vendor_Info or Module_Info entry. These directory types are defined by the CSR architecture for specification by a vendor without the necessity for an explicitly declared directory specifier. Within a vendor or module directory, the meaning of all *key_ID* values in the range 38_{16} to $3F_{16}$ inclusive shall be specified by one of (in decreasing precedence):

- an organization or vendor identified by a Specifier_ID entry in the vendor or module directory;
- a vendor identified by a Vendor_ID entry in the vendor or module directory; or
- the vendor identified by the Vendor_ID entry in the root directory.

Unless a vendor or module directory contains both a Specifier_ID and a Version entry, *key_ID* values in the range 30_{16} to 37_{16} inclusive shall not be used.

In all directories except vendor and module directories, the range of *key_ID* values allocated to directory specifiers, 38_{16} to $3F_{16}$ inclusive (or in the case of a dependent directory, 30_{16} to $3F_{16}$ inclusive) shall not be used unless both a Specifier_ID and a Version entry are present in the directory.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

The format of a leaf depends upon the *key_ID* value of the entry that addresses the leaf and the type of directory that contains the entry. Table 9 identifies the leaf format specifier for all possible cases.

Table 9 – Leaf format specifiers

<table>
<tr><th rowspan="2">key_ID</th><th colspan="3">Location of leaf entry</th></tr>
<tr><th>CSR architecture directory</th><th>Bus-dependent directory</th><th>Dependent directory</th></tr>
<tr><td>$0 – 2F_{16}$</td><td colspan="3">Leaf format and meaning defined by this standard or future CSR architecture standards, with notable exceptions: leaves subsidiary to Bus_Dependent_Info, Vendor_Info, Dependent_Info or Extended_Data leaf entries.</td></tr>
<tr><td>$30_{16} – 37_{16}$</td><td>Leaf format and meaning defined by the bus standard identified in the bus information block.</td><td rowspan="2">Leaf format and meaning defined by the bus standard identified in the bus information block.</td><td rowspan="2">Leaf format and meaning defined by the organization or vendor identified by the directory specifier.</td></tr>
<tr><td>$38_{16} – 3F_{16}$</td><td>Leaf format and meaning defined by the organization or vendor identified by the directory's Specifier_ID entry.</td></tr>
</table>

Within the range of *key_ID* values zero to $2F_{16}$ inclusive, the CSR architecture defines entries that reference leaves whose format and meaning is defined by organizations or vendors other than this standard. The leaf specifier does not depend upon the context of the directory entry that addresses the leaf, but is determined by *key* according to the table below.

key	Name	Leaf format specifier
82_{16}	Bus_Dependent_Info	Bus standard identified in the bus information block.
83_{16}	Vendor_Info	Vendor identified by a Vendor_ID entry in the same directory as the Vendor_Info entry or, in the absence of such an entry, of the Vendor_ID entry in the root directory.
94_{16}	Dependent_Info	Organization or vendor identified by the Specifier_ID entry in the same directory as the Vendor_Info entry.

A Dependent_Info leaf entry shall not be present in any directory that does not also include both a Specifier_ID and a Version entry.

See 7.7 for the *key_ID* values defined by this standard, the scope of directories in which they may appear and how they are used in combination with *type*.

7.5.2 Extended key format

The number of *key_ID* values allocated to a directory specifier, 38_{16} to $3F_{16}$ inclusive, is limited and may not provide adequate scope for the definition of directory entries. In addition, organizations or vendors other than the directory specifier may wish to define entries without recourse to a dependent directory. This standard defines new structures for directory entries to include extended keys, 24-bit key identifiers allocated by the extended key specifier.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Unlike most other configuration ROM entries[3], extended key entries shall be parsed in ascending order of their relative offsets within the directory. This is necessary to establish the context within which both extended key IDs and their associated values are interpreted.

An extended key specifier shall be identified by an entry with the format illustrated by Figure 20.

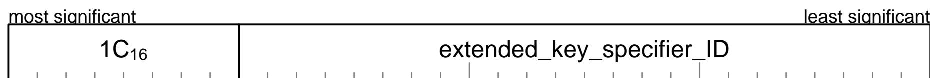

Figure 20 – Extended_Key_Specifier_ID entry format

When this entry is present in a directory, the value of the *extended_key_specifier_ID* field identifies the organization or vendor responsible for the definition of extended key entries that follow. The scope of the *extended_key_specifier_ID* applies to all subsequent extended key entries until another Extended_Key_Specifier_ID entry is encountered.

The extended key format uses paired directory entries illustrated by Figure 21 and Figure 22.

Figure 21 – Extended_Key entry format

The *extended_key_ID* field shall specify a 24-bit key identifier whose meaning and usage shall be defined by the extended key specifier identified by the preceding Extended_Key_Specifier_ID entry. If there is no extended key specifier, the meaning of the *extended_key_ID* field is unspecified.

Figure 22 – Extended_Data entry format

The *type* field shall specify the type of directory entry and the usage of *value*, as defined in Table 7.

The meaning of the *value* field shall be determined by both the *type* field and the *extended_key_ID* field from the prior Extended_Key entry. If there is no prior Extended_Key entry within the directory, the meaning of the *value* field is unspecified. More than one Extended_Data entry may follow an Extended_Key entry.

For the convenience of human readers of configuration ROM, Extended_Data entries should immediately follow the pertinent Extended_Key entry.

7.5.3 Leaf format

All leaves shall have the format specified by Figure 23.

[3] The Descriptor and Modifiable_Descriptor entries are also position-dependent.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

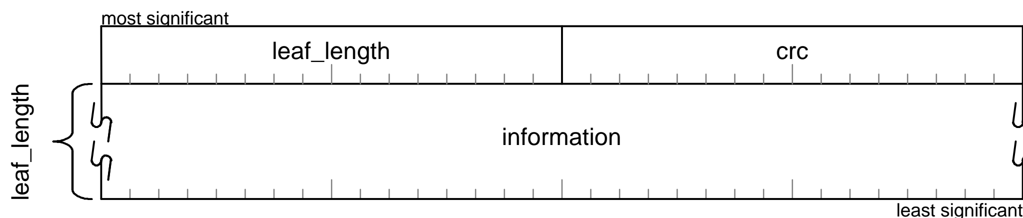

Figure 23 – Leaf format

The *leaf_length* field shall specify the number of following quadlets in the leaf; the total size of the leaf, in bytes, is 4 * (*leaf_length* + 1).

The *crc* field shall be calculated as specified in 7.3 and shall include all *leaf_length* following quadlets in its calculation.

The format and meaning of the remainder of the leaf shall be specified by the organization or vendor responsible for the definition of the directory that contained the leaf entry pointing to the leaf. There is no information within a leaf useful to determine its structure without reference to its parent directory.

7.5.4 Descriptors

A descriptor is a leaf that provides additional information to describe an object associated with a directory entry in configuration ROM. Descriptors commonly contain information in a format suitable for display to human users (such as text or icons), but organizations or vendors may define other formats suitable to particular needs. The format of a descriptor leaf is illustrated by Figure 24.

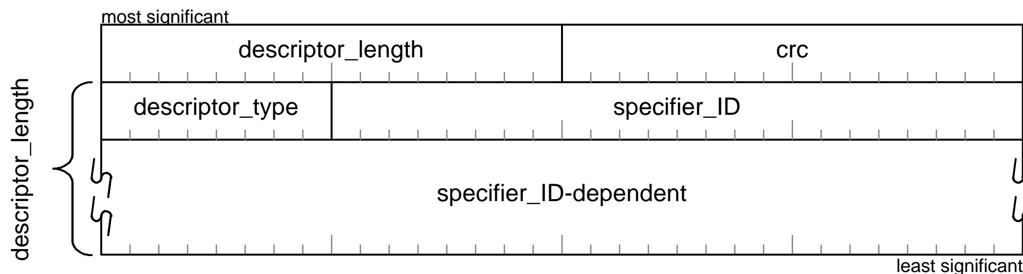

Figure 24 – Descriptor leaf format

The *descriptor_length* field shall specify the number of following quadlets in the descriptor leaf. The value of the *crc* field shall be calculated by the method described in 7.3.

The *descriptor_type* field shall describe the nature of the descriptor, according to the table below. This standard precisely defines the meaning of *descriptor_type* when *specifier_ID* is zero; for nonzero values of *specifier_ID* the *descriptor_type* field is descriptive and intended to aid rapid search of configuration for the desired descriptor. When *specifier_ID* is zero, all values not included in the table are reserved for future standardization by the CSR architecture.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

descriptor_type	Description
0	Textual descriptor
1	Icon
FF_{16}	Unspecified

The *specifier_ID* field shall have a value of zero or else a RID (see 7.1). When the value of *specifier_ID* is nonzero, the organization or vendor identified by *specifier_ID* shall define the format and meaning of all subsequent quadlets in the descriptor leaf. A zero value indicates that the descriptor leaf is defined by the CSR architecture (this standard). The remainder of this clause shall apply when *specifier_ID* is zero.

NOTE – In addition to the textual and icon descriptors standardized by the CSR architecture, many binary object formats may exist. Since binary object formats (which might include microcode, user interface controls described in an interpretive language or executable device drivers) may be highly variable, their definition is beyond the scope of this standard. An organization or vendor identified by a nonzero *specifier_ID* should specify binary object formats.

Descriptor entries, in the format specified by Figure 25, associate one or more descriptor leaves with a directory entry.

Figure 25 – Descriptor entry format

The *type* field shall specify whether the Descriptor references a directory or a leaf; its value shall be two or three.

A Descriptor entry that points to a leaf may appear in any directory. When it is present in any directory except a descriptor directory, the object in the descriptor leaf shall describe the directory entry that immediately preceded the Descriptor leaf entry. Except in a descriptor directory, a Descriptor leaf entry shall not be the first entry in the directory and shall not immediately follow a Descriptor directory or leaf entry. Figure 26 illustrates the use of a descriptor entry to associate a single descriptor with the preceding directory entry.

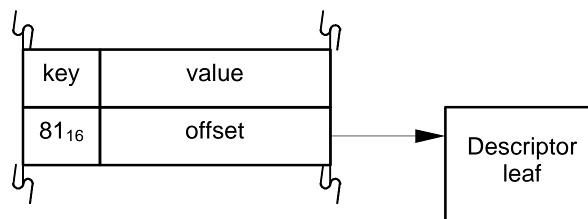

Figure 26 – Descriptor entry for a single descriptor

Descriptors with different characteristics (such as icons with varying display resolutions or textual descriptors in multiple languages may be associated with a single directory entry through the use of an intermediate descriptor directory. A Descriptor directory entry may immediately follow any directory entry except a Descriptor directory or leaf entry and shall not be the first entry in any directory. The referenced descriptor directory in turn points to one or more descriptor leaves; this hierarchy is illustrated by Figure 27.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

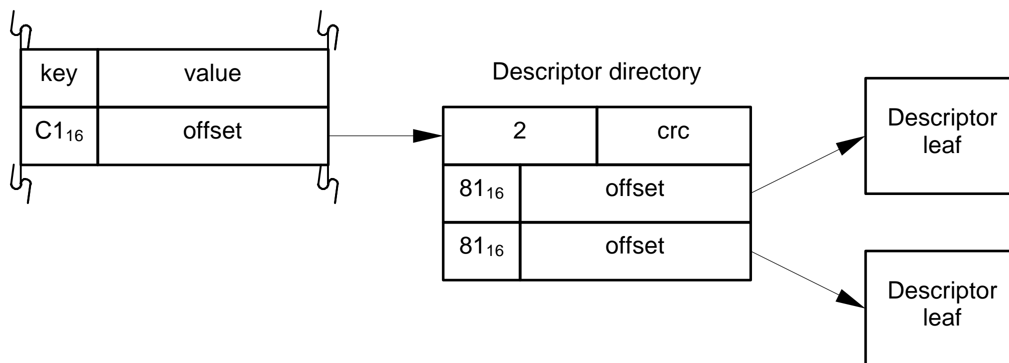

Figure 27 – Descriptor directory for multiple descriptors

Within a descriptor directory only Descriptor leaf entries are permitted. All of the descriptor leaves shall describe the directory entry that immediately preceded the Descriptor directory entry that referenced the descriptor directory.

7.5.4.1 Textual descriptors

A textual descriptor is a descriptor leaf that contains a single text string. Textual descriptors usually provide descriptive information for directory entries in a format appropriate for display to a human user. The format of a textual descriptor leaf is illustrated by Figure 28.

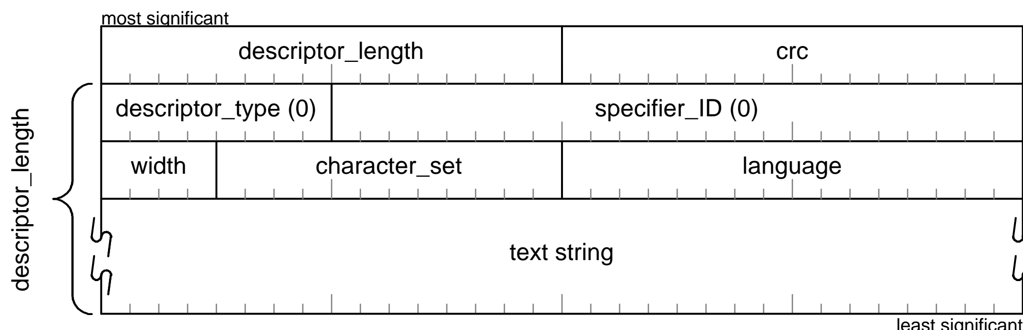

Figure 28 – Textual descriptor leaf format

The *descriptor_length* and *crc* fields shall be as specified by 7.5.4.

The *descriptor_type* and *specifier_ID* field shall have a value of zero.

The *width* field shall specify the width of the character set used in the text string. When the most significant bit of *width* is zero, the character set shall have a fixed width; otherwise the width of the character set is variable but does not exceed the maximum encoded by *width*. The definition of assigned *width* values is given by Table 10.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Table 10 – Character set widths

width	Definition
0	Fixed one-byte characters
1	Fixed two-byte characters
2	Fixed four-byte characters
3 – 7	Reserved for future standardization
8	Not to be used
9	Variable width characters up to a two byte maximum
A_{16}	Variable width characters up to a four byte maximum
B_{16} – F_{16}	Reserved for future standardization

The *character_set* field shall specify the character set used to encode the text string. A value of zero indicates that the minimal ASCII subset specified by 7.4 is used. Nonzero values specify different character sets identified by a MIBenum value registered with IANA. All character sets in use on the Internet (which include both the Unicode and ISO/IEC 10646-1:1993 character sets) are included in the IANA registry, an online version of which registry may be found on the Internet at ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

The *language* field shall be zero when *character_set* is zero; in this case the language is English. Otherwise, when *character_set* is nonzero, the value of *language* shall be derived from either ISO/IEC 639:1998 or ISO/IEC 639-2:1998. The first specifies languages as two-letter codes while the second utilizes three-letter codes. The *language* field is subdivided into three $char_n$ fields to accommodate these codes, as illustrated by Figure 29; the most significant bit of *language* is reserved and shall be zero.

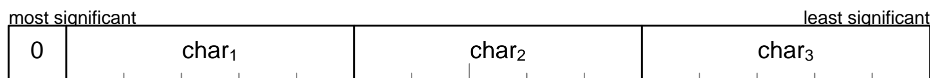

Figure 29 – Language ID format

The $char_n$ fields shall encode the characters from the two- or three-letter codes by assigning zero to the space character and one through 26, inclusive, to the letters *a* through *z*, respectively. For example, the hypothetical language code "esp" is encoded as $char_1$ equal to five, $char_2$ equal to 19 and $char_3$ equal to 16; this yields a *language* value of 5744. In the case of a two-letter code, $char_1$ shall be zero, $char_2$ shall encode the first character and $char_3$ the last. Thus "bk" results in a *language* value of 75.

The text string shall be a sequence of nonzero characters in the character set and language specified. If the length, in bytes, of the text string is not a multiple of four, the last quadlet shall be padded with zero bytes as necessary.

7.5.4.2 Icon descriptors

An icon descriptor is a descriptor leaf that contains a single icon for graphic display. Icon descriptors usually provide visually descriptive information for directory entries in a format appropriate for display to a human user. The format of an icon descriptor leaf is illustrated by Figure 30.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

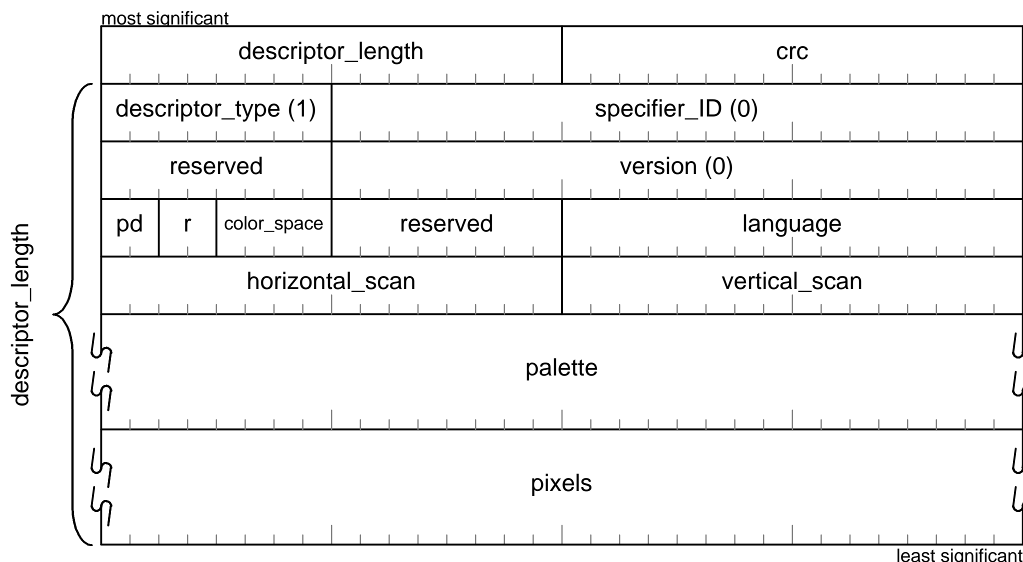

Figure 30 – Icon descriptor leaf format

The *descriptor_length* and *crc* fields shall be as specified by 7.5.4.

The *descriptor_type* field shall be one and the *specifier_ID* and *version* fields shall be zero.

The *pd* bit (abbreviated as *p* in the figure above) shall specify the number of entries (palette depth) in the optional *palette* data, as defined by the table below.

pd	Palette entries
0	None (no *palette* data)
1	4
2	16
3	256

The *color_space* field shall specify the encoding used to represent color for the icon, as specified by the table below and the text that follows.

Table 11 – Color space definitions

color_space	Name	Palette entry or pixel size (bits)	Description
0	RGB	32	24-bit RGB color with optional 8-bit alpha
1	YCbCr	16	Packed YCbCr with 2-bit alpha

The *color_space* field determines the size of a palette entry when *pd* is nonzero else the size of a pixel when no color palette is present. The details of each of the color space formats are given below; they are applicable to either the color palette or pixel encoding according to the value of *pd*.

When *color_space* is zero, a 24-bit RGB encoding is used that conforms to the format illustrated by Figure 31.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Figure 31 – 24-bit RGB format

The *alpha* field shall specify the transparency of the pixel with respect to the background. A value of zero indicates that the pixel is entirely transparent (the background is fully visible) while a value of FF_{16} indicates that the pixel is opaque (the background is not visible). Other values represent intermediate transparencies approximately as fractional opacity, *alpha* / 255.

The *red*, *green* and *blue* fields shall specify the intensity of their respective colors, where zero is off and FF_{16} represents full intensity. Other values represent intermediate intensities approximately as a fraction of 255.

When color_space is one, a packed YCbCr encoding is used that conforms to the format illustrated by Figure 32.

Figure 32 – Packed YCbCr format

The *alpha* field shall specify the transparency of the pixel with respect to the background. A value of zero indicates that the pixel is entirely transparent (the background is fully visible) while a value of three indicates that the pixel is opaque (the background is not visible). Other values represent intermediate transparencies approximately as fractional opacity, *alpha* / 3.

The *y* field shall specify the luminance of the pixel, where zero is off and $3F_{16}$ represents full intensity. Other values represent intermediate intensities approximately as a fraction of 63.

The *cb* and *cr* fields shall specify the intensity of their respective chroma components, where zero is off and F_{16} represents full intensity. Other values represent intermediate intensities approximately as a fraction of 15.

Icons may contain rasterized text, in which case it may be appropriate to specify a language for the icon. When the *language* field is zero no language information is specified. Otherwise the value of *language* shall be derived from either ISO/IEC 639:1998 or ISO/IEC 639-2:1998. The first specifies languages as two-letter codes while the second utilizes three-letter codes. The *language* field is subdivided into three $char_n$ fields to accommodate these codes, as illustrated by Figure 29; the most significant bit of *language* is reserved and shall be zero.

The *horizontal_scan* field shall specify the number of pixels in each of the icon's horizontal rasters.

The *vertical_scan* field shall specify the number of raster lines in the icon.

The *palette*, when present, shall specify the color palette referenced by the *pixels* data. The color palette is an array of entries whose format and size are determined by *color_space*, as specified by Table 11; the number of entries (palette depth) is determined by *pd*. The first entry, *palette*[0], shall be in the most significant bits of the first quadlet of the color palette; the last entry, *palette*[*n*], shall be in the least significant bits of the last quadlet of the color palette. Figure 33 illustrates how *palette* entries are stored when both *pd* and *color_space* are one.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

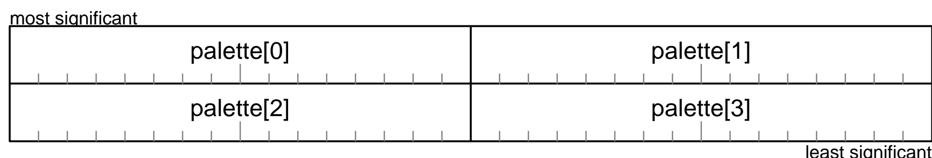

Figure 33 – 4-color palette with packed YCbCr encoding

The remainder of the icon descriptor is occupied by the *pixels* that form the image. Within an individual raster line, the pixels shall be stored left to right in the most significant to least significant positions of each quadlet. Individual raster lines shall be stored top to bottom in the most significant to least significant quadlets of the pixel data. Raster lines shall not be padded in order to occupy an integral number of quadlets. The number of bits occupied by each pixel shall be determined by *pd* and *color_space* and shall be evenly divisible into 32. When *pd* is zero, each pixel is encoded according to the format specified by *color_space*; the number of bits for each pixel is given by Table 11. Otherwise each pixel shall contain an index into the color palette; the size of each pixel is calculated as 2^{pd}.

7.5.4.3 Modifiable descriptors

Although the name configuration ROM suggests an address space whose contents are read-only, modifiable descriptors permit users to associate nicknames or icons of their own design with directory entries in a fashion analogous to Descriptor entries. The principal difference is that a level of indirection is required in order to describe the extent of the modifiable region, as shown by Figure 34.

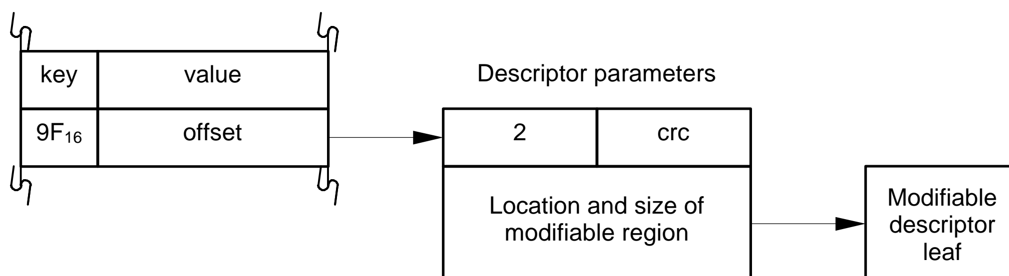

Figure 34 – Descriptor parameters and modifiable descriptor leaf

The descriptor parameters are contained within a configuration ROM leaf that is itself not modifiable; they provide the address and size of a modifiable region. The modifiable region may lie anywhere within a node's address space; when initialized with valid content, the structure of the modifiable region shall be identical to that of a textual descriptor or icon descriptor.

Modifiable descriptor entries, in the format specified by Figure 35 associate a modifiable descriptor leaf with a directory entry.

Figure 35 – Modifiable_Descriptor entry format

When present in a directory, a Modifiable_Descriptor entry shall specify the location of a descriptor parameter leaf in configuration ROM. The value of the *offset* field shall be the offset, in quadlets, of the modifiable ROM descriptor control leaf relative to the Modifiable_Descriptor entry itself. The format of a descriptor parameter leaf is illustrated by Figure 36.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

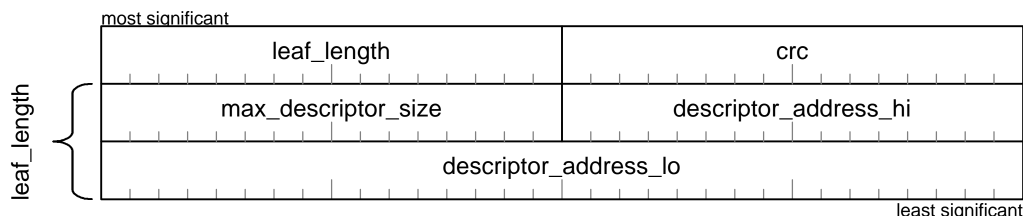

Figure 36 – Descriptor parameters format

The *leaf_length* and *crc* fields shall be as specified by 8.5.3; the *leaf_length* field shall have a value of two.

The *max_descriptor_size* field shall specify the size, in quadlets, of the modifiable region of configuration ROM described by the descriptor parameter leaf. The descriptor written into the modifiable region may be smaller than the maximum. The value of *descriptor_length* in the modifiable region shall be less than *max_descriptor_size*.

The *descriptor_address_hi* and *descriptor_address_lo* fields shall together specify the base address of the modifiable descriptor described by the modifiable descriptor leaf. The 48-bit base address, *descriptor_address*, is relative to zero, the start of node space, and shall be quadlet aligned.

The modifiable region of configuration ROM described by a descriptor parameter leaf shall not overlap any region described by other configuration ROM entries. Whenever the first quadlet of the modifiable region has a value other than zero, the format of the region shall conform to that specified by 7.5.4.1 or 7.5.4.2.

Independent agents that either read or update modifiable descriptors shall use the algorithms below in order to guarantee coherent copies of modifiable descriptors in cases where one agent modifies the descriptor while one or more agents simultaneously access it. Higher level protocols, beyond the scope of this standard, are required if more than one agent simultaneously alters a modifiable descriptor.

If the entire modifiable descriptor cannot be read by a single read transaction with a length of 4 * *max_descriptor_size* bytes, the following procedure shall be used:

a) read the first quadlet of the modifiable descriptor to retrieve the *descriptor_length* and *crc* fields;

b) read the following *descriptor_length* quadlets by some combination of read transactions;

c) read the first quadlet again. If the values of *descriptor_length* or *crc* are different, restart the algorithm from the second step; else

d) if the first quadlet is unchanged and the *crc* is valid for the covered data, the information obtained from the modifiable descriptor is usable.

If the entire modifiable descriptor cannot be updated by a single write transaction with a length less than or equal to 4 * *max_descriptor_size* bytes, the following procedure shall be used:

a) write zero to the first quadlet;

b) by some combination of quadlet or larger write transactions, update the following quadlets with the desired descriptor information; and

c) store the correct *descriptor_length* and calculated *crc* value in the first quadlet.

A Modifiable_Descriptor entry may be used anywhere a Descriptor entry that points to a leaf is permitted. The structure illustrated by Figure 34 is semantically equivalent to that of Figure 26; in both cases a single descriptor leaf is associated with a directory entry.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Just as for descriptors that are unalterable, modifiable descriptors with different characteristics (for example, nicknames in multiple languages) may be associated with a single directory entry through the use of an intermediate descriptor directory. In this case, a Descriptor directory entry immediately follows any directory entry except a Descriptor or Modifiable_Descriptor entry and shall not be the first entry in any directory. The referenced descriptor directory in turn points to one or more descriptor parameter leaves, each of which references a modifiable descriptor; this hierarchy is illustrated by Figure 37.

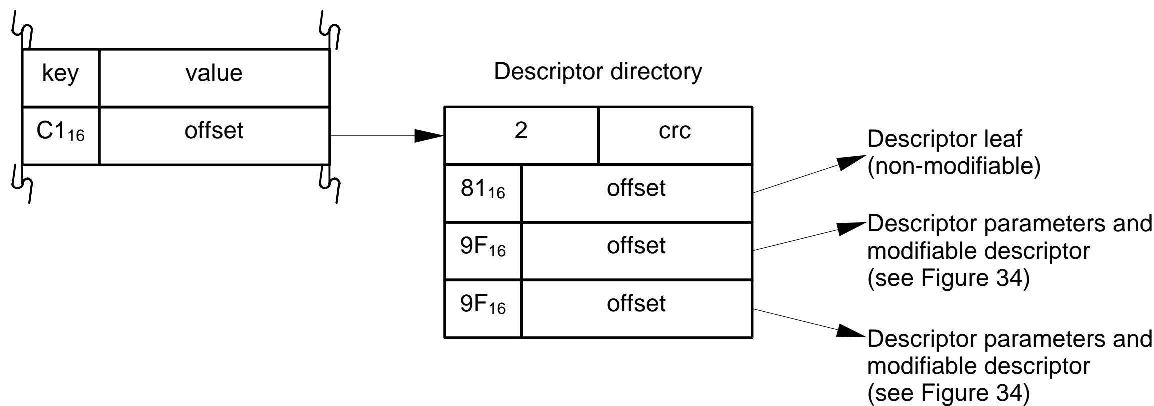

Figure 37 – Descriptor directory for multiple modifiable descriptors

Within a descriptor directory, Descriptor and Modifiable_Descriptor entries may be mixed but only leaf entries are permitted. All of the descriptors, fixed or modifiable, shall describe the directory entry that immediately preceded the Descriptor directory entry that referenced the descriptor directory.

7.6 Required and optional usage

This clause is a guide for implementers confronted with the task of creating configuration ROM for a particular device. A prerequisite for its use is an understanding of the building blocks of configuration ROM, the data structures described in 7.5. General format configuration ROM mandates the presence of some data structures (such as the root directory) and also specifies the manner in which optional features shall be used. Equipped with the information that follows, a designer may lay out configuration ROM that is compliant with this standard and useful in interoperation with other devices.

The tables that follow omit common optional directory entries, such as descriptors and extended key entries, whose usage is similar regardless of directory type. Bus- and specifier-dependent entries (those with *key_ID* values in the range 30_{16} to $3F_{16}$ inclusive) may be used in any directory in accordance with the specifications of 7.5.1.

7.6.1 Root directory

The root directory contains entries that describe features of the node that are not particular to individual software interfaces (unit architectures). The root directory may also contain pointers to other information; it is a starting point from which the configuration ROM hierarchy may be parsed. Table 12 specifies both mandatory directory entries and those whose usage is context-dependent to the root directory (the immediate, CSR offset, directory and leaf offset entry types are abbreviated as I, C, D and L, respectively).

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Table 12 – Common root directory entries

Directory entry Name	Type	Mandatory	Comments
Bus_Dependent_Info	I D L		These entries may provide additional bus-dependent information; see the applicable bus standard.
Vendor_ID	I	Y	24-bit RID of the node's vendor.
Vendor_Info	D L		A directory, leaf or both may provide vendor-dependent information.
Hardware_Version	I		Used in conjunction with the vendor ID to identify diagnostic software for the device.
Module_Primary_EUI_64	L		Used to provide module-dependent information and to distinguish a module's primary and secondary nodes.
Module_Info	D		
Node_Capabilities	I		Identifies which options of the CSR architecture are implemented.
Node_Unique_ID	L		The node unique ID leaf is obsolete. Configuration ROM implementations that contain a node unique ID leaf as specified by ISO/IEC 13213:1994 are compliant with this standard but are cautioned that the *key_ID* value for the Node_Unique_ID entry may be redefined by future standards.
Instance_Directory	D		The instance directories provide a method to group unit architectures (software protocols) to identify shared physical components.
Unit_Directory	D		Pointer to a unit directory that specifies a software interface (unit architecture) for the device instance.
Model_ID	I		Model designation assigned by vendor.
Dependent_Info	D		The dependent directory shall contain a Specifier_ID and Version entry.

7.6.2 Instance directories

A node's configuration ROM may contain zero or more instance directories, each of which describes the function(s) implemented by a particular device instantiation within a node. Table 13 specifies both mandatory directory entries and those whose usage is context-dependent to an instance directory (the immediate, CSR offset, directory and leaf offset entry types are abbreviated as I, C, D and L, respectively).

Table 13 – Common instance directory entries

Directory entry Name	Type	Mandatory	Comments
Vendor_ID	I		24-bit RID of the device's vendor.
Vendor_Info	D L		A directory, leaf or both may provide vendor-dependent information.
Keyword_Leaf	L		"Thumbnail" description of the characteristics of the device.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Directory entry		Mandatory	Comments
Name	**Type**		
Feature_Directory	D		Additional information about the features of the device.
Instance_Directory	D		Pointer to a subsidiary instance directory. The hierarchical organization of instance directories denotes functional views of the device.
Unit_Directory	D		Pointer to a unit directory that specifies a software interface (unit architecture) for the device instance. There may be more than one unit architecture for a single device instance.
Model_ID	I		Model designation assigned by vendor.
Dependent_Info	D		The dependent directory shall contain a Specifier_ID and Version entry.

An instance directory shall contain at least one Instance_Directory or Unit_Directory entry and may optionally contain one Keyword_Leaf entry and multiple Feature_Directory, Instance_Directory and Unit_Directory entries.

All instance directories should contain a Keyword_Leaf entry. When one or more Unit_Directory entries are present in the instance directory, the keyword leaf serves to describe the functional characteristics of the unit(s). When two or more Instance_Directory entries are present in the instance directory, a hierarchical relationship is established with the subsidiary instance directories: they all represent functions that are part of the same physical instance. Without a dependent keyword leaf it may be difficult to characterize device function interdependencies. A keyword leaf is useful even in the case where the instance directory contains no Unit_Directory entries.

The presence of multiple Unit_Directory entries within an instance directory represents multiple software interfaces to the same instance of a device function. The means by which the desired software interface is selected are beyond the scope of this standard.

7.6.3 Unit directories

Each unit directory uniquely identifies the software interface (unit architecture) used to control the unit. The directory may also provide additional information, either in the unit directory itself or in subsidiary directories and leaves, that characterizes the unit.

An important distinction between a unit directory and an instance directory is that the former describes the protocol software uses to access the unit while the latter identifies discrete physical instances of a function. It is common for a single physical instance of a device function to be accessible by multiple software protocols.

NOTE – The CSR architecture standardized by ISO/IEC 13213:1994 was the first to define unit architectures and their corresponding unit directories. That earlier standard did not anticipate multiple software protocols for the same physical instance of a function. The CSR architecture standardized by this document enhances configuration ROM and provides instance directories to eliminate the ambiguity, but the old and the new architectures are not entirely compatible. Legacy bus enumeration software assumes that each unit directory reached from the root represents a physical instance of a function. Bus enumeration software implemented while this standard was under development searches first for instance directories and then for unit directories. It is not always possible to design configuration ROM for a product to work well with both legacy and newer enumeration software. The CSR architecture recommends the following guidelines in order to promote maximal interoperability:

a) If the anticipated environment for the product includes *only* bus enumeration software that is aware of instance directories, no unit directories should be the direct offspring of the root directory. A device designed in this fashion will not be visible to legacy bus enumeration software;

b) If design constraints require the product to operate with legacy bus enumeration software, determine whether or not there is one (and only one) software protocol for the instance that is the search target of the legacy software. If so, link the unit

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

directory that represents this protocol to both the root directory and the instance directories but do not link any of the additional unit directories (that represent alternate software protocols for the same instance) to the root directory; or

c) If neither of the above solutions satisfy your design requirements, link unit directories to the root directory but be aware of possible side-effects: some legacy bus enumeration software may attempt to load device drivers for all unit directories even though they may represent only one physical instance of the function.

In all cases above, all of the unit directories should be accessible *via* the instance directories.

Table 14 specifies both mandatory directory entries and those whose usage is context-dependent to a unit directory (the immediate, CSR offset, directory and leaf offset entry types are abbreviated as I, C, D and L, respectively).

Table 14 – Common unit directory entries

Directory entry			
Name	**Type**	**Mandatory**	**Comments**
Vendor_ID	I		24-bit RID of the unit's vendor.
Vendor_Info	D L		A directory, leaf or both may provide vendor-dependent information.
Model_ID	I		Model designation assigned by vendor
Specifier_ID	I	Y	24-bit RID of the directory specifier.
Version	I	Y	In combination with the directory specifier ID, it identifies the software interface for the unit.
Dependent_Info	any		Additional information for the unit whose format and meaning are defined by the directory specifier.
Feature_Directory	D		Additional information that describes features of the unit. This information is usually independent of the software interface used to control the unit.

7.6.4 Feature directories

While unit directories uniquely identify the software interface for a particular unit, feature directories are intended to aggregate detailed information about device capabilities that may be independent of the protocol used to control the device. The format and meaning of a feature directory is specified by organizations such as accredited standards bodies, trade associations or vendors that have particular expertise with a class of devices. For example, an industry forum of consumer electronic companies might define a set of entries for a feature directory that characterize camcorders, television sets, DVD players and VCRs.

Table 15 specifies both mandatory directory entries and those whose usage is context-dependent to a feature directory (the immediate, CSR offset, directory and leaf offset entry types are abbreviated as I, C, D and L, respectively).

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Table 15 – Common feature directory entries

Directory entry Name	Type	Mandatory	Comments
Specifier_ID	I	Y	24-bit RID of the directory specifier.
Version	I	Y	In combination with the directory specifier ID, it identifies the document(s) that define the format and meaning of entries in the feature directory.
Dependent_Info	any		Additional information related to the feature set; the format and meaning is defined by the directory specifier.

Feature directories serve to segregate the description of device capabilities and features from the software interface (unit architecture) used to access the device. For example, multiple unit directories may all reference the same feature directory. Software that needs to locate particular feature sets does not have to understand the unit architectures in order to search for those features.

If a feature directory is referenced from an instance directory, each subsidiary unit directory referenced from the instance directory shall also point to the same feature directory. This mandates that all features described at the instance directory level shall also be accessible *via* all software interfaces, however there is no requirement that all software interfaces support identical feature sets. Individual unit directories may point to additional feature directories not referenced from other unit directories.

7.6.5 Keyword leaves

A keyword leaf is a collection of one or more ASCII keywords that pertain to the parent directory whose Keyword entry referenced the leaf. A keyword leaf may be a child of the root, in which case it is likely to be a master index of all keywords present in all the keyword leaves in the device's configuration ROM. Or a keyword leaf may be a child of an instance directory, in which case its keywords describe the functions supported by that particular component. The format of a keyword leaf is illustrated by Figure 38.

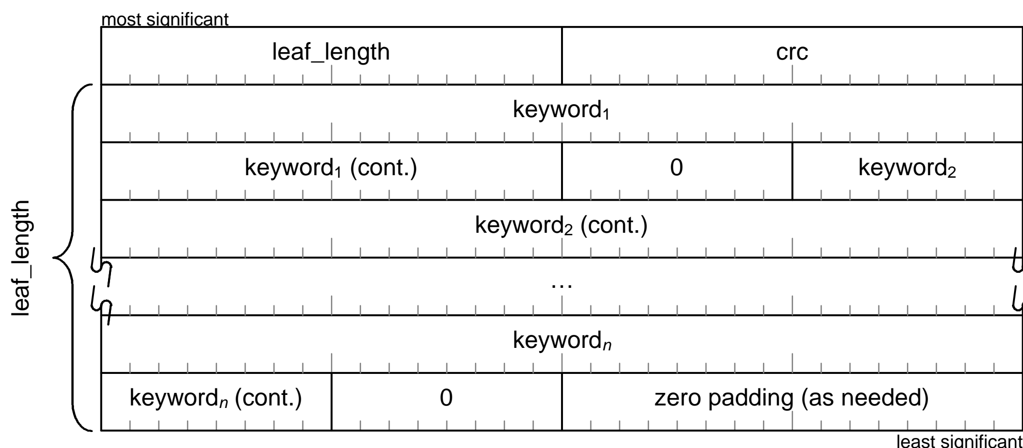

Figure 38 – Keyword leaf format

Individual keywords within a keyword leaf shall be zero-terminated ASCII strings. The character set for keywords is limited to ASCII 'A' through 'Z' (lowercase is not allowed), '0' through '9' and the hyphen '-'; neither spaces nor any other characters, printing or nonprinting, shall appear in keywords. This restricted set has been chosen to minimize

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

the potential for the creation of synonymous keywords. The space is disallowed to avoid the formation of compound keywords; implementers should form aggregations of individual keywords that describe their devices.

Keywords are intended to be self-administered; there are no rules for their use. However, in order that keywords be useful and add value to configuration ROM the following guidelines are important:

- synonymous keywords should be avoided. If a keyword *PRINTER* is already in use, neither vendors, industry groups nor standards bodies should create close variants such as *PRT*, *PRNTR* or even *LPT*. The reasons for this should be clear: if enumeration software in operating systems is written than recognizes *PRINTER* in its search for printers, it will only be harmful to use variants as they likely will not be recognized;
- examine the keywords in use by similar products and emulate them where features overlap. This is an evolutionary advantage of keyword usage: the longer they are in use and the more examples exist of their use, the greater the convergence of commonly accepted meanings for keywords;
- seek feedback before coining a new keyword. Although there are no rules and no registration authority to prevent anyone from creating a new keyword, the best results will be obtained by sharing information;
- migrate the development and usage of specialized keywords to industry groups that have the appropriate expertise. For example, consumer electronic companies that make video equipment might be the best place to develop a list of recommended keywords for features particular to those products; and
- establish a clearinghouse to maintain and update a master index of keywords in known usage for particular bus standards. This list will be invaluable to operating systems vendors and product developers alike.

7.7 Directory entries

The permissible combinations of a directory entry's *key_ID* and *type* are summarized in Table 16 and defined in more detail in the clauses that follow (the immediate, CSR offset, directory and leaf offset entry types are abbreviated as I, C, D and L, respectively).

Table 16 – Key definitions

key_ID	Name	Permitted *type* values	Permitted directories	References
1	Descriptor	D L	any	7.5.4
2	Bus_Dependent_Info	I D L	root	7.7.1
3	Vendor	I D L	any	7.7.2, 7.7.3
4	Hardware_Version	I	any	7.7.4
5 – 6	reserved for future standardization			
7	Module	D L	root	7.7.5, 7.7.6
$8 – B_{16}$	reserved for future standardization			
C_{16}	Node_Capabilities	I	root	7.7.7
D_{16}	EUI_64	L	any	7.7.8
$E_{16} – 10_{16}$	reserved for future standardization			
11_{16}	Unit	D	root or instance	7.7.9
12_{16}	Specifier_ID	I	any	7.7.10
13_{16}	Version	I	any	7.7.11
14_{16}	Dependent_Info	I C D L	any	7.7.12

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

key_ID	Name	Permitted *type* values	Permitted directories	References
15_{16}	Unit_Location	L	unit	7.7.13
16_{16}	reserved for future standardization			
17_{16}	Model	I	any	7.7.14
18_{16}	Instance	D	root or instance	7.7.15
19_{16}	Keyword	L	root or instance	7.7.16
$1A_{16}$	Feature	D	instance or unit	7.7.17
$1B_{16}$	Extended_ROM	L	root	7.7.18
$1C_{16}$	Extended_Key_Specifier_ID	I	any	7.5.2
$1D_{16}$	Extended_Key	I	any	
$1E_{16}$	Extended_Data	I C D L	any	
$1F_{16}$	Modifiable_Descriptor	L	any	7.5.4.3
20_{16}	Directory_ID	I	any	7.7.19
21_{16} – $2F_{16}$	reserved for future standardization			
30_{16} – 37_{16}	allocated for definition by bus standard [4]			7.5.1
38_{16} – $3F_{16}$	allocated for definition by directory specifier [3]			

There is one restriction to the placement of directory entries in "any" directory: the only entries permitted in a descriptor directory are Descriptor or Modifiable_Descriptor leaf entries.

7.7.1 Bus_Dependent_Info entry

Optional, restricted to the root directory.

The three kinds of Bus_Dependent_Info entry (immediate, directory offset and leaf offset) provide bus-dependent information in addition to that specified in the bus information block. The bus standard identified by *bus_name* in the bus information block shall define the meaning and use of these entries. The format of these entries is illustrated by Figure 39.

Figure 39 – Bus_Dependent_Info entry format

The *type* field shall specify one of the immediate, directory offset or leaf offset entry types.

NOTE – Although any of the information referenced by a Bus_Dependent_Info entry may be described within the bus information block, the CSR architecture recommends that bus standards specify a small bus information block. Because the CRC in the first quadlet of configuration ROM provides coverage for the whole bus information block, the block must be read in its entirety in order to verify any particular datum (such as the EUI-64). Bus-dependent information of a less critical nature should be located in directories or leaves referenced by Bus_Dependent_Info entries.

[4] In bus-dependent or dependent directories, *key_ID* values in the range 30_{16} to $3F_{16}$ inclusive are allocated for definition by the bus standard or directory specifier according to the directory context.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

When the directory offset form of the Bus_Dependent_Info entry is used, the meaning of all *key_ID* values in the range 30_{16} to $3F_{16}$, inclusive, in the bus-dependent information directory shall be specified by the bus standard unless the presence of a Specifier_ID entry indicates that *key_ID* values in the range 38_{16} to $3F_{16}$, inclusive, are specified by a different organization or vendor. When the leaf offset form of the Bus_Dependent_Info entry is used, the format of the leaf shall be specified by the bus standard.

7.7.2 Vendor_ID entry

Required in the root directory; optional in other directories.

Figure 40 – Vendor_ID entry format

When the Vendor_ID entry is in the root directory, the value of the *vendor_ID* field shall identify the vendor that manufactured the device. In other directories, the value of *vendor_ID* is useful only in conjunction with the Model_ID entry. In either case, the value shall be a RID (see 7.1).

7.7.3 Vendor_Info entry

Optional.

The Vendor_Info entry, illustrated by Figure 41, provides a pointer to additional vendor-dependent information in either a directory or leaf.

Figure 41 – Vendor_Info entry format

The *type* field shall specify either a directory offset or leaf offset entry type.

The Vendor_Info entry shall be interpreted in combination with the *vendor_ID* value obtained from a Vendor_ID entry in the same directory or, in the absence of such an entry, from the Vendor_ID entry in the root directory. When the directory offset form of the Vendor_Info entry is used, the meaning of all *key_ID* values in the range 30_{16} to $3F_{16}$, inclusive, in the vendor-dependent directory shall be specified by the identified vendor unless the presence of a Specifier_ID entry in the vendor-dependent directory indicates that these *key_ID* values are specified by a different organization or vendor. When the leaf offset form of the Vendor_Info entry is used, the format of the leaf shall be specified by the identifies vendor.

7.7.4 Hardware_Version entry

Optional.

Figure 42 – Hardware_Version entry format

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

The value of the *hardware_version* field may identify diagnostic software for the device or other entity identified by the directory which contains the Hardware_Version entry. If used for this purpose, the 48-bit number formed by appending the value of *hardware_version* to the value of *vendor_ID* obtained from the Vendor_ID entry in the same directory (or, in the absence of such an entry, that in the root directory) shall uniquely identify diagnostic software.

7.7.5 Module_Primary_EUI_64

Optional, restricted to the root directory.

Figure 43 – Module_Primary_EUI_64 entry format

The Module_Primary_EUI_64 entry, if present, is a leaf offset entry that shall reference a leaf that contains the EUI-64 of the module's primary node. The presence of a Module_Primary_EUI_64 entry in a node's root directory identifies the node as one of the module's secondary nodes. The format of the primary node unique ID leaf is illustrated by Figure 44.

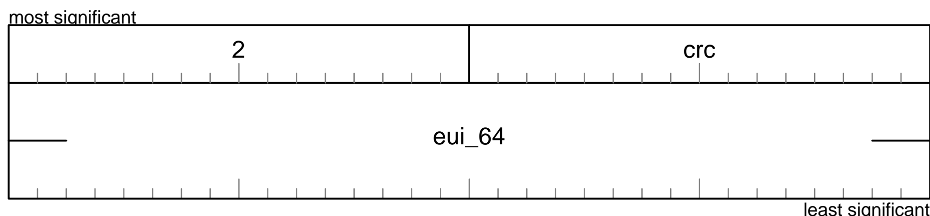

Figure 44 – Primary node unique ID leaf format

The *crc* field shall be calculated as specified in 7.3 and shall include both quadlets of *eui_64* in its calculation.

The *eui_64* field shall have a value identical to the EUI-64 found in the primary node's bus information block.

7.7.6 Module_Info

Optional, restricted to the root directory.

The Module_Info entry, illustrated by Figure 45, provides a pointer to additional module-dependent information in a directory. There shall be no more than one Module_Info entry in the root directory.

Figure 45 – Module_Info entry format

The meaning of all *key_ID* values in the range 30_{16} to $3F_{16}$ inclusive in the module directory shall be specified by one of (in decreasing precedence):

- an organization or vendor identified by a Specifier_ID entry in the module directory;
- a vendor identified by a Vendor_ID entry in the module directory; or

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

– the vendor identified by the Vendor_ID entry in the root directory.

The presence of a Module_Info directory entry identifies a singular node within a module that is in some sense primary. For all nodes the comprise a module, a Module_Info entry shall be present in one and only one node's configuration ROM.

7.7.7 Node_Capabilities entry

Optional, restricted to the root directory.

The 24-bit immediate Node_Capabilities value indicates bus-dependent features. The format of this entry is shown by Figure 46.

Figure 46 – Node_Capabilities entry format

7.7.8 EUI_64 entry

Optional.

Figure 47 – EUI_64 entry format

The EUI_64 entry is a leaf offset entry that, when present, shall reference a leaf that contains an Extended Unique Identifier, 64 bits. The format of the EUI-64 leaf is illustrated by Figure 48.

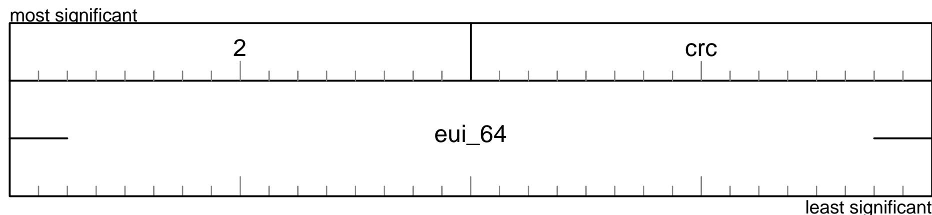

Figure 48 – EUI-64 leaf format

The *crc* field shall be calculated as specified in 7.3 and shall include both quadlets of *eui_64* in its calculation.

The value of the *eui_64* field shall be unique among all EUI-64 values worldwide. The most significant 24 bits of the EUI-64 shall be a RID (see 7.1) and the remaining 40 least significant bits shall be administered by the owner of the RID so as to guarantee the uniqueness of the EUI-64.

Within the root directory, the EUI_64 entry (named the Node_Unique_ID entry by ISO/IEC 13213:1994) is obsoleted and should not be used by devices compliant with this standard. The recommended inclusion of an EUI-64 in the bus information block is equivalent. If an EUI_64 entry is present in the root directory, the value of the *eui_64* fields in the bus information block and any EUI-64 leaf referenced from the root directory shall be identical.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

7.7.9 Unit_Directory entry

Optional, permitted in the root directory or in instance directories.

This entry, whose format is shown by Figure 49, provides a pointer to a directory that is used to identify the software interface (unit architecture) for a device function. The unit directory may provide additional information that characterizes the unit.

Figure 49 – Unit_Directory entry format

Although Unit_Directory entries are optional, the unit directory is the recommended means by which software (device drives) associations are made with independently controllable devices. There is an expectation that at least one unit directory exists in any device compliant with this standard and, consequently, that at least one Unit_Directory entry exists to reference the directory.

7.7.10 Specifier_ID entry

Required in a unit or feature directory, optional in other directories. When present, a Specifier_ID entry should be the first entry in the directory.

Figure 50 – Specifier_ID entry format

When this entry is present in a directory, the value of the *specifier_ID* field identifies the organization or vendor responsible for the definition of directory entries whose *key_ID* value is in the range 30_{16} to $3F_{16}$, inclusive. The value of *specifier_ID* shall be a RID (see 7.1). Except in the case of dependent directories, if a directory contains no Specifier_ID entry, the organization or vendor identified by the Vendor_ID entry in the root directory shall be responsible for the definition of directory entries with *key_ID* values in the range 38_{16} to $3F_{16}$, inclusive. The organization or vendor identified by either the Specifier_ID or Vendor_ID entry is referred to as the directory specifier.

Within a unit directory, the directory specifier shall be responsible for the definition of a software interface (unit architecture) for the unit. The software interface shall be specified by the 48-bit number formed by appending the Version value to the directory specifier ID (see also 7.7.11).

7.7.11 Version entry

Required in a unit or feature directory, optional in other directories. When present, a Version entry should immediately follow the Specifier_ID entry.

Figure 51 – Version entry format

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

When in a unit directory, the combination of *version* and the 24-bit RID of the directory specifier uniquely identify the software interface (unit architecture) for the unit. The 48-bit number formed by appending the value of *version* to the value of the directory specifier ID shall reference document(s) that specify the software interface used to control the unit.

In other directories, the combined value of the directory specifier ID and *version* shall reference document(s) that define the meaning and usage of directory entries with *key_ID* values in the range 38_{16} to $3F_{16}$, inclusive. The same 48-bit number shall also reference document(s) that define the format and meaning of any leaves dependent from the directory.

7.7.12 Dependent_Info entry

Optional.

This entry is used to provide additional information; the context of this entry determines the nature of the information. For example, within a unit directory a Dependent_Info entry provides supplementary information about the unit. The directory specifier shall define the meaning of the Dependent_Info entries found within the directory.

Figure 52 – Dependent_Info entry format

The *type* field shall specify any one of the entry types—immediate, CSR offset, directory offset or leaf offset.

When the directory offset form of the Dependent_Info entry is used, the meaning of all *key_ID* values in the range 38_{16} to $3F_{16}$, inclusive, in the dependent directory shall be specified by the directory specifier of its parent directory unless there is a Specifier_ID entry in the dependent directory. When the leaf offset form of the Dependent_Info entry is used, the format of the leaf shall be specified by the directory specifier of its parent directory.

7.7.13 Unit_Location entry

Optional, restricted to unit directories.

Figure 53 – Unit_Location entry format

The Unit_Location entry is a leaf offset entry that, when present in a unit directory, shall reference a leaf that describes a contiguous region of the node's address space utilized by the unit architecture. There may be more than one Unit_Location entry within a unit directory. The format of the Unit_Location leaf is illustrated by Figure 54.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

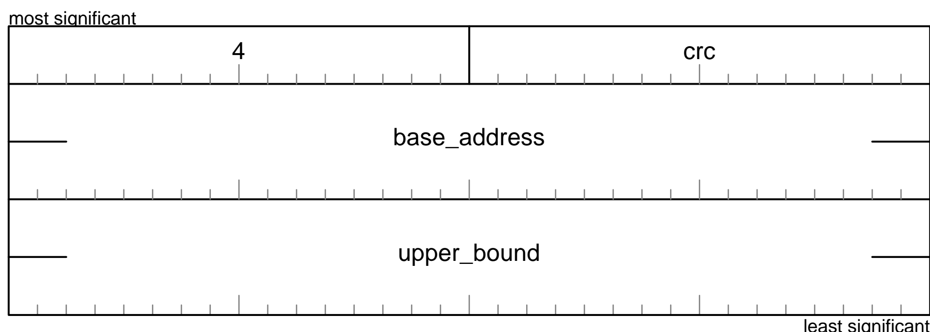

Figure 54 – Unit_Location leaf format

The *crc* field shall be calculated as specified in 7.3 and shall include both *base_address* and *upper_bound* in its calculation.

The value of the *base_address* field shall be specify the quadlet-aligned base address of a contiguous region within node space utilized by the unit. The value of the *upper_bound* field shall be quadlet-aligned and one greater than the highest byte address of the contiguous region. Although the Unit_Location leaf allocates 64 bits for both of these fields, they represent addresses within the node's 256 terabyte address space; as a consequence, the most significant 16 bits of both *base_address* and *upper_bound* shall be zero.

7.7.14 Model_ID entry

Optional.

Figure 55 – Model_ID entry format

The meaning of the *model_ID* field shall be specified by the organization or vendor identified by the Vendor_ID entry in the same directory as the Model_ID entry or, absent such an entry, by the Vendor_ID entry in the root directory. The *model_ID* value should represent a family or class of products and should not be unique to individual devices.

NOTE – The CSR architecture requires that the Specifier_ID and Version values in a unit directory identify a device driver adequate to control the unit. As a pragmatic matter, particular device models may support protocol or command set extensions not supported by the default device driver. A Model_ID entry (or its associated textual descriptor) in a unit directory might identify an alternate device driver.

7.7.15 Instance directory entry

Optional, permitted in the root or in instance directories.

This entry describes instances of particular devices implemented by the node.

Figure 56 – Instance_Directory entry format

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

The Instance_Directory entry references a directory that represents one instance of a device function. Each instance directory in turn references one or more unit directories that specify the software interfaces available to control the device. Instance directories may themselves contain additional Instance_Directory entries; this is appropriate in cases where a device may be represented in several ways to the user. For example, a multifunction peripheral may be controllable by an integrated device driver but might additionally be controllable as separate devices such as a printer, scanner or FAX. The hierarchical structure of configuration ROM guides the order in which device drivers are loaded (which determines the preferred software interface presentation format); if a device driver is available for a particular instance, it should be loaded in preference to device drivers for any of the subsidiary instances should not be presented to the user unless no driver is available for the parent instance.

7.7.16 Keyword_Leaf entry

Optional.

Figure 57 – Keyword_Leaf entry format

The Keyword_Leaf entry, when present in a directory, shall specify the location of a keyword leaf in configuration ROM. The 24-bit immediate value of the Keyword_Leaf entry shall contain the offset of the keyword leaf relative to the location of the Keyword_Leaf entry itself.

7.7.17 Feature_Directory entry

Optional, permitted in instance and unit directories.

This entry provides detailed information about the features supported by a device instance or software interface.

Figure 58 – Feature_Directory entry format

The Feature_Directory entry references a directory whose meaning is specified by other organizations such as accredited standards bodies, trade associations or vendors. The Specifier_ID and Version entries within the feature directory identify the organization responsible for the definition and maintenance of a particular feature directory.

If a Feature_Directory entry is present in an instance directory, each subsidiary unit directory referenced from the instance directory shall also point to the same feature directory. This mandates that all features described at the instance directory level shall also be accessible *via* all software interfaces, however there is no requirement that all software interfaces support identical feature sets. Individual unit directories may contain Feature_Directory entries that point to additional feature directories not referenced from other unit directories.

7.7.18 Extended_ROM entry

Optional, root directory only.

This entry may be used to describe a contiguous region of configuration ROM that exists outside of the address range FFFF F000 0400_{16} through FFFF F000 $07FF_{16}$ inclusive. Figure 59 specifies the format of the Extended_ROM entry.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

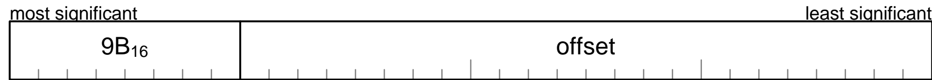

Figure 59 – Extended_ROM entry format

An Extended_ROM entry, when present in the root directory, shall specify the location of a leaf that contains configuration ROM data. If *leaf_length* is obtained from the first quadlet of the leaf, the configuration ROM information may be read by one or more read transactions. If present, the *max_ROM* field in the bus information block shall characterize the size of read requests supported within the leaf.

Configuration ROM data structures within the leaf shall be addressed from other directory entries by relative offsets in the same fashion as data structures within the first kilobyte of configuration ROM.

The leaf described by an Extended_ROM entry shall neither overlap another leaf region described by an Extended_ROM entry nor fall within or overlap any portion of the address range FFFF F000 0400_{16} through FFFF F000 $07FF_{16}$, inclusive.

7.7.19 Directory_ID entry

Optional.

This entry may be used to uniquely identify a directory whose location within configuration ROM is changeable. Figure 60 specifies the format of the Directory_ID entry.

Figure 60 – Directory_ID entry format

There shall be no more than one Directory_ID entry within a directory. When a directory does not contain a Directory_ID entry, the directory's location within configuration ROM shall be unchangeable. In this case, the implicit directory ID shall be the least significant 24 bits of the base address of the directory within node space. The value of *directory_ID* in a Directory_ID entry shall be unique among all of a node's directory IDs, both explicit and implicit, and shall be unchangeable.

Directories that do not contain a Directory_ID entry shall not reuse the same implicit directory ID in cases where they are never simultaneously present within configuration ROM.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Annex A
(informative)

Configuration ROM examples

Configuration ROM is located at base address FFFF F000 0400_{16} within a node's address space. The requirements for general format configuration ROM for devices compliant with this standard are specified in section 7.

This annex contains examples of typical configuration ROM for a variety of devices. All mandatory entries are illustrated; although optional entries may be included in the examples, they are more often omitted for the sake of simplicity.

A.1 Bus information block and root directory

Figure A-1 below shows a typical bus information block, root directory and textual descriptor leaves for devices compliant with this standard. Not shown are the instance, feature and unit directories themselves; these may vary according to the complexity of the device and its supported software interfaces. Consult other clauses in this annex for examples of particular combinations of instance and unit directories.

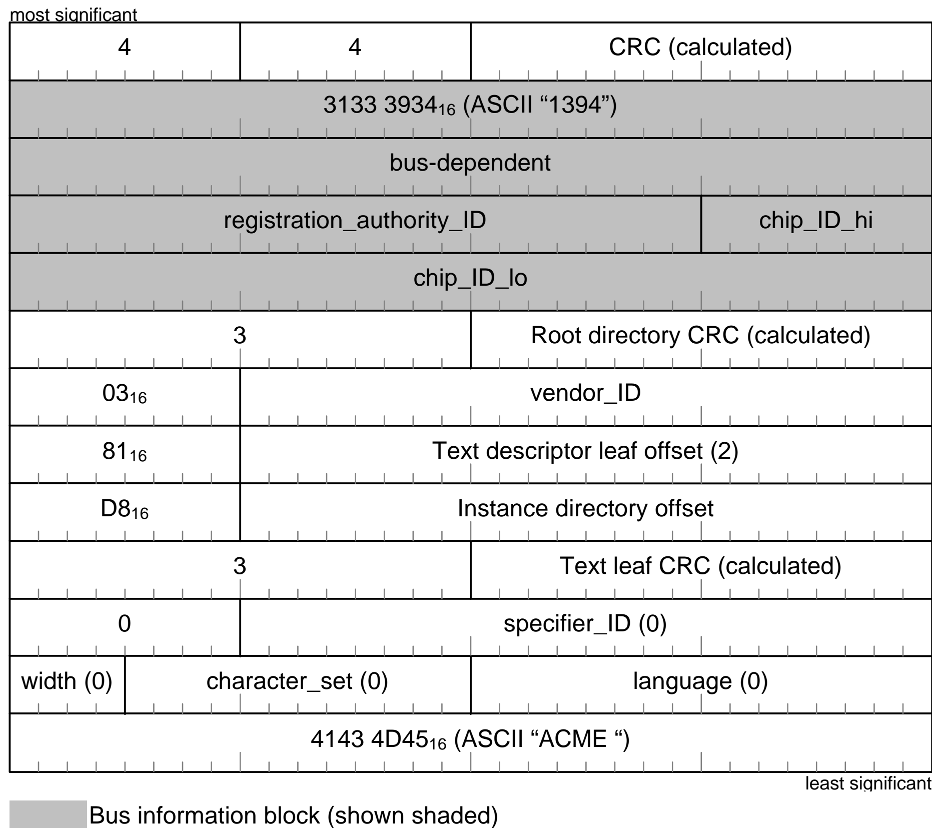

Figure A-1 – Example bus information block and root directory

Since all other configuration ROM structures (the root and its subsidiary directories and leaves) each contain their own CRC, the CRC in the first quadlet should be calculated solely on the bus information block. This minimizes the

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

number of quadlets that must be read in order to verify CRC coverage of the EUI-64. This is an important consideration for bus standards (such as IEEE Std 1394-1995) for which the node address is dynamically determined and, as a result, the relationship between node address and EUI-64 may have to be reconfirmed. Note that the use of "1394" in the bus information block is illustrative and not intended to restrict the applicability of the example.

The EUI-64 in the last two quadlets of the bus information block[5] is shown subdivided into its component fields, *registration_authority_ID*, *chip_ID_hi* and *chip_ID_lo*, to emphasize that the owner of the 24-bit registration authority ID (RID) is responsible to administer the remaining 40 bits of chip ID so as to guarantee the world-wide uniqueness of the EUI-64. The owner of the *registration_authority_ID* is not necessarily the same vendor or organization identified by the Vendor_ID entry in the root directory.

The Vendor_ID entry in the root directory, with a *key* field of 03_{16}, is immediately followed by a textual descriptor leaf entry, with a *key* field of 81_{16}, whose *indirect_offset* value points to a leaf that contains an ASCII string that identifies the vendor (the ACME company). Although the textual descriptor leaf utilizes minimal ASCII, a permissible variant might include a textual descriptor directory in order to provide multiple language support.

The Instance_Directory entry in the root directory, with a *key* field of $D8_{16}$, is the starting point for device discovery (enumeration) software to search configuration ROM for particular function instances.

NOTE – Configuration ROM designers should consult the applicable bus standard to determine whether or not bus-dependent entries, *e.g.*, Bus_Dependent_Info or Node_Capabilities, are mandated in the root directory.

A.2 Single instance with a single unit architecture

The configuration ROM for a simple device that implements only one software protocol (unit architecture) utilizes the bus information block and root directory structure already described in Figure A-1. For the purpose of this example, assume that the Instance_Directory entry in Figure A-1 references the instance directory illustrated by Figure A-2.

most significant

3		Instance directory CRC (calculated)	
99_{16}	Keyword leaf offset (2)		
DA_{16}	Feature directory offset		
$D1_{16}$	Unit directory offset		
2		Keyword leaf CRC (calculated)	
54_{16} ("T")	$4F_{16}$ ("O")	41_{16} ("A")	53_{16} ("S")
54_{16} ("T")	45_{16} ("E")	52_{16} ("R")	0

least significant

Figure A-2 – Instance directory with a single unit architecture

The Keyword_Leaf entry, with a *key* value of 99_{16}, points to a keyword leaf that contains the keyword TOASTER.

The Feature_Directory entry, with a *key* value of DA_{16}, points to a feature directory (not illustrated) that defines additional characteristics of the toaster.

[5] Although the CSR architecture requires the bus information block to contain an EUI-64, its location is left to the applicable bus standard.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Since this is a simple device that supports a single software protocol (unit architecture), there is only one Unit_Directory entry, with a *key* value of $D1_{16}$, in the instance directory. This device follows recommended practice (see 7.6.3) and the unit directory is not addressed by a Unit_Directory entry in the root but may be located only *via* the instance directory.

A.3 Single instance with multiple unit architectures

A common variant on the preceding example is for a single instance of the function (a toaster, in this case) to be controllable by more than one software protocol. The bus information block and root directory structures remain as illustrated by Figure A-1, but the instance directory referenced from the root is altered. This is shown by Figure A-3.

most significant

4		Instance directory CRC (calculated)	
99_{16}	Keyword leaf offset (4)		
DA_{16}	Feature directory offset		
$D1_{16}$	Unit directory offset		
$D1_{16}$	Unit directory offset		
2		Keyword leaf CRC (calculated)	
54_{16} ("T")	$4F_{16}$ ("O")	41_{16} ("A")	53_{16} ("S")
54_{16} ("T")	45_{16} ("E")	52_{16} ("R")	0

least significant

Figure A-3 – Instance directory with two unit architectures

The Keyword_Leaf entry, with a *key* value of 99_{16}, points to a keyword leaf that contains the keyword TOASTER.

The Feature_Directory entry, with a *key* value of DA_{16}, points to a feature directory (not illustrated) that defines additional characteristics of the toaster.

Since two methods are available to control the device, the instance directory contains a Unit_Directory entry for each software protocol (unit architecture) supported by the device. The means by which one is selected in preference to the other are beyond the scope of this standard.

A.4 Multiple instances with identical unit architectures

Consider a device such as the toaster from the first example but assume it is manufactured for a different environment, for example, a restaurant kitchen, in which multiple instances of the same function are useful. The configuration ROM for such a device still utilizes the same bus information block already described in Figure A-1, but its instance directory hierarchy is more complicated than in the preceding example. Because there are multiple instances, the root directory differs from that illustrated by Figure A-1; the altered root directory and both instances it references are shown by Figure A-4.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

most significant

<table>
<tr><td>4</td><td>4</td><td colspan="2">CRC (calculated)</td></tr>
<tr><td colspan="2">3</td><td colspan="2">Root directory CRC (calculated)</td></tr>
<tr><td>03_{16}</td><td colspan="3">vendor_ID</td></tr>
<tr><td>81_{16}</td><td colspan="3">Text descriptor leaf offset (3)</td></tr>
<tr><td>$D8_{16}$</td><td colspan="3">Instance directory offset (6)</td></tr>
<tr><td>$D8_{16}$</td><td colspan="3">Instance directory offset (10)</td></tr>
<tr><td colspan="2">3</td><td colspan="2">Text leaf CRC (calculated)</td></tr>
<tr><td>0</td><td colspan="3">specifier_ID (0)</td></tr>
<tr><td>width (0)</td><td>character_set (0)</td><td colspan="2">language (0)</td></tr>
<tr><td colspan="4">4143 $4D45_{16}$ (ASCII “ACME “)</td></tr>
<tr><td colspan="2">3</td><td colspan="2">Instance directory CRC (calculated)</td></tr>
<tr><td>99_{16}</td><td colspan="3">Keyword leaf offset (7)</td></tr>
<tr><td>DA_{16}</td><td colspan="3">Feature directory offset</td></tr>
<tr><td>$D1_{16}$</td><td colspan="3">Unit directory offset</td></tr>
<tr><td colspan="2">3</td><td colspan="2">Instance directory CRC (calculated)</td></tr>
<tr><td>99_{16}</td><td colspan="3">Keyword leaf offset (3)</td></tr>
<tr><td>DA_{16}</td><td colspan="3">Feature directory offset</td></tr>
<tr><td>$D1_{16}$</td><td colspan="3">Unit directory offset</td></tr>
<tr><td colspan="2">2</td><td colspan="2">Keyword leaf CRC (calculated)</td></tr>
<tr><td>54_{16} ("T")</td><td>$4F_{16}$ ("O")</td><td>41_{16} ("A")</td><td>53_{16} ("S")</td></tr>
<tr><td>54_{16} ("T")</td><td>45_{16} ("E")</td><td>52_{16} ("R")</td><td>0</td></tr>
</table>

least significant

Figure A-4 – Two instance directories with an identical unit architecture

The Vendor_ID entry in the root directory, with a *key* field of 03_{16}, is immediately followed by a textual descriptor leaf entry, with a *key* field of 81_{16}, whose *indirect_offset* value points to a leaf that contains an ASCII string that identifies the vendor (the ACME company). Although the textual descriptor leaf utilizes minimal ASCII, a permissible variant might include a textual descriptor directory in order to provide multiple language support.

Unlike the root directory illustrated by Figure A-1, this device utilizes two Instance_Directory entries in the root directory, one for each separate instance of the TOASTER function.

In each of the instance directories, a Keyword_Leaf entry, with a *key* value of 99_{16}, points to the same keyword leaf that contains the keyword TOASTER. When instances are identical, they should share the same keyword leaf.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

Similarly, the Feature_Directory entry in each instance directory, with a *key* value of DA_{16}, should point to a shared feature directory (not illustrated) that defines additional characteristics of the toaster.

When the software protocol (unit architecture) used by multiple instances is identical, it is necessary to distinguish one instance from another. The simplest strategy is to address distinct unit directories from each instance directory. On the other hand, if a unit directory is addressed by more than one instance directory, it is essential that the instance directory (or its subsidiary directories or leaves) contain unit-dependent information to distinguish one instance from another.

A.5 Extended keys

Extended keys permit organizations or vendors other than the directory specifier to define key values and their usage within any directory. More than one organization or vendor may define extended keys within the same directory. Figure A-5 illustrates how the scope of the different extended key entries depends upon the order in which they are encountered in the directory. Entries with smaller addresses in configuration ROM logically precede entries with larger addresses. The reader should assume that the example below is contained within a directory whose other contents are not shown.

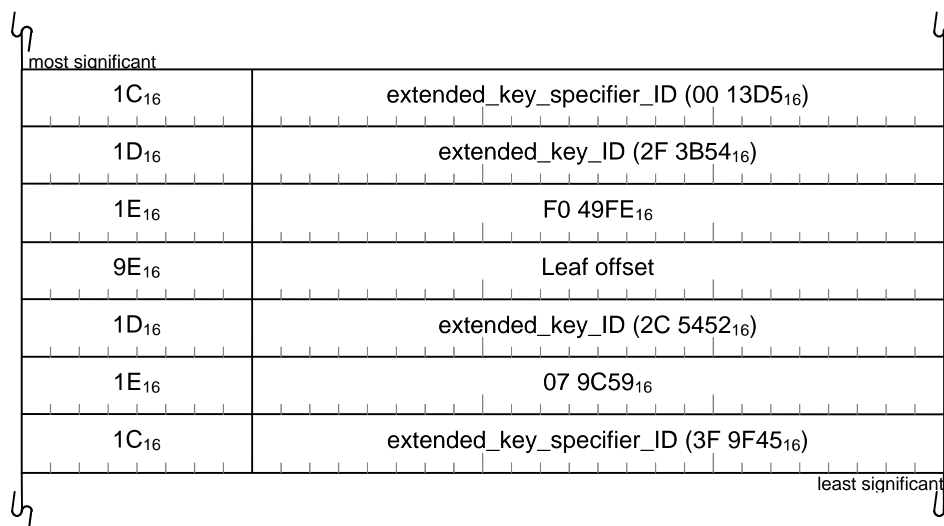

Figure A-5 – Example extended key usage

The first Extended_Key_Specifier_ID entry, with a *key* value of $1C_{16}$, identifies the organization or vendor responsible for the definition of all extended key entries that follow until a subsequent Extended_Key_Specifier_ID entry. The RID shown, $00\ 13D5_{16}$, is intended to represent the imaginary Acme company—not an actual organization or vendor.

The Extended_Key entry that immediately follows, with a *key* value of $1D_{16}$, uniquely identifies an extended key identifier value, which is analogous to the 6-bit *key_ID* values defined by this standard. In this case the 48-bit number formed by the concatenation of $00\ 13D5_{16}$ and $2F\ 3B54_{16}$ is the key identifier. The extended key identifier may be used to reference different data types—immediate values or CSR, directory and leaf offsets—according to the *type* field of the Extended_Data entries that follow.

The Extended_Data entry with a *key* value of $1E_{16}$, contains immediate data, in this example the value $F0\ 49FE_{16}$, whose meaning is specified by the organization or vendor identified by the RID value $00\ 13D5_{16}$ (the Acme company).

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.

The next Extended_Data entry, with a *key* value of $9E_{16}$, references a leaf elsewhere in configuration ROM (the leaf is omitted from the figure). The format and meaning of the leaf is the responsibility of the organization or vendor identified by the RID value 00 $13D5_{16}$; the 48-bit number formed by concatenating 2F $3B54_{16}$ to that RID uniquely identifies the leaf format.

Although not shown in the example, multiple Extended_Data entries with identical *type* values are permitted within the scope of a single Extended_Key entry.

Another Extended_Key entry follows; it defines a new 48-bit value in combination with the RID for the Acme company. This value in turn specifies how the next Extended_Data is to be interpreted, which in this case contains immediate data with a value of 07 $9C59_{16}$.

The last entry shown in the directory is an Extended_Key_Specifier_ID entry that identifies a different organization or vendor responsible for any following extended key entries (none are shown). There is no restriction on the number of extended key specifiers present in a directory.

Although other directory entries (whether defined by the CSR architecture, applicable bus standard or directory specifier) may be interspersed among extended key entries, extended key entries should be grouped as shown for the convenience of human readers of configuration ROM Software developers who write code to parse configuration ROM should not make any assumptions about the absence or presence of other entries within groups of extended key entries.

Copyright © 1999 IEEE. All rights reserved
This is an unapproved IEEE Standards Draft, subject to change.