TA Document 1999025
AV/C Descriptor Mechanism Specification Version 1.0

July 23, 2001

Sponsored by:
1394 Trade Association

Accepted for Release by:
1394 Trade Association Board of Directors.

Abstract:
This specification defines AV/C general descriptors and information blocks and their protocol, which is a standard way for AV/C devices to share information.

Keywords:
Descriptor, Info Block.

Table of Contents

List of Figures

List of Tables

This page is left intentionally blank

1. Overview

1.1 Purpose

This document specifies a command set used to create, read, write, select and delete information from descriptor and info block data structures between consumer electronic audio/video equipment. The descriptor and info block mechanism is used to share information in a standard way between AV/C devices.

This mechanism limits the media- and subunit-type specific knowledge required for performing media discovery and access. AV/C devices are not required to implement descriptor and info block data structures. Whether a subunit uses the descriptor mechanism is determined by its subunit specification.

1.2 Scope

This specification concerns itself narrowly with AV/C descriptor and info block structures and the syntax and semantics of a general set of commands to allow external controllers create, read, write, select and delete information from these structures. This document is intended to clarify and augment the descriptor and info block mechanism that was presented in the previous AV/C Digital Interface Command Set General Specification version 3.0 [R7], and Enhancement to the AV/C General Specification 3.0 [R8]. It provides a comprehensive look at descriptor and info block structures, and a full-featured command set for their manipulation.

The suite of AV/C documentation has been separated into this AV/C General Descriptor Mechanism specification and the AV/C General 4.0 specification [R9]. AV/C Information Block Type Specification [R10] defines the types and formats of general info block structures.

2. References

The following standards contain provisions, which through reference in this document constitute provisions of this standard. All the standards listed are normative references. Informative references are given in Annex A.

2.1 Reference sources

All listed references are available at various web sites. Some web sites require membership to access the references, and other sites require payment for each reference. The following sites contain the references used in this document. The reader is encouraged to always consult these sites for information on the latest versions of specifications mentioned here, as well as specifications that may be developed in the future.

[R1] 1394TA web site, http://www.1394TA.org. This web site is kept up to date with the latest released and draft versions of AV/C specifications. You need to be a member to access specifications.

[R2] International Electro-technical Commission web site, http://www.iec.ch. This web site contains specifications that must be purchased.

[R3] IEEE standards web site. http://standards.ieee.org. This web site contains specifications that must be purchased.

2.2 Specifications

At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

[R4] IEEE Std 1394–1995, *Standard for a High Performance Serial Bus*, August 30 1996.

[R5] IEEE Std 1394a-2000, *Standard for High Performance Serial Bus-Amendment 1*

[R6] ISO/IEC 13213:1994, *Control and Status Register (CSR) Architecture for Microcomputer Buses*, First Edition, October 5, 1994.

[R7] TA Document 1998003, *AV/C Digital Interface Command Set General Specification, Version 3.0*, April 15, 1998.

[R8] TA Document 2000004, *Enhancement to the AV/C General Specification Version 3.0, Version 1.1*, October 24, 2000.

[R9] TA Document 1999026, *AV/C Digital Interface Command Set General Specification Version 4.0*, Draft.

[R10] TA Document 1999045, *AV/C Information Block Types Specification Version 1.0* Draft.

3. Changes from previous version

This document contains information that was originally in the AV/C Digital Interface Command Set General Specification 3.0 [R7], and Enhancements to the AV/C General Specification 3.0 [R8]. These two documents were merged, and much of the text was rewritten and examples provided to explain the descriptor mechanism more clearly. Also, figures and tables were added to help elucidate the subject.

This section provides a cross-reference to the major changes that were made from the original document to this document. As of this writing, the editor cannot guarantee that the cross-reference will be entirely accurate and complete. A strong effort was made to reasonably reference all the changes that were made to the document, but there were many minor editorial changes that the editor felt was not necessary to cross reference. For example, changing “a” to “an” was considered unremarkable. The reader is encouraged to reference the documentation in the areas where there is interest.

Please note that there are some minor architectural changes to the technology in this document. There were also decisions made about the behavior and the meaning of some ambiguous features. These decisions may or may not conflict with systems that are already in production that used the previous documents as a baseline.

As an overview, the following updates were made:

1. Descriptor field name changes include:

Old Descriptor Field Name	New Descriptor Field Name
descriptor_length	subunit_identifier_descriptor_length list_descriptor_length entry_descriptor_length
size_of_object_position	size_of_entry_position
number_of_root_object_lists	number_of_root_lists
subunit_dependent_length	subunit_dependent_information_length
manufacturer_dependent_length	manufacturer_dependent_information_length
size_of_list_specific_information	list_specific_information_length
number_of_entries	number_of_entry_descriptors
object_entry	entry_descriptor
size_of_entry_specific_information	entry_specific_information_length

2. Other name changes:

Old Name	New Name
Object List Descriptor	List Descriptor
Object Entry Descriptor	Entry Descriptor
descriptor_identifier	descriptor_specifier

The following table shows the changes other than name changes above. For further inquiries, please refer to the document itself.

Table 3.1 – Document Changes

Category	Description
Editorial	The new AVWG Template was added to the document.
Editorial	Drawings were added in various places to clarify document content.
Editorial	The glossary of terms was revamped.
Editorial	The section on command-frame figures was copied from AV/C General 4.0.
Technical	Unit Identifier Descriptor was added.
Editorial	Information Block information was merged from the enhancements document.
Technical	Primary fields in information blocks may contain mandatory info blocks.
Editorial	A specific chapter was added explaining data structure conventions in the document.
Editorial	Root, parent, child descriptors were clarified.
Editorial	Multiple parent descriptors were described.
Editorial	All ways for identifying descriptors were put under “Descriptor identification” heading.
Editorial	Sections were added explaining how to assign and use list_IDs and object_IDs.
Editorial	Extended information fields were added to each descriptor.
Editorial	Descriptor fields were assigned a specific length and read/write property.
Technical	Descriptor attributes fields were carefully and more clearly defined. The *has_more_attributes* bit was deprecated. The *skip* and *up_to_date* bits were re-defined.
Editorial	The section of Descriptor_specifier was removed from the OPEN DESCRIPTOR command area, and combined with info_block_reference_path.
Editorial	Separate attribute tables were made for list and entry descriptors.
Editorial	A separate chapter was made to describe how to reference descriptors and info blocks in descriptor and info block commands.
Editorial	The descriptor_specifier for an entry specified only by object_ID was included.
Editorial	The descriptor_specifier for creating an entry was included.
Editorial	Information about how to use the different descriptor_specifiers was provided.
Editorial	Created a descriptor access rule summary table.
Editorial	Described legacy device behavior when devices send descriptor commands with reserved fields and values.
Editorial	Provided examples of newly-created descriptor structures created by the CREATE DESCRIPTOR command.
Technical	Deprecated List-only status from the OPEN DESCRIPTOR status command.
Editorial/ Technical	WRITE DESCRIPTOR command subfunctions were more clearly defined and either recommended or not recommended for usage.

Category	Description
Editorial/ Technical	Access rules were rewritten to include info blocks and were separated as follows: – Access rules for opening descriptors and info blocks – Access rules for writing descripors and info blocks – Access for closing descriptors and info blocks – Reset Rule – Timeout Rules
Technical	Controller Read/Write is introduced into descriptor structures and the access rules for Controller Read/Write are defined.
Technical	The WRITE DESCRIPTOR command with a descriptor specifier for a list can not write an area of descriptor entries of the list.
Technical	Clarify how to create and write an info block using WRITE DESCRIPTOR.
Editorial	Subunit Requirements were enumerated for subunits that want to support the descriptor and info block mechanism.
Editorial	All commands were given a command-response table describing fields in the response frame.
Editorial	Specific examples were given on how to use the CREATE DESCRIPTOR, READ DESCRIPTOR, and WRITE DESCRIPTOR commands.
Technical	The CREATE DESCRIPTOR command now returns the object_position of a newly created entry in the response frame. It also returns the list_ID of a newly created list in the response frame.
Technical	A study of deleting descriptors was made, and rules were determined about how to do this in a descriptor hierarchy using the WRITE DESCRIPTOR command.
Technical	Two new subfunction values in the response of WRITE DESCRIPTOR and WRITE INFO BLOCK are defined.
Editorial	Provided examples of using the WRITE DESCRIPTOR command.
Editorial	The SEARCH DESCRIPTOR command was largely untouched. Just some minor editorial changes were made.
Editorial	The OBJECT NUMBER SELECT command had only some editorial clarifications.
Editorial	Added an anatomy of AV/C descriptor in Annex A.

4. Definitions

4.1 Conformance levels

4.1.1 expected: A key word used to describe the behavior of the hardware or software in the design models *assumed* by this Specification. Other hardware and software design models may also be implemented.

4.1.2 may: A key word that indicates flexibility of choice with *no implied preference*.

4.1.3 shall: A key word indicating a mandatory requirement. Designers are *required* to implement all such mandatory requirements.

4.1.4 should: A key word indicating flexibility of choice with a strongly preferred alternative. Equivalent to the phrase *is recommended*.

4.1.5 reserved fields: A set of bits within a data structure that are defined in this specification as reserved, and are not otherwise used. Implementations of this specification shall zero these fields. Future revisions of this specification, however, may define their usage.

4.1.6 reserved values: A set of values for a field that are defined in this specification as reserved, and are not otherwise used. Implementations of this specification shall not generate these values for the field. Future revisions of this specification, however, may define their usage.

NOTE — The IEEE is investigating whether the “may, shall, should” and possibly “expected” terms will be formally defined by IEEE. If and when this occurs, draft editors should obtain their conformance definitions from the latest IEEE style document.

4.2 Glossary of terms

4.2.1 Asynchronous: asyn “any”– chronous – “time”. Asynchronous is an adjective used to describe data transfers are not sent at fixed time intervals. Asynchronous transfers are usually used for time in-sensitive data such as CONTROL commands.

4.2.2 AV/C: Audio/video control. The AV/C Digital Interface Command Set of which a part is specified by this document.

4.2.3 AV/C subunit: A part of an AV/C unit that is uniquely defined and offers a subset of functions that belong to the unit.

4.2.4 AV/C unit: An electronic device that deals with Audio and/or Video data, *e.g.*, a camcorder or a VCR, attached as a Serial Bus node.

4.2.5 Byte: Eight bits of data.

4.2.6 Controller: A device at a serial bus node that sends AV/C commands to control a remote device.

4.2.7 CSR: A Control and Status Register within a node, as defined by IEEE Std 1394–1995.

4.2.8 Data structure: A grouping of data fields in a particular and recognizable format. Examples of data structures are AV/C commands, descriptors, and info blocks.

4.2.9 AV/C Descriptor: A data structure with defined fields used for sharing information with and/or modifying information by other AV/C devices. Descriptors have a set of commands for manipulation of their contents.

4.2.10 Entry descriptor: A data structure that describes a set of information. A list descriptor generally contains a series of entry descriptors.

4.2.11 EUI-64: Extended Unique Identifier, 64-bits, as defined by the IEEE. The EUI-64 is a concatenation of the 24-bit *company_ID* obtained from the IEEE Registration Authority Committee (RAC) and a 40-bit number (typically a silicon serial number) that the vendor identified by *company_ID* guarantees to be unique for all of its products. The EUI-64 is also known as the node unique ID and is redundantly present in a node's configuration ROM in both the *Bus_Info_Block* and the *Node_Unique_ID* leaf.

4.2.12 Info block: An information block is a data structure embedded within a descriptor that describes a common type of AV/C data.

4.2.13 Isochronous: iso – "same" chronous – "time". Isochronous is an adjective used to describe data transfers that occur at regular intervals. Isochronous transfers are used for time sensitive data such as audio and video.

4.2.14 List descriptor: A list descriptor is a type of descriptor that contains a series of entry descriptors.

4.2.15 Nibble: Four bits of data. A byte is composed of two nibbles.

4.2.16 Node: An addressable device attached to Serial Bus with at least the minimum set of control registers defined by IEEE Std 1394–1995 [R4].

4.2.17 Node ID: A 16-bit number, unique within the context of an interconnected group of Serial Buses, and is defined in IEEE 1394-1995 [R4]. The node ID is used to identify both the source and destination of Serial Bus asynchronous data packets. It can identify one single device within the addressable group of Serial Buses (unicast), or it can identify all devices (broadcast).

4.2.18 Quadlet: Four bytes of data.

4.2.19 Serial Bus: The hardware interconnects and software protocols for the peer-to-peer transport of serialized data, as defined by IEEE Std 1394–1995 [R4] .

4.2.20 Target: A unit or one of its subunits at a serial bus node that receives and responds to AV/C commands from a remote controller device.

4.3 Acronyms and abbreviations

FCP: Function Control Protocol as defined by IEC 61883

IEEE: Institute of Electrical and Electronics Engineers, Inc.

lsb	least significant bit
LSB	Least Significant Byte
msb	most significant bit
MSB	Most Significant Byte
ONS	Object Number Select
SID	Subunit Identifier Descriptor

UID	Unit Identifier Descriptor

5. Data structure conventions

The following information explains the conventions used in this specification for presenting information in tables and figures.

5.1 Endian-ness

Structures and command frames are always defined with the most significant byte (MSB) of multi-byte fields at the lowest address offset or operand (by number) in the structure or command frame. The most significant bit (msb) of a field is at the highest bit position. For example:

address offset	***msb***							***lsb***
00_{16} (***MSB***)								
01_{16}								
02_{16} (***LSB***)								

Figure 5.1 – MSB/LSB and msb/lsb positions

5.2 Command frame figures

Command frame figures in this specification contain a column for each of opcode/operands, field length, ck validation of fields, and the field name or value.

Field length may have a fixed and/or variable number of bytes. A column for field length denotes the length of the field in bytes.

The length of some fields may be determined by a preceding field or a formula. In the figure below, the length is transferred from the *field_C_length* field as *i* to the length column for *field_C*.

	length	ck	msb							lsb
operand[x]	2	√	field_A							
:										
:	1	√	field_B							
:	2	–	field_C_length = *i*							
:										
:	*i*	√	field_C							
:										

Figure 5.2 – Example of a variable length field

Variable length fields are denoted by a “see^x ”if a length field does not precede the field. The lengths of these fields are determined by some other means and are described in the footnote.

The following command frame example illustrates these concepts:

	length	ck	msb							lsb
opcode	1	√	COMMAND OPCODE (XX_{16})							
operand[0]	1	√	field A							
operand[1]	2	√	field B							
operand[2]										
operand[3]	see[1]	–	field C							
:										
:	1	–	field D							

[1] The length of this field is described here.

Figure 5.3 – Example command frame

NOTE — The opcode/operand column on the far left is used to map the fields to the opcode and operands of the command frame. When a field has a variable length, this mapping can no longer be determined, and colons ":" are used for all remaining operands.

Command frames shall specify the *ck* (check) column. A check "√" in this column indicates a field that should be validated to return a response of NOT_IMPLEMENTED. A dash "–" in this column indicates a field that does not need the validation. For more information on error checking levels, see reference [R9].

It is recommended that other subunit-type specifications include the *ck* column for their commands.

5.3 Command-response tables

Command-response tables in this specification contain a column for fields in the command frame, a column for their values or description of their values, and unless otherwise indicated, a column for each response type except NOT IMPLEMENTED.

Table 5.1 – Generic command-response table example

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
field 1	FF_{16}	FF_{16}	FF_{16}	00_{16}
field 2	00_{16}	←	←	←
field 3	Explanation of the values for field 3	←	←	←

The arrow "←" for the fields in the response frame indicate that the value is identical to that of the command frame.

5.4 Descriptor field qualifiers

Descriptor and info block fields contain field qualifiers in columns two and three as shown below:

Address	Length, bytes	Controller Read/Write	Contents
00 00_{16}	2	R	field A
00 01_{16}			
00 02_{16}	1	R/W/I	field B
00 03_{16}	1	R/W	field C
00 04_{16}	1	R	field_D_length = i
00 05_{16}	i	–	field D…
:			
:			

Figure 5.4 – Descriptor fields example

These qualifiers help the reader understand more about each descriptor field, and should be used in unit and subunit-type specifications. They are defined below.

Length, bytes: Each field in a descriptor structure has a fixed or variable length, which is indicated in the *Length, bytes* column of the descriptor. The way it is denoted is identical to how it is denoted for commands shown in section 5.2.

Controller Read/Write: Each field in a descriptor can either be read/write(R/W), read/write/ignore(R/W/I) read/ignore(R/I), or read(R) by an external controller. This qualifier indicates whether the field can be written using WRITE DESCRIPTOR and WRITE INFO BLOCK commands with the *partial_replace* subfunction. No field shall be write only. For more information, see cause 9.2.2.2.

NOTE — A dash "–" in this column indicates that Controller Read/Write attribute is not defined in the figure.

5.5 General data structures

The following is an example of a general data structure that is used throughout this document, and is used to describe data that is contained within descriptor structures and command frames.

Address	Length, bytes	Contents
00 00_{16}	2	field A
00 01_{16}		
00 02_{16}	1	field B
00 03_{16}	1	field C
00 04_{16}	1	field_D_length = i
00 05_{16}	i	field D…
:		
:		

Figure 5.5 – General data structure example

5.6 Naming convention in specifications (informative)

To make the specification easier to read and understand, the following guidelines are suggested for naming fields in data structures:

1) Info block names should end with "*_info_block*" (example: *name_info_block*)
2) Fixed-definition field names can have any name, but should avoid using the term "*_info_block*" at the end of the name.
3) Length fields, which precede fixed named fields, are named "*xxxx_length*."
4) An AV/C command with *ctype*=XXXX is denoted by "a XXXX command". For example, "a CONTROL command".
5) AV/C command names are placed in UPPERCASE. For example, "an OPEN DESCRIPTOR command" or "an OPEN DESCRIPTOR control command" when *ctype* is explicitly specified.
6) An AV/C response with *response*=YYYY is denoted by "a YYYY response". For example, "an ACCEPTED response".

5.7 User-modifiable text fields (informative)

Some implementations or technologies may place restrictions on the number of characters that can be entered for a given field. Such restrictions might be due to storage or display limitations. Text fields may also have an indicator of this maximum size.

Controllers may want to be aware of these limitations when letting users work with the text entries of various subunit objects.

6. AV/C descriptor and info block mechanism

6.1 Overview

The descriptor and info block mechanism is a general way for AV/C devices to store and share information across a 1394 network. This mechanism is implemented by AV/C devices, but it is not required for all AV/C devices.

The AV/C descriptor and info block mechanism contains two types of structures for sharing data between devices: the *descriptor* and the *information block*. These two structures are defined in detail below.

6.1.1 Descriptors

A descriptor is a structured data interface presented by a device that *represents* its features and/or other descriptive information. Using this interface and the common data structures it defines, AV/C devices can exchange information. The internal storage strategy used by a particular device is device dependent and should not affect the final descriptor format. Data can be stored in any manner or location as required internally.

When an external device wants to access and/or manipulate information in a target, it uses descriptor commands to specify which descriptor it interfaces with. When a controller reads a descriptor, the target is responsible for presenting the response in the format these structures define. The following figure shows conceptually how the descriptor mechanism shares information.

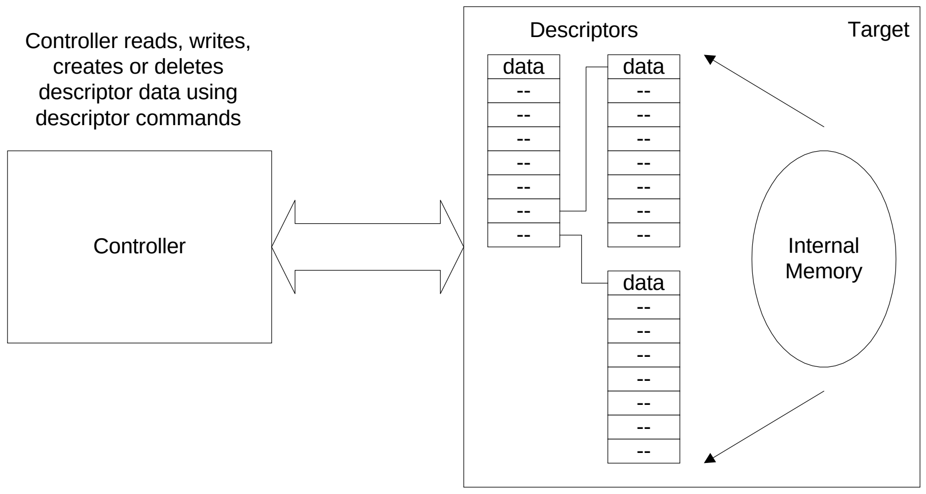

Controller reads, writes, creates or deletes descriptor data using descriptor commands
Controller
Descriptors
Target
data
--
Internal Memory

Figure 6.1 – Descriptor mechanism high-level view

One example of a descriptor is the subunit identifier descriptor, which is a data structure containing various pieces of information about a particular type of subunit. The format of the subunit identifier descriptor is common among all subunit types. However, contents of the subunit identifier descriptor are unique to each type of subunit. Most of the information in this structure does not change. However, depending on the type of subunit and its particular data requirements, it is possible that some of the information *may* change from time to time.

6.1.1.1 Descriptor types

There are two categories of descriptors: general descriptors and unit and subunit-type specific descriptors. This document defines general descriptors. Unit or subunit-type specific documents may define subunit-type specific descriptors to support functionality not available in general descriptors. There are three types of general descriptors in a unit or subunit: entry descriptors (or entries), list descriptors (or lists), and the (sub) unit identifier descriptor. A subunit can also contain its own specific descriptor type; however, this type of descriptor is defined in each subunit-type specification. The descriptor types are defined in the following table.

Table 6.1 – General and unit and subunit-type specific descriptors

Descriptor Category	Descriptor Type	Meaning
General Descriptors	unit identifier descriptor	If the descriptor mechanism is supported in a unit implementation, regardless of its subunits' implementations, this descriptor must exist. It contains information about the unit's descriptor structures, information about the unit and its manufacturer, and references to list descriptors at the root level (if any).
	Subunit identifier descriptor	This is the same as the unit identifier descriptor except that it applies to a subunit.
	list descriptor	This descriptor contains list information and entry descriptors. When list descriptors are referenced from the (sub)unit identifier descriptor, they are termed root list descriptors. When referenced from entry descriptors, they are termed child list descriptors.
	Entry descriptor	This descriptor is an addressable set of information in a list. It can also contain references to other list descriptors for presenting data in a hierarchical manner.
Subunit-type Specific descriptors	subunit-type specific descriptor	Subunit-type specific descriptors are not defined in this specification. Please refer to the subunit-type specific documentation for more information.

These descriptors can be combined in such a way as to produce a hierarchy of information called a descriptor hierarchy. This concept is shown in Figure 6.2 – AV/C descriptor on page 26.

6.1.2 Information blocks

Information blocks (or info blocks) are extensible data structures that exist inside of descriptors. Unique info-block data structures are given unique type values. Their structures can be applied to a broad range of subunit types.

Information blocks can be placed end-to-end within a descriptor's information field, or they can be nested. When they are nested, the result is a hierarchical representation of data. Note that they are NOT hierarchical in the same way that descriptors are hierarchical. For more information on the info block hierarchy see section 8.3 "Information block reference path" on page 55.

Whether they are nested or not, they can be easily parsed by a controller, even if the controller does not know the types of info blocks it encounters. In some units or subunits, the placement of information blocks within the information field is important, while in others it is not. A unit or subunit specification will specify if the information blocks it supports must be in a particular order or not. For more information on info blocks, see section 7.6 "Information blocks" on page 45.

6.2 Hierarchies using general descriptor types

The general descriptor types shown in the table above can be combined to produce a descriptor hierarchy that is most useful when representing unit or subunit information that is or can be categorized hierarchically.

6.2.1 Root list descriptor

A root list descriptor is the term for a list descriptor that is at the top of a descriptor hierarchy, and appears one level below the (sub)unit identifier descriptor. The root list is accessible by its *root_list_ID* found within the (sub)unit identifier descriptor. A (sub)unit identifier descriptor can contain more than one *root_list_ID*, and thus multiple hierarchies can exist in a unit or subunit.

Though it appears that the (sub)unit identifier descriptor is the root in the diagrams below, AV/C convention stipulates that this descriptor actually references multiple roots.

When traversing away from the root, we say that we are moving down in the descriptor hierarchy. Conversely, when moving toward the root, we are moving up in the descriptor hierarchy. There is only one root for the descriptor hierarchy.

6.2.2 Parent and child descriptors

When an entry descriptor has a child list, the entry is considered a parent of the child list, and its list is also considered the parent of the child list. A list descriptor without child lists exists at the bottom of a hierarchy and is termed a leaf list descriptor. Depending on which way it is approached, a list descriptor between root and leaf descriptors can be considered a parent or a child.

The parent – child relationship does not exist between lists and their entries or info blocks. Lists simply *contain* entries, and list or entries *contain* info blocks.

All unit and subunit types can use list descriptors, entry descriptors, and info block structures defined in this document, but would differ in the contents of their list-specific and entry-specific information. The difference is reflected in each descriptor and info block type value.

The following diagram illustrates the general relationship between the kinds of descriptors, and how they form a hierarchy:

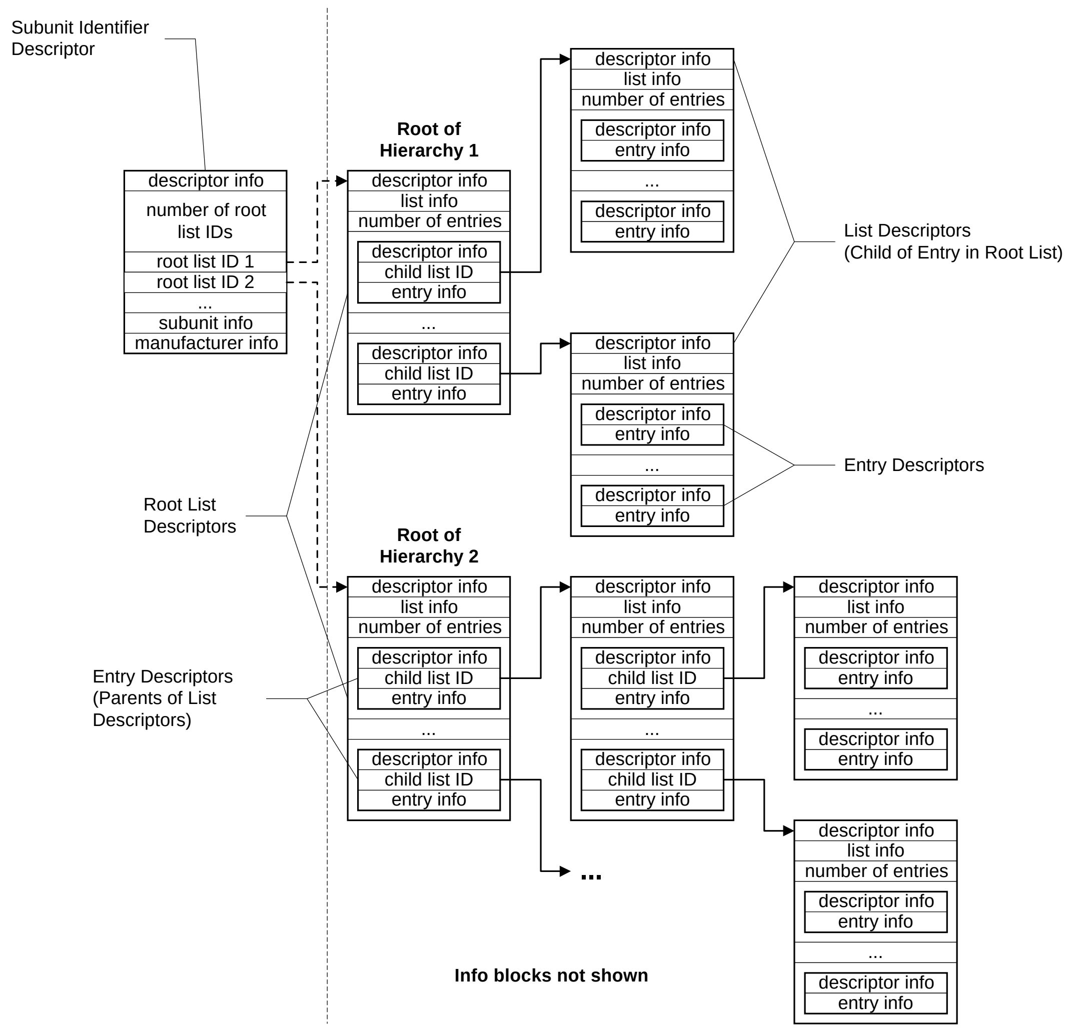


Figure 6.2 – AV/C descriptor hierarchy

Entries shall not contain *child_list_IDs* that cyclically refer to their containing list or ascendants.

6.2.3 Multiple parents

It is possible for child list descriptors to have multiple parents. This is shown by the following figure.

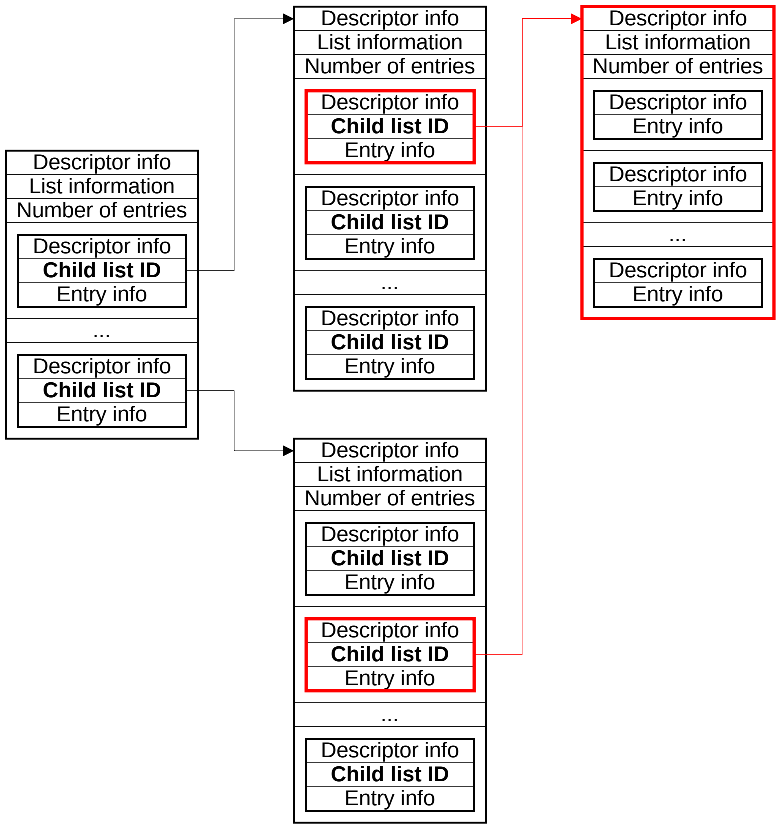


Figure 6.3 – List descriptor with multiple parents

In the figure above, the right-most list descriptor contains two parents. This is possible if the parent entry descriptors contain identical *child_list_ID* values. If a child list descriptor with multiple parents is deleted, the unit or subunit shall ensure that all parent entry descriptors shall have their *child_list_ID* field removed.

6.3 Descriptor identification

List descriptors are identified using *descriptor_specifier* structures by their type or by their ID. Entry descriptors are identified by their type, ID, or position in their list. A unit or subunit may have multiple list or entry descriptors of a particular type. However, each list descriptor must have a unique ID within the scope of a unit or within the scope of a subunit. Note that if a unit contains descriptors, it does not preclude subunits from containing descriptors with same IDs as those in the unit. The ID of an entry is optional. When entries have IDs, each entry descriptor must have a unique ID within the scope of its list.

A descriptor's type is determined by opening it and reading its *list_type* or *entry_type* field. A list descriptor's list ID is determined from its parent's *root_list_ID* or *child_list_ID* field. An entry's ID can be determined by its *object_ID* field, when it is included. An entry's position is determined by counting through the entries in a list.

6.3.1 List, entry, and info block type fields

Lists, entries, and info blocks contain a type field that identifies the structure and existence of the fields in their data structures. By reading an entry type, a controller can interpret the data in a descriptor or info block if it knows its type.

When these data structures are defined with a type value, their fields shall be defined in detail as to their existence and/or format.

These type fields have values that exist within two possible scopes: there are types that are general to all units and subunits, and types that are unit or subunit-type specific.

The following table shows the *list_type* assignment ranges and their scope:

Table 6.2 – Assignment ranges for the two list_type scopes

Range	Scope of list_type
00_{16} – $7F_{16}$	General
80_{16} – FF_{16}	Unit or subunit-type specific

Because the *list_type* definitions within the *unit or subunit-type specific* range are unique to a unit or a given type of subunit (the combination of *subunit_type* and *list_type* are unique), different subunit types and their unit may define list types with the same value in this range.

The scope for entry descriptors is defined in the same manner as list descriptors. The following table shows the *entry_type* assignment ranges and their scope:

Table 6.3 – Assignment ranges for the two entry_type scopes

Range	Scope of entry_type
00_{16} – $7F_{16}$	General
80_{16} – FF_{16}	Unit or subunit-type specific

Entry_type values that are unit or subunit-type specific may contain overlap. That is, it is possible that two different unit or subunit-types may have the same *entry_type* values for different types of descriptors. Descriptor types in the unit or subunit-type specific range and the structure of their particular information fields are defined in unit and subunit-type specifications.

The scope for info blocks are also defined in the same manner as descriptors, except that the range is much broader. There are a range for General, and a range for unit or subunit-type specific according to the following table:

Table 6.4 – Assignment ranges for the two info_block_type scopes

Range	Scope of info_block_type
$00\ 00_{16}$ – $7F\ FF_{16}$	General
$80\ 00_{16}$ – $FF\ FF_{16}$	Unit or subunit-type specific

An entry descriptor may contain multiple info blocks of different types. Likewise, a list descriptor may contain multiple entries and/or info blocks of different types.

In the unit or subunit-type specific range, info blocks shall not contain overlapping values between unit and subunit-types.

NOTE — *list_types* and *entry_types* may contain overlapping values. However, *info_block_type* shall not contain overlapping values.

When new *info_block_type* values are assigned for (sub)unit-type specific *info_block_types*, the reservation of their range is required in reference [R10]. See that specification for more information.

It is permitted to define *info_block_types* with the same meaning in other (sub)units.

See references [R10] for information about general *list_types*, *entry_types* and *info_block_types* that can exist in all units and subunits.

6.3.2 List ID values

A unique list ID, which is assigned by the unit or subunit, identifies each list descriptor within the unit or subunit. The size of this ID is specified in the (sub)unit identifier descriptor in the *size_of_list_ID* field. List IDs appear in the (sub)unit identifier descriptor as *root_list_ID*, and in entry descriptors as *child_list_ID*. The following table specifies the list ID value range assignments:

Table 6.5 – List ID value assignment ranges

Range	List Definition
0000_{16} - $0FFF_{16}$	Reserved
1000_{16} - $3FFF_{16}$	Unit or subunit-type dependent
4000_{16} - $FFFF_{16}$	Reserved
1 0000_{16} - max list ID value	Unit or subunit-type dependent

The size of the maximum list ID value is determined by the (sub)unit identifier descriptor's *size_of_list_ID* field.

The list ID of a list descriptor shall stay constant during the life of the list.

Note that there are no list ID values that fall into a general range.

6.3.2.1 Assigning list IDs (informative)

Some unit and subunit-types may create and destroy list descriptors throughout their product life cycle. In the case where a new list is created, it is unit or subunit-type dependent whether the new list shall use a new list ID value, or a previously used list ID value.

1) If an entirely new list is created, it is recommended to use a new list ID value. This ensures that controllers know that it is a new list. If a target has created more lists than are possible within the constraints given by the *size_of_list_ID* value in the (sub)unit identifier descriptor, then it may reuse any available list ID of previously deleted lists.

2) The same list ID as one in a previously deleted list may be used if the new list should be recognized by controllers to be that same list.

There may be other conditions or reasons for using a new list ID, or for reusing a previous list ID. Either way, one should consider the effects on controllers. Refer to the unit or subunit-type specification regarding how to assign specific list IDs.

6.3.3 Object_IDs

Entries may have an *object_ID*, which facilitate in searching for and accessing the entry descriptor. It is up to the unit or subunit-type specification to define *object_ID*s for a particular implementation. *Object_ID*s can be assigned by an external controller, or internally by the unit or subunit. The unit or subunit is responsible for ensuring that the setting of *object_ID* accords with the following rules:

1) *object_ID* values must be unique within their list.
2) A particular type of unit or subunit may define additional conditions on *object_ID* uniqueness. For example, a subunit may want to have unique entries within a group of lists one level below the root list. For details, please refer to the appropriate unit or subunit-type specification.
3) If a controller attempts to set an *object_ID* to a value that conflicts with an existing *object_ID* within this scope, the unit or subunit shall return a REJECTED response to the operation.
4) If an *object_ID* assignment is accepted, then the unit or subunit shall not change it.
5) All entries within a unit or subunit shall have the same number of bytes for their ID values. The number of bytes used to specify the *object_ID* is defined in the (sub)unit identifier descriptor's *size_of_object_ID* field.

When a unit or subunit assigns *object_ID*s, it can use various schemes, for example:

1) A unit or subunit can use a sequential or unused number.
2) A unit or subunit-type specification can specify a scheme for *object_ID* values that is patterned after its data and which is understandable and duplicable by a controller. That is, a controller that understands the scheme can re-construct the *object_ID* and access the entry immediately without having to do any searches on the target.
3) In some lists, the *object_ID's* value may be the same as the entry's position, but this is not guaranteed nor required by the architecture.

NOTE — If an *object_ID* is unique within the scope of a list, it is possible to specify two or more entries with the same *object_ID* when using descriptor specifier type = 21_{16}, causing an ambiguity. Depending on the unit or subunit-type specification, this may or may not be desired. If an ambiguous specification is made within the *descriptor_specifier*, the target shall return an arbitrary descriptor that matches the specifier. For more information on the *descriptor_specifier*, see section 8.1, "Descriptor specifier" on page 50.

Please refer to the unit or subunit-type specification for more information on assigning *object_ID*s.

6.3.4 Identifying entries by position

A unit or subunit-type specification may specify entries by their position in their list, and require controllers to access them in such a way. A controller may need to know about how the unit or subunit orders its entries to access them correctly.

6.4 Object and object group representations (informative)

Descriptors and info blocks describe objects and object groups. An object is an abstract term used to refer to something in a unit or subunit that can be presented by a descriptor. For example, in a device, objects could be files and object groups could be the directories that contain the files.

A list descriptor includes features that are shared by each entry it contains. For example, a disc subunit can implement a list descriptor that contains information about its files. Each entry descriptor in the list, therefore, represents a file. An info block within an entry could be used as a common interface to present properties of that file.

List descriptors and entry descriptors can be used to create hierarchical data relationships. For a hypothetical example, consider a device that contains two directories of files (See footnote [1]). The volume could be described as an object containing a group of directories. Each directory is considered an object containing a group of file objects. The descriptors can be structured to support this hierarchical order, as shown below.

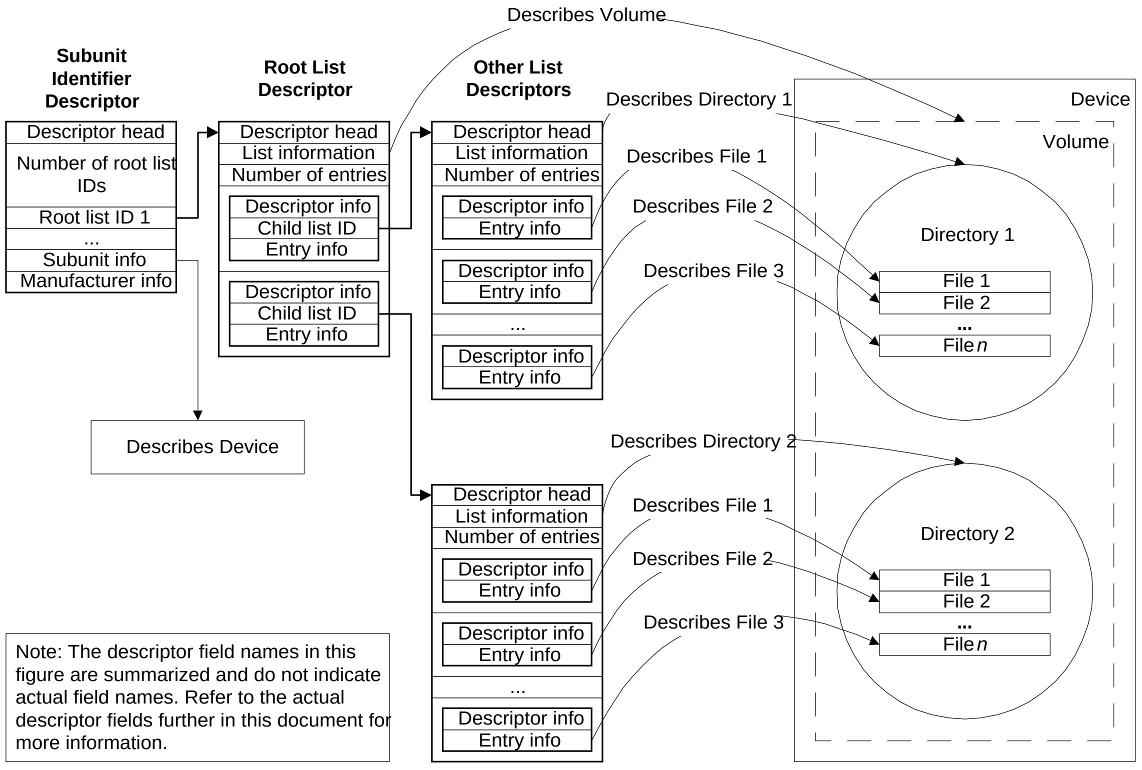


Figure 6.4 – List and entry descriptors describing an object group with objects

The hierarchical model of lists and objects can be continued to any level as needed to correspond to the way a device structures its data.

In this example, the information about directories can be located in the *list_specific_information* of a list or the *entry_specific_information* of the list's parent. A unit or subunit-type specification shall define where to locate information at this level.

[1] This hypothetical example is used to illustrate the hierarchical nature of the descriptor mechanism, and is not intended to represent the architecture of the any subunit-type in any way.

7. General descriptor and info block data structures

7.1 Unit identifier descriptor

The unit identifier descriptor is the highest-level descriptor in a unit. The unit identifier descriptor is required if the unit supports the descriptor mechanism at the unit level. The following figure shows the unit identifier descriptor:

Address	Length, bytes	Controller Read/Write	Contents
00 00$_{16}$	2	R	unit_identifier_descriptor_length
00 01$_{16}$			
00 02$_{16}$	1	R	generation_ID
00 03$_{16}$	1	R	size_of_list_ID = i
00 04$_{16}$	1	R	size_of_object_ID
00 05$_{16}$	1	R	size_of_entry_position
00 06$_{16}$	2	R	number_of_root_lists = n
00 07$_{16}$			
00 08$_{16}$	i	R	root_list_id[0]
:			
:			:
:	i	R	root_ list_id[n-1]
:			
:	2	R	unit_information_length = j
:			
:			
:	j	–[1]	unit_information…
:			
:	2	R	manufacturer_dependent_information_length = k
:			
:			
:	k	–[2]	manufacturer_dependent_information…
:			
:		–[1]	extended_information (optional)…
:			

[1] depends on the unit specification.
[2] depends on the how the vendor defines these fields.

Figure 7.1 – Unit Identifier Descriptor

7.1.1 Unit identifier descriptor fields

unit_identifier_descriptor_length: The *unit_identifier_descriptor_length* field contains the number of bytes which follow in this descriptor structure. The value of this field does *not* include the length field itself. The unit identifier descriptor has a maximum size of 64 kbytes.

generation_ID: The *generation_ID* field specifies which AV/C descriptor format is used by this unit for all data structures it maintains, and the command sets that manipulate them. This field can have one of the following values:

Table 7.1 – Generation_ID values

generation_ID	Meaning
00_{16}	Data structures and command sets as specified in the AV/C General Specification, version 3.0.
01_{16}	Data structures and command sets as specified in the AV/C General Specification, version 3.0 and the Enhancement to the AV/C General Specification 3.0, vrsion 1.0 and 1.1.
02_{16}	Data structures and command sets as defined in this specification.
all others	Reserved for future specification.

size_of_list_ID: The *size_of_list_ID* field indicates the number of bytes used to specify a list ID field in descriptors and commands. Note that list descriptors do not contain list IDs, instead they exist in their parent descriptor as *root_list_ID* or *child_list_ID*. The value of this field shall not change. If this value is 0_{16}, then list descriptors are not supported by the unit.

size_of_object_ID: The *size_of_object_ID* field indicates the number of bytes used to indicate an *object_ID* for this unit. All entries maintained within the unit that have an *object_ID* (some lists may contain entries without *object_ID* fields) shall have this number of bytes for their ID. The value of this field may change depending on the unit specification. If the unit does not use the *object_ID* field in entry descriptors, the value of this field shall be 0_{16}.

size_of_entry_position: The *size_of_entry_position* field indicates the number of bytes used when referring to an entry by its position in its list. All such references used within the unit shall have this number of bytes for the position reference. The value of this field may change depending on the unit specification. If this field is 0, then usage of *descriptor_specifier* 20_{16} in descriptor commands is not supported.

number_of_root_lists: The *number_of_root_lists* field contains the number of root list descriptors in this unit.

root_list_id[x]: The *root_list_id[x]* fields are the ID values for each of the root list descriptors in this unit.

unit_information_length: The *unit_information_length* field specifies the number of bytes in the *unit_information* field.

unit_information: The *unit_information* field contains information whose format and contents will depend on the unit this is describing. If there is no unit information in the descriptor, then the *unit_information_length* field shall be zero and the *unit_information* field shall not exist. The following structure shows how fields in the *unit_information* are organized.

Address Offset	Length, bytes	Controller Read/Write	Contents
00 00_{16}	2	R	general_unit_info_length = *l*
00 01_{16}			
00 02_{16}	*l*	– [1]	general_unit_info
00 03_{16}			
00 04_{16}			
00 05_{16}	–	– [2]	info_block_area
00 06_{16}			

[1] to be defined by a future revision of this specification.
[2] depends on other unit specifications.

Figure 7.2 – unit_information fields

general_unit_info_length: The *general_unit_info_length* field indicates the length of the *general_unit_info* field. Currently, the value of this field shall be 0000_{16}.

general_unit_info: The *general_unit_info* field contains non-info block information, and may be defined by a future revision of this specification.

info_block_area: The *info_block_area* field contains info blocks for describing unit dependent information.

manufacturer_dependent_information_length: The *manufacturer_dependent_information_length* field specifies the number of bytes in the *manufacturer_dependent_information* field.

manufacturer_dependent_information: The *manufacturer_dependent_information* field is used for vendor-specific data. The format and contents are completely up to the manufacturer. If there is no manufacturer dependent information in the descriptor, then the *manufacturer_dependent_information_length* field shall be zero and the *manufacturer_dependent_information* field shall not exist.

extended_information: The *extended_information* field is used to place information in later generation descriptors, and is ***not to be used in descriptors using this version of the specification***. As controllers access the unit identifier descriptor with a *generation_ID* higher than was used for their own descriptor mechanism, they shall be aware that this field may exist. A controller can check if data exists in this field using the following formula:

unit_identifier_descriptor_length >
10 + (*number_of_root_lists* * *size_of_list_ID*) + *unit_information_length* +
manufacturer_dependent_information_length

Under the condition above, a legacy controller can assume that there is extended data that it cannot access. In these circumstances, controllers must not assume that an error condition has occurred; rather, they should assume that the descriptor has been extended.

A *unit_identifier_descriptor_length* less than the value from the equation above is an error.

It is recommended that *extended_information* field is extended with the general info block structure defined in clause 7.6 “Information blocks” on page 45.

7.2 Subunit identifier descriptor

The subunit identifier descriptor is the highest-level descriptor in a subunit. The subunit identifier descriptor is required if the subunit supports the descriptor mechanism and, therefore, it shall not be deleted. The following figure shows the subunit identifier descriptor:

Address	Length, bytes	Controller Read/Write	Contents
00 00$_{16}$	2	R	subunit_identifier_descriptor_length
00 01$_{16}$			
00 02$_{16}$	1	R	generation_ID
00 03$_{16}$	1	R	size_of_list_ID = i
00 04$_{16}$	1	R	size_of_object_ID
00 05$_{16}$	1	R	size_of_entry_position
00 06$_{16}$	2	R	number_of_root_lists = n
00 07$_{16}$			
00 08$_{16}$	i	R	root_list_id[0]
:			
:			:
:	i	R	root_ list_id[n-1]
:			
:	2	R	subunit_type_dependent_information_length = j
:			
:			
:	j	– [1]	subunit_type_dependent_information…
:			
:	2	R	manufacturer_dependent_information_length = k
:			
:			
:	k	– [2]	manufacturer_dependent_information…
:			
:		– [1]	extended_information (optional)…
:			

[1] depends on the subunit-type specification.
[2] depends on the how the vendor defines these fields.

Figure 7.3 – Subunit Identifier Descriptor

7.2.1 Subunit identifier descriptor fields

subunit_identifier_descriptor_length: The *subunit_identifier_descriptor_length* field contains the number of bytes which follow in this descriptor structure. The value of this field does *not* include the length field itself. The subunit identifier descriptor has a maximum size of 64 kbytes.

generation_ID: The *generation_ID* field specifies which AV/C descriptor format is used by this subunit for all data structures it maintains, and the command sets that manipulate them. See Table 7.1 on page 33 for generation_ID values and meanings.

size_of_list_ID: The *size_of_list_ID* field indicates the number of bytes used to specify a list ID field in descriptors and commands. Note that list descriptors do not contain list IDs, instead they exist in their parent descriptor as *root_list_ID* or *child_list_ID*. The value of this field shall not change. If this value is 0_{16}, then list descriptors are not supported by the subunit.

size_of_object_ID: The *size_of_object_ID* field indicates the number of bytes used to indicate an *object_ID* for this subunit. All entries maintained within the subunit that have an *object_ID* (some lists may contain entries without *object_ID* fields) shall have this number of bytes for their ID. The value of this field may change depending on the subunit-type specification. If the subunit does not use the *object_ID* field in entry descriptors, the value of this field may shall be 0_{16}.

size_of_entry_position: The *size_of_entry_position* field indicates the number of bytes used when referring to an entry by its position in its list. All such references used within the subunit shall have this number of bytes for the position reference. The value of this field may change depending on the subunit-type specification. If this field is 0, then usage of *descriptor_specifier* 20_{16} in descriptor commands is not supported.

number_of_root_lists: The *number_of_root_lists* field contains the number of root list descriptors in this subunit.

root_list_id[x]: The *root_list_id[x]* fields are the ID values for each of the root list descriptors in this subunit.

subunit_dependent_information_length: The *subunit_dependent_information_length* field specifies the number of bytes in the *subunit_dependent_information* field.

subunit_dependent_information: The *subunit_dependent_information* field contains information whose format and contents will depend on the type of subunit this is describing. If there is no subunit-dependent information in the descriptor, then the *subunit_dependent_information_length* field shall be zero and the *subunit_dependent_information* field shall not exist. For more information on how to define this field, see section 7.5 “Specific information fields in descriptors” on page 43.

manufacturer_dependent_information_length: The *manufacturer_dependent_information_length* field specifies the number of bytes in the *manufacturer_dependent_information* field.

manufacturer_dependent_information: The *manufacturer_dependent_information* field is used for vendor-specific data. The format and contents are completely up to the manufacturer. If there is no manufacturer dependent information in the descriptor, then the *manufacturer_dependent_information_length* field shall be zero and the *manufacturer_dependent_information* field shall not exist.

extended_information: The *extended_information* field is used to place information in later generation descriptors, and is ***not to be used in descriptors using this version of the specification***. As controllers access the subunit identifier descriptor with a *generation_ID* higher than was used for their own descriptor mechanism, they shall be aware that this field may exist. A controller can check if data exists in this field using the following formula:

> *subunit_identifier_descriptor_length* >
> 10 + (*number_of_root_lists* * *size_of_list_ID*) + *subunit_dependent_information_length* +
> *manufacturer_dependent_information_length*

Under the condition above, a legacy controller can assume that there is extended data that it cannot access. In these circumstances, controllers must not assume that an error condition has occurred; rather, they should assume that the descriptor has been extended.

A *subunit_identifier_descriptor_length* less than the value from the equation above is an error.

It is recommended that *extended_information* field is extended with the general info block structure defined in clause 7.6 “Information blocks” on page 45.

7.3 List descriptor

A list descriptor is a container of entry descriptors. It also has fields for information related to the list. All list descriptors have the same layout, which includes standard fields at the beginning, and then a collection of entries. The list descriptor is shown below.

Address	Length, bytes	Controller Read/Write	Contents
00 00_{16}	2	R	list_descriptor_length
00 01_{16}			
00 02_{16}	1	R	list_type
:	1	R	attributes
:	2	R	list_specific_information_length = i
:			
:	i	– [2]	list_specific_information…
:			
:			
:	see[4]	R	number_of_entry_descriptors = n
:			
:	see[1]	– [5]	entry_descriptor[0]…
:			
:			
:			:
:	see[1]	–[5]	entry_descriptor[n-1]…
:			
:			
:	see[3]	– [2]	extended_information (optional)…
:			

1 the length of this field is determined by reading its first two bytes, that is, length = value in first two bytes + 2.
2 list type and subunit-type dependent.
3 for the length of this field, see the description of *extended_information* below.
4 The length of this field is equal to the value of the *size_of_entry_position* field in the (sub)unit identifier descriptor.
5 See Figure 7.5 – General entry descriptor

Figure 7.4 – General List Descriptor

7.3.1 List descriptor fields

list_descriptor_length: The *list_descriptor_length* field contains the number of bytes which follow in the list descriptor structure. This field is two bytes in length. The value of this field does *not* include the length field itself.

list_type: The *list_type* field indicates the type of the list descriptor and the formats of its fields to the extent that a controller knowing the type can also know how to read or write to the list. Lists are defined either by this specification or by their unit or subunit-type specifications and are assigned list type values. For information on the *list_type*, refer to clause 6.3.1 “List, entry, and info block type fields” on page 27.

attributes: The *attributes* contains bit flags which indicate attributes that pertain to the entire list structure.

The following table illustrates the attributes that are defined for list descriptors.

Table 7.2 – List descriptor attribute values

Attribute	Meaning
1xxx xxxx	**has_more_attributes:** If this bit is set to 1, then the next byte is also an attributes byte. All extended attribute bytes shall use this bit to indicate an extension to the next byte. In this and future versions of this specification, this bit shall be 0.
x1xx xxxx	**skip:** It is recommended for unit and subunits to not use this bit. If used, this bit indicates that the entire list's underlying data is no longer available or valid. If this bit is set to 1, the controller must ignore the list. If this bit is set to 0, the data in the list prior to its entries is valid, but entries within the list may still need to be skipped. Refer to the entrys' *skip* attribute to determine whether they should be skipped. The use of this bit is unit, subunit-type and list type dependent. If a unit, subunit-type or list type does not use this attribute, its default value shall be 0. The unit or subunit can use this bit to perform a "delayed memory clean up" operation. By setting this bit after a list deletion request, the unit or subunit can defer the actual deletion and reclamation of this memory until a convenient time. The unit or subunit may set this bit to 1 as a result of a controller deleting the list.
xx1x xxxx	Reserved
xxx1 xxxx	**entries_have_object_ID:** If this bit is set to 1, then all entries in this list have an *object_ID* field. If it is 0, then none of the entries in this list have an *object_ID* field. This attribute bit shall not change.
xxxx 1xxx	**up_to_date:** This bit indicates the *validity* of data. If this bit is set to 1, then the entire list is known by the unit or subunit to match with its underlying data at the time the list was opened. If this bit is set to 0, then some data in the list may be stale. Refer to the entry's *up_to_date* attribute to determine whether it is up to date. The use of this bit is unit or subunit-type dependent. If a unit or subunit-type does not use this attribute, its default value shall be 1. When data is stale, the unit or subunit may or may not be sure of this, and the controller should take this into consideration when dealing with the data. This attribute shall not change while the descriptor is open.
all others	Reserved for future specification.

list_specific_information_length: The *list_specific_information_length* field specifies the number of bytes used for the following *list_specific_information* field. The size of this field is two bytes, and is NOT included in the calculation.

list_specific_information: The *list_specific_information* field contains information that is specific to a particular *list_type*, and may be partially writeable dependent on that type. For more information on general guidelines about how to define this field, see section 7.5 “Specific information fields in descriptors” on page 43.

Refer to the specific *list_type* definitions in unit and subunit-type specifications for more details on this field.

number_of_entry_descriptors: The *number_of_entry_descriptors* field contains the number of entry descriptors in this list.

entry_descriptor[x]: The *entry_descriptor[x]* field is an entry descriptor. See section 7.4 “Entry descriptor” on page 41 for more information.

extended_information: The *extended_information* field is used to place added information for future extension of the general list descriptor, and is ***not to be used in descriptors using this version of the specification***. It is recommended that these fields are extended with the general info block structure defined in clause 7.6 “Information blocks” on page 45. If a controller needs to, it can check if data exists in this field using the following formula:

$$
\begin{aligned}
&\mathit{list\_descriptor\_length} > \\
&4 + \mathit{list\_specific\_information\_length} + \mathit{size\_of\_entry\_position} \\
&+ \sum_{i=1}^{number\_of\_entry\_descriptors} (entry\_descriptor\_length(i-1) + 2)
\end{aligned}
$$

A *list_descriptor_length* field less than a value from the equation above is an error.

As controllers access the list descriptor with a *generation_ID* (found in the (sub)unit identifier descriptor) higher than was used for their own descriptor mechanism, they shall be aware that this field may exist. Under this condition, a legacy controller can assume that there is extended data that it cannot access. In these circumstances, controllers must not assume that an error condition has occurred; rather, they should assume that the descriptor has been extended.

7.4 Entry descriptor

All entry descriptors share a common format. Each entry includes entry-specific information describing an object within the unit or subunit. The entry descriptor has the following structure:

Address Offset	Length, bytes	Controller Read/Write	Contents
00_{16}	2	R	entry_descriptor_length
01_{16}			
02_{16}	1	R	entry_type
03_{16}	1	R	attributes
:	see[1]	R	child_list_ID (optional)
:			
:	see[2]	– [3]	object_ID (optional)
:			
:			
:			
:	2	R	entry_specific_information_length = i
:			
:	i	– [3]	entry_specific_information…
:			
:			
:	see[4]	– [3]	extended_information (optional)…
:			

[1] the length of this field is equal to the *size_of_list_ID* in the (sub)unit identifier descriptor.
[2] the length of this field is equal to the *size_of_object_ID* in the (sub)unit identifier descriptor.
[3] entry type and subunit-type dependent.
[4] for the length of this field, see the description of *extended_information* below.

Figure 7.5 – General entry descriptor

NOTE — The *child_list_ID* and *object_ID* fields exist only if specified in the attributes field of the entry descriptor and the list descriptor that contains the entry descriptor, respectively.

7.4.1 Entry descriptor fields

entry_descriptor_length: The *entry_descriptor_length* field contains the number of bytes which follow in this descriptor structure. The value of this field does not include the length field itself.

entry_type: The *entry_type* field indicates the type of the entry descriptor and the formats of its fields to the extent that a controller knowing the type can also know how to read or write to the entry. Entries are defined either by this specification or by their unit or subunit-type specifications (at the time of this writing, there are no general entry types defined in this specification) and are assigned entry type values. For information on the *entry_type*, refer to 6.3.1, "List, entry, and info block type fields" on page 27.

attributes: The *attributes* field contains bit flags which indicate attributes that pertain to the entire entry structure. The following table illustrates the attributes that are defined for entry descriptors.

Table 7.3 – Entry descriptor attribute values

Attribute	Meaning
1xxx xxxx	**has_mor** **e_attributes:** If this bit is set to 1, then the next byte is also an attributes byte. All extended attribute bytes shall use this bit to indicate an extension to the next byte. In this and future versions of this specification, this bit shall be 0.
x1xx xxxx	**skip:** This bit indicates that the entry's underlying data is no longer available or valid. If this bit is set to 1, the controller must skip the entry. If this bit is set to 0, the data is available and the controller may read or write to the information in the entry. The use of this bit is unit, subunit-type and entry type dependent. If a unit, subunit-type or entry type does not use this attribute, its default value shall be 0. The unit and subunit can use this bit to perform a "delayed memory clean up" operation. By setting this bit after an entry deletion request, the unit and subunit can defer the actual deletion and reclamation of this memory until a convenient time. The unit or subunit may set this bit to one as a result of a controller deleting the entry.
xx1x xxxx	**has_child_ID:** If this bit is set to 1, then the entry has the *child_list_ID* field. If this bit is set to 0, then the entry does not have this field.
xxx1 xxxx	Reserved
xxxx 1xxx	**up_to_date:** This bit indicates the *validity* of data. If this bit is set to 1, then the entry is known by the unit or subunit to match with its underlying data. If this bit is set to 0, then the entry might be stale. The use of this bit is unit or subunit-type dependent. If a unit or subunit type does not use this attribute, its default value shall be 1. When data is stale, the unit or subunit may or may not be sure of this, and the controller should take this into consideration when dealing with the data. This attribute shall not change while the descriptor is open.
all others	Reserved for future specification.

child_list_ID: The *child_list_ID* is an optional field that holds the list ID of the child list descriptor that this entry refers to. If the entry does not have a child list, then the *has_child_ID* bit of the attributes field in the entry descriptor is set to 0, and this field shall not exist in the structure. An entry can have only one *child_list_ID* field.

object_ID: The *object_ID* is an optional field that contains a value that uniquely identifies the object that is represented by this entry, and can be used to reference the entry when executing descriptor commands. The *object_ID* can be writeable depending on the entry's type definition. Either all or no entries in a list contain *object_IDs*.

For more information on *object_IDs*, see clause 6.3.3, “Object_IDs” on page 29.

entry_specific_information_length: The *entry_specific_information_length* field specifies the number of bytes used for the following *entry_specific_information* field. The length field is two bytes and is not included in this calculation. If the value of this field is zero, then the *entry_specific_information* field shall not exist.

entry_specific_information: The *entry_specific_information* area will have a format and contents specific to the type of entry being referenced, which is defined by the entry type’s unit or subunit-type specification. For more information on general guidelines about how to define this field, see section 7.5 “Specific information fields in descriptors” on page 43.

Refer to the specific *entry_type* definitions in the unit or subunit specification for more details on this field.

extended_information: The *extended_information* field is used to place information for future extension of the general entry descriptor, and **is not used in this version of the specification**. It is recommended that these fields are extended with the general info block structure defined in clause 7.6 “Information blocks” on page 45. If a controller needs to, it can check if data exists in this field using the following formula:

> *entry_descriptor_length* >
> 4 + (if *has_child_ID* attribute exists, *size_of_list_ID*) +
> (if *has_object_ID* attribute exists, *size_of_object_ID*) + *entry_specific_information_length*

An *entry_descriptor_length* field less than the equation above is an error.

As controllers access the entry descriptor with a *generation_ID* (found in the (sub)unit identifier descriptor) higher than was used for their own descriptor mechanism, they shall be aware that this field may exist. Under this condition, a legacy controller can assume that there is extended data that it cannot access. In these circumstances, controllers must not assume that an error condition has occurred; rather, they should assume that the descriptor has been extended.

7.5 Specific information fields in descriptors (informative)

All three general descriptors contain specific information fields. These fields usually contain the primary information carried by the descriptor. The specific information fields for each descriptor are defined as follows:

Table 7.4 – General Descriptor’s Specific Information Fields

Descriptor Type	Specific Information Fields
Unit Identifier Descriptor	unit_information manufacturer_dependent_information
Subunit Identifier Descriptor	subunit_dependent_information manufacturer_dependent_information
List Descriptor	list_specific_information
Entry Descriptor	entry_specific_information

There are three suggestions for the configuration of these fields when designing their format as shown in the figures below. Unit and subunit-type specifications will contain the actual formats of these fields.

Address Offset	Length, bytes	Controller Read/Write	Contents
00 00_{16}	see[1]	– [2]	Non Info Block data...
00 01_{16}			
:			

[1] The size of this block is equal to the specific information length.
[2] unit or subunit-type dependent.

Figure 7.6 – Specific information fields with non info block data

Address Offset	Length, bytes	Controller Read/Write	Contents
00 00_{16}	see[1]	– [2]	Info Block data...
00 01_{16}			
:			

[1] The size of this block is equal to the specific information length.
[2] unit or subunit-type dependent.

Figure 7.7 – Specific information fields with info blocks only

Address Offset	Length, bytes	Controller Read/Write	Contents
00 00_{16}	2	– [2]	non_info_block_length = I
00 01_{16}			
00 02_{16}	i	– [3]	Non Info Block data...
:			
:	see[1]	– [3]	Info Block data...
:			

[1] The size of this block is equal to the specific information length – 2 – i.
[2] unit or subunit-type dependent. (Either R or R/W)
[3] unit or subunit-type dependent.

Figure 7.8 – Specific information fields with both

If non info block only or info block only data shall exist in these fields, then the specific information does not need an internal length field, since the length is provided by the previous fields-length field.

If info block data exists with non-info block data, then the non-info block data shall precede the info block data, and a two byte *non_info_block_length* field shall precede the non-info block data.

7.6 Information blocks

Information blocks (or info blocks) are defined as a group of extensible and common data structures that may be used across a broad range of units and subunits. They further increase the organization of AV/C data, and are consistent with the descriptor navigation model. Information blocks can be placed within the following areas:

— *unit_dependent_information* and *manufacturer_dependent_information* field of the unit identifier descriptor

— *subunit_dependent_information* and *manufacturer_dependent_information* field of the subunit identifier descriptor

— *list_specific_information* field of the list descriptor

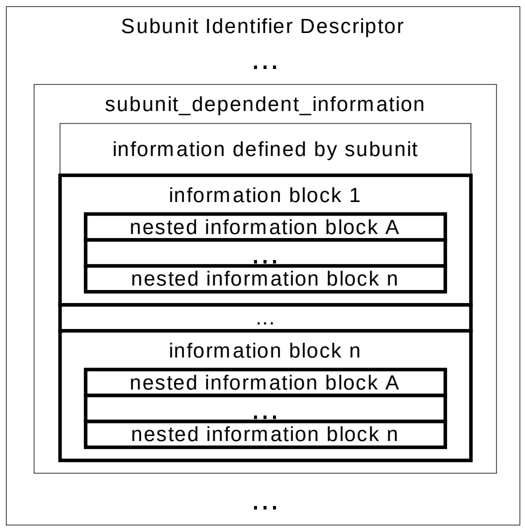
— *entry_specific_information* field of the entry descriptor

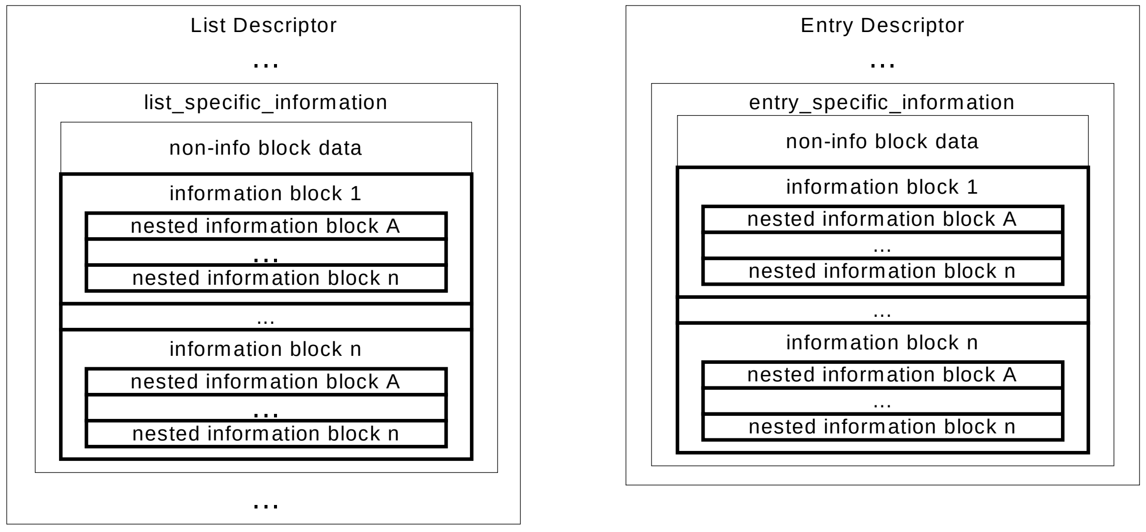
— *extended fields* of all general descriptors

— nested within another info block

— inside subunit-dependent descriptors – refer to the subunit-type specification for more details.

This information block structure is depicted in each general descriptor in the following figure:


Subunit Identifier Descriptor
...
subunit_dependent_information
information defined by subunit
information block 1
nested information block A
...
nested information block n
...
information block n
nested information block A
...
nested information block n
...


List Descriptor
...
list_specific_information
non-info block data
information block 1
nested information block A
...
nested information block n
...
information block n
nested information block A
...
nested information block n
...
Entry Descriptor
...
entry_specific_information
non-info block data
information block 1
nested information block A
...
nested information block n
...
information block n
nested information block A
...
nested information block n

Note: Nested information blocks may contain other nested information blocks, producing an internal hierarchy.

Figure 7.9 – Information blocks within (sub)unit identifier, list and entry descriptors

Information Blocks have a data structure extensibility mechanism, similar to the IEEE 1212 tagged-field model. However, information blocks differ from the 1212 model in one basic way: where 1212 defines a mechanism for EVERY field to have its own tag (and length), an information block contains several fields which are associated with the one information block tag. In this way, information blocks are closer to the leaf data structures defined by IEEE 1212.

Controllers that understand the info block model should be designed to expect any info blocks to occur at any location in a descriptor within the constraints of where info blocks are expected, and to not treat their appearance as an error. If a controller discovers an info block of an unknown type, it shall skip that info block and any nested info blocks it may contain.

Info blocks generally do not have dependencies on their order of appearance in a descriptor, so controllers should also not assume significance in the order in which they are discovered. However, certain units and subunit types may set rules on the order of appearance of info blocks; controllers that understand these unit or subunit-specific rules may assume the appearance order of info blocks. Refer to the appropriate unit or subunit-type specification for details.

7.6.1 Information block structure

The basic structure of an information block is as follows:

Address Offset	Length, bytes	Controller Read/Write	Contents
$00\ 00_{16}$	2	R/W	compound_length = i
$00\ 01_{16}$			
$00\ 02_{16}$	2	R/W	info_block_type
$00\ 03_{16}$			
$00\ 04_{16}$	2	R/W	primary_fields_length = j
$00\ 05_{16}$			
$00\ 06_{16}$	j	–[1]	primary_fields… (info block type-specific)
:			
:			
:	i-j-4	–[1]	secondary_fields… (optional)
:			
:			

[1] unit or subunit-type dependent.

Figure 7.10 – General information block

7.6.1.1 Information block fields

compound_length: The *compound_length* field specifies the number of bytes for the remainder of this info block structure, which is all fields that follow this length field. The two bytes used for the *compound_length* field are NOT included in this value.

info_block_type: The *info_block_type* field describes the meaning and format of this info block. The type definitions are divided into two scopes: *general* and *subunit-type specific*. Within the subunit-type specific range, additional classifications may apply (for example, the disc subunit defines information blocks that are specific to a given type of disc media). For more information, see clause 6.3.1, “List, entry, and info block type fields” on page 27.

For details on the general information blocks that are currently defined, please refer to [R10]. For details on subunit-type specific info block type definitions, please refer to the appropriate unit or subunit-type specification.

primary_fields_length: The *primary_fields_length* field specifies the number of bytes for the following *primary_fields*.

primary_fields: The *primary_fields* area contains a set of fields in a data structure that are specific to the type of info block represented by the *info_block_type* field. The meaning and format of these fields is fixed, and will NOT be re-defined once the info block type is defined.

The interpretation of the primary fields depends not only on the info block type, but also on the SCOPE of where the info block is found. For example, if an info block is found inside of a *list_specific_information* structure, then that info block applies to the list. If the info block is found inside of an *entry_specific_information* structure, then it applies to that entry. Likewise, if an info block is nested inside another info block, then the nested block applies to its container block. A controller which understands the *primary_fields* area will always know the basic structure of the given info block. Note that while this area is well defined, it MAY vary in length depending on its definition. For example, it might consist of a set of fields which is common in the AV/C descriptor mechanism, such as {*number_of_XXX_fields*, *XXX_field[0]*,…, *XXX_field[n – 1]*}. Depending on the value of *number_of_XXX_fields*, the size of the well-defined area might be different from one info block to the next, but the MEANING and interpretation is the same. Thus, controllers will always be able to navigate the structure (or navigate around the structure by using the length fields to jump past bytes).

The primary field may contain mandatory info blocks (for example, the name_info_block contains the character_code_info_block, the language_code_info_block, and the raw_text_info_block.) Info blocks in the primary field share the level for their references with optional info blocks in the secondary field.

secondary_fields: The *secondary_fields* area represents optional info blocks that may be nested inside the one being parsed. Generally, there may be any number and any type of info blocks in this area, but unit or subunit-type specifications may impose certain restrictions. Those info blocks may further contain other info blocks.

Generally, there are no theoretical limits to the level of nesting, however, for practical reasons only a few levels would likely be useful. Unit and subunit-type specifications may impose limits on the level of nesting; refer to the appropriate unit or subunit-type specification for details.

A controller can detect the presence of *secondary_fields* by comparing the *compound_length* field to the *primary_fields_length* field. If *compound_length* = *primary_fields_length* + 4, then there are no *secondary_fields* (note that the value of *compound_length* includes the two bytes used for *primary_fields_length*). If the *compound_length* field is greater than *primary_fields_length* + 4, then one or more nested info blocks are present. If the *compound_length* field is LESS THAN the *primary_fields_length* + 4, then an error has occurred.

7.6.2 Expanding information block structures

There are two main reasons for expanding an information block structure:

1) To add new fields which were not included in the original definition.
2) To allow a flexible means of associating pieces of data with other pieces of data, based on the needs and/or capabilities of the user, the controller, and the unit or subunit (for example, to associate a set of lyrics with an audio track, to associate some user data with an image, etc.).

If an existing information block type needs to have additional fields added to it, then new information blocks shall be defined and nested inside the existing information block. The *primary_fields* shall NOT be changed to accommodate new field definitions.

7.6.3 Restrictions on information block contents

The general information block allows for any type of information to be embedded in the block, as long as its *information_block_type* field is defined and optional nested information blocks are present. It is possible that some restrictions may be placed on the allowable contents based on a unit or subunit implementation, or a unit or subunit-type specification.

For example, a name_info_block that describes the title of an audio track on a disc includes the language code for the text. Even though a large number of language codes may be defined, a particular implementation of the unit or subunit might not support some of them, so it places a restriction on which language codes can be stored in the name_info_block.

Also, as mentioned previously, additional restrictions such as the number, type, and level of nested information blocks may be imposed by an implementation or specification.

For unit, subunit- and media-type restrictions, please refer to the appropriate specification documents.

8. Referencing descriptors and info blocks

When AV/C commands are used to operate on descriptors and info blocks, *descriptor_specifier* structures reference the particular descriptors, and *info_block_reference_path* structures reference info blocks.

The following clause describes each of these descriptor specifier types. The *info_block_reference_path* is discussed in 8.3 “Information block reference path” on page 55.

8.1 Descriptor specifier

In order to reference existing descriptors and info blocks or create new descriptors in descriptor hierarchies, descriptor and info block commands use a data structure called the *descriptor_specifier*. The *descriptor_specifier* has been enhanced to include specifying info blocks.

The descriptor specifier is a versatile data structure that allows the designer several options for specifying descriptors based how a unit or subunit organizes its data. The exact type of the specifier used will vary based on the kind of descriptor, the command used, and the unit or type of subunit that is managing that descriptor.

The general AV/C model defines the (sub)unit identifier descriptor, entries, lists, and info blocks as the kinds of data structures that are accessed through descriptor and info block commands. It is possible that subunit-dependent descriptors may be defined exclusively for a particular type of subunit. Such descriptors would also be accessed using the descriptor specifier structure in commands.

The following table illustrates the general descriptor specifier data structure:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type							
:	see[1]	descriptor_specifier_type_specific_fields							
:									
:									

[1] The length of this field depends on the descriptor specifier type

Figure 8.1 – General descriptor specifier

descriptor_specifier_type: The *descriptor_specifier_type* field indicates what kind specifier is used and determines the information that comes next in the *descriptor_specifier_type_specific_fields*. The following table lists the *descriptor_specifier_type* encoding:

Table 8.1 – Descriptor_specifier_type meanings

descriptor_ specifier_type	Purpose	Meaning
00_{16}	Reference	(Sub)unit identifier descriptor
10_{16}	Reference	List descriptor - specified by *list_ID*
11_{16}	Reference/ Create	List descriptor - specified by *list_type*
20_{16}	Reference	Entry descriptor - specified by entry position in a list specified by *list_ID*
21_{16}	Reference	Entry descriptor - specified by *object_ID* in a list specified by *list_type* under a root list specified by *root_list_ID*
22_{16}	Create	Entry descriptor – specified by *entry_type*
23_{16}	Reference	Entry descriptor - specified by *object_ID*
30_{16}	Reference	Information block – specified by its type and instance count in a containing info block or descriptor
31_{16}	Reference	Information block – specified by position in the containing info block or descriptor
80_{16} - BF_{16}	Reference	Subunit dependent descriptor. Refer to the subunit-type specification for more details
all others		Reserved for future specification

See each unit or subunit-type specification for the *descriptor_specifiers* that are supported by that unit or subunit type. Each descriptor command will use a subset of descriptor specifiers above. See each command for details.

descriptor_specifier_type_specific_fields: Each of the *descriptor_specifier_type* values in the table indicates the format and contents of the *descriptor_specifier_type_specific_fields* and are given in the sections below.

8.2 Descriptor_specifiers for descriptors

8.2.1 (Sub)unit identifier descriptor specifier type

The descriptor specifier for (sub)unit identifier descriptor is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 00_{16}							

Figure 8.2 – Descriptor_specifier for a (sub)unit identifier descriptor

The *descriptor_specifier_type_specific_fields* does not exist (the *descriptor_specifier* consists of only the *descriptor_specifier_type* field) because there can be only one (sub)unit identifier descriptor for a unit or subunit.

NOTE — Whether the unit identifier descriptor or the subunit identifier descriptor is being referenced in a command using this descriptor specifier depends on whether the command is addressed to the unit or subunit, as defined in its *subunit_type* field. For more information, see "*subunit_type*" in reference [R9].

8.2.2 List descriptor specified by list ID

The descriptor specifier for a list descriptor specified by list ID is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 10_{16}							
:	see[1]	list ID (child or root)							
:									
:									

[1] The length shall be equal to the value given in the (sub)unit identifier descriptor's *size_of_list_ID* field.

Figure 8.3 – Descriptor_specifier for a list descriptor specified by list ID

list ID: The *descriptor_specifier_type_specific_fields* consists of the desired list ID. All lists within the scope of a unit or subunit shall have unique list ID values, so there is no need to resolve the scope any further.

8.2.3 List descriptor specified by list_type

The *descriptor_specifier* for a list descriptor specified by its *list_type* is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 11_{16}							
:	1	list_type							

Figure 8.4 – Descriptor_specifier for a list descriptor specified by list_type

list_type: This descriptor specifier is normally used when units or subunits that have only one list per list type specified.

NOTE — If multiple lists exist in a subunit with the same list type, the specified descriptor is ambiguous. When used with all descriptor commands except the SEARCH DESCRIPTOR command, a target shall return a response with an arbitrary descriptor that matches the specifier.

8.2.4 Entry descriptor specified by position in its list

Entries can be referenced by their position in a specified list as shown in the figure below.

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 20_{16}							
:	see[1]	list ID (child or root)							
:									
:									
:	see[2]	entry_position							
:									
:									

[1] The length shall be equal to the value given in the subunit identifier descriptor's *size_of_list_ID* field.
[2] The length shall be equal to the value given in the subunit identifier descriptor's *size_of_entry_position* field.

Figure 8.5 – Descriptor_specifier for referencing an entry's position

list ID: This is the list ID of the list containing the entry.

entry_position: The *entry_position* field indicates the position of the target entry within the list that was specified by the list ID field. Entry positions start at 0. The value of all FF_{16} bytes for the *entry_position* is reserved, and specifies the end of the list when used with the WRITE DESCRIPTOR and CREATE DESCRIPTOR control commands. Please refer to the definition of those commands for details.

8.2.5 Entry descriptor specified by object_ID

For the *object_ID* reference, the *descriptor_specifier* is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 21_{16}							
:	see[1]	root_list_ID							
:									
:									
:	1	list_type							
:	see[2]	object_ID							
:									
:									

[1] The length shall be equal to the value given in the subunit identifier descriptor's *size_of_list_ID* field.
[2] The length shall be equal to the value given in the subunit identifier descriptor's *size_of_object_ID* field.

Figure 8.6 – Descriptor_specifier for an object_ID reference

root_list_ID: The root list ID under which the entry with the *object_ID* exists.

list_type: The list type value, either *subunit-type specific* or *general*, of the list(s) that contains the desired entry.

object_ID: The *object_ID* field indicates the unique *object_ID*. This reference may be used when *object_ID* values are unique among the scope of all lists that share the same *list_type* value, within the descriptor hierarchy indicated by *root_list_ID*.

WARNING: In some cases, this reference may not uniquely identify a specific entry within a descriptor hierarchy. In case one or more entries in the descriptor hierarchy indicated by the root list have the same ID and belong to lists with the same list type, then an arbitrary entry will be addressed. This will depend on the particular technology being represented by the list and entry descriptors.

NOTE — The SEARCH DESCRIPTOR command can use an ambiguous descriptor specification to search through multiple descriptors.

It is also possible that *object_ID*s do not exist in entry descriptors for some list or entry types. Whether *object_ID*s exist in child entries is determined by the attributes byte(s) within the list descriptor. In this case, the entry must be specified by its position in the list using the *descriptor_specifier_type* = 20_{16}.

8.2.6 Entry descriptor specified by entry_type

The format of *descriptor_specifier_type* 22_{16} is only valid for creating descriptors, and is as follows:

	length	msb							lsb
operand[x]	1	Descriptor_specifier_type = 22_{16}							
:	1	entry_type							

Figure 8.7 – Descriptor_specifier for an entry specified by entry_type

entry_type: Entry type can be either a *subunit-type specific* or *general* type.

8.2.7 Entry descriptor specified only by object_ID

An entry may be specified by object_ID only when the (sub) unit is designed to contain entries with unique object_IDs within the entire descriptor hierarchy. The format of *descriptor_specifier_type* 23_{16} is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 23_{16}							
:	see[1]	Object_ID							

[1] The length of *object_ID* is determined by the *size_of_object_ID* field in the subunit identifier descriptor

Figure 8.8 – Descriptor_specifier for an entry specified by object_ID only

object_ID: The requested *object_ID* of the entry to access.

8.2.8 Advantages and disadvantages of specifying by object_ID and entry_position (informative)

This section is intended to help subunit-type specification writers determine which descriptor_specifier to use for their specification.

The following table shows the advantages and disadvantages for using the descriptor specifiers discussed in the previous three sections.

Table 8.2 – Advantages and disadvantages to using the various entry specifiers

Descriptor_Specifier_Type	Advantage	Disadvantage
20_{16} (by entry_position)	1. Doesn't require object ID. 2. Can be used for contents discovery.	1. No guarantee that the same data will be read each time due to possible changes in the number of entries in the list descriptor. 2. Contents may change during discovery if entries are added or deleted.
21_{16} (by object_ID)	Controllers can access entries without concern for whether entries are added or removed from their lists.	1. Controller cannot know IDs without discovery, unless controller originally wrote data, or unless an object_ID scheme is used. 2. object_IDs must be present in descriptors.
23_{16} (by object_ID)	Does not require list_type or list_ID specifiers to access entry.	Object_IDs must be unique throughout subunit.

A controller can specify by entry_position effectively for contents discovery. If entries contain object_IDs, then once a controller has discovered and knows those IDs, it is more reliable for a controller to access a descriptor by object_ID. Refer to the unit or subunit-type specification to determine which method(s) are used.

NOTE — If a unit or subunit supports creating a descriptor, it must support specifying entries by entry_position.

8.3 Information block reference path

The information block reference path is used to reference info blocks in info block commands.

Info blocks, whether nested or not, can be thought of as existing at various levels hierarchically within a descriptor. The descriptor which contains info blocks is considered to be at the highest level or level[0]. A non-nested info block within the descriptor would exist at the next level down or level[1]. Nested info blocks within info blocks are at even further levels.

At a specific level, info blocks may be referred to by their position, that is, the order in which they appear when reading the container structure, i.e., their descriptor or info block, from the beginning to the end. Another reference method is by the instance count of a specified info block type within its container structure (e.g. the third block of type “x”).

Consider the following diagram, which shows a list descriptor, which contains some entry descriptors. One of the entry descriptors contains two info blocks at level[1] (info blocks A and B). Info block B contains two others (blocks X and Y) at level[2].

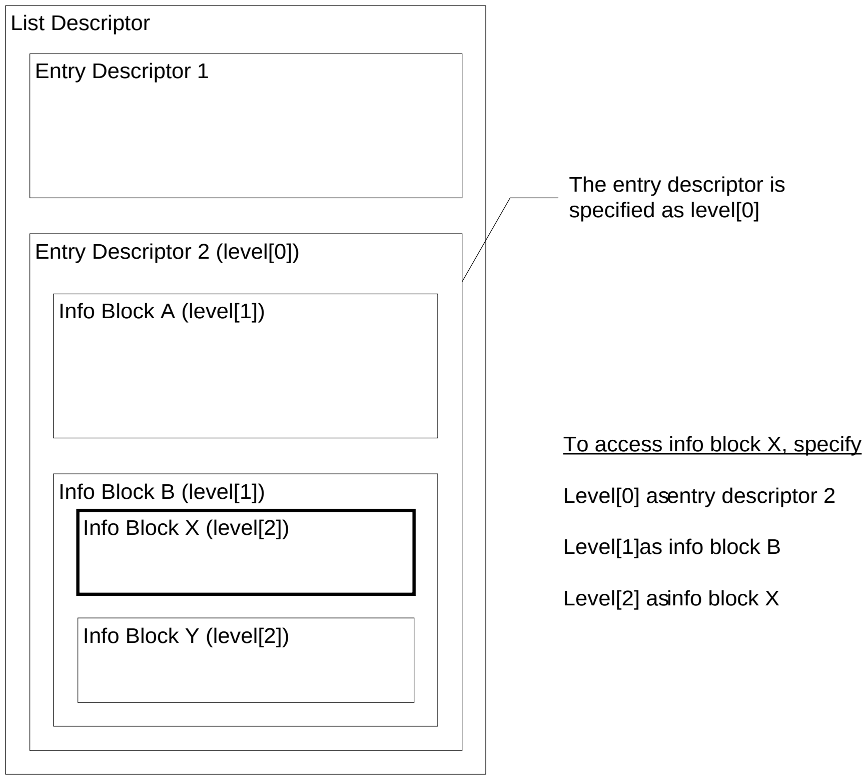


Figure 8.9 – Referencing info blocks

When referencing info blocks that are inside of list descriptors or entry descriptors, the following rules apply:

1) To access an info block in the *list_specific_information* area of a list, specify the list as level 0, then the info block(s) at the subsequent levels.
2) To access an info block inside an entry descriptor, specify the entry as level 0, and the info block(s) at the subsequent levels. Do NOT specify the list as level 0, and the entry as level 1.

Information blocks outside the entry descriptor are not candidates for access when the entry descriptor is chosen as level[0].

The scope of the info blocks in Figure 8.9 above is that of entry descriptor 2.

8.3.1 The info_block_reference_path structure

The number of levels and the specification of each descriptor or info block at each level are referred to as the *info_block_reference_path*. This structure has the following general format:

	length	msb							lsb
Operand[x]	1	number_of_levels = n							
:									
:	see[1]	descriptor_specifier for descriptor at level[0]							
:									
:									
:	see[1]	descriptor_specifier for info block at level[1]							
:									
:		:							
:									
:	see[1]	descriptor_specifier for info block at level[n – 1]							
:									

[1] The length of this field is determined by the descriptor specifier types used.

Figure 8.10 – Info_block_reference_path structure

number_of_levels: The *number_of_levels* field specifies the number of levels from the top of the info block hierarchy to the level of the information block being specified.

level[n]: Level [0] is a descriptor specifier for a descriptor described in clause 8.2 “Descriptor_specifiers for descriptors” on page 51. *Level[1]* to *level[n-1]* fields are descriptor specifiers for info blocks and each specify an info block at the specified level (the level is the same as the index value). Descriptor specifiers for info blocks are described in 8.4, “Descriptor_specifiers for info blocks ” on page 56.

Note the following:

1) There may be more levels in the full info block hierarchy than are specified in the structure (for example, if the info block being referenced contains other info blocks, there are more levels to the info block hierarchy).
2) The level [0] of a path is an identifiable descriptor structure. In Figure 8.9 above, entry descriptor 2 can be specified in a level [0] field of the *info_block_reference_path*, so it is the top of the path (even though it looks like the list structure is the top of the path).
3) The minimum number of levels is 2.

8.4 Descriptor_specifiers for info blocks

Descriptor_specifiers for Info blocks are used within info block reference paths as described above.

8.4.1 Info block specified by info block type and instance count

The format of *descriptor_specifier_type* 30_{16} is only used in info block reference paths and is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 30_{16}							
:	2	info_block_type							
:									
:	1	instance_count							

Figure 8.11 – Info_block_specifier for an info block specified by its type and instance count

info_block_type: The *info_block_type* field specifies the type of info block.

instance_count: The *instance_count* specifies which info block instance of the specified type is being referenced. For the first instance of an info block from the top of the container, *instance_count* = 0.

8.4.2 Info block specified by position in container structure

The format of *descriptor_specifier_type* 31_{16} is only used in info block reference paths, and is as follows:

	length	msb							lsb
operand[x]	1	descriptor_specifier_type = 31_{16}							
:	1	info_block_position							

Figure 8.12 – Descriptor_specifier for an info block, specified by its position

info_block_position: The *info_block_position* specifies the position, from the top of the container, defined as the order of occurrence at a given info block hierarchy level of the information block.

This info block doesn't consider the info block type when referencing the info block.

8.5 Info block reference path examples

For the following examples, refer to Figure 8.9.

The *info_block_reference_path* for info block X would appear as follows:

	length	msb							lsb
operand[x]	1	number_of_levels (n) = 3							
:	5	20_{16} (entry ref. by position in list) Level 0[1]							
:		XX_{16} (list ID MSB)							
:		XX_{16} (list ID LSB)							
:		00_{16} (entry position MSB)							
:		01_{16} (entry position LSB)							
:	2	31_{16} (info block ref. by position) Level 1							
:		01_{16} (info block B position)							
:	2	31_{16} (info block ref. by position) Level 2							
:		00_{16} (info block X position)							

[1] Assumes 2-byte list ID and object position.

Figure 8.13 – Example info_block_reference_path structure

The use of specifier type 30_{16} (info block reference by type and instance count) would be similar; for this example, assume that blocks A and B are different types and that X and Y are the same type.

The *info_block_reference_path* for info block X would then appear as follows:

	length	msb							lsb
operand[x]	1	number_of_levels (n) = 3							
:	5	20_{16} (entry ref. by position in list) Level 0[1]							
:		XX_{16} (list ID MSB)							
:		XX_{16} (list ID LSB)							
:		00_{16} (entry position MSB)							
:		01_{16} (entry position LSB)							
:	4	30_{16} (info block ref. by type and instance count) Level 1							
:		XX_{16} (info block B type MSB)							
:		XX_{16} (info block B type LSB)							
:		00_{16} (info block B instance count)							
:	4	30_{16} (info block ref. by type and instance count) Level 2							
:		XX_{16} (info block X type MSB)							
:		XX_{16} (info block X type LSB)							
:		00_{16} (info block X instance count)							

[1] Assumes 2-byte list ID and object position

Figure 8.14 – Example info_block_reference_path structure

9. Descriptor and info block commands

9.1 Descriptor commands overview

This section defines commands that operate on descriptor and info block structures. Table 9.1 below summarizes these commands.

Table 9.1 – Descriptor and info block commands

Opcode	Value	Support level (by *ctype*)			Target	Comments
		C	S	N		
CREATE DESCRIPTOR	$0C_{16}$	O	–	–	Unit or subunit	Create a new list or entry descriptor
OPEN DESCRIPTOR	08_{16}	M[1]	O	O	Unit or subunit	Gain and relinquish access to a specified descriptor
READ DESCRIPTOR	09_{16}	M[1]	–	–	Unit or subunit	Reads data from a specified descriptor
WRITE DESCRIPTOR	$0A_{16}$	O	O	–	Unit or subunit	Writes data into a specified descriptor
OPEN INFO BLOCK	05_{16}	O	O	–	Unit or subunit	Gain and relinquish access to a specified info block
READ INFO BLOCK	06_{16}	O	–	–	Unit or subunit	Read a specified info block
WRITE INFO BLOCK	07_{16}	O	–	–	Unit or subunit	Write data into a specified info block
SEARCH DESCRIPTOR	$0B_{16}$	O	–	–	Unit or subunit	Search for a pattern of data within a descriptor set
OBJECT NUMBER SELECT	$0D_{16}$	O	O	O	Unit or subunit	Select one or more entries using an *object_ID* and list ID

[1] mandatory only if the (sub)unit supports descriptors.

In the preceding table, a dash in one of the support level columns indicates that the command is not defined for the *ctypes*, CONTROL, STATUS or NOTIFY, as indicated.

Closing a descriptor is done with an OPEN DESCRIPTOR subfunction. Deleting a descriptor is done with a WRITE DESCRIPTOR subfunction.

NOTE — As of this revision of this specification, info blocks cannot be created or deleted apart from being created as the result of a CREATE DESCRIPTOR / WRITE DESCRIPTOR command, or deleted as a result of a WRITE DESCRIPTOR command deleting an entry or list.

9.2 Reading and writing AV/C descriptor structures

Prior to the creation of the information block model, all reading and writing of AV/C descriptors was accomplished through the READ DESCRIPTOR and WRITE DESCRIPTOR commands.

The AV/C descriptor structures were designed to abstract the way a (sub)unit chooses to actually store data internally. Only when it is time to provide or receive this data is it necessary to use the descriptor structure formats.

This same concept applies to the information block model. Because information blocks are contained inside of descriptor structures (or nested in other information blocks), the READ DESCRIPTOR command can be used to read them.

However, information blocks are "encapsulated" blocks of information. Restrictions on their use and location might be decided by a subunit implementation or type specification; therefore, a new command is needed for modifying information blocks.

The WRITE INFO BLOCK command is used for this purpose. In the command, the specific info block and the new data for that block is specified. The (sub)unit has the opportunity to check for violations of any particular info block usage policies that may be in effect. If no violations are detected, the (sub)unit is free to store the data internally in a proprietary manner (or a media-type-specific manner). However, if a violation is detected, then the (sub)unit can return an explanation of why the command was rejected, and the controller can attempt to fix the situation (or at least inform the user as to why the operation failed).

For details on the OPEN INFO BLOCK, READ INFO BLOCK and WRITE INFO BLOCK command, please refer sections 9.7, 9.8 and 9.9.

Another reason for the command is to allow the (sub)unit to prepare any necessary supporting data structures or other state information, and to link them together if necessary, in order to accommodate the new information block.

The following diagram illustrates the relationship between descriptor structures and info blocks, and the READ DESCRIPTOR, WRITE DESCRIPTOR, READ INFO BLOCK and WRITE INFO BLOCK commands:

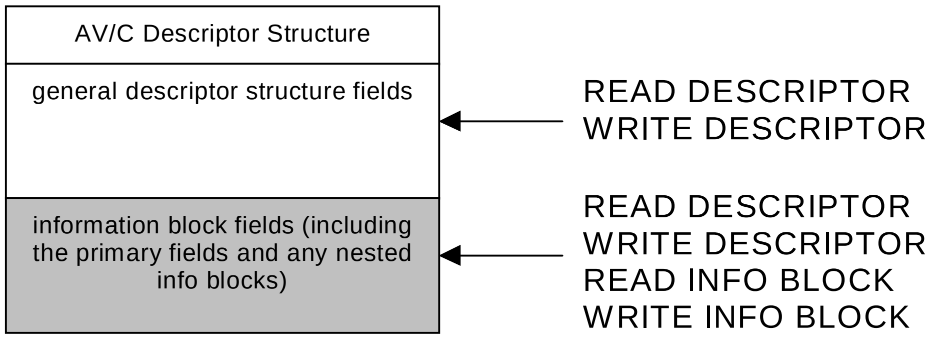

AV/C Descriptor Structure
general descriptor structure fields
information block fields (including the primary fields and any nested info blocks)
READ DESCRIPTOR
WRITE DESCRIPTOR
READ DESCRIPTOR
WRITE DESCRIPTOR
READ INFO BLOCK
WRITE INFO BLOCK

Figure 9.1 – AV/C Descriptor Structure

As shown above, the READ INFO BLOCK and WRITE INFO BLOCK control commands can only be used on info blocks; the READ DESCRIPTOR and WRITE DESCRIPTOR control commands can be used on descriptors or info blocks. For more information about creating and writing an info block using WRITE DESCRIPTOR control commands, see clause 9.2.2.2 "Access rules for writing descriptors and info blocks" on page 62. It is also possible to perform a READ/WRITE INFO BLOCK control command after issuing OPEN DESCRIPTOR control command.

IMPORTANT: Some (sub)unit specifications might place restrictions on the use of READ/WRITE INFO BLOCK/DESCRIPTOR combinations. Please refer to the (sub)unit specification document(s) for details.

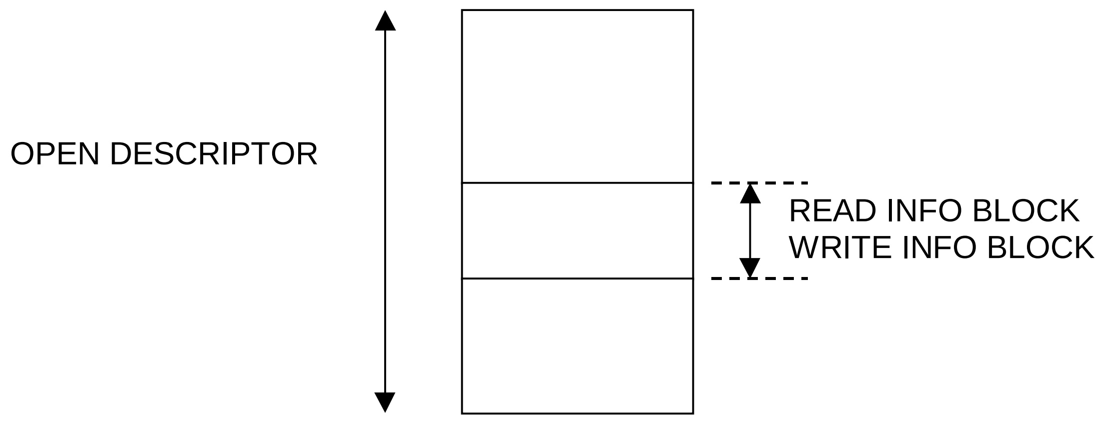


Figure 9.2 – relation between OPEN DESCRIPTOR and READ/WRITE INFO BLOCK

READ/WRITE INFO BLOCK commands are available to access info blocks in the descriptor which is opened by OPEN DESCRIPTOR command. Please refer to section 9.2.2.1 "Access rules for opening descriptors and info blocks" on page 61.

9.2.1 Access support

A (sub)unit may support three levels of access as follows:

— **Descriptor Level 1:** Access to entries and info blocks by opening their list (mandatory).

— **Descriptor Level 2:** Access to entries and info blocks by opening the entry (optional and not recommended).

— **Descriptor Level 3:** Access to info blocks by opening the info block (optional and not recommended).

While it is mandatory for a (sub)unit that has descriptors to support the OPEN DESCRIPTOR control commands, it is not mandatory nor recommended for it to support that command down to the individual entry descriptor or info block levels (levels 2 & 3). For example, it is not required that a (sub)unit be sophisticated enough to allow one controller to have read/write access to entry 2, and a separate controller to simultaneously have read/write access to entry 7 in the same list. It is sufficient and recommended to allow access control only at the list descriptor level.

9.2.2 Access rules

When a descriptor or info block is accessed, the rules below shall be followed. These rules are intended to help the target maintain a fair and stable environment for itself and external controllers.

9.2.2.1 Access rules for opening descriptors and info blocks

1) **General open rule:** A descriptor can be accessed only for the controller that opened it. If a second controller wishes to access a descriptor that has been opened by a first controller, the second controller must still issue an OPEN DESCRIPTOR control command.

2) **Multiple read-only opens:** If a descriptor or info block is closed or has only been opened for read access, a (sub)unit may accept any number of OPEN DESCRIPTOR control commands with a subfunction of read-open as long as the (sub)unit is able to accommodate additional controllers accessing for read-only.

3) **Only one write-open accepted:** If a descriptor or info block is closed or has only been opened for read access, a (sub)unit may accept a single OPEN DESCRIPTOR control command or OPEN INFO BLOCK control command with a subfunction of write-open. If accepted, this open operation for write access forces any existing read-only opens of itself and all of its embedded entries or info blocks, and its enclosing descriptor(s) to be closed. Though it is possible for multiple controllers to access different entries in a list, or info blocks in an entry, any attempt to do the following shall be REJECTED by the (sub)unit:

    a) While one controller has write access to a list, another controller attempts to gain access (write or read) to any of its entries or info blocks within that list.

    b) While one controller has write access to an entry, another controller attempts to gain access (write or read) to any of its info blocks within it.

    c) While one controller has write access to an entry, another controller attempts to gain access (write or read) to its list.

    d) While one controller has write access to an info block, another controller attempts to gain access (write or read) to its enclosing entry or list.

4) **Accept identical open commands:** The (sub)unit shall always accept a command from a controller to close a descriptor, even when the controller has not opened or has already closed the descriptor. If a controller has opened a descriptor or info block for read or write access, the target shall accept a request to open the descriptor or info block for the same access by the same controller. However, if a controller has opened a descriptor or info block for write access, the target shall reject any requests to open the descriptor or info block for read access by the same controller.

5) **Write-open enclosing descriptor before adding or deleting:** If a controller intends to add or delete an entry, then it shall first gain write access control to its list. If a controller intends to add or delete a child list, then it shall first gain write access control to its parent entry's list.

9.2.2.2 Access rules for writing descriptors and info blocks

1) WRITE DESCRIPTOR and WRITE INFO BLOCK commands shall write "0" to data locations that are reserved if writing to an area that includes them. Any attempt to write other than "0" to them shall be REJECTED.

2) WRITE DESCRIPTOR and WRITE INFO BLOCK commands that attempt write data that is out of range or is otherwise invalid according to this or the target's (sub)unit specification shall be REJECTED.

3) An entry descriptor can be created using CREATE DESCRIPTOR commands. An entry descriptor can be written using WRITE DESCRIPTOR commands with a descriptor specifier for the entry descriptor. Attempt to create or write an entry descriptor using WRITE DESCRIPTOR commands with a descriptor specifier for the list descriptor should be REJECTED.

4) The non_info_block_length field can be created or written by WRITE DESCRIPTOR commands only when it attempts to write an area that includes both the non_info_block_length field and the whole non info block data.

5) An info block can be created or written by WRITE DESCRIPTOR commands only when it attempts to write to an area that includes the whole info block structure (from compound_length field to secondary_fields).

6) Each field in a descriptor structure can be read/write(R/W), read/write/ignore(R/W/I) read/ignore(R/I) or read(R) as indicated in the Controller Read/Write column. See section 5.4 for more information about Controller Read/Write. The following table defines the target behavior

when a controller sends WRITE DESCRIPTOR and WRITE INFO BLOCK control commands. It is recommend that (sub)unit specifications define Controller Read/Write for fields in descriptor and info block structures that are not defined in this specification.

Table 9.2 –Controller Read/Write attribute and target behaviors

Controller Read/Write	Rule
read/write(R/W)	If ACCEPTED, the target changes these fields as specified in the command frame.
read/write/ignore(R/W/I)	If ACCEPTED, the target changes these fields as specified in the command frame, changes them to other values, or does not update them. For example, the target may round to the values to the acceptable values, truncate the character strings, or the target implementation or state does not allow to change the fields.
read/ignore (R/I)	If ACCEPTED, the target does not change these fields. For example, those fields that indicate the capabilities of the target and are located in the writeable area may be R/I. Note that *replace* and *insert* subfunctions may write these fields.
read(R)	The target shall return a response of REJECTED when the command frame attempts to write an area that includes these fields. For example, those fields that indicate the capabilities of the target may be R. Note that *replace* and *insert* subfunctions may write these fields.

NOTE — The target shall return the subfunction value in the response frame according to the result of write operation. See Table 9.26 and Table 9.38.

9.2.2.3 Access rules for closing descriptors and info blocks

1) **Close read or write-enabled descriptor only by opener:** If a descriptor or info block is open for read or write access by a controller, only that controller can close it for itself.

2) **Accept identical closure:** If a controller has not opened a descriptor or info block, a (sub)unit shall accept OPEN DESCRIPTOR or OPEN INFO BLOCK commands with close subfunction from the same controller to close the descriptor or info block. However, if other controllers have opened the descriptor or info block, the (sub)unit shall keep the descriptor or info block open for them.

3) **Force-closure:** A descriptor or info block that is open for write may be force-closed by a high priority command, by a request from a front panel, or at the (sub)unit’s convenience. The controller shall construct its write operations carefully so that if a force-closure occurs, descriptor integrity is not compromised. After a force-closure, if a controller regains access to the descriptor, it should assume the descriptor has changed as a result of the force-closure.

4) **Atomic operations:** All descriptor and info block commands executions shall be atomic – that is, force closure shall not occur during the execution of any one descriptor command. For example, if a WRITE DESCRIPTOR control command writes five bytes, all five bytes must be written before a force-closure. If a bus reset occurs during the execution of a command, any changes to the descriptor shall be canceled.

5) **Close when done:** If a controller intends to read or write to a descriptor or info block, and it is able to gain access to that descriptor or info block, then it shall relinquish control of that same descriptor or info block when it is finished. Controllers are strongly recommended to keep descriptors and info block open (for read only or read/write access) only for the duration that access is needed, and to relinquish access as soon as possible.

9.2.2.4 Summary of the access rules for opening and closing descriptors and info blocks (informative)

The table below indicates the responses (A = ACCEPTED or R = REJECTED) received when a controller sends various OPEN DESCRIPTOR control commands under various open states of the descriptor.

Descriptor Z is a descriptor or info block on a target that controllers X and Y may be accessing. The columns indicate the current state of the descriptor Z as would be returned in the status field of the response of OPEN DESCRIPTOR status command, and the columns are also divided by which controller is presently opening the descriptor.

Controller X will receive the response indicated by the crossing cell corresponding to the subfunction of the OPEN DESCRIPTOR control command that it sends.

Table 9.3 – Descriptor access rule summary

OPEN DESCRIPTOR control command from controller X	Current state of the descriptor Z (Value of the status field in the response of OPEN DESCRIPTOR status command) (Currently opened by X, Y, or none)								
	CLOSE 00	READ OPEN (accept more read-only req.) 01			READ OPEN (reject more read-only req.) 11			WRITE OPEN 33	
	none	X	X, Y	Y	X	X, Y	Y	X	Y
OPEN DESCRIPTOR (CLOSE)	A	A	A	A	A	A	A	A	A
OPEN DESCRIPTOR (READ OPEN)	A	A	A	A	A	A	R	R	R
OPEN DESCRIPTOR (WRITE OPEN)	A	A	A	A	A	A	A	A	R

9.2.2.5 Reset rule

1) **Closed after reset:** After a power reset or serial bus reset (see reference [R4] for details), the descriptors and info blocks of all (sub)units shall be in a closed state.

9.2.2.6 Time-out rules

1) **Timeout between OPEN and READ/WRITE:** When the descriptor or info block has been opened for read/write access, a time out period should be measured between the (sub)unit's response to the initial OPEN DESCRIPTOR/INFO BLOCK control command (read/write subfunction) and the first ensuing READ DESCRIPTOR/INFO BLOCK control command or WRITE DESCRIPTOR/INFO BLOCK control command issued by the controller. If the READ DESCRIPTOR/INFO BLOCK control command or WRITE DESCRIPTOR/INFO BLOCK control command is not issued before the time out period, then the descriptor should be closed by

the (sub)unit for read/write access. The time-out period is recommended to be longer than one minute.

2) **Timeout between consecutive READs and WRITEs:** The (sub)unit should also measure the time out between its response to a READ DESCRIPTOR/INFO BLOCK control command or WRITE DESCRIPTOR/INFO BLOCK control command, and the subsequent READ DESCRIPTOR/INFO BLOCK control command or WRITE DESCRIPTOR/INFO BLOCK control command issued from the controller. If the controller fails to either close the descriptor or issue another READ DESCRIPTOR/INFO BLOCK control or WRITE DESCRIPTOR/INFO BLOCK control command within the time out period, then the (sub)unit should close the descriptor or info block for read/write access. The descriptor is then available to be opened again, by any controller. The time-out period is recommended to be longer than one minute.

NOTE — These time out measurements are measured per controller; when measuring the time between a response and a subsequent command, it is important to remain consistent about which controller interaction is being measured.

9.2.3 Unit/subunit requirements

1) **keep node_ID of openers:** The unit/subunit shall maintain the *node_ID* about each controller that opens a descriptor or info block.

2) **verify node_ID controllers:** When a descriptor is open, the unit/subunit shall check the *node_ID* of all received descriptor control commands to ensure that the controller has rights to the descriptor.

3) **Update length and "number_of…" :** The unit/subunit shall update the fields shown in the table below as data is added or removed from them. Other fields of descriptors may be updated by the target or by the controller according to the unit or the subunit-type specification. In case the target updates these fields reflecting the result of CREATE DESCRIPTOR, WRITE DESCRIPTOR or WRITE INFO BLOCK control command, the target shall not close the descriptor or the info block.

Table 9.4 – Fields maintained by target

	Field Name in descriptor and info block structure
(sub)unit descriptor	(sub)unit_identifier_descriptor_length number_of_root_lists unit_information_length or subunit_type_dependent_information_length general_unit_info_length[1] manufacturer_dependent_information_length
list descriptor	list_descriptor_length number_of_entry_descriptors list_specific_information_length non_info_block_length[2]
entry descriptor	entry_descriptor_length entry_specific_information_length non_info_block_length[2]
info block	Compound_length[2] primary_field_length[2]

[1] Unit descriptor

[2] These fields are written by a controller using WRITE DESCRIPTOR control commands. See clause 9.2.2.2 about the access rules.

9.2.4 Legacy device behavior

The rules below were created to define the behavior of legacy devices when they receive descriptor commands from devices using a future version of this document as a baseline.

Table 9.5 – Rules for reserved fields

row	Situation	Rule
1	A new controller writes data other than “0” to a descriptor field that is reserved in a legacy target.	The legacy target should return REJECTED.
2	A legacy controller reads a descriptor in a new target that contains data that was previously reserved.	The legacy controller shall ignore the new data.

NOTE — Any extensions using reserved fields to affect the meanings of other fields shall not be made.

Table 9.6 – Rules for reserved values

row	Situation	Rule
1	New controller writes previously reserved values to a descriptor in a legacy target.	The legacy target should return an AV/C response frame of REJECTED.
2	A legacy controller reads a previously reserved field value in a descriptor of a new target.	This is implementation dependent. It shall not be considered as an error.

9.3 CREATE DESCRIPTOR command

The CREATE DESCRIPTOR command is a unit/subunit command and manages the creation of descriptor structures. This command supports four ways to create descriptor:

1) Create a new root list from the subunit identifier descriptor.
2) Create a new child list from an existing parent entry.
3) Create a new entry in a list (See footnote[2]).
4) Create a new entry in a list and a child list from that entry.

All entry descriptors are created by *entry_type*, and all list descriptors are created by *list_type*. When an entry is created, whether it has an object ID is determined by its list's *has_object_ID* attribute. When a list is created, the subunit is responsible for assigning it a unique list ID and whether its entries contain *object_IDs* is *list_type* dependent. Refer to the subunit-type specific documentation for how particular types of lists and entries are created.

The CREATE DESCRIPTOR command supports descriptor specifiers 00_{16}, 11_{16}, 20_{16}, and 22_{16}. See section 8.2 and the sections below for the combination of descriptor specifiers in detail.

9.3.1 CREATE DESCRIPTOR control command

The CREATE DESCRIPTOR control command causes the subunit to create a specified type of descriptor structure in a specified location. The control command has the following format:

	length	ck	msb							lsb
opcode	1	√	CREATE DESCRIPTOR ($0C_{16}$)							
operand[0]	1	√	Result							
operand[1]	1	√	subfunction_1							
operand[2]	1	√	Reserved							
operand[3]	see[1]	see[2]	subfunction_1_specification							
:										
:										

[1] The size of this field depends on the *subfuntion_1* value, and is described below.
[2] Refer to the figures of each subfunction_1_specification field below.

Figure 9.3 – CREATE DESCRIPTOR control command frame

9.3.1.1 Field definitions

result: The *result* field shall be set to FF_{16} in the command frame by the controller. In the response frame, the subunit shall update the field with a result code according to Table 9.8 and Table 9.9 on page 69.

subfunction_1: The *subfunction_1* field specifies the descriptor to create:

[2] In legacy devices, the WRITE DESCRIPTOR control command, subfunction = insert or partial replace commands may have been used to achieve this same functionality.

Table 9.7 – Subfunction_1 field in command frame

subfunction_1	Meaning
00_{16}	Create a new root list, child list, or entry without a child list
01_{16}	Create a new entry *and* its child list
all other values	Reserved for future specification

To discern the kind of descriptor to create when *subfunction_1* = 00_{16}, the subunit shall inspect the *subfunction_1_specification* fields for the *descriptor_specifier* type combination used.

reserved: The reserved field shall be treated according to the rules defined in reference [R9].

subfunction_1_specification: The *subfunction_1_specification* fields have formats that depend on the *subfunction_1* value.

For *subfunction_1* = 00_{16}, the *subfunction_1_specification* field has the following format:

	length	ck	msb							lsb
operand[3]	see[1]	√	descriptor_specifier_where							
:		see[2]								
:										
:	see[1]	√	descriptor_specifier_what							
:		see[2]								
:										

[1] The length of this field depends on the length of the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.4 – Subfunction_1_specification for subfunction_1 = 00_{16}

descriptor_specifier_where: The *descriptor_specifier_where* field is a standard *descriptor_specifier* structure, as defined in section 8.1 "Descriptor specifier" on page 50, and defines from where the new descriptor is created.

descriptor_specifier_what: The *descriptor_specifier_what* field is also a standard *descriptor_specifier* structure, and it specifies the *entry_type* or *list_type* to be created.

For *subfunction_1* = 01_{16}, the *subfunction_1_specification* field has the following format:

	length	ck	msb							lsb
operand[3]	see[1]	√	descriptor_specifier_where							
:		see[2]								
:										
:	see[1]	√	descriptor_specifier_what_1							
:		see[2]								
:										
:	see[1]	√	descriptor_specifier_what_2							
:		see[2]								
:										

[1] The length of this field depends on the length of the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.5 – Subfunction_1_specification for subfunction_1 = 01_{16}

descriptor_specifier_where: The *descriptor_specifier_where* field is a standard *descriptor_specifier* structure, specifying from where the subunit creates the new descriptors.

descriptor_specifier_what_1: The *descriptor_specifier_what_1* field is also a standard *descriptor_specifier* structure, and it specifies the *entry_type* to be created.

descriptor_specifier_what_2: The *descriptor_specifier_what_2* field is also a standard *descriptor_specifier* structure, and it specifies the *list_type* of the child list to be created.

NOTE — When *subfunction_1* = 01_{16}, the subunit shall return an ACCEPTED response to the command only when both descriptors are created. If there is a failure creating both descriptors, then the subunit shall return a REJECTED response to the command, and the descriptors shall remain in their original state.

9.3.1.2 CREATE DESCRIPTOR control command responses

The following table shows the returned field values in REJECTED, INTERIM, and ACCEPTED response frames when *subfunction_1* = 00_{16} in the command frame.

Table 9.8 – Field values in the CREATE DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames for subfunction_1 = 00_{16}

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
result	FF_{16}	FF_{16}	FF_{16}	00_{16}
subfunction_1	00_{16}	←	←	←
descriptor_specifier _where	Descriptor specifier of the UID, SID or an entry	←	←	←[1]
descriptor_specifier _what	Descriptor specifier of the new list or entry	←	←	←[2]

[1] When appending an entry using FF, the response frame shall return the resultant object position.

[2] When creating a list, the accepted response frame shall return the list ID using *descriptor_specifier* 10_{16}.

← means "same as the command frame"

The following table shows the returned field values in REJECTED, INTERIM, and ACCEPTED response frames when *subfunction_1* = 01_{16} in the command frame.

Table 9.9 – Field values in the CREATE DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames for subfunction_1 = 01_{16}

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
result	FF_{16}	FF_{16}	FF_{16}	00_{16}
subfunction_1	01_{16}	←	←	←
descriptor_specifier _where	Descriptor specifier of an entry	←	←	←
descriptor_specifier _what_1	Descriptor specifier of the new entry	←	←	←[1]
descriptor_specifier _what_2	Descriptor specifier of the new list	←	←	←[2]

[1] When appending an entry using FF, the response frame shall return the resultant object position using *descriptor_specifier* 20_{16}.
[2] The accepted response frame shall return the list ID using *descriptor_specifier* 10_{16}.
← means "same as the command frame"

9.3.1.3 Creating descriptors

When a descriptor is created using the CREATE DESCRIPTOR control command, the structure of the newly created descriptor depends on the *list_types* and *entry_types* that are created, which are given in each subunit-type specification. As new entries and lists are created, the *entry_type* and *list_type* shall determine the info blocks and other data that are present in the new descriptors. For details, please refer to the subunit-type specification.

9.3.1.3.1 Creating a root list

A subunit may or may not support creating a root list. To determine this, use the CREATE DESCRIPTOR specific inquiry command with *subfunction_1* = 00_{16} and the descriptor specifiers shown in Table 9.10 below.

To create a new root list from an external controller, issue CREATE DESCRIPTOR control command, *subfunction_1* = 00_{16} using the descriptor specifier types shown in Table 9.10 below. When creating a root list, all other descriptor specifier type combinations are invalid.

Table 9.10 – Descriptor specifier types for creating a new root list

	descriptor_specifier_where	descriptor_specifier_what
descriptor_specifier_type	00_{16} (Subunit identifier descriptor)	11_{16} (List descriptor specified by list type)

The controller shall write-open the subunit identifier descriptor prior to issuing the CREATE DESCRIPTOR control command for the new root list descriptor and the subunit shall be responsible for updating the UID or SID while it is open. The subunit shall append the new list ID to the end of the root list IDs in the UID or SID. The controller can then read this new list ID to access the new descriptor. After the UID or SID closes, the subunit may reorder the *root_list_ID* entries as appropriate, such as to group lists IDs of the same list type.

The new root list descriptor shall be created as follows:

Address	length, bytes	Contents
00 00_{16}	2	list_descriptor_length = 06_{16} + list_specific_information_length + bytes for default entries
00 01_{16}		
00 02_{16}	1	list_type = type in descriptor_specifier_what
00 03_{16}	1	attributes = 000x 1000
00 04_{16}	2	list_specific_information_length = I
00 05_{16}		
00 06_{16}	i	default list_specific_information
:		
:	2	number_of_entry_descriptors = j
:		
:	–	default entry_descriptor[0]
:	–	:
:	–	default entry_descriptor[j-1]

Figure 9.6 – A new root list created by the CREATE DESCRIPTOR control command

After the CREATE DESCRIPTOR control command, the new list descriptor shall be left in the write-open state.

The value of “x” in the *attributes* field (*has_object_ID*) depends on the *list_type* definition. Also note that the *attributes* field may be extended, depending on the *generation_ID* of the target.

A subunit may not create default *list_specific_information*, in which case, the *list_specific_information_length* field shall be 00_{16} and the default *list_specific_information* field shall not exist.

After the CREATE DESCRIPTOR control command, the new root list descriptor and the subunit identifier descriptor shall remain open.

A subunit-type specification may create one or more default entries and info blocks in the new list.

9.3.1.3.2 Creating a child list from an existing entry

A subunit may or may not support creating a child list from an existing entry. To determine this, use CREATE DESCRIPTOR specific inquiry command with *subfunction_1* = 00_{16} and the descriptor specifiers shown in Table 9.11 below.

A subunit can create a new child list only from an entry that doesn’t already have a child list. To create a new child list, issue CREATE DESCRIPTOR control command with *subfunction_1* = 00_{16} and the descriptor specifier types shown in Table 9.11 below. All other descriptor specifier type combinations for creating a child list are invalid.

Table 9.11 – Specifier types for creating a child list from and existing entry

	descriptor_specifier_where	descriptor_specifier_what
descriptor_specifier_type	20_{16} (Entry descriptor specified by position)	11_{16} (List descriptor specified by list type)

If the descriptor specified by *descriptor_specifier_where* does not exist, or if it is already open by another controller, or if it already contains a *child_list_ID*, the command shall be REJECTED.

The controller shall open the parent entry's list descriptor for write prior to creating the new child list descriptor. The subunit is responsible for updating the parent entry and its list while they are open. The controller can then read the *child_list_ID* data to determine the list ID of the new list.

The structure of a new child list is identical to that of the new root list shown in Figure 9.6 – A new root list created by the CREATE DESCRIPTOR control command.

Whether the child list's entries will contain *object_IDs* is determined by the *list_type* of the created list.

A subunit may not create default *list_specific_information*, in which case, the *list_specific_information_length* field shall be 00_{16} and the *list_specific_information* field shall not exist.

After the CREATE DESCRIPTOR control command, the parent entry's list and the new child list shall remain open.

9.3.1.3.3 Creating an entry in a list

A subunit may or may not support creating an entry in an existing list. To determine this, use the CREATE DESCRIPTOR specific inquery command with *subfunction_1* = 00_{16} and the descriptor specifiers shown in Table 9.12 below.

To create a new entry in a list, the controller must first open the entry's list descriptor using the OPEN DESCRIPTOR control command with subfunction = 03_{16} (write). It then issues CREATE DESCRIPTOR control command with *subfunction_1* = 00_{16} and the descriptor specifier types shown in Table 9.12 below. All other descriptor specifier type combinations for creating an entry are invalid.

Table 9.12 – Descriptor specifier types for creating an entry descriptor in an existing list

	descriptor_specifier_where	descriptor_specifier_what
descriptor_specifier_type	20_{16} (Entry descriptor specified by position in a list)	22_{16} (Entry descriptor specified by entry type)

If the list of the descriptor specified by *descriptor_specifier_where* is already opened for write by another controller, the command shall be REJECTED.

When an entry is newly created by the CREATE DESCRIPTOR control command, the subunit inserts the entry at the position specified by *descriptor_specifier_where* of the list. As the insertion is made, the entries following the specified position (if any) are automatically shifted one by one. If no entries exist in the list, specify all 00_{16} or all FF_{16} for the position. To append an entry to the end of a list, specify the value in the *number_of_entry_descriptors* field or all FF_{16} for the position. If a position is specified that is greater than the number of entries, then the command shall be rejected. The subunit is responsible for incrementing the *number_of_entry_descriptors* field of the list descriptor and its overall length in order to accommodate the new entry.

The new entry descriptor shall be created as follows:

Address	Length, bytes	Contents
00 00_{16}	2	entry_descriptor_length = 04_{16} +
00 01_{16}		size_of_object_ID[1] + entry_specific_information_length
00 02_{16}	1	entry_type = type in descriptor_specifier_what
00 03_{16}	1	attributes = “0000 1000”
:	see[2]	object_ID (optional) = all 00_{16}
:		
:		
:		
:	2	entry_specific_information_length = i
:		
:	i	default entry_specific_information

[1] This value is included only when the list contains entries with object IDs
[2] The length of this field is determined by the *size_of_object_ID* field in the (sub)unit identifier descriptor.

Figure 9.7 – A new entry without a child list created by CREATE DESCRIPTOR control command

The *size_of_object_ID* that is found in the subunit identifier descriptor determines the length of the *object_ID* field. Whether the entry contains an *object_ID* is determined by its list’s *has_object_ID* attribute. The *has_child_ID* attribute shall be 0.

A unit or subunit may not create default *entry_specific_information*, in which case, the *entry_specific_information_length* field shall be 00_{16} and the *entry_specific_information* field shall not exist.

After the CREATE DESCRIPTOR control command, the new entry’s list descriptor shall remain open and the controller may want to issue some WRITE DESCRIPTOR commands on the new entry to complete the creation process.

9.3.1.3.4 Creating a new entry and its child list

A subunit may or may not support creating an entry and its child list from an existing list. To determine this, use CREATE DESCRIPTOR specific inquiry command with *subfunction_1* = 01_{16} and the descriptor specifiers shown in Table 9.13 below.

To create a new entry and its child list, the controller must first open the list where the entry will be added using the OPEN DESCRIPTOR control command with *subfunction* = 03_{16} (write). The controller shall then issues CREATE DESCRIPTOR control command with *subfunction_1* = 01_{16} with the specifier types shown in Table 9.13 below. All other specifier type combinations for creating a new entry and its child list are invalid.

Table 9.13 – Descriptor specifier types for creating a new entry and its child list

	descriptor_specifier_where	descriptor_specifier_what_1	descriptor_specifier_what_2
descriptor_specifier_type	20_{16} (Entry descriptor specified by position in a list)	22_{16} (Entry descriptor specified by entry type)	11_{16} (List descriptor specified by list type)

If the list of the descriptor specified by *descriptor_specifier_where* is already opened for write by another controller, the command shall be rejected.

When an entry is newly created by the CREATE DESCRIPTOR control command, the subunit inserts the entry at the position specified by *descriptor_specifier_where* of the list. As the insertion is made, the entries following the specified position (if any) are automatically shifted one by one. If no entries exist in the list, specify all 00_{16} or all FF_{16} for the position. To append an entry to the end of the list, specify the value in the *number_of_entries* field or all FF_{16} for the position. If a position is specified that is greater than the number of entries, then the command shall be rejected.

The subunit shall be responsible for adding the new entry to the list. After creating the new entry, the controller can then read the new entry by object position to determine the new *child_list_ID*.

The new entry descriptor with a child list shall be created as follows:

Address	Length, bytes	Contents
00 00_{16}	2	entry_descriptor_length = 04_{16} + size_of_object_ID
00 01_{16}		+ size_of_list_ID + entry_specific_information_length
00 02_{16}	1	entry_type = type in descriptor_specifier_what_1
00 03_{16}	1	attributes = "0010 1000"
:	see[1]	child_list_ID =
:		
:		Unique value assigned by subunit
:	see[2]	object_ID (optional) = all 0_{16}
:		
:		
:		
:	2	entry_specific_information_length = i
:		
:	i	default entry_specific_information

[1] The length of this field is determined by the *size_of_list_ID* field in the (sub)unit identifier descriptor.
[2] The length of this field is determined by the *size_of_object_ID* field in the (sub)unit identifier descriptor.

Figure 9.8 – A new entry with a child list created by CREATE DESCRIPTOR control command

Whether the new entry contains an *object_ID* is determined by its list's *has_object_ID* attribute. The *has_child_ID* attribute shall be set to 1.

A unit or subunit might not create default *entry_specific_information*, in which case, the *entry_specific_information_length* field shall be 00_{16} and the *entry_specific_information* field shall not exist.

The new child list descriptor shall be created as follows:

Address	length	Contents
00 00_{16}	2	list_descriptor_length = 06_{16} +
00 01_{16}		list_specific_information_length + bytes for default entries
00 02_{16}	1	list_type = type in descriptor_specifier_what_2
00 03_{16}	1	attributes = "000x 1000"
00 04_{16}	2	list_specific_information_length = I
00 05_{16}		
00 06_{16}	i	default list_specific_information
:	2	number_of_entry_descriptors = j
:		
	see[1]	default entry_descriptor[0]
		…
	see[1]	default entry_descriptor[j-1]

[1] The size of these fields = 2 + *entry_descriptor_length*

Figure 9.9 – A new list created by CREATE DESCRIPTOR control command

The *has_object_ID* value in the newly created list is determined by its *list_type*.

A unit or subunit might not create default *list_specific_information*, in which case, the *list_specific_information_length* field shall be 00_{16} and the *list_specific_information* field shall not exist.

Depending on the list type, the new list may be created with default entries.

After the CREATE DESCRIPTOR control command, both descriptors shall remain open. No entry descriptors shall be created in the new list by the (sub)unit. While both descriptors remain open, the controller may want to write to them before closing them.

9.4 OPEN DESCRIPTOR command

The OPEN DESCRIPTOR command is a unit/subunit command and is used to manage the access of descriptors in target devices.

The OPEN DESCRIPTOR command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.4.1 OPEN DESCRIPTOR control command

The OPEN DESCRIPTOR control command is used to gain read-only or write-enabled access to a descriptor within a target. It is also used to relinquish that access. The format of the OPEN DESCRIPTOR control command is shown by the figure below:

	bytes	ck	msb							lsb
opcode	1	√	OPEN DESCRIPTOR (08_{16})							
Operand[0]										
Operand[1]	see[1]	√	descriptor_specifier							
:		see[2]								
:	1	√	subfunction							
:	1	√	reserved							

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.10 – OPEN DESCRIPTOR control command frame

9.4.1.1 Field definitions

descriptor_specifier: The *descriptor_specifier* operand in the command frame specifies which, among possibly many, of the structures that the controller wants to access. The response frame contains the same information as the command frame. For more information about the descriptor specifier, see section 8.1 "Descriptor specifier" on page 50.

NOTE — If a list_type descriptor_specifier is supported and used to open a list, and if there are multiple lists of the same type, then an arbitrary list will be opened.

subfunction: The *subfunction* operand determines the operation performed by the target, as defined by the table below:

Table 9.14 – Values of the subfunction operand

Subfunction	Action
00_{16}	**Close:** Relinquish use of the descriptor.
01_{16}	**Read open:** Open the descriptor for read-only access.
03_{16}	**Write open:** Open the descriptor for read or write access.
All others	Reserved

reserved: The reserved field shall be treated according to the rules defined in reference [R9].

9.4.1.2 OPEN DESCRIPTOR control command responses

The following table shows the REJECTED, INTERIM, and ACCEPTED responses to the OPEN DESCRIPTOR control command.

Table 9.15 – Field values in the OPEN DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
descriptor_specifier	The descriptor specifier of the descriptor to open or close	←	←	←
subfunction	see Table 9.14	←	←	←
reserved	00_{16}	←	←	←

← means “same as the command frame”

If the OPEN DESCRIPTOR control command returns a REJECTED response, use the OPEN DESCRIPTOR status command to determine why.

9.4.2 OPEN DESCRIPTOR status command

OPEN DESCRIPTOR may also be used as a STATUS command to inquire about the current open condition of the descriptor. It can be issued on an open or closed descriptor.

The format of the OPEN DESCRIPTOR status command is shown below:

	bytes	ck	msb							lsb
opcode	1	√	OPEN DESCRIPTOR (08_{16})							
Operand[0]	see[1]	√ see[2]	descriptor_specifier							
Operand[1]										
:										
:	1	√	status = FF_{16}							
:	1	√	reserved							
:	2	√	node_ID = FF FF_{16}							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.11 – OPEN DESCRIPTOR status command frame

9.4.2.1 Field definitions

descriptor_specifier: The *descriptor_specifier* operand specifies which, among possibly many, of the structures that the controller wants to access. For more information about the descriptor specifier, see section 8.1 “Descriptor specifier” on page 50.

status: The *status* field is set to FF_{16} in the command frame. In the STABLE response frame, the *status* field indicates the current status of the descriptor. See Table 9.17 on page 79 for more information. In the IN TRANSITION response frame, the *status* field indicates the target’s expected status of the open operation.

reserved: The reserved field shall be treated according to the rules defined in reference [R9].

node_ID: The *node_ID* is set to FF FF_{16} in the command frame. In the STABLE response frame with *status* = 33_{16}, the unit or subunit shall update the *node_ID* operand. See Table 9.17 on page 79 for more information.

9.4.2.2 OPEN DESCRIPTOR status command responses

The following table shows the OPEN DESCRIPTOR status response frame field values for the STABLE and REJECTED responses.

Table 9.16 – Field values in the OPEN DESCRIPTOR status command: REJECTED, IN TRANSITION and STABLE response frames

Fields	Command	Response		
		REJECTED	IN TRANSITION	STABLE
descriptor_specifier	The descriptor specifier of the descriptor to check open status	←	←	←
status	FF_{16}	←	see Table 9.17	see Table 9.17
reserved	00_{16}	←	←	←
node_ID	FF_{16}	←	see Table 9.18	see Table 9.18

← means "same as the command frame"

The following table shows the values for *status* and their meaning.

Table 9.17 – status field values in the OPEN DESCRIPTOR status command: STABLE response frame

Status	Meaning
00_{16}	**Ready for open:** The descriptor specified by *descriptor_specifier* is closed and can be accessed. No controllers currently have access (either read or write).
01_{16}	**Read opened:** The descriptor or its list is open for read-only access to its data by one or more controllers, and is able to accept additional read-only open requests.
04_{16}	**Non Existent:** No descriptor specified by *descriptor_specifier* exists.
05_{16}	**List-only (deprecated):** Access control for individual entries in a list is not supported - the controller must obtain access to the entire list. In this case, a NOT IMPLEMENTED response will be returned, therefore, this value shall not be used.
11_{16}	**At capacity:** The descriptor is open for read-only access to the data, and is unable to accept any additional read-only open requests.
33_{16}	**Write opened:** The descriptor or its list is open for read or write access to the data. No access (for read-only or for read or write) by other controllers is allowed.

The following table describes the value for *node_ID* in the response frame.

Table 9.18 – node_ID values in the response frame

Status	Value of node_ID
33_{16}	**Controller with write access:** When the descriptor is open for write access by a controller, the unit or subunit shall update this field to contain the node ID of the controller.
all other values	The value of this field is not changed in the response frame; it remains as FF FF_{16}.

If for some reason the target cannot return any of the status responses in the STABLE response frame, then the target shall return a REJECTED response. Under this condition, the descriptor is not presently accessible.

9.4.3 OPEN DESCRIPTOR notify command

The OPEN DESCRIPTOR command may also be used as a NOTIFY command when controllers wish to be advised of a possible change of status of the descriptor. A controller may want to know several things about the status of a descriptor:

1) If the controller is waiting for a descriptor to be available for opening, it may want to know when the descriptor is free to access. The controller then should determine if the data it cares about has changed.
2) If the controller is waiting for a descriptor to be available for opening, it may want to know if another controller deleted the descriptor.

The format of the NOTIFY command is the same as STATUS command, but with a ctype value of NOTIFY.

	bytes	ck	msb							lsb
opcode	1	√	OPEN DESCRIPTOR (08_{16})							
operand[0]	see[1]	√ see[2]	descriptor_specifier							
operand[1]										
:										
:	1	√	status = FF_{16}							
:	1	√	Reserved							
:	2	√	node_ID = FF FF_{16}							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.12 – OPEN DESCRIPTOR notify command frame

9.4.3.1 Field definitions

descriptor_specifier: In the command frame, the *descriptor_specifier* operand specifies which, among possibly many, of the structures that the controller wants access notification. For more information about the descriptor specifier, see section 8.1 “Descriptor specifier” on page 50.

status: In the command frame, *status* is set to FF_{16}. In the INTERIM response frame, *status* contains the present access state of the descriptor specified by the *descriptor_specifier* operand. If the response is CHANGED, *status* contains the new access state of the descriptor specified by the *descriptor_specifier* operand.

node_ID: The controller shall set this field to FF FF_{16} in the command frame, and the target shall update it with the node ID in the INTERIM and CHANGED response frames when *status* = 33_{16}.

9.4.3.2 OPEN DESCRIPTOR notify command responses

The following table shows the OPEN DESCRIPTOR notify response frame field values for the REJECTED, INTERIM, and CHANGED responses.

Table 9.19 – Field values in the OPEN DESCRIPTOR notify command: REJECTED, INTERIM and CHANGED response frames

Fields	Command	Response		
		REJECTED	INTERIM	CHANGED
Descriptor_specifier	The descriptor specifier of the descriptor to open or close	←	←	←
status	FF_{16}	←	see Table 9.17	see Table 9.17
reserved	00_{16}	←	←	←
node_ID	$FFFF_{16}$	←	see Table 9.18	see Table 9.18

← means "same as the command frame"

9.5 READ DESCRIPTOR command

The READ DESCRIPTOR command is a unit/subunit command and is used to manage the retrieval of data from a descriptor.

The READ DESCRIPTOR command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.5.1 READ DESCRIPTOR control command

The READ DESCRIPTOR control command is used to read the data from the descriptor specified by the *descriptor_specifier*. Before a descriptor can be read, it must be opened using the OPEN DESCRIPTOR control command. Refer to section 9.4 "OPEN DESCRIPTOR command" on page 76 for more information.

The format of the READ DESCRIPTOR control command is shown by the figure below:

	bytes	ck	msb							lsb
opcode	1	√	READ DESCRIPTOR (09_{16})							
operand[0]	see[1]	√ see[2]	descriptor_specifier							
operand[1]										
:										
:	1	√	read_result_status							
:	1	√	Reserved							
:	2	–	data_length							
:										
:	2	–	Address							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.13 – READ DESCRIPTOR control command frame

9.5.1.1 Field definitions

descriptor_specifier: The *descriptor_specifier* is specified in the command frame and describes which, among possibly many, of the structures that the controller wants to read. For more information about the descriptor specifier, see section 8.1 "Descriptor specifier" on page 50.

If a list descriptor is specified in the *descriptor_specifier* in the OPEN DESCRIPTOR control command, the list or any of its entries may be specified in the *descriptor_specifier* of the subsequent READ DESCRIPTOR control commands. Otherwise, the same descriptor specifier should be used in the READ DESCRIPTOR control command as was used in the OPEN DESCRIPTOR control command.

read_result_status: The *read_result_status* shall be set to FF_{16} by the controller. The ACCEPTED response frame contains values indicating the status of the read. See Table 9.21 on page 85 for more information. For details on what this field means, please refer also to the meaning of the *data_length* field below.

reserved: The reserved field shall be treated according to the rules defined in reference [R9].

data_length: The *data_length* operand specifies the number of bytes to be read from the target. There is a special case when *data_length* = 00_{16} in the command frame, which means that the address field shall be ignored and the entire descriptor is specified to be read. In the ACCEPTED response frame, this field will be updated to contain the actual number of bytes that were read.

IMPORTANT: If a target receives a read request with a *data_length* that would result in reading past the end of the data boundary indicated by the *descriptor_specifier* operand, then it shall return only the legitimate data, a *read_result_status* of 12_{16}, and a *data_length* to reflect the size of this data.

address: The *address* field specifies the address offset of the starting point to be read. It is an offset from the beginning of the descriptor specified by the *descriptor_specifier*. When *data_length* = 00_{16}, then the *address* field shall be ignored.

9.5.1.2 READ DESCRIPTOR command responses

When the READ DESCRIPTOR command succeeds, the descriptor data is inserted after the address field with a length of *data_length*. Command and response fields are summarized in the table below:

Table 9.20 – Field values in the READ DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
descriptor_specifier	The descriptor specifier of the descriptor to read	←	←	←
read_result_status	FF_{16}	←	←	see Table 9.21
reserved	00_{16}	←	←	←
data_length	XX_{16} (specific amt) or 00_{16} (all)	←	←	YY_{16} (actual read amt)
address	address offset	←	←	←
data	none	none	none	Inserted descriptor data

← means "same as the command frame".

Table 9.21 – read_result_status field values in the ACCEPTED response frame

read_result_status	Meaning
10_{16}	**Complete read:** The entire data specified by the *descriptor_specifier*, *data_length*, and *address* fields in the command frame was returned.
11_{16}	**More to read:** A portion of the data specified by the *descriptor_specifier*, *data_length*, and *address* fields in the command frame was returned. The *data_length* field in the response frame indicates exactly how much data was returned.
12_{16}	**Data length too large:** There is less data in the descriptor than was specified by *data_length* in the command frame. The response frame contains the actual *data_length* of the returned data, which includes the data at the end of the descriptor.

9.5.2 Reading the (sub)unit identifier descriptor example

To read the (sub)unit identifier descriptor, use the following command sequence:

1) Open the (sub)unit identifier descriptor for read: OPEN DESCRIPTOR, *descriptor_specifier* type = 00_{16}, *subfunction* = 01_{16}.

2) Read the (sub)unit identifier descriptor: READ DESCRIPTOR, *descriptor_specifier* type = 00_{16} or READ INFO BLOCK, *info_block_reference_path* … (see section 9.8, “READ INFO BLOCK command” on page 103)

3) Close the (sub)unit identifier descriptor: OPEN DESCRIPTOR, *descriptor_specifier* type = 00_{16}, *subfunction* = 00_{16}.

9.5.3 Reading a list or an entry example

To read a list or entry, use the following command sequence:

1) Open the list descriptor for read: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16} , *subfunction* = 01_{16}.

2) Read the descriptor: READ DESCRIPTOR, *descriptor_specifier* type = 10_{16} (for list), or 20_{16} or 21_{16} (for entry), or READ INFO BLOCK, *info_block_reference_path* … (see section 9.8, “READ INFO BLOCK command” on page 103).

3) Close the list: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16} (for list), 20_{16}, or 21_{16}, (for entry) *subfunction* = 00_{16}.

9.6 WRITE DESCRIPTOR command

The WRITE DESCRIPTOR command is a unit/subunit command and is used to support variable-length write operations on an open descriptor.

The WRITE DESCRIPTOR command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.6.1 WRITE DESCRIPTOR control command

The WRITE DESCRIPTOR control command is used to write variable-length data in the descriptor of the target. Before writing to a descriptor, the descriptor must be opened for write access using the OPEN DESCRIPTOR control command. For more information, see section 9.4 “OPEN DESCRIPTOR command” on page 76. A WRITE DESCRIPTOR control command is also used to create an info block and to write an info block. For access rules for writing descriptors and info blocks, see clause 9.2.2.2, “Access rules for writing descriptors and info blocks” on page 62.

After the write operations, it is recommended that the controller closes the descriptor as soon as possible to allow other controllers to access the descriptor.

The format of the WRITE DESCRIPTOR control command is shown by the figure below:

	bytes	ck	msb							lsb
opcode	1	√	WRITE DESCRIPTOR ($0A_{16}$)							
Operand[0]	see[1]	√ see[2]	descriptor_specifier							
Operand[1]										
:										
:	1	√	subfunction							
:	1	√	group_tag							
:	2	–	data_length = i							
:										
:	2	–	address							
:										
:										
:	i	–	data							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.14 – WRITE DESCRIPTOR control command frame

9.6.1.1 Field definitions

descriptor_specifier: The *descriptor_specifier* describes which descriptor is being written to. The exact format of this field will vary based on the type of descriptor specifier used.

For more information about the descriptor specifier, see section 8.1 “Descriptor specifier” on page 50.

If a list descriptor is specified in the *descriptor_specifier* in the OPEN DESCRIPTOR control command, the list or any of its entries may be specified in the *descriptor_specifier* of the WRITE DESCRIPTOR control command, since opening a list implies opening its entries. Otherwise, the same descriptor specifier should be used in the WRITE DESCRIPTOR command as was used in the OPEN DESCRIPTOR command.

The *descriptor_specifier* fields in the response frame shall contain the same information as in the command frame.

subfunction: The *subfunction* operand specifies the way information is being written by the controller. The following table describes the legal values for this operand and what those values mean:

Table 9.22 – Values for the subfunction operand

Subfunction	Action
10_{16}	**Change:** (not recommended, use **partial replace** instead) Overwrite the descriptor part indicated by the *address* and *data_length* operands. The controller is responsible for making sure that exactly the same number of bytes are being used to replace existing data. Uncontrolled writing beyond the end of a descriptor's data boundary is not permitted. If a target receives a write request with a *data_length* that would result in writing past the end of the data indicated by the *descriptor_specifier* operand, then it shall return a REJECTED response to the command. This applies to the list and individual entries within the list. For fields within an entry, the controller is responsible for making sure not to write beyond the field's or entry's boundary.
20_{16}	**Replace:** Overwrite a complete descriptor. The target is responsible for extending or shrinking descriptor storage to accommodate differences in size between this descriptor and the one being replaced. The *address* operand is ignored for this subfunction. Note – read and read/ignore fields can be written by the controller using this subfunction.
30_{16}	**Insert:** Insert an entry or subunit dependent descriptor in a list, inserting the new descriptor before the one specified by the *descriptor_specifier* operand. The *address* operand is ignored for this subfunction. This subfunction shall not be used for inserting list descriptors. To add the descriptor to the end of the list, specify *object_ID* = all FF_{16}, or *entry_position* = all FF_{16}, in the *descriptor_specifier*. Note – read and read/ignore fields can be written by the controller using this subfunction.
40_{16}	**Delete:** Delete the list, entry, or subunit dependent descriptor specified by *descriptor_specifier*. For this subfunction, *address* and *data_length* are ignored. The target is responsible for adjusting the size of descriptor storage to accommodate this operation. Note – A descriptor that includes read and read/ignore fields can be deleted using this subfunction. If the descriptor being deleted or any descriptors in the hierarchy under the descriptor being deleted are open for write during a delete operation, it is unit or subunit-type dependent whether the delete operation shall be REJECTED (with no descriptors deleted) or ACCEPTED (with all descriptors deleted). See clause 9.6.1.3, "Deleting list descriptors" on page 92 for more information. If a root list is deleted, the unit or subunit is responsible for updating the *number_of_root_lists* field and *(sub)unit_identifier_descriptor_length* field in the (sub)unit identifier descriptor. When a child list is deleted, the unit or subunit is responsible for updating the parent entry, which includes removing the *child_list_ID* field, setting the *has_child_ID* attribute to 0, updating the entry's length, and updating the entry's list's length. WARNING: If an ambiguous descriptor is specified by the *descriptor_specifier*, then an arbitrary descriptor will be deleted! Use of the UID or SID *descriptor_specifier* for this subfunction is prohibited.

Subfunction	Action
50_{16}	**Partial replace:** Replace a portion of the descriptor with data of a different or same size. This subfunction can also be used to partially delete or partially insert data into a descriptor. The target is responsible for adjusting the size of descriptor storage and to update any related fields as necessary to accommodate this operation. A controller shall not use the partial replace subfunction to delete general fields (length, size_of…, number_of…, descriptor type, attribute, object_ID, and child_list_ID fields) individually. Any attempt to do so shall be REJECTED. Partial replace is a special case. See clause 9.6.1.2, “Partial replace operations” on page 91 for the command frame and the definitions of the fields.
all others	reserved for future definition

The target shall modify the low nibble of the *subfunction* field in the response frame. See Table 9.26 on page 94 for more information on the subfunction response values. Refer also to the definitions for *data_length*, *address*, and *data* for more information about the information in this table.

NOTE — A write operation that results in a change of descriptor size indirectly affects length fields that appear prior to the memory area that changed. Figure A. on page 138 maps all the descriptor and info block length fields with areas of the descriptor. As indicated by this figure, a unit or subunit can determine which length fields, in descriptors and info blocks, are affected by write operations.

A subunit specification may place restrictions on the fields in the WRITE DESCRIPTOR command when using these subfunctions. For example, the combination of length and address may be limited.

group_tag: The *group_tag* operand is used for grouped update operations on a descriptor, in which several WRITE DESCRIPTOR control commands must be issued. The controller may use this field to specify an arbitrary number of update operations that must be performed on an “all or nothing” basis.

The *group_tag* is useful due to the limitation of the FCP frame, which is 512 bytes, whereas a descriptor could contain up to 65535 bytes. If the data to write to the descriptor is longer than 512 bytes (minus fields required prior to the *data* field), then the entire write operation must be grouped into multiple smaller write operations.

Furthermore, some targets impose a limit to the size of the write packet it can accept (being less than 512 bytes). It is recommended that the controller know the limitations of the amount of data a target can accept per write operation before issuing group update operations. If necessary, this can be determined by issuing a WRITE DESCRIPTOR status command.

The *group_tag* is also useful when writing to divided parts of a descriptor that must be committed to memory at the same time.

Whether the *group_tag* is supported or not is unit or subunit-type dependent, and can be determined by using the WRITE DESCRIPTOR specific inquiry command.

This field may take one of the following values:

Table 9.23 – Group_tag values

group_tag	Action
00_{16}	**Immediate:** Immediately write the data to the descriptor. Whether the data is committed after it is written or after it is closed is unit or subunit-type dependent.
01_{16}	**First:** Begin a grouped update sequence, with this command being the first of several which may follow. The target shall take the necessary actions to prepare for this sequence, and to ensure that the original descriptor structure is preserved, should the sequence not finish normally.
02_{16}	**Continue:** Any number of subsequent commands may be issued using this tag, after the “first” has been issued.
03_{16}	**Last:** This command signals the last of the grouped update sequence. When the target receives this *group_tag* value, it commits all of the data written in the sequence, including the data in this command, to the specified descriptor. If the target does NOT receive this command before a bus reset or the time out period used for descriptor access, then it shall discard ALL of the data written in this sequence and leave the specified descriptor unmodified.
all others	reserved for future definition

NOTE — The purpose of indivisible updates is to provide a safe mechanism for updating a descriptor structure in the distributed environment, where interruptions and partially-updated descriptors may have a fatal effect on the unit or subunit.

Supporting the *group_tag* is optional; if a unit or subunit receives a non-zero *group_tag*, it may return a response of NOT IMPLEMENTED. The controller will then have to fall back to using immediate operations. The controller shall verify that all write operations in a “group update” are ACCEPTED by the target. If any of the write operations are REJECTED, then the target will return a REJECTED response to all further group updates, and the controller should stop the “group update” operation.

data_length: The *data_length* operand in the command frame specifies the number of bytes in the *data* field to be written to the descriptor.

If *data_length* specifies more bytes than can be accepted by the target in a single operation then the target may return either an ACCEPTED response or a REJECTED response to the command. If the command is ACCEPTED and data was dropped, then this must be indicated to the controller in the response frame by updating the *data_length* field to the number of accepted bytes so that it can respond accordingly. If the command is REJECTED, then the controller should issue the WRITE DESCRIPTOR status command to determine the maximum length of data the target can accept.

If the combination of *data_length* and *address* (below) reference data out of bounds, then the target shall return a REJECTED response to the command.

address: The *address* field specifies the address of the starting point to be written. It is an offset from the beginning of the particular entity described by *descriptor_specifier*.

If the *address* references data out of bounds, then the target shall return a REJECTED response to the command.

data: The *data* field in the command frame contains the data bytes to be stored, the length of which is indicated by *data_length*. The target shall check the validity of the data written by the controller to the extent possible described by the unit or subunit-type specification, and if any data is invalid, the target shall return with a REJECTED response.

If the target returns ACCEPTED, INTERIM or REJECTED responses, then the frame shall NOT contain the data in the *data* field that was specified in the original command frame. This avoids possibly large data transfers in the case that such responses are returned.

9.6.1.2 Partial replace operations

The WRITE DESCRIPTOR control command has the following frame when the *partial_replace* subfunction is specified:

	bytes	ck	msb							lsb
opcode	1	√	WRITE DESCRIPTOR ($0A_{16}$)							
operand[0]	see[1]	√ see[2]	descriptor_specifier							
operand[1]										
:										
:	1	√	Subfunction “partial_replace” = 50_{16}							
:	1	√	group_tag							
:	2	–	Replacement_data_length = i							
:										
:	2	–	address							
:										
:	2	–	original_data_length							
:										
:	i	–	replacement_data							
:										
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descriptor_specifier_type should be evaluated.

Figure 9.15 – WRITE DESCRIPTOR control command frame, subfunction = partial_replace (50_{16})

The *descriptor_specifier*, *subfunction* and *group_tag* operands are as described above.

replacement_data_length: The *replacement_data_length* field specifies the number of bytes in the *replacement_data* operands.

address: The *address* operand specifies where the operation (replace, insert or delete) is to be performed. In the case of insert, it indicates where to begin inserting bytes. In the case of delete, it indicates where to begin deleting bytes.

If the address references invalid memory, the target shall return a REJECTED response to the command.

original_data_length: The *original_data_length* field specifies the number of bytes in the original descriptor data. All descriptor data specified by the *original_data_length* will be removed in the partial replace operation. If the combination of *address* and *original_data_length* reference invalid memory, the target shall return a REJECTED response to the command.

The *replacement_data_length* along with the *original_data_length* field can be specified for partial insert or partial delete operations:

1) **Partial Insert:** When the *original_data_length* operand = 0, the operation is a partial insert. In this case, the *replacement_data_length* operand shall be greater than 0, indicating the number of bytes to be inserted.

2) **Partial Delete:** When *replacement_data_length* = 0, the operation is a partial delete and the *replacement_data* operand does not exist. In this case, the *original_data_length* operand shall be greater than 0, indicating the number of bytes to be deleted.

The combination of *replacement_data_length* = 0 and *original_data_length* = 0 is illegal. If they are both zero, the target shall return a REJECTED response to the command.

replacement_data: *replacement_data* shall contain the data that replaces the original descriptor data.

The *partial_replace* subfunction shall not be used to delete any general descriptor fields in this specification, except fields in the *entry_specific_information*, *list_specific_information*, *(sub)unit_dependent_information* and *manufacturer_dependent_information* areas, whose internal formats are defined by their unit or subunit-type specification. Any attempt to do so shall be REJECTED.

9.6.1.3 Deleting list descriptors

List descriptors can be deleted with the *delete* subfunction.

There are two ways to delete list descriptors, which are unit or subunit-type dependent:

1) **Deleting a list only:** Open the list descriptor and list descriptor's parent entry's list (or the UID or SID) for write, then delete the child (or root) list descriptor. Under this circumstance, the list's parent entry (or UID or SID) shall be updated by the target.
2) **Deleting a list and its parent entry:** Open the list descriptor and list descriptor's parent entry's list for write, and delete the entry. The (sub)unit shall be responsible for updating the parent entry's list.

NOTE — When deleting a root list, use option 1) above.

The delete operation shall be atomic, that is, no other descriptor command shall be executed on the descriptor structure while the delete operation occurs.

Furthermore, a (sub)unit may support deleting only leaf lists or deleting lists and entries with child lists (cascade deleting).

9.6.1.3.1 Deleting leaf lists

If only delete leaf lists is supported, then only those descriptors without child lists can be deleted (those at the bottom of the hierarchy). A (sub)unit can support either one or both of the options above for deleting leaf lists. If the descriptors are already open for write by another controller(s), then the controller must wait until the descriptors can be accessed before deleting the list.

9.6.1.3.2 Cascade deleting lists

When a (sub)unit supports cascade delete, all descriptors in the hierarchy below the list descriptor being deleted are also deleted during the operation subject to certain restraints given below.

If cascade-delete is supported, and if any list descriptor in the hierarchy is open for write by another controller at the start of the delete operation, it shall be (sub)unit-type dependent whether the delete command shall be ACCEPTED or REJECTED. If ACCEPTED, then those open descriptors are force-closed and deleted. If REJECTED, then none of the descriptors (opened or closed) are deleted.

9.6.1.4 WRITE DESCRIPTOR control command responses

The following table summarizes the command and response field values for the WRITE DESCRIPTOR command when *subfunction* $\neq 50_{16}$.

Table 9.24 – Field values in the WRITE DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
descriptor_specifier	The descriptor specifier of the descriptor to write	←	←	←
subfunction	see Table 9.22	see Table 9.26	←	see Table 9.26
group_tag	see Table 9.23	←	←	←
data_length	size of data	←	←	number of accepted bytes
address	address of starting point from beginning of descriptor_specifier	←	←	←
data	Variable	none	none	none

← means "same as the command frame"

The following table summarizes the command and response field values for the WRITE DESCRIPTOR command when *subfunction* = 50_{16}.

Table 9.25 – Field values in the WRITE DESCRIPTOR control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
descriptor_specifier	The descriptor specifier of the descriptor to write	←	←	←
subfunction	50_{16}	see Table 9.26.	←	see Table 9.26
group_tag	see Table 9.23	←	←	←
replacement_ data_length	size of replacement data	←	←	number of accepted bytes
address	address of starting point from beginning of descriptor	←	←	←
original_data_length	size of original data	←	←	←
data	Variable	none	none	none

← means "same as the command frame"

The following table describes the *subfunction* field in the ACCEPTED and REJECTED response frames.

Table 9.26 – subfunction in the response frame

Subfunction in the response	Response	Meaning
x0$_{16}$	ACCEPTED	The specified subfunction was performed with no problem. R/I fields were not changed as specified in the command frame.
x1$_{16}$	ACCEPTED	This response applies only to the *partical_replace* subfunction. The original descriptor data, specified by *address* and *origial_data_length* fields, was deleted and the lower part of replacement data was inserted. R/I fields were not changed as specified in the command frame. The target shall return the *replacement_data_length* field to be equal to the size of the inserted part. The controller may need to issue more *partial_ replace_ operation(s)* to insert the remaining part.
x2$_{16}$	REJECTED	The *address* and *data_length* fields specify an invalid address, **or** the target cannot support the amount of data specified by *data_length,* so the write operation was not performed. This may also be returned if the target prevents controllers from writing any fields as specified in *data* field, or *descriptor_specifier* is invalid.
x3$_{16}$	ACCEPTED	The specified subfunction was performed but some R/W/I fields were not changed as specified in the command frame. This response does not apply to the *delete* subfunction.
x4$_{16}$	ACCEPTED	This response applies only to the *partical_replace* subfunction. The origial descriptor data, specified by *address* and *origial_data_length* fields, was deleted and the lower part of replacement data was inserted. Some R/W/I fields were not changed as specified in the command frame. The target shall return the *replacement_data_length* field to be equal to the size of the performed part. The controller may need to issue more *partial_ replace_ operation(s)* to insert the remaining part.

NOTE — If subfunction x3$_{16}$ or x4$_{16}$ is returned, then the controller may read the descriptor to determine which R/W/I fields were not changed. Note that the unit or subunit that is compliant to the previous versions of this specification does not return subfunction x3$_{16}$ or x4$_{16}$.

9.6.1.5 Modifying the (sub)unit identifier descriptor – example

To modify the (sub)unit identifier descriptor, use the following command sequence:

1) Open the (sub)unit identifier descriptor for write: OPEN DESCRIPTOR, *descriptor_specifier* type = 00$_{16}$, *subfunction* = 03$_{16}$.
2) Read the descriptor as necessary: READ DESCRIPTOR, *descriptor_specifier* type = 00$_{16}$.
3) Write to the descriptor: WRITE DESCRIPTOR, *descriptor_specifier* type = 00$_{16}$, *subfunction* = 50$_{16}$, or WRITE INFO BLOCK, see section 9.9, “WRITE INFO BLOCK command” on page 106.
4) Close the (sub)unit identifier descriptor: OPEN DESCRIPTOR, *descriptor_specifier* type = 00$_{16}$, *subfunction* = 00$_{16}$.

9.6.1.6 Modifying list or entry specific information, extended information or Object_ID – example

To modify the list specific information of an entry or list or the object_ID of an entry, use the following command sequence:

1) Open the list for write: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 03_{16}.
2) Read the descriptor as necessary: READ DESCRIPTOR, *descriptor_specifier* type = 10_{16}, 20_{16}, 21_{16}.
3) Write to the descriptor(s): WRITE DESCRIPTOR, *descriptor_specifier* type = 10_{16}, 20_{16}, 21_{16}, *subfunction* = 50_{16}.
4) Read info blocks as necessary: READ INFO BLOCK, … (see section 9.8, “READ INFO BLOCK command” on page 103)
5) Write to the info block(s) as necessary: WRITE INFO BLOCK, ...(see section 9.9, “WRITE INFO BLOCK command” on page 106)
6) Close the list: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 00_{16}.

9.6.1.7 Deleting a list descriptor – example

To delete a list descriptor, use the following command sequence:

1) Open the list for write: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 03_{16}.
2) Open the parent entry’s list for write: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 03_{16}.
3) Delete the list descriptor: WRITE DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 40_{16}.
4) Or, delete the parent entry descriptor: WRITE DESCRIPTOR, *descriptor_specifier* type = 20_{16}, *subfunction* = 40_{16}.
5) Close the parent entry’s list descriptor: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 00_{16}.

9.6.1.8 Deleting an entry descriptor without a child list – example

To delete an entry descriptor, which has no child list, use the following command sequence:

1) Open the entry’s list for write: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 03_{16}.
2) Delete the entry: WRITE DESCRIPTOR, *descriptor_specifier* type = 20_{16}, *subfunction* = 40_{16}.
3) Close the list: OPEN DESCRIPTOR, *descriptor_specifier* type = 10_{16}, *subfunction* = 00_{16}.

9.6.2 WRITE DESCRIPTOR status command

The WRITE DESCRIPTOR status command may be used to determine the (sub)unit's capability to accept data in its descriptor in a single operation. The command frame for the WRITE DESCRIPTOR status command is the same as the WRITE DESCRIPTOR control command frame, except that the *address* and *data* operands are not included. In this case, the format shown by the figure below is used:

	bytes	ck	msb							lsb
Opcode	1	√	WRITE DESCRIPTOR ($0A_{16}$)							
operand[0]	see[1]	√ see[2]	descriptor_specifier							
:										
:										
:	1	–	subfunction = FF_{16}							
:	1	–	group_tag = FF_{16}							
:	2	–	data_length = FF FF_{16}							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Only its descripor_specifier_type should be evaluated

Figure 9.16 – WRITE DESCRIPTOR status command frame

9.6.2.1 Field definitions

descriptor_specifier: The *descriptor_specifier* describes which descriptor structure to check for write. The exact format of this specifier will vary based on the kind of descriptor (list or entry), the method of specifying this descriptor, and on the type of unit or subunit for which the descriptor is defined. For more information about the descriptor specifier, see section 8.1 "Descriptor specifier" on page 50.

The *descriptor_specifier* does not change in the response frame.

The remaining 4 bytes shall be set to FF_{16} on input.

The WRITE DESCRIPTOR status command can be issued on a descriptor that is closed or already opened for write by another controller.

9.6.2.2 WRITE DESCRIPTOR status command responses

The response frame for the WRITE DESCRIPTOR status command is the same as the control command frame, except the variable sized *data* operand and the *address* operand are not included. The following table summarizes the control and response frames of the WRITE DESCRIPTOR status command:

Table 9.27 – Field values in the WRITE DESCRIPTOR status command: REJECTED, IN TRANSITION and STABLE response frames

Fields	Command	Response		
		REJECTED	IN TRANSITION	STABLE
descriptor_specifier	The descriptor specifier of the descriptor to inquire	←	←	←
subfunction	FF_{16}	←	←	←
group_tag	FF_{16}	←	←	00_{16}
data_length	$FF\ FF_{16}$	←	←	max number of accepted bytes

← means "same as the command frame"

The *data_length* field shall return the maximum number of bytes that may be written into the specified descriptor in a single WRITE DESCRIPTOR operation. If this operand is returned with a value of zero, then there is no write access possible for the descriptor because it can't be modified.

9.7 OPEN INFO BLOCK command (not recommended)

The OPEN INFO BLOCK command is a unit/subunit command and is used to manage the access of info blocks in target devices. This command is not recommended. Implementers should use OPEN DESCRIPTOR on a list (or the (sub)unit identifier descriptor) instead.

If a unit or subunit supports the OPEN INFO BLOCK command, it should also support the OPEN DESCRIPTOR command.

The OPEN INFO BLOCK command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, 30_{16}, 31_{16} and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.7.1 OPEN INFO BLOCK control command

The OPEN INFO BLOCK control command is used to gain read-only or write-enabled access to an info block including its *secondary_fields*. It is also used to relinquish that access. It is similar to the OPEN DESCRIPTOR command in terms of rules of operation and access. See section 9.2 ”Reading and writing AV/C descriptor structures” on page 59 for more information.

The OPEN INFO BLOCK control command has the following format:

	bytes	Ck	msb							lsb
Opcode	1	√	OPEN INFO BLOCK (05_{16})							
operand[0]	see[1]	√ See[2]	info_block_reference_path							
:										
:										
:	1	√	subfunction							
:	1	√	reserved							

[1] The size of this field depends on the info block reference path used.
[2] Its descriptor_specifier_types in descriptor_specifiers should be evaluated. The evaluation of other fields is unit or subunit-type dependent.

Figure 9.17 – OPEN INFO BLOCK control command frame

9.7.1.1 Field definitions

info_block_reference_path: The *info_block_reference_path* field provides a description of which info block is to be opened by this command. For more information about the info block reference path, see section 8.3 “Information block reference path” on 55.

subfunction: The *subfunction* field determines the operation performed by the target, as defined by the table below:

Table 9.28 – Values of the subfunction operand

subfunction	Action
00_{16}	**Close:** Relinquish use of the info block
01_{16}	**Read open:** Open the info block for read-only access
03_{16}	**Write open:** Open the info block for read or write access
others	Reserved

9.7.1.2 OPEN INFO BLOCK control command responses

The following table shows the REJECTED, INTERIM, and ACCEPTED responses to the OPEN INFO BLOCK control command.

Table 9.29 – Field values in the OPEN INFO BLOCK control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
Info_block_reference_path	The reference path of the info block to open or close	←	←	←
subfunction	See Table 9.28	←	←	←
reserved	00_{16}	←	←	←

← means “same as the command frame”

If the OPEN INFO BLOCK control command returns REJECTED, use the OPEN INFO BLOCK status command to determine why.

9.7.2 OPEN INFO BLOCK status command

OPEN INFO BLOCK may also be used as a STATUS command to inquire about the current open condition of the info block. It can be performed on an open or closed info block. The format of the OPEN INFO BLOCK status command is shown below:

	bytes	ck	msb							lsb
Opcode	1	√	OPEN INFO BLOCK (05_{16})							
operand[0]	see[1]	√ see[2]	info_block_reference_path							
:										
:										
:	1	√	status							
:	1	√	reserved							
:	2	√	node_ID							
:										

[1] The size of this field depends on the info block reference path used.
[2] Its descriptor_specifier_types in descriptor_specifiers should be evaluated. The evaluation of other fields is unit or subunit-type dependent.

Figure 9.18 – OPEN INFO BLOCK status command frame

9.7.2.1 Field definitions

info_block_reference_path: The *info_block_reference_path* operand specifies which, among possibly many, of the info blocks that the controller wants to know access status. For more information about the info block reference path, see section 8.3 “Information block reference path” on 55.

status: In the command frame, *status* shall be set to FF_{16}. If the response is STABLE, *status* contains the current access state of the info block specified by the *info_block_reference_path* operand. See clause 9.7.2.2 “OPEN INFO BLOCK status command responses” on page 100 for more information.

node_ID: In the command frame, *node_ID* shall be set to $FFFF_{16}$. In the STABLE response frame with *status* = 33_{16}, the unit or subunit shall update the *node_ID* operand. See Table 9.32 – Value of node_ID based on status for more information.

9.7.2.2 OPEN INFO BLOCK status command responses

The following table shows the OPEN INFO BLOCK status response frame field values for the REJECTED, IN TRANSITION and STABLE responses.

Table 9.30 – Field values in the OPEN INFO BLOCK status command: REJECTED, IN TRANSITION and STABLE response frames

Fields	Command	Response		
		REJECTED	IN TRANSITION	STABLE
Info_block_reference_path	The reference path of the info block to check open status	←	←	←
status	FF_{16}	←	See Table 9.31	See Table 9.31
reserved	00_{16}	←	←	←
node_ID	FF_{16}	←	See Table 9.31	See Table 9.32

← means “same as the command frame”

Table 9.31 – status in the OPEN INFO BLOCK status command: STABLE response frame

Status	Meaning
00_{16}	**Ready for open:** The info block specified by *info_block_reference_path* is closed and can be accessed. No controllers currently have access (either read or write).
01_{16}	**Read opened:** The info block or its containing descriptor(s) is open for read-only access to its data by one or more controllers, and is able to accept additional read-only open requests.
04_{16}	**Non Existent:** No info block specified by *info_block_reference_path* exists.
05_{16}	**List/entry-only (deprecated):** Access control for individual info blocks in a list or an entry is not supported – the controller must obtain access to the entire list or the entire entry. In this case, the NOT IMPLEMENTED response will be returned and, therefore, this value will not be used.
11_{16}	**At capacity:** The info block is open for read-only access to the data and is unable to accept any additional read-only open requests.
33_{16}	**Write opened:** The info block or its containing descriptor(s) is open for read or write access to the data. No access (for read-only or read or write) by other controllers is allowed.

Table 9.32 – Value of node_ID based on status

Status	Value of node_ID
33_{16}	**Controller with write access:** When the info block is open for write access by a controller, the (sub)unit shall update this field to contain the node ID of the controller.
all other values	The value of this field is not changed in the response frame; it remains as FF FF_{16}.

If a controller opens an info block, its status is write-open and the status of its entry and list shall be write-open as a result. However, the statuses of other info blocks in the list, such as in another entry may be “ready for open”. To determine the status of other info blocks, issue the OPEN INFO BLOCK status command on them, not their entry or list.

If for some reason the target cannot return any of the status responses in the STABLE response frame, then the target shall return a REJECTED response. Under this condition, the info block is not accessible at this time.

9.8 READ INFO BLOCK command

The READ INFO BLOCK command is a unit/subunit command and is used to manage the reading of information from info blocks in descriptor structures. The OPEN DESCRIPTOR command should precede any consecutive READ INFO BLOCK commands. Refer to section 9.4 "OPEN DESCRIPTOR command" on page 76 for more information.

The READ INFO BLOCK command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, 30_{16}, 31_{16} and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.8.1 READ INFO BLOCK control command

The READ INFO BLOCK control command is used to read the contents of a specified info block. It is similar to the READ DESCRIPTOR control command, in terms of rules of operation. The control command has the following format:

	bytes	ck	msb							lsb
opcode	1	√	READ INFO BLOCK (06_{16})							
operand[0]	see[1]	√ see[2]	Info_block_reference_path							
operand[1]										
:										
:	1	√	read_result_status							
:	1	√	reserved							
:	2	–	data_length							
:										
:	2	–	address							
:										

[1] The size of this field depends on the descriptor specifier used.
[2] Its descriptor_specifier_types in descriptor_specifiers should be evaluated. The evaluation of other fields is unit or subunit-type dependent.

Figure 9.19 – READ INFO BLOCK control command frame

9.8.1.1 Field definitions

info_block_reference_path: The *info_block_reference_path* specifies the path to the info block that shall be read by this operation. For more information about the info block reference path, see section 8.3 "Information block reference path "on 55.

read_result_status: The controller shall set the *read_result_status* field to FF_{16} in the command frame. In the response frame, the (sub)unit shall update this field to indicate the status of the operation. See Table 9.34 for more information. For details on what this field means, please refer also to the meaning of the *data_length* field below.

reserved: The reserved field shall be treated according to the rules defined in reference [R9].

data_length: The *data_length* operand specifies the number of bytes to be read from the target. There is a special case when *data_length* = 0, which means that the address field shall be ignored and the entire info block (the header, the *primary_fields* and the *secondary_fields*) is specified to be read. In the ACCEPTED response frame, this field will be updated to contain the actual number of bytes that were read.

IMPORTANT: If a target receives a read request with a *data_length* that would result in reading past the end of the data boundary indicated by the *info_block_reference_path* operand, then it shall return only the legitimate data, a *read_result_status* of 12_{16}, and a *data_length* to reflect the size of this data.

address: The *address* field specifies the relative location of the starting point to be read. It is an offset from the beginning of the info block header. Note that this is different from the WRITE INFO BLOCK command, where write operations are relative to the beginning of the *primary_fields* area. When *data_length* = 0, then the *address* field shall be ignored.

If there is no data at the specified *address* when *data_length* is > 0, then the target shall return a REJECTED response to the command.

If an ACCEPTED response frame is returned by the target after a READ INFO BLOCK control command, the response data consists of additional fields inserted after the *address* field that contain the data bytes requested. If the target is not able to return the number of bytes indicated by the *data_length* input field in a single operation due to data transfer limitations, then it shall return the maximum quantity of data it is able to and set the *data_length* field in the response frame to this value. It shall also update the *read_result_status* field as described in section 9.5.1, “READ DESCRIPTOR control command” on page 83.

When reading an info block, the entire block, including nested info blocks, may be read in a single operation (depending on the data transfer limits of the controller and target).

9.8.1.2 READ INFO BLOCK control command responses

When the READ INFO BLOCK command succeeds, the info block’s data is inserted after the address field with a length of *data_length*. Fields in the command and response frames are summarized in the table below:

Table 9.33 – Field values in the READ INFO BLOCK control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
info_block_reference _path	The info block reference path of the info block to read	←	←	←
read_result_status	FF_{16}	←	←	see Table 9.34
reserved	00_{16}	←	←	←
data_length	XX_{16} (specific amt) or 00_{16} (all)	←	←	YY_{16} (actual amt)
address	address offset	←	←	←
data	none	none	None	Inserted info block data

← means “same as the command frame”

Table 9.34 – read_result_status field values in the ACCEPTED response frame

Value	Meaning
10_{16}	**Complete read:** The entire data specified by the *info_block_reference_path*, *data_length*, and the *address* fields in the command frame was returned.
11_{16}	**More to read:** A portion of the data specified by the *info_block_reference_path*, *data_length*, and *address* fields in the command frame was returned. The *data_length* field in the response frame indicates exactly how much data was returned.
12_{16}	**Data length too large:** There is less data in the info block than was specified by *data_length* in the command frame. The response frame contains the actual *data_length* of the returned data, which includes data at the end of the info block.

9.8.1.3 READ INFO BLOCK – examples

See section 9.5.3, “Reading a list or an entry” on page 85 for examples of reading info blocks.

9.9 WRITE INFO BLOCK command

The WRITE INFO BLOCK command is a unit/subunit command and is used to manage the writing of info blocks within descriptor structures.

The WRITE INFO BLOCK command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, 30_{16}, 31_{16} and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.9.1 WRITE INFO BLOCK control command

The WRITE INFO BLOCK control command allows a controller to change the contents of a specified information block. It is similar to the WRITE DESCRIPTOR control command in terms of rules of operation. The control command has the following format:

	bytes	ck	msb							lsb
opcode	1	√	WRITE INFO BLOCK (07_{16})							
operand[0]	see[1]	see[2]	info_block_reference_path							
:										
:										
:	1	√	subfunction "partial_replace" = 50_{16}							
:	1	√	group_tag							
:	2	–	replacement_data_length = I							
:										
:	2	–	Address							
:										
:	2	–	original_data_length							
:										
:	i	–	replacement_info_block_data							
:										
:										

[1] The size of this field depends on the info block reference path used.
[2] Its descriptor_specifier_types in descriptor_specifiers should be evaluated. The evaluation of other fields is unit or subunit-type dependent.

Figure 9.20 – WRITE INFO BLOCK control command frame

9.9.1.1 Field definitions

info_block_reference_path: The *info_block_reference_path* specifies the path to the info block that shall be changed by this operation. For more information about the info block reference path, see section 8.3 “Information block reference path”on 55.

subfunction: The *subfunction* field specifies which write operation subfunction to perform. The subfunctions are the same as those for the WRITE DESCRIPTOR control command, however, ONLY the *partial_replace* subfunction is valid for this command. If the controller specifies any other subfunctions, then the unit or subunit shall return a response of NOT IMPLEMENTED. See clause 9.6.1, “WRITE DESCRIPTOR control command” on page 86 for more information on the *partial_replace* subfunction.

NOTE — A write operation that results in a change of the size of an info block indirectly affects length fields that appear prior to the memory area that changed. Figure A.1 – Structure of list descriptor with entries and info blocks on page 138 maps all the descriptor and info block length fields with areas of the descriptor. From this figure, it can be determined which length fields are affected by write operations anywhere in the info block.

A subunit specification may place restrictions on the fields in the WRITE INFO BLOCK command. For example, the combination of length and address may be limited.

group_tag: The *group_tag* field is also the same as defined in the WRITE DESCRIPTOR control command.

Table 9.35 – Group_tag values

group_tag	Action
00_{16}	**Immediate:** Immediately write the data to the info block. Whether the data is committed after it is written or after it is closed is unit or subunit-type dependent.
01_{16}	**First:** Begin an “grouped update” sequence, with this command being the first of several which may follow. The target shall take the necessary actions to prepare for this sequence, and to ensure that the original info block structure is preserved, should the sequence not finish normally.
02_{16}	**Continue:** Any number of subsequent commands may be issued using this tag, after the “first” has been issued.
03_{16}	**Last:** This command signals the last of the grouped update sequence. When the target receives this *group_tag* value, it commits all of the data written in the sequence, including the data in this command, to the specified info block structure. If the target does NOT receive this command before a bus reset or the time out period used for info block access, then it shall discard ALL of the data written in this sequence and leave the specified info block unmodified.
all others	reserved for future definition

replacement_data_length: The *replacement_data_length* field specifies the number of bytes in the *replacement_info_block_data* field.

NOTE — Subunit-type specifications may restrict the usage of *replacement_data_length* and *address* in this command, such as requiring the *replacement_data_lengh* to be equal to the *original_data_length*, and requiring that the *address* be 0000_{16}.

address: The *address* field specifies a zero-based offset, relative to the beginning of the *primary_fields* area, at which the data should be written.

original_data_length: The *original_data_length* field specifies the number of bytes to be deleted.

replacement_info_block_data: The *replacement_info_block_data* field contains the new data to be written into the info block.

9.9.1.2 Writing to info blocks

When writing into an info block, only the *primary_fields* area shall be modified by the WRITE INFO BLOCK command. If an info block contains several nested info blocks, each of them must be changed individually by writing into their primary fields. The following diagram illustrates this concept:

Table 9.36 – Information block basic structure

Address	Contents
$00\ 00_{16}$	compound_length
$00\ 01_{16}$	
$00\ 02_{16}$	info_block_type
$00\ 03_{16}$	
$00\ 04_{16}$	primary_fields_length
$00\ 05_{16}$	
$00\ 06_{16}$	primary_fields... (info block type-specific)
:	
:	
:	
:	secondary_fields (if defined)
:	
:	

offset = 0

This area modified by WRITE INFO BLOCK command

The *primary_fields* area, marked by the lighter gray color, is the only area that can be changed by a controller using the WRITE INFO BLOCK command. The (sub)unit is responsible for updating the appropriate length fields for the info block being written into as well as any containing info blocks. It is also responsible for adjusting any descriptor offset values. See Figure A. for more information.

NOTE — Using a WRITE DESCRIPTOR commands, the controller can create or write an info block with whole data of info block basic structure. See clause 9.2.2.2 for more information about the rule. In this case, the controller is responsible for ensuring the appropriate length fields in the command frame as well as any containing info block.

The following rules are defined for the WRITE INFO BLOCK command:

1) Before making any changes to an info block, the controller shall first obtain write access to that info block. This can be achieved in two ways:

   a) Info blocks are always nested inside of descriptor structures (entries, lists, UID, SID, etc.), thus, the controller may use the OPEN DESCRIPTOR control command with write access to the descriptor. This is the recommended method. Those info blocks that are inside the scope of the descriptor will be available for modification.

   b) Using the OPEN INFO BLOCK control command with write access. Supporting this command is optional and not recommended for units and subunits, and would allow several controllers to change separate info blocks independently from each other. If the unit or subunit supports the WRITE INFO BLOCK control command, then it may also support the OPEN INFO BLOCK control command; however it is supported, it is not required to support multiple simultaneous controllers (supporting only one controller with write access at a time is allowed).

2) Other access rules as given in clause 9.2.2.2 shall be adhered to.

9.9.1.3 WRITE INFO BLOCK control command responses

The following table summarizes the command and response field values for the WRITE INFO BLOCK control command.

Table 9.37 – Field values in the WRITE INFO BLOCK control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
info_block_reference_ path	The info block reference path of the info block to write	←	←	←
subfunction	50_{16}	see Table 9.38	←	see Table 9.38
group_tag	see Table 9.35	←	←	←
replacement_ data_length	size of replacement data		←	Number of accepted bytes
address	address of starting point from beginning of primary fields	←	←	←
original_data_length	size of original data	←	←	←
data	Variable	none	None	None

← means “same as the command frame”

Table 9.38 – subfunction in the response frame

subfunction in the response	Response	Meaning
x0$_{16}$	ACCEPTED	The specified subfunction was performed with no problem. R/I fields were not changed as specified in the command frame.
x1$_{16}$	ACCEPTED	The origial descriptor data, specified by *address* and *origial_data_length* fields, was deleted and the lower part of replacement data was inserted. R/I fields were not changed as specified in the command frame. The target shall return the *replacement_data_length* field to be equal to the size of the performed part. The controller may need to issue more *partial_ replace_ operation(s)* to insert the reamaining part.
x2$_{16}$	REJECTED	The *address* and *data_length* fields specify an invalid address, **or** the target cannot support the amount of data specified by *data_length,* so the write operation was not performed. This may also be returned if the target prevents controllers from writing any fields in *replacement_info_block_data* field, or *info_block_reference_path* is invalid.
x3$_{16}$	ACCEPTED	The specified subfunction was performed but some R/W/I fields were not changed as specified in the command frame.
x4$_{16}$	ACCEPTED	The origial descriptor data, specified by *address* and *origial_data_length* fields, was deleted and the lower part of replacement data was inserted. Some R/W/I fields were not changed as specified in the command frame. The target shall return the *replacement_data_length* field to be equal to the size of the performed part. The controller may need to issue more *partial_ replace_ operations(s)* to insert remaining part.

If subfunction x3$_{16}$ or x4$_{16}$ is returned, then the controller may read the info block to determine which R/W/I fields were not changed. Note that the unit or subunit that is compliant to the previous versions of this specification does not return subfunction x3$_{16}$ or x4$_{16}$.

9.10 SEARCH DESCRIPTOR command

The SEARCH DESCRIPTOR command is a unit/subunit command and allows a controller to request the unit or subunit to execute a search within the descriptor data space (NOT within the **content** data space) looking for a specified entity. If a search is successful, the returned results will be a specifier that identifies the first candidate that was found; multiple specifiers are not returned by the search operation. The controller must specify additional searches to find additional instances that match the search criteria.

The SEARCH DESCRIPTOR command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.10.1 SEARCH DESCRIPTOR control command

The control command has the following format:

	bytes	ck	msb							Lsb
opcode	1	√	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	See[1]	see[2]	search_for							
:										
:										
:	See[1]	see[2]	search_in							
:										
:										
:	See[1]	see[2]	start_point							
:										
:										
:	1	√	Direction							
:	1	√	response_format							
:	1	√	Status							

[1] The length of this field is variable.
[2] Refer to the figures of search_for, search_in and search_point fields described below.

Figure 9.21 – SEARCH DESCRIPTOR control command frame

9.10.1.1 Field definitions

search_for: The *search_for* operand specifies what the controller would like the unit or subunit to search for. This operand has the following format:

address offset	bytes	ck	msb							Lsb
00_{16}	1	√	Length							
01_{16}	See[1]	–	search_data							
:										
:										

[1] The length of this field is variable

Figure 9.22 – Search_for operand of SEARCH DESCRIPTOR control command

length: The *length* operand specifies the number of bytes for the following *search_data* operand. To perform a wild card search on the *search_data* operand, the controller can set the length operand to 0 and not include any *search_data*. The wildcard search is useful for searching through all object IDs, entry types, and list types in descriptors.

search_data: The *search_data* operand contains the subject of the search. These bytes can represent text such as "CNN" or any numeric value.

search_in: The *search_in* operand specifies the location and scope of the search. The controller is not required to have read or read/write access to the descriptor space indicated by *search_in*, in order to request the search. The unit or subunit shall search through all descriptors that match the *search_in* operand, even if they are open for modification by a controller.

This operand has the following format:

address offset	bytes	ck	msb							lsb
00_{16}	1	√	Length							
01_{16}	1	√	Type							
02_{16}	See[1]	–	type_specific_info							
:										
:										

[1] The length of this field is variable

Figure 9.23 – Search_in operand of SEARCH DESCRIPTOR control command

length: The *length* operand specifies the number of bytes in the *type_specific_info* operand.

type: The *type* operand specifies the type of specification for the search location. It can have one of the following values:

Table 9.39 – Type value for search_in

Type for search_in	Meaning
10_{16}	List descriptors
20_{16}	Entry descriptors
30_{16}	Other descriptors
50_{16}	Fields specified by an offset address and length in list descriptors
52_{16}	*list_type* fields in list descriptors
60_{16}	Fields specified by an offset address and length in entry descriptors
62_{16}	*entry_type* fields in entry descriptors
64_{16}	*child_list_ID* fields in entry descriptors
66_{16}	*object_ID* fields in entry descriptors
70_{16}	Fields specified by an offset address and length in Other Descriptors
all others	reserved for future specification

type_specific_info: The *type_specific_info* operand specifies the scope and location of the search. Its format is defined by the *type* values in the table above. The *type_specific_info* operand is specified below for each of the *type* values in clause 9.10.2 “Type_specific_info for the search_in operand” on page 116.

start_point: The *start_point* operand specifies where to begin the search. It has the following basic structure:

address offset	bytes	ck	msb							Lsb
00_{16}	1	√	Length							
01_{16}	1	√	Type							
02_{16}	See[1]	–	type_specific_info							
:										
:										

[1] The length of this field is variable

Figure 9.24 – Start_point operand of SEARCH DESCRIPTOR control command

length: The *length* operand specifies the number of bytes in the *type_specific_info* operand.

type: The *type* operand specifies how the starting point is indicated, in the *type_specific_info* operand.

Table 9.40 – Type values for start_point

Type for start_point	Starting Point for the Search
00_{16}	The controller does not care where the start point is – the (sub)unit chooses where to start the search operation.
02_{16}	At the “current location”, where the current location is defined by the currently selected entry descriptor.
03_{16}	At the “current location”, where the current location is defined by the position of the last search result.
10_{16}	At the point specified by an offset address in the list descriptor, where the list is specified by its list ID.
11_{16}	At the *list_type* field in the specified list descriptor, where the list is specified by its list ID.
20_{16}	At the point specified by an offset address in the specified entry descriptor, where the entry is specified by *entry_position*.
21_{16}	At the point specified by an offset address in the specified entry descriptor, where the entry is specified by *object_ID*.
22_{16}	At the *entry_type* field in the specified entry descriptor, where the entry is specified by *entry_position*.
23_{16}	At the *entry_type* field in the specified entry descriptor, where the entry is specified by *object_ID*.
24_{16}	At the *child_list_ID* field in the specified entry descriptor, where the entry is specified by *entry_position*.
25_{16}	At the *child_list_ID* field in the specified entry descriptor, where the entry is specified by *object_ID*.
26_{16}	At the *object_ID* field in the specified entry descriptor, where the entry is specified by *entry_position*.
27_{16}	At the *object_ID* field in the specified entry descriptor, where the entry is specified by *object_ID*.
30_{16}	At the point specified by an offset address in the Other Descriptor, where that descriptor is specified by a *descriptor_specifier* structure.
all others	reserved for future specification

type_specific_info: The *type_specific_info* operand indicates the starting point. Its format depends on the value of the type operand. The *type_specific_info* operand is specified below for each of the *type* values in clause 9.10.2 “Type_specific_info for the search_in operand” on page 116.

direction: The *direction* operand specifies how the search should proceed, as indicated in the following table:

Table 9.41 – Direction operand meanings

Direction	Meaning
00_{16}	The controller does not care about the direction of the search - the (sub)unit chooses the direction.
10_{16}	Up - in the increasing order of the *search_for* specifier.
12_{16}	Up - in the increasing order of the *search_for* specifier, based on the *entry_position*.
13_{16}	Up - in the increasing order of the *search_for* specifier, based on the *object_ID*.
20_{16}	Down - in the decreasing order of the *search_for* specifier.
22_{16}	Down - in the decreasing order of the *search_for* specifier, based on the *entry_position*.
23_{16}	Down - in the decreasing order of the *search_for* specifier, based on the *object_ID*.
all others	reserved for future specification

The order of searching, as specified by the *direction* operand, has the following rules:

Table 9.42 – Order of searching rules

search_in	Rules for Search Direction
a field	The address value within the field using the increasing direction $=10_{16}$, or the decreasing direction $=20_{16}$
An entry descriptor	
a list descriptor	
fields in entries	The *entry_position* value using the increasing direction $=12_{16}$, or the decreasing direction $= 22_{16}$
entries	The *object_ID* value using the increasing direction $=13_{16}$, or decreasing direction $= 23_{16}$
fields in lists	The list ID value using the increasing direction $= 10_{16}$, or decreasing direction $= 20_{16}$
lists	

response_format: The *response_format* operand specifies how the controller would like the return data to be presented, as defined in the following table:

Table 9.43 – Response_format operand

Response_ format	Meaning
00_{16}	Not specified - the (sub)unit may choose how to present the data. This is also called the “don’t care” response.
10_{16}	By descriptor_specifier_type 10_{16} (specified by list ID)
11_{16}	By descriptor_specifier_type 11_{16} (specified by list_type)
20_{16}	By descriptor_specifier_type 20_{16} (entry_position)
21_{16}	By descriptor_specifier_type 21_{16} (object_ID)
all others	reserved for future specification

status: The *status* operand is set to FF_{16} by the controller in the control frame. It is updated in the ACCEPTED response frame to indicate the result of the search operation.

9.10.2 Type_specific_info for the search_in operand

The following diagrams illustrate the format of the *type_specific_info* fields of the *search_in* operand. These fields are defined by the *type* field of the *search_in* operand.

9.10.2.1 Supporting data structures

Many of the *type_specific_info* structures for the *search_in* operand make use of the *entry_descriptor_specifier* and the *list_descriptor_specifier*. Other data structures include the standard *descriptor_specifier* and an offset and length structure. All are described below.

9.10.2.1.1 Entry_descriptor_specifier

The *entry_descriptor_specifier* is a data structure that specifies an entry. In the context of the *search_in* operand, it specifies an entry or a collection of entries in which the search operation should be performed. It has the following format:

address offset	Bytes	msb							lsb
00_{16}	1	Type							
:	See[1]	type_specific							
:									
:									

[1] The length of this field is variable

Figure 9.25 – Entry_descriptor_specifier

type: The *type* field defines how the entry(s) is indicated in the *type_specific* field.

type_specific: The following table illustrates the relationship between the *type* and *type_specific* fields:

Table 9.44 – Relationship between type and type_specific fields

Type	Meaning	type_specific Field	Size of type_specific Field
20_{16}	A specified entry (by position)	entry_position	k bytes (see NOTE below)
21_{16}	A specified entry (by object_ID)	Object_ID	k bytes (see NOTE below)
22_{16}	Any entries with the specified entry_type field	Entry_type	1 byte
$2F_{16}$	Any entries	None	zero bytes
all others	Reserved for future specification	---------------	---------------

NOTE — When an entry is specified by its object ID, the size of the *type_specific* field is indicated by the *size_of_object_ID* field of the (sub)unit identifier descriptor. When an entry is specified by its entry position, the size of the *type_specific* field is indicated by the *size_of_entry_position* field of the (sub)unit identifier descriptor.

9.10.2.1.2 List_descriptor_specifier

The *list_descriptor_specifier* is a data structure that identifies a list descriptor (or more than one list). In the context of the *search_in* operand, it specifies the list or a collection of lists in which the search operation should be performed. It has the following format:

Address offset	bytes	msb							lsb
00_{16}	1	type							
:	see[1]	Type_specific							
:									
:									

[1] The length of this field is variable

Figure 9.26 – List_descriptor_specifier

type: The *type* field defines how the list(s) are indicated in the *type_specific* field.

type_specific: The following table illustrates the relationship between the *type* and *type_specific* fields:

Table 9.45 – Relationship between type and type_specific fields

Type	Meaning	type_specific Field	Size of type_specific Field
10_{16}	A specified list (by child or root list ID)	Child_list_ID or root_list_ID	n bytes (see Note below)
12_{16}	Any lists with the specified list_type	list_type	1 byte
all others	Reserved for future specification	---------------	---------------

NOTE — The number of bytes for the *type_specific* field when using a list ID will depend on the *size_of_list_ID* field of the target's (sub)unit identifier descriptor.

9.10.2.1.3 Descriptor_specifier

The general *descriptor_specifier* structure is already defined. For the SEARCH DESCRIPTOR control command, it is used to specify one of the (non-entry and non-list) descriptor structures (such as the (sub)unit identifier descriptor) in which the search operation is to be performed.

9.10.2.1.4 Offset_address and length

The *offset_address* field specifies the starting address within the specified descriptor structure to begin the search. This field is always 2 bytes in length.

The *length* field specifies the number of bytes over which to perform the search. This field is always 1 byte in length.

9.10.2.2 Type_specific_info data structures

address offset	Bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.27 – Type_specific_info for the search_in operand, type 10_{16}

address offset	Bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									
:	See[1]	entry_descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.28 – Type_specific_info for the search_in operand, type 20_{16}

address offset	Bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.29 – Type_specific_info for the search_in operand, type 30_{16}

address offset	Bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									
:	2	offset_address							
:									
:	1	Length							

[1] The length of this field is variable

Figure 9.30 – Type_specific_info for the search_in operand, type 50_{16}

Address offset	bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.31 – Type_specific_info for the search_in operand, type 52_{16}

Address offset	bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									
:	See[1]	entry_descriptor_specifier							
:									
:									
:	2	Offset_address							
:									
:	1	length							

[1] The length of this field is variable

Figure 9.32 – Type_specific_info for the search_in operand, type 60_{16}

Address offset	bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									
:	See[1]	entry_descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.33 – Type_specific_info for the search_in operand, type 62_{16}

Address offset	bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
	See[1]	entry_descriptor_specifier							

[1] The length of this field is variable

Figure 9.34 – Type_specific_info for the search_in operand, type 64_{16}

address offset	Bytes	msb							lsb
00_{16}	See[1]	list_descriptor_specifier							
:									
:									
:	See[1]	entry_descriptor_specifier							
:									
:									

[1] The length of this field is variable

Figure 9.35 – Type_specific_info for the search_in operand, type 66_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier							
:									
:									
:	2	offset_address							
:									
:	1	Length							

[1] The length of this field is variable

Figure 9.36 – Type_specific_info for the search_in operand, type 70_{16}

9.10.3 Type_specific_info for the start_point operand

The following diagrams illustrate the format of the *type_specific_info* fields of the *start_point* operand. These fields are defined by the *type* field of the *start_point* operand.

9.10.3.1 Supporting data structures

The *type_specific_info* structures of the *start_point* operand make use of the *descriptor_specifier*, **including** those for the entry and list descriptors. Note that this is different from the *search_in* operand, which by necessity had to define its own entry and list descriptor specifiers.

Some of the *type_specific_info* structures of the *start_point* operand also make use of an *address_offset* field. This offset is from the beginning of the descriptor structure specified in the *start_point* operand.

The *entry_type* field used in some of the structures refers to the type of entry, as defined by the *entry_type* field of the entry descriptor structures.

9.10.3.2 Type_specific_info data structures

address offset	bytes	msb							lsb
------	------	There is no type_specific_info for type 00_{16}							

Figure 9.37 – Type_specific_info for the start_point operand, type 00_{16}

address offset	bytes	msb							lsb
------	------	There is no type_specific_info for type 02_{16}							

Figure 9.38 – Type_specific_info for the start_point operand, type 02_{16}

address offset	bytes	msb							lsb
------	------	There is no type_specific_info for type 03_{16}							

Figure 9.39 – Type_specific_info for the start_point operand, type 03_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for a list specified by list ID							
	2	Offset_address							

[1] The length of this field is variable

Figure 9.40 – Type_specific_info for the start_point operand, type 10_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for a list specified by list ID							

[1] The length of this field is variable

Figure 9.41 – Type_specific_info for the start_point operand, type 11_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an entry position reference							
:									
:									
:	2	Offset_address							
:									

[1] The length of this field is variable

Figure 9.42 – Type_specific_info for the start_point operand, type 20_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an object_ID reference							
:									
:									
:	2	offset_address							
:									

[1] The length of this field is variable

Figure 9.43 – Type_specific_info for the start_point operand, type 21_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an entry position reference							
:									
:									
:	1	entry_type							

[1] The length of this field is variable

Figure 9.44 – Type_specific_info for the start_point operand, type 22_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an object_ID reference							
:									
:									
:	1	entry_type							

[1] The length of this field is variable

Figure 9.45 – Type_specific_info for the start_point operand, type 23_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an entry position reference							
:									
:									

[1] The length of this field is variable

Figure 9.46 – Type_specific_info for the start_point operand, type 24_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an object_ID reference							
:									
:									

[1] The length of this field is variable

Figure 9.47 – Type_specific_info for the start_point operand, type 25_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an entry position reference							
:									
:									

[1] The length of this field is variable

Figure 9.48 – Type_specific_info for the start_point operand, type 26_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier for an object_ID reference							
:									
:									

[1] The length of this field is variable

Figure 9.49 – Type_specific_info for the start_point operand, type 27_{16}

address offset	bytes	msb							lsb
00_{16}	See[1]	descriptor_specifier "other" descriptor							
:									
:									
:	2	Offset_address							
:									

[1] The length of this field is variable

Figure 9.50 – Type_specific_info for the start_point operand, type 30_{16}

9.10.4 Examples of the SEARCH DESCRIPTOR control command (Informative)

This clause presents some examples of how to use the SEARCH DESCRIPTOR control command for various types of searches. For those examples that refer to a specific type of (sub)unit, supporting information may be found in the unit or subunit-type specification.

9.10.4.1 Example 1: Search for the *service_name* "NHK" in all DVB service lists (tuner subunit)

This example demonstrates how an arbitrary field of an entry descriptor structure can be searched, looking for a specified value. This type of search could be used to find fields, which may be unknown to the target subunit.

The control command frame for this search would be defined as follows:

	msb							lsb
opcode	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	**search_for** (03_{16}) length							
operand[1]	($4E_{16}$) "N"							
operand[2]	(48_{16}) "H"							
operand[3]	($4B_{16}$) "K"							
operand[4]	**search_in** (07_{16}) length							
operand[5]	(60_{16}) type: search in specified fields							
operand[6]	(12_{16}) in any lists with the specified list_type field							
operand[7]	(82_{16}) list_type: **service**							
operand[8]	($2F_{16}$) search in any entries in the lists							
operand[9]	(00_{16}) offset address...							
operand[10]	(19_{16}) ...for the **service_name** field							
operand[11]	(03_{16}) length							
operand[12]	**start_point** (00_{16}) not specified - subunit chooses							
operand[13]	**direction** (00_{16}) not specified – subunit chooses							
operand[14]	**response_format** (21_{16}) return the data as an object_ID reference							
operand[15]	**status** (FF_{16})							

Figure 9.51 – SEARCH DESCRIPTOR control command frame, example 1

9.10.4.2 Example 2: Search for the next service entry in the service lists (tuner subunit)

This example shows how to search on one of the basic fields defined for all entry descriptor structures. This type of search is useful for well-defined fields that all target subunits must know about.

The control command frame for this type of search would appear as follows:

	msb							lsb
opcode	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	**search_for** (00_{16}) not specified							
operand[1]	**search_in** (04_{16}) length							
operand[2]	(66_{16}) type: search in **object_ID** fields							
operand[3]	(12_{16}) in any lists with the specified list_type field							
operand[4]	(82_{16}) list_type: **service**							
operand[5]	($2F_{16}$) search in any entries in the lists							
operand[6]	**start_point** (02_{16}) start at the currently selected entry							
operand[7]	**direction** (13_{16}) search up (increasing order of object_ID value)							
operand[8]	**response_format** (21_{16}) return the data as an object_ID reference							
operand[9]	**status** (FF_{16})							

Figure 9.52 – SEARCH DESCRIPTOR control command frame, example 2

9.10.4.3 Example 3: Search for the parent entry of the service list with list ID 2000_{16} (tuner subunit)

The control command frame for this type of search would appear as follows:

	msb							lsb
opcode	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	**search_for** (02_{16}) length							
operand[1]	(20_{16}) “20”							
operand[2]	(00_{16}) “00”							
operand[3]	**search_in** (04_{16}) length							
operand[4]	(64_{16}) type: search in **child_list_ID** fields							
operand[5]	(12_{16}) in any lists with the specified list_type field							
operand[6]	(80_{16}) list_type: **multiplex**							
operand[7]	($2F_{16}$) search in any entries in the lists							
operand[8]	**start_point** (00_{16}) not specified - subunit chooses the start point							
operand[9]	**direction** (00_{16}) not specified - subunit chooses the direction							
operand[10]	**response_format** (20_{16}) return the data as an entry position reference							
operand[11]	**Status** (FF_{16})							

Figure 9.53 – SEARCH DESCRIPTOR control command frame, example 3

9.11 OBJECT NUMBER SELECT command

The OBJECT NUMBER SELECT command is a unit/subunit command (See footnote[3]), and performs a selection of an entry (or many entries), and routes the data that the entry represents to a subunit source plug. This is achieved by specifying, for each desired entry, either a path to that entry in its list, or a direct specification of that entry, if the entry is unique within its unit or subunit. The type of unit or subunit receiving the command will define the nature of what it means to “select” an entry. For details on subunit-specific functionality, please refer to the OBJECT NUMBER SELECT reference in the appropriate subunit-type specifications.

The OBJECT NUMBER SELECT command supports descriptor specifiers 00_{16}, 10_{16}, 11_{16}, 20_{16}, 21_{16}, 23_{16}, 30_{16}, 31_{16} and 80_{16} – BF_{16}. See section 8.2 and the sections below for details.

9.11.1 OBJECT NUMBER SELECT control command

The general operation of OBJECT NUMBER SELECT is to allow the controller to specify which entry(s) to select, a subunit source plug that should receive the data from the entry(s), and a subfunction which modifies the command in a subunit-specific way. The format of the OBJECT NUMBER SELECT control command frame is as follows:

	bytes	ck	msb							lsb
opcode	1	√	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	1	√	Source_plug							
operand[1]	1	√	Subfunction							
operand[2]	1	√	Status							
operand[3]	1	–	number_of_ons_selection_specifications (n)							
operand[4]	See[1]	–	ons_selection_specification[0]							
:										
:										
:										
:	-	-	:							
:	See[1]	–	ons_selection_specification[n - 1]							
:										
:										
:										

[1] The length of this field is variable

Figure 9.54 – OBJECT NUMBER SELECT control command frame

9.11.1.1 Field definitions

source_plug: the *source_plug* operand indicates the subunit source plug number which shall output the specified entry(s). Values for plug addresses are shown by Table 9.46.

[3] Though this is a unit/subunit command, it presently doesn’t support being addressed to a unit.

Table 9.46 – AV/C Subunit Plug Address

source_plug	Source Plug
00_{16} – $1E_{16}$	Source Plug 0 – 30
$1F_{16}$ – FC_{16}	Reserved for future Specification
FD_{16}	Reserved for future Specification
FE_{16}	Do not output
FF_{16}	Any available source plug

The plug value FE_{16} is a special case; it means that the specified item(s) should be “selected”, but not output to any plug. This is used when the controller does not want the selection specification to affect the output signal. An example is the case of a CD changer subunit; selecting a CD would mean that the CD is taken from the changer and placed into the player mechanism and has no meaning for the output signal.

subfunction: The *subfunction* field specifies the operation of the CONTROL command. The general AV/C model defines a set of subfunctions for this command, but the meaning of “selection” may vary according to the type of unit or subunit receiving this command. Different types of subunits may not be able to support all of the general subfunctions. Subunit type specifications shall not define new subfunctions. Please refer to the appropriate unit or subunit-type specification document for specific details of the subfunctions supported by that subunit type, and what those subfunctions mean for that subunit type.

For the general AV/C model, the following subfunctions are defined. Note that the description of what action to take refers to the output signal, but the same concepts are applicable to the non-output signal, plug FE_{16} cases:

Table 9.47 – Subfunctions for the general AV/C model

subfunction	Meaning	Action
$C0_{16}$	clear	Stop the output of all selections on the specified plug. No selection specifiers shall be included in the command frame for this subfunction.
$D0_{16}$	remove	Remove the specified selection from the output stream on the specified plug.
$D1_{16}$	append	Add (multiplex) the specified selection(s) to the current output.
$D2_{16}$	replace	Remove the current selection(s) from the specified plug, and output or multiplex the specified selection(s).
$D3_{16}$	new	Output the specified selection(s) on the specified plug if the plug is currently unused; otherwise, return a REJECTED response to the selection command.
all others	X	reserved for future specification.

status: The *status* field shall be set to FF_{16} on input to the CONTROL command.

number_of_ons_selection_specifications: The *number_of_ons_selection_specifications* operand shows the number of ONS selection specifiers that are provided in the parameter block.

ons_selection_specification[x]: The *ons_selection_specification[x]* operands each define a single entry to be selected. The *ons_selection_specification* structure has two basic forms: a full path specification from the root of a descriptor hierarchy down to the entry being selected; and a “don’t care” version of this specifier which does not indicate a path to an entry. These are illustrated in the following clauses.

Some of the contents of this specification will vary based on the type of subunit which is receiving this command. For details on subunit-specific descriptors, please refer to the OBJECT NUMBER SELECT reference in the appropriate subunit-type specific clauses.

9.11.1.2 Subfunction implementation rules

The following rules shall be adhered to when a subunit implements the OBJECT NUMBER SELECT command subfunctions. Note that some subunit-type specifications may add further rules, but they shall not conflict with these general rules:

1) The remove subfunction shall be REJECTED if the specified information instances are not present on the specified output plug.
2) If a controller wants to be sure that it is making a selection on an unused plug, then it should use the new subfunction to establish an initial selection on that plug. Subsequent selections may be appended to that plug.

9.11.1.3 The general ons_selection_specification structure

The general *ons_selection_specification* is as follows:

offset	bytes	msb							lsb
00_{16}	2	root_list_ID							
:									
:	1	selection_indicator							
:	1	target_depth (m)							
:									
:	see[1]	path_specifier[0]							
:									
:		:							
:									
:	see[1]	path_specifier[m-1]							
:									
:									
:	see[2]	Target							
:									
:									

[1] The length of this field depends on the *descriptor_specifier_type*.
[2] See Figure 9.56 on page 129.

Figure 9.55 – General ons_selection_specification (full path specification)

The fields of this *ons_selection_specification* define exactly one path, among possibly many, to the desired entry being selected. This is necessary because in general, an entry may have more than one parent and hence more than one path specification.

root_list_ID: The *root_list_ID* field contains the ID of the root list that defines the beginning of the path. This list must be the top of a descriptor hierarchy (e.g., it must be referenced from the (sub)unit identifier descriptor). The root lists will have fixed or well-known ID values.

selection_indicator: The *selection_indicator* field indicates how the entry references are specified, either by position or unique object ID. Each *path_specifier* and the *target* must be specified in the same manner. The *selection_indicator* field is encoded as follows:

Table 9.48 – Selection_indicator field encoding

selection_ indicator	Meaning
1xxx xxxx	**descriptor_specifier_type_flag:** when set to 1, this indicates that the path specifiers and the specifiers in the *target* are by object ID. When it is 0, it indicates that the path specifiers and specifiers in the target are by entry position.
xxxx xxx1	**target_format_flag** - this flag indicates the format of the *target* field. When set to 1, this flag indicates that the target is to be selected using specified CHILDREN of that entry. When this flag is zero, then the entire entry is to be selected (no specific CHILDREN are specified). Please see the example selections for details.
all others	reserved for future specification.

target_depth: The *target_depth* field indicates how deep in the descriptor hierarchy to go in order to find the list which holds the target entries(s) to be selected. The root of the descriptor hierarchy has a depth of zero.

path_specifier[x]: The *path_specifier[x]* fields are *descriptor_specifier* data structures (see section 8.1 for more information) that appear in order from the top of the descriptor hierarchy down. A level corresponds to a list, and each *path_specifier* indicates an entry in the list at that level of the descriptor hierarchy. The root has level zero. Since the path specification always starts with the root list and an entry in that list, and since an entry always contains exactly zero or one child reference, we always know exactly which list we are looking in and what the next list is that we will be looking at. If the value of the *target_depth* is zero, then the path specifier fields don't exist.

target: The *target* field will indicate which entry is to be selected from the target level list indicated by the path specification as follows:

offset	bytes	msb							lsb
00_{16}	see[2]	target_object_reference							
:									
:									
:									
:	1	Number_of_children (m)[1]							
:	see[2]	child_object_reference [0][1]							
:									
:									
:		:							
:	see[2]	child_object_reference [m - 1][1]							
:									
:									

[1] Present only when *target_format_flag* selection indicator is 1.
[2] The length of this field depends on the *descriptor_specifier_type*.

Figure 9.56 – Format of the target field (full path specification)

Target_object_reference, child_object_reference: The *target_object_reference* and *child_object_reference* fields are *descriptor_specifier* structures for entry descriptors.

The format of the *target* field shall be defined in subunit-specific ways.

9.11.1.4 The “don’t care” specification

In some cases a controller may know the unique object ID of an item that it wants to select, and it does not need to traverse a descriptor hierarchy to find it. In some cases, the controller may not be able to determine an exact path for the item that it wants, and is willing to accept whatever path, among possibly many, the subunit may choose. In other situations, the particular technology may be defined with non-ambiguous paths among all levels of the descriptor hierarchy so a full path specification is not required.

In these cases, the controller may not want to or may not be able to fill out a full path specification for the *ons_selection_specification*. For this, we define the “don’t care” specification, which only points at the entry to be selected. The format of the “don’t care” *ons_selection_specification* is as follows:

offset	bytes	msb							lsb
00_{16}	2	root_list_ID							
:									
:	1	selection_indicator							
:	1	target_depth (m) = FF_{16}							
:	see[1]	Target							
:									
:									
:									

[1] 1 See section 8.2.7, “Entry descriptor specified only by object_ID” on page 54.
Figure 9.59 on page 131 and 1 See section 8.2.4, “Entry descriptor specified by position in its list” on page 52.
Figure 9.58 on page 131.

Figure 9.57 – ons_selection_specification (“don’t care” specification)

root_list_ID: The *root_list_ID* specifies the root list of the descriptor hierarchy from which the entries will be selected. This narrows down the scope of where an entry with a given object ID is located.

selection_indicator: The *selection_indicator* field is described in the table below. For the don’t care specification, the *target_depth* field shall be FF_{16}.

Table 9.49 – Selection_indicator for the “don’t care” specification

selection_ indicator	Meaning
1xxx xxxx	**descriptor_specifier_type_flag:** When set to 1, this indicates that the specifiers in the *target* are by list type and object ID. When it is 0, it indicates that the specifiers in the target are by list ID and entry position.
xxxx xxx1	**target_format_flag:** This flag indicates the format of the *target* field. When set to 1, this flag indicates that the target is to be selected using specified CHILDREN of that entry. When this flag is zero, then the entire entry is to be selected (no specific CHILDREN are specified). Please see the example selections for details.
all others	reserved for future specification

When the *descriptor_specifier_type_flag* is zero, then the *target* field shall have the following format:

offset	bytes	msb							lsb
00_{16}	2	list ID							
:									
:	see[1]	target_object_reference							
:		(entry_position)							
:	1	Number_of_children (m)							
:									
:	see[1]	child_object_reference[0]							
:		(entry_position)							
:		:							
:									
:	see[1]	child_object_reference[m - 1]							
:		(entry_position)							

[1] See section 8.2.4, "Entry descriptor specified by position in its list" on page 52.

Figure 9.58 – Target field ("don't care" specification) when descriptor_specifier_type_flag = 0

In the above diagram, the fields from *number_of_children* through *child_object_reference[m - 1]* exist only if the *target_format_flag* is set to one.

When the *specifer_type_flag* is one, then the target field shall have the following format:

offset	bytes	msb							lsb
00_{16}	1	list_type							
01_{16}		target_object_reference							
:	see[1]								
:		(object_ID)							
:	1	Number_of_children (m)							
:									
:	see[1]	child_object_reference[0]							
:		(object_ID)							
:		:							
:									
:	see[1]	child_object_reference[m - 1]							
:		(object_ID)							

[1] See section 8.2.7, "Entry descriptor specified only by object_ID" on page 54.

Figure 9.59 – Target field ("don't care" specification) when descriptor_specifier_type_flag = 1

In the above diagram, the fields from *number_of_children* through *child_object_reference[m - 1]* exist only if the *target_format_flag* is set to one.

9.11.1.5 OBJECT NUMBER SELECT control command responses

For all responses, the response frame shall consist only of the first three fields (up to the *status* field). All other fields of the CONTROL command frame shall not be returned. In case of the INTERIM or REJECTED responses, the contents of the *status* field shall be set to FF_{16}. The following table summarizes the commands and responses of this command.

Table 9.50 – Field values in the OBJECT NUMBER SELECT control command: REJECTED, INTERIM and ACCEPTED response frames

Fields	Command	Response		
		REJECTED	INTERIM	ACCEPTED
output/source_plug	See Table 9.46	←	←	←
subfunction	See Table 9.47	←	←	←
status	FF_{16}	←	←	see Table 9.51
number_of_ons_selection_ specifications (n)		N/A	N/A	N/A
ons_selection_ specifications [x]	Specific entry(s)	N/A	N/A	N/A

← means "same as the command frame"

In the ACCEPTED response frame, the status field shall be updated with the appropriate value as defined here:

Table 9.51 – STATUS value in ACCEPTED response frame

status	Meaning
00_{16}	The selection specification indicated a unique item, which was selected.
01_{16}	The selection specification was ambiguous, so the unit or subunit selected one (please refer to the "don't care" path specification description).
all others	reserved for future specification.

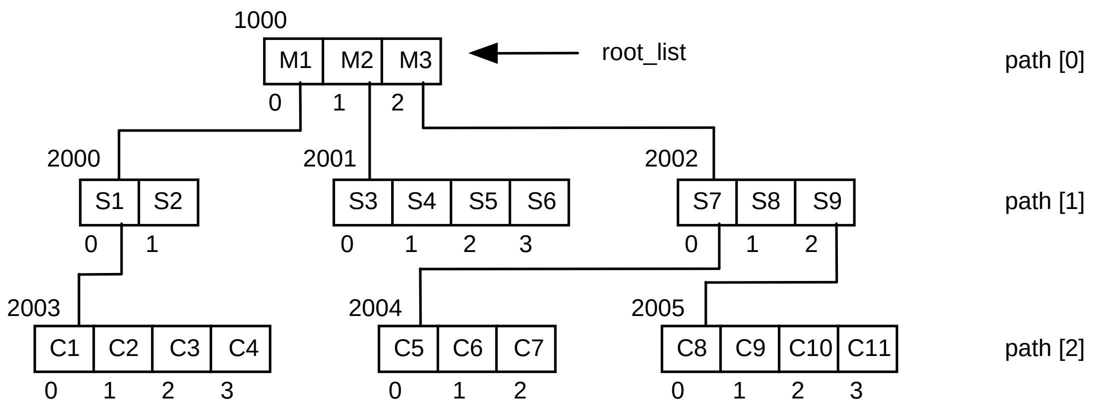
9.11.1.6 Object selection examples

The following diagrams illustrate various types of entry selection, as indicated by the *selection_indicator* described above.

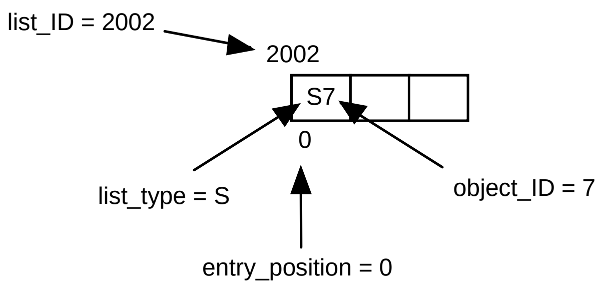
Example 1 shows the selection of an entire entry (S7) with the entry reference being the object ID. The results of the selection will be entry S7 composed of all three of its children (C5, C6 and C7).

Example 2 shows the selection of the same entry (S7), but this time it will be composed of only children C5 and C6.

These are the two ONS selection methods defined for the general AV/C list model. Specific types of subunits (such as tuners) may define additional selection methods.



Legend for this example:



Example 1: ons_selection_specification for the entry “S7”

		Notes
root_list_ID	10	list ID = 1000
	00	
selection_indicator	1xxx xxx0	specifier_type flag = by object ID
		target_format_flag = without children
target_depth	01	
path [0]	00	path [0] (object ID = 3)
	00	
	00	
	03	
target	00	target (object ID = 7)
	00	
	00	
	07	

Example 2: ens_selection_specification for the entry "S7" using specified children

Field	Value	Notes
root_list_ID	10	list ID = 1000
	00	
selection_indicator	1xxx xxx1	reference_type flag = by object ID
		target_format_flag = using specified children
target_depth	01	
path [0]	00	(object ID = 3)
	00	
	00	
	03	
target	00	(object ID = 7)
	00	
	00	
	07	
number_of_children	02	the target consists of two children
child[0]	00	the object ID in the child list = 5
	00	
	00	
	05	
child [1]	00	the object ID in the child list = 6
	00	
	00	
	06	

In example 2, we specify how to create the desired entry by specifying a subset of its child entries (the entries from its child list). Note that when specifying the path, we stop at the target entry and do not go down to the child list level. Because there is only one child list for the target entry, there is no ambiguity; we then just specify which child entries to use from the implied child list. In this example, all references (including the child references) were defined using object ID.

9.11.1.7 Entry selection semantics

The semantics of selecting an entry from a list will vary based on the kind of subunit being controlled, the nature of the data relationships that are involved, and the details provided in the *entry_selection_reference* field. There are some general rules defined for all lists in the AV/C model, and specific types of subunits may add further definitions.

9.11.2 OBJECT NUMBER SELECT status command

OBJECT NUMBER SELECT command may also be used with a ctype of STATUS, in which case the *ons_selection_specifications* of the currently selected information instances on the specified source plug are returned. The format of the OBJECT NUMBER SELECT status command is shown by the figure below:

	bytes	ck	msb							lsb
opcode	1	√	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	1	√	source_plug							
operand[1]	1	√	status = FF_{16}							

Figure 9.60 – OBJECT NUMBER SELECT status command frame

9.11.2.1 Field definitions

source_plug: The *source_plug* operand indicates the subunit plug number for which status is being requested.

9.11.2.2 OBJECT NUMBER SELECT status Responses

If the subunit is able to return a STABLE response to the OBJECT NUMBER SELECT status command, the AV/C response frame has the format illustrated by the figure below:

	bytes	msb							lsb
opcode	1	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	1	source_plug							
operand[1]	1	Status							
operand[2]	1	number_of_ons_selection_specifications (n)							
operand[3]	see[1]	ons_selection_specification[0]							
:									
:									
:									
:		:							
:	see[1]	ons_selection_specification[n - 1]							
:									
:									
:									

[1] The length of this field is variable.

Figure 9.61 – OBJECT NUMBER SELECT status command: STABLE response frame

9.11.2.2.1 Field definitions

status: The *status* field describes the current situation of the specified entry(s). The meaning of this field will be subunit-type-specific. For details, please refer to the appropriate subunit-type-specific OBJECT NUMBER SELECT command description.

All other operands are as described for the control command.

NOTE — There is no *subfunction* field in the status command.

Only entries which have an entry in the list(s) can be returned. For some types of subunits, it is possible to direct data to a source plug by means other than using the ONS control command (for one example of this, please refer to the DIRECT SELECT INFORMATION TYPE command in the tuner subunit specification). In these cases, information about the ons_selection_specifications may not be returned by the ONS status command. The controller must use the appropriate mechanisms to retrieve status information about selections made via those other commands.

Table 9.52 – Field values in the OBJECT NUMBER SELECT status command: REJECTED, IN TRANSITION and STABLE response frames

Fields	Command	Response		
		REJECTED	IN TRANSITION	STABLE
source_plug	See Table 9.46	←	←	←
status	FF_{16}	←	See [1]	See [1]
number_of_ons_selection_ specifications	N/A	N/A		
ons_selection_specification[x]	N/A	N/A		

[1] Dependent on the unit or subunit-type specification.
← means "same as the command frame"

9.11.3 OBJECT NUMBER SELECT notify command

In addition, the OBJECT NUMBER SELECT notify command is also used so that the controller shall be notified when the output of the specified source plug has changed. If the source plug has new data directed to it using some means other than the ONS command, then a controller will not be notified. The CHANGED response notification from ONS will only occur when the ONS command has been used to change the source plug, or when the target must change the ONS selection(s) on the plug for internal reasons (for example, if the data runs out). The controller must request notification for other changes to the source plug using the appropriate mechanisms. The format of the OBJECT NUMBER SELECT notify command frame is shown by the figure below:

	bytes	ck	msb							lsb
opcode	1	√	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	1	√	source_plug							
operand[1]	1	√	FF_{16}							

Figure 9.62 – OBJECT NUMBER SELECT notify command frame

9.11.3.1 Field definitions

source_plug: The *source_plug* has the same meaning as in the STATUS command.

9.11.3.2 OBJECT NUMBER SELECT: NOTIFY responses

The format of associated INTERIM and CHANGED responses is the same as the OBJECT NUMBER SELECT status response described above.

Annexes

Annex A: Anatomy of AV/C descriptor (informative)

This clause describes an anatomy of AV/C descriptor and info block structures. Each descriptor or info block field contains either implicit or explicit length. If a controller does not have knowledge of a descriptor or info block’s contents, the fields’ explicit length information could be used to skip over the data, thus, allowing for backwards compatibility.

A controller that wishes to read or write to a descriptor must know about the format and content of the descriptor’s specific information. A controller can determine if it knows about the format and content of the descriptor’s (or info block’s) data by inspecting its type field. If the type field is recognized, it can expect a particular format for its specific information. Though a controller could parse through a descriptor that it doesn’t know about, it is unlikely that it will have any use for the information it contains.

A.1 Understanding descriptor and info block structure

AV/C descriptors were designed so that any legacy or future AV/C controller could navigate through descriptor structures and info blocks without parsing failures. A controller can know what to expect from a descriptor structure by reading the *generation_ID* in the (sub) unit identifier descriptor of the target. If the *generation_ID* of the target is greater than that of the controller, then the controller should expect data structures that contain some extended parts that it cannot understand. If the *generation_ID* is the same or less, then the controller can expect that it will understand the descriptors presented.

The following figure shows the internal hierarchical levels of a list descriptor containing entry descriptors and information blocks in the current unextended structure (*generation_ID* = 02_{16}). This figure shows the descriptor with length fields represented as lines that extend to the end of the blocks of data they represent. Entry descriptors, info blocks, and information fields are all represented at different levels in an info block hierarchy within the list descriptor.

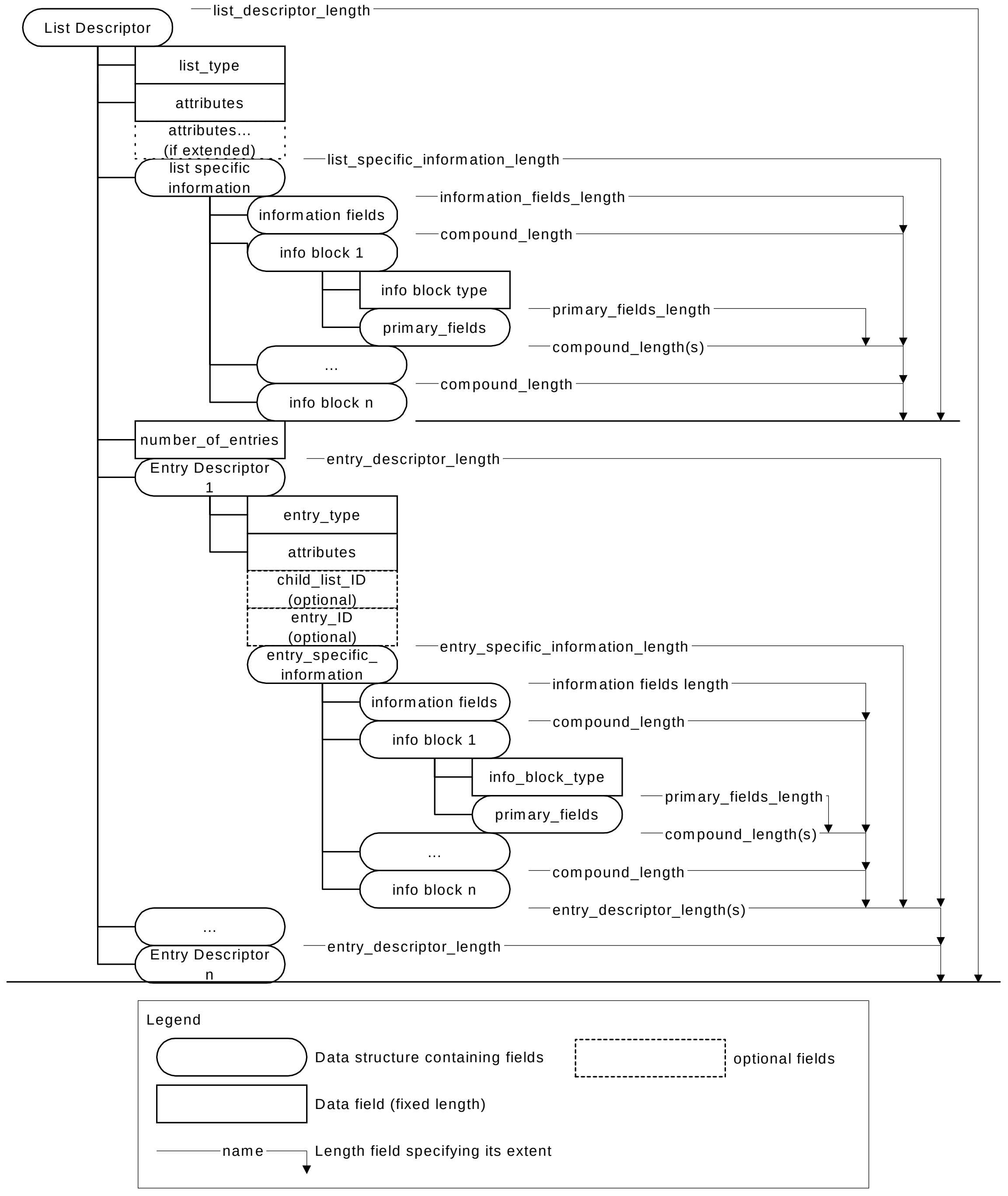


Figure A.1 – Structure of list descriptor with entries and info blocks

Note that the overall AV/C descriptor and info block data structure includes a mixture of fields with fixed length and data blocks with a variable length. Length fields precede all data blocks that have variable length. All AV/C controllers and targets should know about fixed-length data fields, since they were specified under *generation_ID* = 0.

Information block nesting is shown in the following figure.

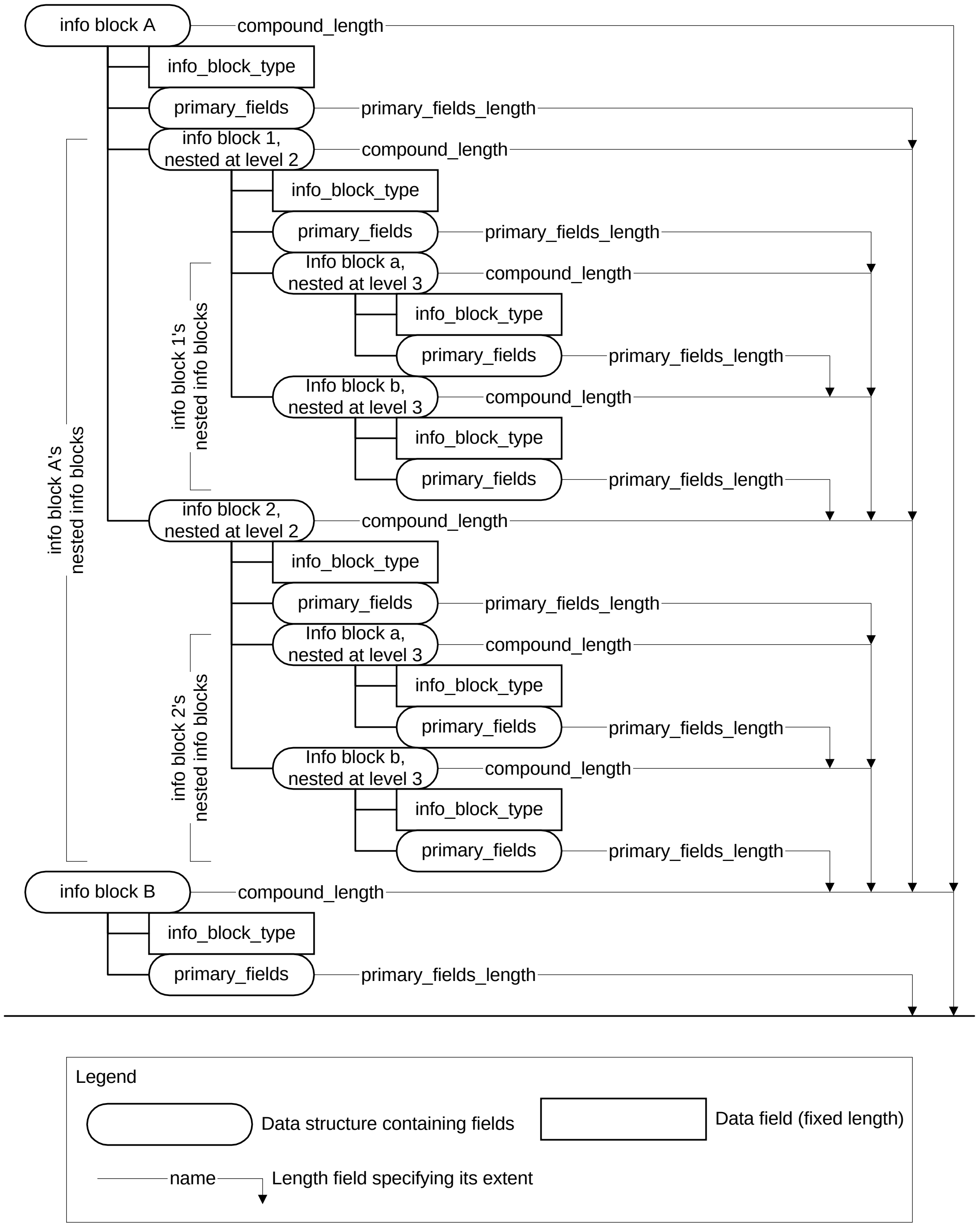


Figure A.2 – Structure of info block

The above figure shows three levels of nesting info blocks. At the final level – info blocks a and b under info blocks 1 and 2 –no *secondary_fields* exist. Info block B also does not contain *secondary_fields*.

The fields within the AV/C descriptor structures that are used for parsing descriptors and info blocks are described below.

A.1.1 Size_of fields

Before a controller accesses any descriptor in a unit or subunit, it will need to open the (sub)unit identifier descriptor, and read its *size_of* fields. These fields are used for accessing list and entry descriptors and are not needed for info blocks. The fields are as follows:

— **size_of_list_ID:** This field determines how many bytes the *child_list_ID* field has in entry descriptors that contain them, and the *root_list_ID* field has in the (sub)unit identifier descriptor.

— **size_of_object_ID:** This field determines how many bytes the *object_ID* field has in entry descriptors that contain them.

— **size_of_entry_position:** This field is used in commands and is not present in any of the descriptors.

The *size_of* fields are constant and are normally defined when the (sub)unit is manufactured.

A.1.2 Descriptor and info block length fields

The first field inside of each (sub)unit identifier descriptor, list descriptor, entry descriptor, and info block is its length (the length field is always two bytes). This length field refers to the compound length of the descriptor or info block, which may include any extended (in the case of descriptors) or *secondary_fields* (in the case of info blocks). By specifying the length fields, controllers know exactly how much data is in the descriptor or info block.

A.1.3 Fields-length fields

General descriptor structures and info blocks may contain fields with variable-length data. Preceding these variable-length fields is a fields-length field specifying their length. After the variable-length field changes in size, the unit or subunit that owns the descriptor or info block is responsible for updating the preceding fields-length field. Note that the fields-length field specifies the number of bytes for variable-length fields; the length value never includes itself in the calculation.

An example of this concept is the *manufacturer_dependent_information* field in the (sub)unit identifier descriptor. This field is preceded by the *manufacturer_dependent_information_length* field, which specifies the number of bytes used for the *manufacturer_dependent_information* field that follows.

A.1.4 Bit fields indicating the presence of other fields

Some bit fields may have bits indicating the presence of data in the descriptor or info block. The *attributes* fields in list descriptors and entry descriptors provide information on whether an entry descriptor contains *child_list_ID* and *object_ID* fields (see Table 7.2 – List descriptor attribute values on page 39). The bits are properly termed *has_child_ID* and *has_object_ID*.

Though these bits only indicate the presence of these fields, the lengths of these fields are defined in the (sub)unit identifier descriptor as *size_of_list_ID* and *size_of_object_ID*.

A.1.5 Number_of fields

Descriptors and info blocks may contain *number_of…* fields that help a controller determine the number of following fields or blocks. The following fields or blocks of data must either have a static known size or, if the size is not known, must be preceded by a length field. The *number_of...* fields are only valid for the descriptor they are found in. In the present AV/C descriptors, these fields are as follows:

— **number_of_root_lists:** This field exists in the (sub)unit identifier descriptor. A controller can use this field to prevent over-reading the following *root_list_ID* fields. The total number of bytes of root list data is equal to *number_of_root_lists* * *size_of_list_ID.*

— **number_of_entry_descriptors:** This field exists in the list descriptor. A controller can use this field to prevent from over-reading the list descriptor. The total number of bytes of entries is equal to the following formula:

$$\text{Total entry bytes} = \sum_{n=1}^{number\_of\_entry\_descriptors} entry\_descriptor\_length(n-1)$$

An info block may contain *number_of* fields as well. Refer to the specific info block for more information in references [R1] and [R10].

The location of the *number_of* fields must be known by a controller to exist based on the specified architecture.

A.2 Extending descriptors and info blocks

Due to technology advances, new fields may need to be added to descriptors and info blocks. Because info blocks are structured to be navigable and extensible, it is highly recommended that info blocks are used to extend all AV/C descriptors and even info blocks themselves.

The AV/C descriptor structure can be extended using info blocks in the following locations:

— Within the (sub)unit identifier descriptor’s extended fields area.

— Within the list or entry descriptor’s extended fields area.

— Within the list or entry descriptor’s *specific_information* fields.

— Within an info block’s *secondary_fields.*

The following figure shows the locations in a list descriptor structure where info blocks can be used to extend it. It also contains information about length fields and how they are associated with extended fields. Note that since entry descriptors are contained within list descriptors, this structure also serves to show where info block extensions apply to entry descriptors. Extended fields are shown in gray. The legend is the same as shown in Figure A.1 – Structure of list descriptor with entries and info blocks on page 138.

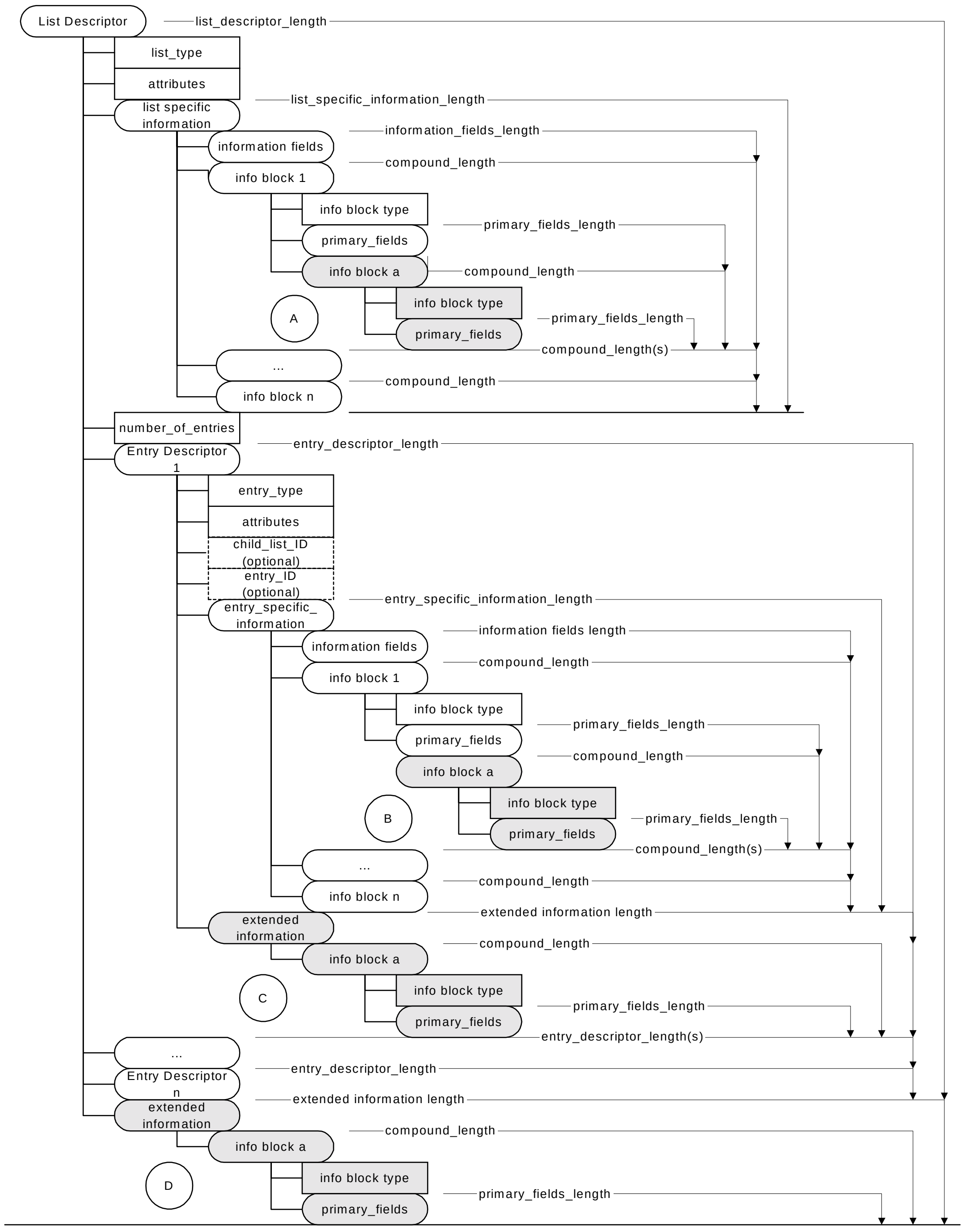


Figure A.3 – Extended structure of the list descriptor and its entries with info blocks

At point (A) in the figure above, nesting an info block within a pre-existing info block extends the *list_specific_information* fields.

At point Ⓑ in the figure above, nesting an info block within a pre-existing info block extends the *entry_specific_information* fields.

At point Ⓒ in the figure above, the entry descriptor is extended at the end of its structure using an info block.

At point Ⓓ in the figure above, the list descriptor is extended at the end of its structure using an info block.

Though a controller may read a greater *generation_ID* in a new unit or subunit (in the (sub)unit identifier descriptor of the target), this is no guarantee that all general descriptors in the unit or subunit are extended. If the target's *generation_ID* is greater than the controller's is, then the controller can only assume that there *may* be extended descriptors.

A legacy controller can know if a descriptor has been extended by comparing its first two-byte length field with the computed length of its known fields (using the parsing mechanisms described in clause A.1 "Understanding descriptor and info block" on page 137). If the computed length is less than the first length field, then the descriptor is extended. The legacy controller shall not assume that an extended descriptor is an error. Rather, it shall ignore the extended information.

Because extended fields may have variable lengths, the first field in the extended area must be a two-byte length field.

Information blocks are extended by nesting new information blocks in the information block's *secondary_fields*. Primary fields shall NOT be changed to accommodate new field definitions. Controllers that understand the info block model should be designed to expect any info block to occur at any time, and to not treat their appearance as an error.

A.2.1 Extending info block-aware structures

As with all of the AV/C descriptor structures, new info block-aware structures might be defined with general and specific areas. General areas are usually well defined and usually contain header information about the descriptor. Specific areas are then designed to appear after the general area, and should be composed of info blocks.

The overall strategy of placing well-defined fields at the front of a data structure and a variable number of info blocks at the end of a data structure applies to both the general area as well as the specific area(s). However, it is important to note that the specific area is generally considered to be part of the well-defined fields of the general area. As a result, the extra info blocks used to expand a specific area might seem to be nested "inside" of the well-defined fields of the general area. The following diagram illustrates how an info block-aware structure can be extended:

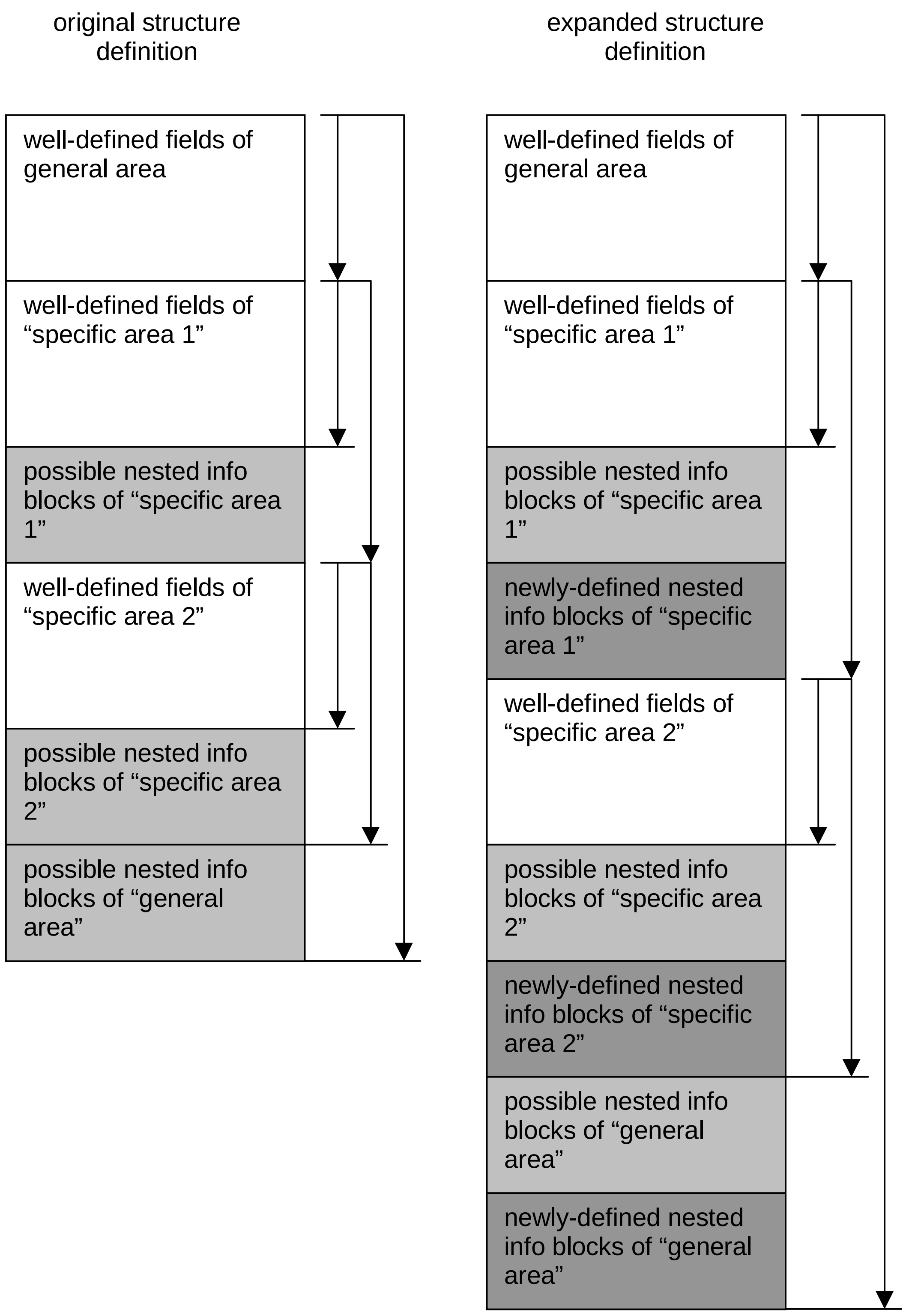

original structure definition
well-defined fields of general area
well-defined fields of “specific area 1”
possible nested info blocks of “specific area 1”
well-defined fields of “specific area 2”
possible nested info blocks of “specific area 2”
possible nested info blocks of “general area”
expanded structure definition
well-defined fields of general area
well-defined fields of “specific area 1”
possible nested info blocks of “specific area 1”
newly-defined nested info blocks of “specific area 1”
well-defined fields of “specific area 2”
possible nested info blocks of “specific area 2”
newly-defined nested info blocks of “specific area 2”
possible nested info blocks of “general area”
newly-defined nested info blocks of “general area”

Figure A.4 – Extending block-aware structures

The diagram shows a hypothetical structure that contains a general area, and two "specific" areas (for example, subunit-specific and media type-specific areas). Note that the general area is composed of well-defined fields at the beginning of the entire structure, and info blocks at the end of the entire structure. In the middle are the specific areas.

Each area has two length fields; the first indicates the overall size of that area (the general area indicates the total size of the structure), and the second indicates the size of the well-defined fields for that area.

Controllers can use this information to navigate through and/or around the various components of any data structure. Info blocks also have length information allowing the nesting of blocks within blocks, and still allowing controllers to safely navigate around those block types that are not understood.

The second part of the diagram shows how each of the areas can be expanded. The well-defined fields are NOT changed, but new info blocks (either mandatory or optional, depending on the definition) are added in the info block portion of each area.

Note that because the info block descriptor mechanism is designed to be quite flexible, all controllers should be prepared to find ANY info blocks in these areas. The diagram shows that new info blocks are added AFTER the existing info block definitions. Unless there are restrictions on info block placement, it is acceptable for newly-defined info blocks to appear in any order along with previously-defined info blocks.

Restrictions on info block placement might be imposed by the info block definitions; by unit or subunit type restrictions; or by media type restrictions. Of course, vendors are free to place info blocks in any order within the constraints of the technology restrictions.