1394 TRADE ASSOCIATION
THE MULTIMEDIA CONNECTION

AV/C Digital Interface Command Set General Specification

Version 3.0
April 15, 1998

Sponsored by:
Audio/Video Working Group of the 1394 Trade Association

Approved for Release by:
This document has been approved for release by the 1394 Trade Association Board of Directors

Abstract: This specification defines a command set for consumer and professional audio/video equipment over IEEE Std. 1394-1995. The command set makes use of the Function Control Protocol (FCP) defined by IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment, for the transport of audio/video command requests and responses. The audio/video devices are implemented as a common unit architecture within IEEE Std. 1394-1995.

Keywords: Audio, Video, 1394, Digital, Interface

1394 Trade Association
3925 W. Braker Lane, Austin, TX 78759 USA
http://www.1394TA.org

1394 Trade Association Specifications are developed within Working Groups of the 1394 Trade Association, a non-profit industry association devoted to the promotion of and growth of the market for IEEE 1394-compliant products. Participants in working groups serve voluntarily and without compensation from the Trade Association. Most participants represent member organizations of the 1394 Trade Association. The specifications developed within the working groups represent a consensus of the expertise represented by the participants.

Use of a 1394 Trade Association Specification is wholly voluntary. The existence of a 1394 Trade Association Specification is not meant to imply that there are not other ways to produce, test, measure, purchase, market or provide other goods and services related to the scope of the 1394 Trade Association Specification. Furthermore, the viewpoint expressed at the time a specification is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the specification. Users are cautioned to check to determine that they have the latest revision of any 1394 Trade Association Specification.

Comments for revision of 1394 Trade Association Specifications are welcome from any interested party, regardless of membership affiliation with the 1394 Trade Association. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally, questions may arise about the meaning of specifications in relationship to specific applications. When the need for interpretations is brought to the attention of the 1394 Trade Association, the Association will initiate action to prepare appropriate responses.

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

Editor, 1394 Trade Association
3925 W. Braker Lane
Austin, TX 78759
USA

1394 Trade Association Specifications are adopted by the 1394 Trade Association without regard to patents which may exist on articles, materials or processes, or to other proprietary intellectual property which may exist within a specification. Adoption of a specification by the 1394 Trade Association does not assume any liability to any patent owner or any obligation whatsoever to those parties who rely on the specification documents. Readers of this document are advised to make an independent determination regarding the existence of intellectual property rights which may be infringed by conformance to this specification.

Table of Contents

1. REFERENCES....................2

2. CHANGES FROM PREVIOUS VERSION....................3

3. DEFINITIONS AND ABBREVIATIONS....................4
3.1 Conformance glossary....................4
3.2 Technical glossary....................4

4. FUNCTION CONTROL PROTOCOL (INFORMATIVE)....................6

5. AV/C FRAMES....................7
5.1 AV/C command frame....................7
5.2 AV/C response frame....................7
5.3 AV/C frame components....................8
5.3.1 Command type (*ctype*)....................8
5.3.2 Response code (*response*)....................8
5.3.3 AV/C address (*subunit_type*, *subunit_ID*)....................9
5.3.4 Operation (*opcode*)....................11
5.3.5 Operands....................11
5.3.6 Subunit classification process....................11

6. AV/C OPERATIONS....................13

7. AV/C COMMANDS....................16
7.1 Support levels....................16
7.2 Control commands....................16
7.3 Status commands....................17
7.4 SPECIFIC INQUIRY commands....................18
7.5 Notify commands....................18
7.6 GENERAL INQUIRY commands....................19

8. AV/C DESCRIPTOR MECHANISM....................20
8.1 The General Subunit Identifier Descriptor....................22
8.2 Object Lists....................24
8.3 Object Entries....................26
8.4 Object References....................29
8.4.1 Object Position Reference....................30
8.4.2 Object ID Reference....................30
8.5 Parsing and Navigating the General AV/C Descriptor Structures (Informative)....................30
8.5.1 Endian-ness....................30
8.5.2 Variable Length Fields Within Data Structures....................30
8.5.3 The Length of Object Entry Descriptors....................32
8.5.4 Extended Data Structures....................32
8.6 Rules for Reserved Fields....................34

9. UNIT COMMANDS....................35
9.1 CHANNEL USAGE command....................35
9.2 CONNECT command....................37
9.3 CONNECT AV command....................40
9.4 CONNECTIONS command....................42
9.5 DIGITAL INPUT command....................43
9.6 DIGITAL OUTPUT command....................43
9.7 DISCONNECT command....................43
9.8 DISCONNECT AV command....................44

9.9 INPUT PLUG SIGNAL FORMAT command....44
9.10 OUTPUT PLUG SIGNAL FORMAT command....45
9.11 SUBUNIT INFO command....46
9.12 UNIT INFO command....47

10. COMMON UNIT AND SUBUNIT COMMANDS....49
10.1 OPEN DESCRIPTOR command....49
10.1.1 Subunit Identifier....50
10.1.2 Object List....50
10.1.3 Object Entry....51
10.1.4 Descriptor Access Support....52
10.1.5 The OPEN DESCRIPTOR Status Command....53
10.2 READ DESCRIPTOR command....55
10.3 WRITE DESCRIPTOR command....56
10.3.1 Partial Replace Operations....59
10.3.2 WRITE DESCRIPTOR Status....60
10.4 SEARCH DESCRIPTOR command....61
10.4.1 *type_specific_info* for the *search_in* operand....66
10.4.2 *type_specific_info* for the *start_point* operand....69
10.4.3 Examples of the SEARCH DESCRIPTOR Command (Informative)....71
10.5 OBJECT NUMBER SELECT command....73
10.5.1 Subfunction Implementation Rules....75
10.5.2 The General ons_selection_specification Structure....75
10.5.3 The “Don’t Care” Specification....76
10.5.4 Object Selection Examples....78
10.5.5 Object Selection Semantics....80
10.5.6 The OBJECT NUMBER SELECT Status Command....81
10.6 POWER command....82
10.7 RESERVE command....82
10.8 PLUG INFO command....84
10.9 VENDOR-DEPENDENT commands....85

A. AV/C COMMANDS IN NUMERICAL ORDER (NORMATIVE)....87

B. UNRESOLVED ISSUES (INFORMATIVE)....88

List of Figures

FIGURE 4-1 — FCP FRAME WITHIN A SERIAL BUS PACKET....6
FIGURE 5-1 — AV/C COMMAND FRAME....7
FIGURE 5-2 — AV/C RESPONSE FRAME....7
FIGURE 5-3 — AV/C COMMAND FRAME WITH EXTENDED TYPE AND ID ADDRESSES....10
FIGURE 6-1 — AV/C IMMEDIATE TRANSACTION....13
FIGURE 6-2 — AV/C DEFERRED TRANSACTION....14
FIGURE 9-1 — CHANNEL USAGE STATUS COMMAND FORMAT....36
FIGURE 9-2 — CHANNEL USAGE STATUS RESPONSE FORMAT....36
FIGURE 9-3 — CONNECT CONTROL COMMAND....37
FIGURE 9-4 — CONNECT CONTROL COMMAND WITH EXTENDED SUBUNIT_TYPE AND EXTENDED SUBUNIT_ID....38
FIGURE 9-5 — CONNECT STATUS COMMAND FORMAT FOR A SOURCE PLUG....39
FIGURE 9-6 — CONNECT STATUS COMMAND FORMAT FOR A DESTINATION PLUG....39
FIGURE 9-7 — CONNECT AV CONTROL COMMAND FORMAT FOR AUDIO/VIDEO STREAM....40
FIGURE 9-8 — CONNECT AV STATUS COMMAND FORMAT FOR AUDIO/VIDEO STREAM....41
FIGURE 9-9 — CONNECTIONS STATUS COMMAND FORMAT....42
FIGURE 9-10 — CONNECTIONS RESPONSE FORMAT....42
FIGURE 9-11 — DIGITAL INPUT COMMAND FORMAT....43

FIGURE 9-12 — DIGITAL OUTPUT COMMAND FORMAT 43
FIGURE 9-13 — DISCONNECT COMMAND FORMAT 44
FIGURE 9-14 — DISCONNECT AV COMMAND FORMAT.......... 44
FIGURE 9-15 — INPUT PLUG SIGNAL FORMAT CONTROL COMMAND FORMAT.......... 44
FIGURE 9-16 — INPUT PLUG SIGNAL FORMAT STATUS COMMAND FORMAT 45
FIGURE 9-17 — OUTPUT PLUG SIGNAL FORMAT CONTROL COMMAND FORMAT 45
FIGURE 9-18 — OUTPUT PLUG SIGNAL FORMAT STATUS COMMAND FORMAT.......... 46
FIGURE 9-19 — SUBUNIT INFO STATUS COMMAND FORMAT.......... 46
FIGURE 9-20 — SUBUNIT INFO RESPONSE FORMAT.......... 47
FIGURE 9-21 — SUBUNIT TABLE ENTRY.......... 47
FIGURE 9-22 — UNIT INFO STATUS COMMAND FORMAT 47
FIGURE 9-23 — UNIT INFO RESPONSE FORMAT 48
FIGURE 10-1 — POWER COMMAND FORMAT.......... 82
FIGURE 10-2 — RESERVE CONTROL COMMAND FORMAT 83
FIGURE 10-3 — RESERVE STATUS COMMAND FORMAT.......... 84
FIGURE 10-4 — PLUG INFO STATUS COMMAND FORMAT 84
FIGURE 10-5 — PLUG INFO STATUS RESPONSE FORMAT FROM AN AV SUBUNIT 85
FIGURE 10-6 — PLUG INFO RESPONSE FORMAT FROM AN AV UNIT.......... 85
FIGURE 10-7 — VENDOR-DEPENDENT COMMAND FORMAT 85

List of Tables

TABLE 5.3-1 — SUBUNIT TYPE ENCODING.......... 9
TABLE 5.3-2 — SUBUNIT ID ENCODING.......... 9
TABLE 8.6-1 — U NIT COMMANDS 35
TABLE 9.2-1 — SERIAL BUS AND EXTERNAL PLUG NUMBERS 38
TABLE 9.12-1 — COMMON UNIT AND SUBUNIT COMMANDS 49

Preface

This document specifies a command set used to control consumer electronic audio/video equipment. The command set builds upon an extensive body of standards work, underway and completed, as referenced in section 1. Serial Bus, an IEEE standard, is the digital interface used to transport commands from controllers to AV devices (targets) and to return responses to the controllers. The unit architectures of these AV devices are defined within the scope of the configuration ROM and CSR architecture standardized by ISO/IEC. The commands themselves are encapsulated within a generic Function Control Protocol (FCP) developed by the HD Digital VCR Conference and now part of the IEC 61883 standard. Similarly, the format of the isochronous data itself has been developed by the HD Digital VCR Conference.

This specification concerns itself narrowly with the syntax and semantics of a general set of commands transmitted by controllers to AV devices and the resultant actions that occur at the AV device. The suite of AV/C documentation has been separated into this general AV/C specification document and separate documents for each type of subunit (VCR, Tuner, Disc, etc.). The reader is strongly encouraged to read this document in conjunction with the references given below, as well as with any AV/C-related documents which may be created in the future.

1. References

AV/C Master Index: Guide to AV/C Specification Documents - this document is available on the 1394 Trade Association web site noted above, and is kept up to date with the latest released versions of AV/C specifications. The reader is encouraged to always consult this document for information on the latest versions of specifications mentioned here, as well as specifications which may be developed in the future.

AV/C Digital Interface Command Set for VCR Subunit Specification, version 2.0.1, January 5, 1998

AV/C Tuner Model Working Specification Version 1.0W

AV/C Tuner Model and Command Set Version 1.0

IEEE Std 1394–1995, Standard for a High Performance Serial Bus

IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment

ISO/IEC 13123:1994, Control and Status Register (CSR) Architecture for Microcomputer Buses

HD Digital VCR Conference, Specifications of Consumer-Use Digital VCR's using 6.3 mm magnetic tape (December 1995)

2. Changes from previous version

Version 3.0 of this document differs from version 2.0.1 in the following ways:

- The AV/C Descriptor Mechanism chapter was added

- The OPEN DESCRIPTOR, READ DESCRIPTOR, WRITE DESCRIPTOR, SEARCH DESCRIPTOR and OBJECT NUMBER SELECT commands were added

Version 2.0.1 of this document differs from version 2.0 in the following ways:

- The AV/C Digital Interface Command Set 2.0 manual was separated into two books: General Specification and the VCR Subunit Specification, each assigned version 2.0.1

Version 2.0 of this document differs from version 1.0 in the following ways:

- Table 5.3-1 — Subunit type encoding - the disc recorder/player type has been added, and the subunit type 05 has been changed to “Tuner” from “TV Tuner”.

- A model for extended subunit addressing has been defined in section 5.3.3. As a result, item C3 in Annex C (extended subunit addressing) has been removed (what was item C4 - Notification Support - is now item C3).

- A process for defining new device types and command sets has been defined in section 5.3.6.

- The ctype GENERAL INQUIRY (value 4) was added. This allows a controller to ask a target “do you support this opcode?” without passing any specific operands.

- The original ctype INQUIRY (value 2) was renamed SPECIFIC INQUIRY, to indicate that a set of operands must be supplied along with the opcode when issuing the command.

3. Definitions and abbreviations

3.1 Conformance glossary

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

expected: A keyword 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.

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

shall: A keyword indicating a mandatory requirement. Designers are required to implement all such mandatory requirements to ensure interoperability with other products conforming to this specification.

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

3.2 Technical glossary

ATN: A sequential reference number recorded as part of each track of a DVCR cassette. Within the context of a single, uninterrupted recording session, these reference numbers are monotonically increasing and, in that sense, *absolute track numbers*. However, if the medium has been recorded at different times there may be gaps between different recorded areas and there is no guarantee of relationship between the absolute track numbers in one area and those in another.

AV unit: The physical instantiation of a consumer electronic device, *e.g.*, a camcorder or a VCR, within a Serial Bus node. This document describes a command set that is part of the software unit architecture for AV units.

AV subunit: an instantiation of a virtual entity that can be identified uniquely within an AV unit and offers a set of coherent functions.

AV/C: Audio/video control, as in the AV/C Digital Interface Command Set specified by this document.

byte: Eight bits of data.

CSR: A node or unit Control and Status Register, as defined by IEEE Std 1394–1995.

DVCR: Digital video cassette recorder as defined by the HD Digital VCR Conference, Specifications of Consumer-Use Digital VCR's using 6.3 mm magnetic tape.

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.

FCP: Function Control Protocol, as defined by IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment.

IEEE: The Institute of Electrical and Electronics Engineers, Inc.

isochronous: A term that indicates the essential characteristic of a time-scale or signal, such that the time intervals between consecutive instances either have the same duration or durations that are integral multiples of the shortest duration. In the context of Serial Bus, “isochronous” is taken to mean a bounded worst-case latency for the transmission of data; physical and logical constraints that introduce jitter preclude the exact definition of “isochronous."

MIC: An acronym for memory in cassette, a feature of DVCR cassettes that provides a limited amount of nonvolatile memory that may be used for any purpose. Standard MIC formats have been specified by the HD Digital VCR Conference.

module: The smallest component of physical management, *i.e.*, a replaceable device.

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

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

node ID: A 16-bit number, unique within the context of an interconnected group of Serial Buses. 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).

PCR: Plug Control Register, as defined by IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment.

iPCR: Input plug PCR, as defined by IEC 61883.

oPCR: Output plug PCR, as defined by IEC 61883.

plug: A physical or virtual end-point of connection implemented by an AV unit or subunit that may receive or transmit isochronous or other data. Plugs may be Serial Bus plugs, accessible through the PCR’s; they may be external, physical plugs on the AV unit; or they may be internal virtual plugs implemented by the AV subunits.

quadlet: Four bytes of data.

Serial Bus: The physical interconnects and higher level protocols for the peer-to-peer transport of serial data, as defined by IEEE Std 1394–1995.

SMPTE/EBU time code**:**
Time code format for professional use.

stream: A time-ordered set of digital data originating from one source and terminating at zero or more sinks. A stream is characterized by bounded bandwidth requirements and by synchronization points, or time stamps, within the stream data.

unit architecture: The formal specification of the format and function of the software-visible resources and behaviors of a class of units. This document, in conjunction with the references above, defines a unit architecture for the class of AV devices.

4. Function Control Protocol (informative)

The AV/C commands and responses are transported by the Function Control Protocol (FCP) defined by IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment. FCP provides a simple means to encapsulate device commands and responses within IEEE Std 1394–1995 asynchronous block write transactions. The format of an FCP frame, encapsulated within a Serial Bus block write packet, is illustrated below in Figure 4-1.

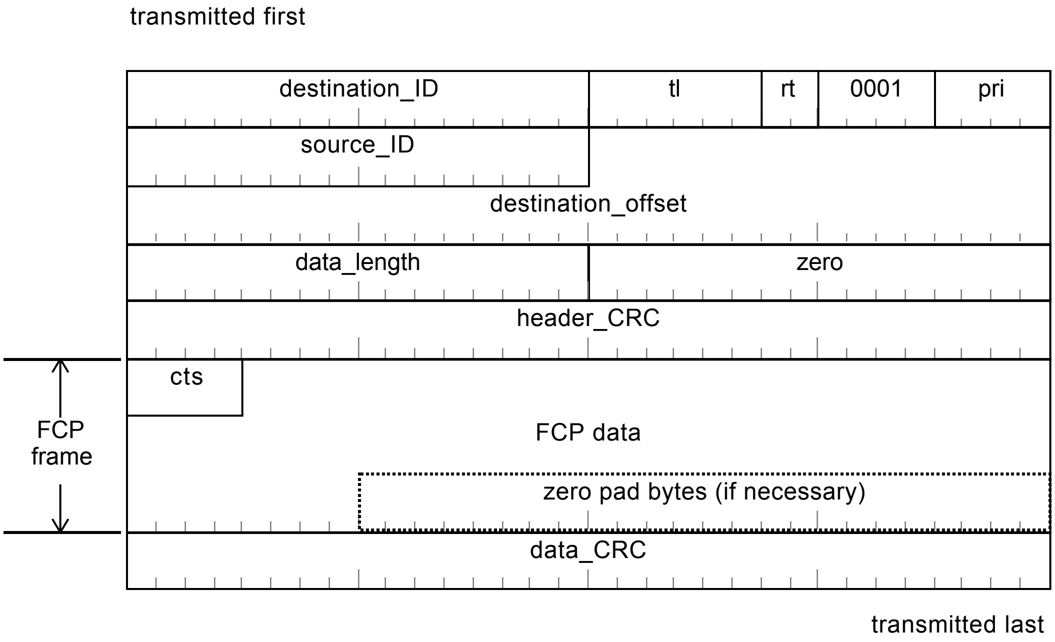


Figure 4-1 — FCP frame within a Serial Bus packet

The ***destination_ID***, ***tl***, ***rt***, ***tcode*** (write request for data block, 0001_2), ***pri***, ***source_ID***, ***data_length*** and CRC fields are as defined by IEEE Std 1394–1995.

The ***cts*** field defines the command transaction format used by the FCP frame. For the AV/C commands defined by this document, the *cts* field shall be zero.

Commands originated by a device at a Serial Bus node, the controller, are addressed to the FCP_COMMAND register, ***destination_offset*** FFFF F000 $0B00_{16}$ at the Serial Bus node that contains the device to be controlled, the target. The remotely controlled device in turn returns its response(s) to the FCP_RESPONSE register, *destination_offset* FFFF F000 $0D00_{16}$, at the controller.

The data payload of both FCP request and response packets, specified by ***data_length***, is limited to a maximum of 512 bytes.

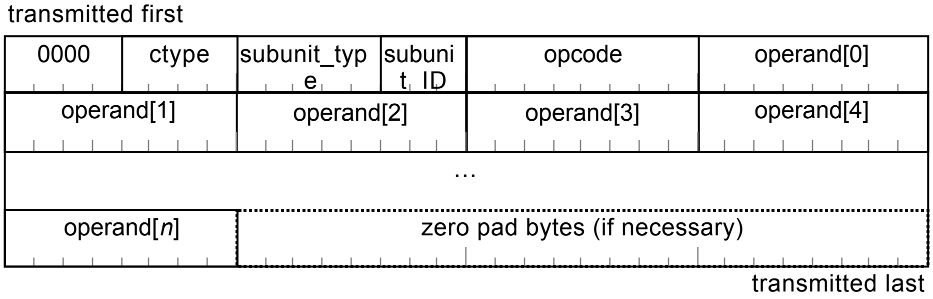
> **NOTE:** If the size of an FCP frame is exactly four bytes, a Serial Bus quadlet write transaction shall be used to transmit the data instead of the block write packet illustrated above.

5. AV/C frames

AV/C commands and responses are encapsulated within FCP frames, as described above, and are transmitted between the controller and target FCP_COMMAND and FCP_RESPONSE registers. The format of both the AV/C command and the AV/C response frames are similar, as described in the clauses that follow.

5.1 AV/C command frame

An AV/C command frame is up to 512 bytes of command payload with the structure shown in Figure 5-1 below:



transmitted first

<table>
<tr><td>0000</td><td>ctype</td><td>subunit_type</td><td>subunit_ID</td><td>opcode</td><td>operand[0]</td></tr>
<tr><td colspan="2">operand[1]</td><td colspan="2">operand[2]</td><td>operand[3]</td><td>operand[4]</td></tr>
<tr><td colspan="6">…</td></tr>
<tr><td colspan="2">operand[<i>n</i>]</td><td colspan="4">zero pad bytes (if necessary)</td></tr>
</table>

transmitted last

Figure 5-1 — AV/C command frame

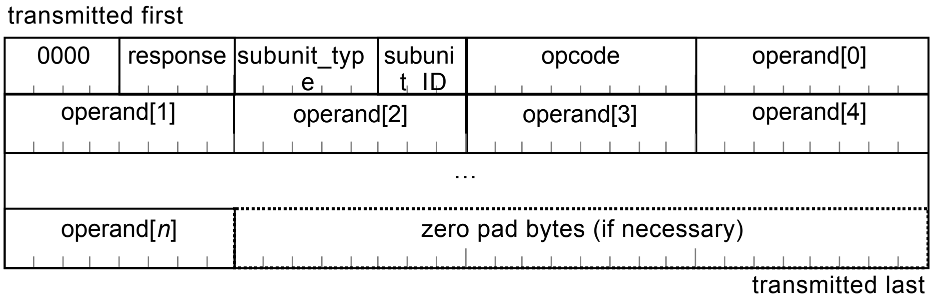
All of the operands, up to a maximum permitted by the overall payload limit of 512 bytes, are optional and are defined by *ctype*, *subunit_type* and *opcode*.

> **NOTE:** If an AV/C command frame exceeds the maximum capacity of an AV unit or subunit to which it is addressed, it may be ignored.

The *subunit_type* and *subunit_ID* fields form an AV/C address which identifies the destination of the AV/C command frame and the source of the AV/C response frame. If either the subunit type or subunit ID values have been extended, then there will be additional bytes used before the opcode byte. Please refer to the section titled “AV/C address (subunit_type, *subunit_ID)*” for details.

5.2 AV/C response frame

An AV/C response frame is up to 512 bytes of response payload with the structure shown in Figure 5-2 below:



transmitted first

<table>
<tr><td>0000</td><td>response</td><td>subunit_type</td><td>subunit_ID</td><td>opcode</td><td>operand[0]</td></tr>
<tr><td colspan="2">operand[1]</td><td colspan="2">operand[2]</td><td>operand[3]</td><td>operand[4]</td></tr>
<tr><td colspan="6">…</td></tr>
<tr><td colspan="2">operand[<i>n</i>]</td><td colspan="4">zero pad bytes (if necessary)</td></tr>
</table>

transmitted last

Figure 5-2 — AV/C response frame

All of the operands, up to a maximum permitted by the overall payload limit of 512 bytes, are optional and are defined by *response*, *subunit_type* and *opcode*.

The *subunit_type* and *subunit_ID* fields form an AV/C address which identifies the source of the responding AV/C entity and equals the destination to which the corresponding AV/C command frame was sent. As with the command frame, it is possible that the subunit type and/or subunit ID have been extended, thus requiring more bytes before the opcode field.

5.3 AV/C frame components

The component fields and code values for AV/C command and response frames are defined in this clause.

Except as otherwise indicated, reserved codes and fields within an AV/C frame are reserved for future specification. All reserved fields shall be set to zero by the sender of the AV/C frame. The sender shall not use reserved or invalid values for any components of an AV/C frame.

Responses to reserved or invalid codes and fields are defined in section 6.

5.3.1 Command type (*ctype*)

The 4-bit command type, ctype, defines one of five types of commands, as defined by the table below:

Value	Command type
0	CONTROL
1	STATUS
2	SPECIFIC INQUIRY
3	NOTIFY
4	GENERAL INQUIRY
5 – 7	Reserved for future specification
8 – F_{16}	Reserved for response codes

5.3.2 Response code (*response*)

The 4-bit response code, ***response***, defines one of seven types of response, as defined by the following table:

Value	Response
0 – 7	Reserved for command types
8	NOT IMPLEMENTED
9	ACCEPTED
A_{16}	REJECTED
B_{16}	IN TRANSITION
C_{16}	IMPLEMENTED / STABLE
D_{16}	CHANGED
E_{16}	Reserved for future specification
F_{16}	INTERIM

5.3.3 AV/C address (*subunit_type*, *subunit_ID*)

Taken together, the ***subunit_type*** and ***subunit_ID*** fields define the address of the recipient of the command or the source of the response. Version 1.0 of the AV/C specification limited subunit addressing to 32 subunit types and 5 subunits of a given type within a unit (refer to Table 5.3-1 and Table 5.3-2 of the original 1.0 specification for details). To allow for growth beyond these limitations, a backward compatible model for an *extended subunit address* has been devised using previously reserved subunit_type and subunit_ID values. The following tables illustrate the new definitions:

Subunit type	Meaning
0	Video monitor
1 – 2	Reserved for future specification
3	disc recorder/player (audio or video)
4	tape recorder/player (audio or video)
5	Tuner
6	Reserved for future specification
7	Video camera
8 – $1B_{16}$	Reserved for future specification
$1C_{16}$	Vendor unique
$1D_{16}$	Reserved for all subunit types
$1E_{16}$	subunit_type extended to next byte
$1F_{16}$	Unit

Table 5.3-1 — Subunit type encoding

Subunit ID	Meaning
0 - 4	Instance number
5	subunit_ID extended to next byte
6	Reserved for all instances
7	Ignore

Table 5.3-2 — Subunit ID encoding

An AV/C address with subunit_type value $1F_{16}$ and subunit_ID value 7 addresses the complete AV unit instead of one of its subunits. The combinations of subunit_type value $1F_{16}$ and subunit_ID values 0 through 6 are reserved.

If the subunit_type value is not equal to $1F_{16}$, the subunit_ID indicates the ordinal of the subunit as indicated by subunit_type. In this case, the subunit_ID commences at zero and is consecutively numbered up to the total instances less one.

IMPORTANT: The subunit_ID value specified in *extended* fields shall be equal to the exact instance number, NOT less one. This is required because there are restrictions on the value

0 in extended address fields. Please refer to the tables below, which define the meaning of extended subunit_type and subunit_ID fields.

The subunit_ID field can be used as a normal instance number in the case where an extended subunit_type is specified. This gives an identical numbering scheme for extended subunit_types and normal subunit_types.

If extended addressing is used, each extra byte is either completely used for an extended subunit_type or an extended subunit_ID. This differs from the normal subunit address byte, in which both the type and ID are specified within a single byte.

If byte n in an AV/C frame contains an AV/C address value that indicates that both the subunit-type and subunit-ID are extended to the next byte, the subunit_type is extended using byte n+1 of the AV/C frame and the subunit-ID is extended using byte n+2. Compare Figure 5-1 to the following diagram of an AV/C command frame with extended type and ID addresses:

transmitted first

0000	ctype	subunit_type	subunit_ID	extended subunit_type	extended subunit_ID
opcode		operand[0]		operand[1]	operand[2]
...				...	
operand[n]		zero pad bytes (if necessary)			

transmitted last

Figure 5-3 — AV/C command frame with extended type and ID addresses

The following tables illustrate the definitions for extended subunit_type and subunit_ID values:

Extended subunit_type value	Meaning
0	reserved for future specification
1..FE_{16}	extended subunit_type
FF_{16}	extended subunit_type extended to next byte

Extended subunit_ID value	Meaning
0	reserved for future specification
1..FE_{16}	extended subunit_ID
FF_{16}	extended subunit_ID extended to next byte

Note that by using the value FF_{16} in the extended type or ID fields, it is possible to specify an unlimited number of subunit types or ID's of a given type.

Extended subunit_ID values continue counting where the previous (extended) subunit_ID stopped. For example, if 3 bytes (one normal and two extended) are used for the subunit_ID, the value 1 in the third byte addresses instance number: 5 + 254 + 1 = the 260th instance. Note that the actual entries in the fields would be 5, FF_{16}, 1.

In this example, the instance number 5 is derived from the normal subunit_ID field entry of 5 which indicates that this is an extended ID. The instance number is indicated by the highest legitimate ID for that field. In the case of the normal address field, the highest ID value is 5. For extended ID fields, it is FE_{16}, or 254. Remember that the ID entries in the normal subunit_ID field are equal to the “instance number - 1” and that the entries in all of the extended fields are equal to the “instance number”.

5.3.4 Operation (*opcode*)

Within the five types of AV/C commands, CONTROL, STATUS, SPECIFIC INQUIRY, NOTIFY and GENERAL INQUIRY, the ***opcode*** field defines the operation to be performed or the status to be returned. The permissible values of *opcode* are divided into ranges valid for commands addressed to AV subunits, AV units or both, as follows.

Value	Addressing mode
0 – F_{16}	Unit and subunit commands
10_{16} – $3F_{16}$	Unit commands
40_{16} – $7F_{16}$	Subunit commands
80_{16} – $9F_{16}$	Reserved for future specification
$A0_{16}$ – BF_{16}	Unit and subunit commands
$C0_{16}$ – DF_{16}	Subunit commands
EE_{16} – FF_{16}	Reserved for future specification

5.3.5 Operands

The number and meaning of the ***operand[n]*** fields are determined by the *ctype*, *subunit_type* and *opcode* fields, as defined in clauses that follow.

5.3.6 Subunit classification process

The AV/C protocol has been designed for future growth, to accommodate the creation of new types of products that were not envisioned when the protocol was originally developed. When a manufacturer is designing such a new piece of equipment, the following guidelines should be used for determining if the device falls into an existing category (as defined in Table 5.3-1), or if a new type needs to be defined. Note that new device types may require modifications to existing commands or the creation of entirely new commands.

The basic approach to type classification is a two step process:

1) Examine the MAIN functionality of the subunit, in terms of the following:

- transport mechanism - does it have one?

- signal input - is the usefulness of this subunit defined mostly by the fact that a signal ends up here (regardless of the fact that it may be propagated without changes)?

- signal output - is this subunit a signal source?

- signal processing - accepts input, performs some sort of processing, and then outputs modified data

- no signal input or output - a utility of some kind

2) If a set of commands do not apply equally to audio or video data, then decompose the subunit type into separate audio and video categories.

While many subunits may have both input and output signals, the important item to consider is the MAIN functionality - in other words, what is the purpose of this subunit? The main purpose of a video camera subunit is to capture data through its lens and send that signal somewhere - it's a signal source. The main purpose of a television monitor is for viewing the input signal - it's a signal input or destination.

A utility such as a timer or a mechanism that can pan/tilt a camera does not deal with signal input or output, but it may be part of a controllable subunit.

When selecting a new type value and the appropriate command set, the following guidelines should be followed:

1) Find an unused subunit type value from the table of pre-defined types (Table 5.3-1).

2) Select only the specified new subunit type from the table; other codes for unused types must remain reserved.

3) Define a (relatively) complete set of commands for this new type. This step includes the definition of new commands that are unique to this type, as well as the verification that existing commands (where applicable) will work as defined. New devices that have similar functionality to existing devices should map their control features to the existing commands.

6. AV/C operations

AV/C commands transmitted by a controller and the associated response(s) returned by the target are called an AV/C transaction. An AV/C transaction consists of one AV/C command frame addressed to the target's FCP_COMMAND register and zero or more AV/C response frames addressed to the controller's FCP_RESPONSE register. Unless stated otherwise within individual command descriptions, it is assumed that at least one response will be returned.

The target's node_ID identifies either a specific AV unit or it identifies all AV units (broadcast). Unless stated otherwise within individual command descriptions, it is assumed that a single AV unit is addressed by the command. An example of a simple AV/C transaction, in which the target is able to complete the request before responding, is shown below in Figure 6-1.

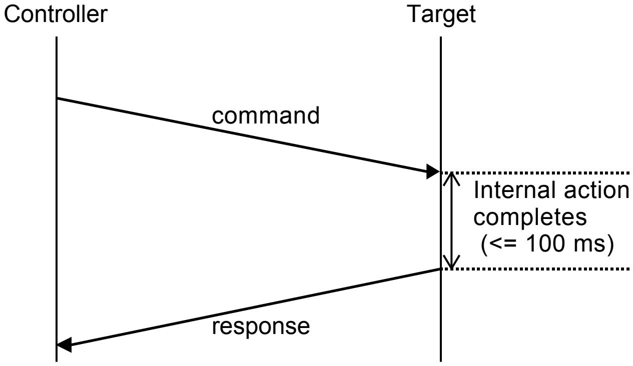


Figure 6-1 — AV/C immediate transaction

In an immediate transaction any response code, except INTERIM or CHANGED, is permitted. The transaction is complete when the target writes the AV/C response frame to the controller's FCP_RESPONSE register.

For some transactions the target may not be able to complete the request (or determine if it is possible to complete the request) within the 100 milliseconds allowed. In this case, the target shall return an initial response of INTERIM with the expectation that a final response will follow later. Figure 6-2 below illustrates an AV/C transaction with an intermediate response.

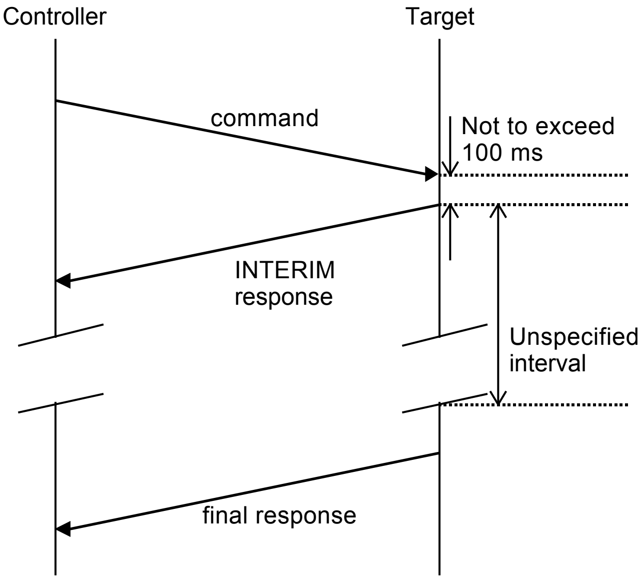


Figure 6-2 — AV/C deferred transaction

A target shall follow the following procedures when AV/C response frames are returned to the controller:

The target shall generate a response frame within 100 milliseconds of the receipt of the AV/C command frame. Targets should respond as quickly as possible.

If the AV/C command frame contains a reserved value in the *ctype* field, the target shall ignore the command and shall not generate a response frame.

If the target is already occupied with a previous command, it may ignore any AV/C command frames received. Note that the receipt of an AV/C command frame shall always be acknowledged by a target. The target ignores a command frame by a failure to return a response frame.

NOTE: A controller that does not receive a response frame for an AV/C command frame within 100 milliseconds may retry the command by resending the same command frame.

If the target is not occupied with a previous command, it shall create an AV/C response frame from the command frame by first copying all of the bytes of the command frame that precede *opcode* and then inserting the correct response code. The remainder of the response frame, *opcode* and operands, is dependent upon the *ctype*, *subunit_type* and *opcode* of the original command.

NOTE: Response frames returned after control commands are usually identical to the original command frame except for the *response* field. A response to a status or notify command typically has different *response* and *operand* fields and, in some cases, a different *opcode* field.

If the target receives a command frame whose *subunit_type* and *subunit_ID* fields address the command to a nonexistent subunit, the target shall return a response code of NOT IMPLEMENTED.

If the target is able to initiate the requested command in less than 100 milliseconds, it shall return a response code other than INTERIM. This includes those cases where the target determines that it cannot execute the command, such as a REJECTED, response. The return of any response code other than INTERIM marks the transaction completed and the target is normally ready to accept other transactions at its FCP_COMMAND register.

> If the target is unable to complete the command within 100 milliseconds, it shall promptly return an intermediate response code of INTERIM. Subsequent to an initial response of INTERIM, the target shall not send any additional INTERIM responses for this command. There is no time limit on command completion once an INTERIM response has been sent. The target shall ultimately send a final response when the command completes.
>
> If the target detects a Serial Bus reset, it shall reset its state to be able to accept AV/C command frames at its FCP_COMMAND register. Any in progress transactions shall be discarded without the return of a response frame.
>
> If a target receives an AV/C command frame using the broadcasting node_ID, and a response of NOT IMPLEMENTED would be required, no response shall be returned.

In order to correlate a response frame with an outstanding AV/C command, a controller shall examine certain fields in the response frame. The *subunit_type* and *subunit_ID* fields are never modified by the target. The *ctype* field is overwritten with the response code returned. The *opcode* and *operand[n]* fields may or may not be altered, dependent upon the command type, *subunit_type* and *opcode*. For any particular *opcode*, consult the clauses that follow for the details of the response frame returned by the target.

7. AV/C commands

AV/C commands are variable-length strings of bytes that are embedded within a command frame and addressed to a particular AV unit or subunit. A command consists of a command type (*ctype*), a unit or subunit to which the command is addressed (*subunit_type*), an operation code (*opcode*) and one or more operands. Commands are described in the clauses that follow according to their command type, specified by *ctype* values of CONTROL, STATUS, SPECIFIC INQUIRY, NOTIFY or GENERAL INQUIRY.

7.1 Support levels

Each AV unit or subunit may implement a subset of the AV/C command set. An unsupported command shall be rejected with a response of NOT IMPLEMENTED. Support for the different commands is characterized as mandatory, recommended, optional and vendor-dependent, as defined below:

Mandatory
The command shall be supported by any audio/video device that claims compliance with this specification and that implements the AV unit or subunit type(s) for which the command is defined. AV/C compliant devices are identified by configuration ROM entries.

Recommended
For an AV/C compliant device, the command is optional but it represents a basic functionality, *e.g.*, video and audio insert modes for a VCR subunit's RECORD command. If the device supports unit or subunit type(s) that have the functionality corresponding to the command, it is recommended that the command be implemented.

Optional
The command is optional for an AV/C compliant device.

Vendor-dependent
Support for and interpretation of the command are defined by the device vendor.

Support levels for the different commands vary according to *ctype*.

7.2 Control commands

A control command is sent by a controller to another AV device, the target, to instruct the target to perform an operation. Either the AV unit or a subunit may be the recipient of the command, as determined by the *subunit_type* and *subunit_ID* fields in the command frame. The remaining fields, *opcode* and *operand[n]*, specify the command.

Subject to the procedures described in section 6, a target that receives a control command shall return an AV/C response frame with one of the four response codes described below.

NOT IMPLEMENTED
The target does not support the control command specified by *opcode* and *operand[n]* or the command is addressed to a subunit not implemented by the target. Target state is not modified.

ACCEPTED
The target implements the control command specified by *opcode* and *operand[n]* and the target state permits execution of the command. Note that command execution may not be complete at the time a response of ACCEPTED is returned. For example, a PLAY control command sent to a VCR may be acknowledged as accepted before the head mechanisms have engaged and the tape has started to move. The return of a response of ACCEPTED does not distinguish between a

command that has completed immediately and one that is deferred but expected to complete without error.

REJECTED The target implements the control command specified by *opcode* and *operand[n]* but the target state does not permit execution of the command. For example, a PLAY control command sent to a VCR that has no cassette inserted would be rejected. The target state may be modified as a result of the control command.

INTERIM If the control command specified by *opcode* and *operand[n]* is implemented but the target is unable to respond with either ACCEPTED or REJECTED within 100 milliseconds, it shall return a response frame that indicates INTERIM. Unless a subsequent bus reset causes the AV/C transaction to be aborted, the target shall ultimately return a response frame with a response code of ACCEPTED or REJECTED.

7.3 Status commands

A status command is sent by a controller to an AV device to request the device's current status. Status commands may be sent to either AV units or subunits. No target state is altered by the receipt of a status command.

NOTE: With some notable exceptions, for example the status commands that deal with a VCR's transport states, the status commands bear a family resemblance to the control commands. The same *opcode* that is used to issue a control command to a target is generally used to request corresponding status.

A target that receives a status command shall return an AV/C response frame with one of the four response codes described below:

NOT IMPLEMENTED The target does not support the status command specified by *opcode* and *operand[n]* or the command is addressed to a subunit not implemented by the target.

REJECTED The target implements the status command specified by *opcode* and *operand[n]* but the target state does not permit the return of status for the command.

IN TRANSITION The target implements the status command specified by *opcode* and *operand[n]* but the target state is in transition, possibly because of an already acknowledged command or a manual operation. A subsequent status command, at an unspecified future time, may result in the return of STABLE status.

STABLE The target implements the status command specified by *opcode* and *operand[n]* and the information requested is reported in the *opcode* and *operand[n]* values in the AV/C response frame.

NOTE: Stable information may be returned for target information that is changing because of command execution. For example, the tape position reported by a VCR may be an accurate snapshot at the time the status command was accepted, but a subsequent status command could yield a different result.

Except for the STABLE and IN TRANSITION responses, the AV/C response frame data contains the same *opcode*, operands and addressing fields as the command frame. When status information is available, both the *opcode* field and one or more of the *operand[n]* fields may be updated with the status information.

7.4 SPECIFIC INQUIRY commands

Inquiry commands may be used by a controller to determine whether or not a target AV device supports a particular control command. Except for the *ctype* field, the AV/C command frame for an inquiry command is identical to the corresponding control command.

A controller may reliably use inquiry commands to probe the capabilities of a target, since the target shall not modify any state nor initiate any command execution in response to an inquiry command.

Only two response codes, IMPLEMENTED or NOT IMPLEMENTED are permitted in the response frame returned by the target. All other fields in the response frame are exact copies of the command frame. A response of IMPLEMENTED specifies that the corresponding control command specified by *opcode* and *operand[n]* is implemented by the target AV device. An AV device implementation may validate all of the operands or it may validate only *opcode* and enough of the operands to uniquely identify the control command and determine its support level.

> **NOTE:** If a controller wishes to determine whether or not a particular status command is supported, it should issue the command. This is safe because status commands, whether or not implemented by a target, shall not cause state changes in the target.

Unlike the other command types, inquiry commands do not have a support level since they return information about the support level of the corresponding control command. However, the ability of an AV device to provide a response to an inquiry command for any *opcode* is mandatory. This insures that a controller shall always receive a response to a support level inquiry command.

The broadcasting node_ID shall not be used for inquiry commands.

7.5 Notify commands

A controller that desires to receive notification of future changes in an AV device's state may use the notify command. Responses to a notify command shall indicate the current state of the target and then, at some indeterminate time in the future, indicate the changed state of the target.

A target that receives a notify command shall not modify its state but shall generate an immediate response frame with one of the three response codes described below:

NOT IMPLEMENTED	The target does not support the notify command specified by *opcode* and *operand[n]* or the command is addressed to a subunit not implemented by the target.
REJECTED	The target implements event notification for the condition specified by *opcode* and *operand[n]* but is not able to supply the requested information.
INTERIM	The target supports the requested event notification and has accepted the notify command for any future change of state. The current state is indicated by the *opcode* and *operand[n]* data returned in the response frame. At a some future time, the target shall return an AV/C response frame with either a REJECTED or CHANGED response code.

Once a target has accepted a notify command by the return of an INTERIM response frame, the target is primed to return a subsequent response frame upon the first change in target state. The future change of target state could be the result of an operation in progress when the notify command was received or it could be the result of a control command not yet received by the target.

CHANGED

The target supports the event notification specified by *opcode* and *operand[n]* and the target state differs from the target state at the time the INTERIM response was returned. The altered target state is indicated by the *opcode* and *operand[n]* data returned in the response frame.

A typical example of the use of a notify command might involve a VCR whose cassette is being rewound. The initial response to a TRANSPORT STATE notify command is an indication of INTERIM and a “rewinding” state. When the cassette’s beginning of medium is reached, the target generates a final response frame of CHANGED and a state that indicates “stopped”.

Note that notification is a one-shot operation. If the controller wishes to be notified of additional changes in a target, the controller must issue a notify command after each CHANGED response.

7.6 GENERAL INQUIRY commands

General inquiry commands may be used by a controller to determine whether or not a target AV device supports a particular control command WITHOUT being required to specify a particular set of parameters for that command. The format of the GENERAL INQUIRY command frame consists of only the opcode of the command which is being queried.

As with the SPECIFIC INQUIRY command, the target shall not modify any state nor initiate any command execution in response to a general inquiry command.

Only two response codes, IMPLEMENTED or NOT IMPLEMENTED are permitted in the response frame returned by the target. The response frame shall also contain the opcode that was originally passed in. A response of IMPLEMENTED specifies that at least one of the corresponding control command variations specified by *opcode* is implemented by the target AV device. For example, a VCR which supports the BACKWARD control command with the *video scene* operand, but not the *video frame* or *index* operands, shall return IMPLEMENTED for the BACKWARD general inquiry command.

Unlike the other command types, general inquiry commands do not have a support level since they return information about the support level of the corresponding control command. However, the ability of an AV device to provide a response to a general inquiry command for any *opcode* is mandatory. This insures that a controller shall always receive a response to a support level inquiry command.

The general inquiry command type was defined after the original AV/C specification, and some products, were created. Hence, there will be some devices which do not respond to this command type. A controller which does not receive a response should try the SPECIFIC INQUIRY command as a fallback measure.

The broadcasting node_ID shall not be used for general inquiry commands.

8. AV/C Descriptor Mechanism

The AV/C descriptor mechanism supports the creation of various data structures, both static and dynamic, which contain useful information about AV/C units, subunits, and their associated components such as plugs. Additionally, these structures can be used to model the media contents provided by these subunits.

The structures can be combined to form a content navigation and selection mechanism which allows controllers to discover and access all media contents in a general way, limiting the media- and subunit-type specific knowledge required to perform such tasks.

These structures are applicable to any type of unit and subunit definitions of the AV/C protocol.

A *descriptor* is an address space on a target which contains attributes or other descriptive information. One example is the *subunit identifier descriptor*, which is a data block containing various pieces of information regarding a particular type of subunit. The format and contents of the subunit identifier descriptor are unique to each type of subunit. For example, all tuner subunits will have the same kind of subunit identifier descriptor. Most of the information in this structure is static in nature. However, depending on the type of subunit and the particular technologies that it implements, it is possible that some of the information may change from time to time.

Other standard descriptor structures include an *object list* and the *object entries* that it contains. An object is a generic concept that applies to a particular type of subunit, which can be defined as needed. For example, a tuner subunit implements an object list which contains information about the various services which are available on the system. Each service is represented by an *object.*

Other examples of where objects may be useful might be for disc players, where the objects represent tracks on the media. For digital still cameras, objects could be the pictures that have been taken.

An *object list* is a generic container concept; each entry in the list is an object descriptor structure. These may also be referred to as *object entries,* or simply, *objects.*

Objects and object lists can be used to model data relationships where one entity is composed of several sub-entities. For example, an audio compact disc (CD) can be composed of the collection of audio tracks on that disc.

In the above example, an *object* would be used to describe the CD. That object would, in turn, have a reference to a list of objects, where each one represents an audio track. Further extending the example, a collection of several CD's can be represented by a list, where each object entry describes one of those CD's.

The hierarchical model of lists and objects can be continued to any arbitrary level of complexity.

We define the ***root*** or beginning of a hierarchy to be a list, which may contain one or more objects. This list will be accessible by its list_ID value, which can be found in the subunit identifier descriptor. A subunit identifier descriptor may refer to several root lists.

When traversing away from the root, we say that we are moving down in the hierarchy. Conversely, when moving toward the root, we are moving up in the hierarchy. There is only one root for the hierarchy, and there may be any number of leaf nodes (end points).

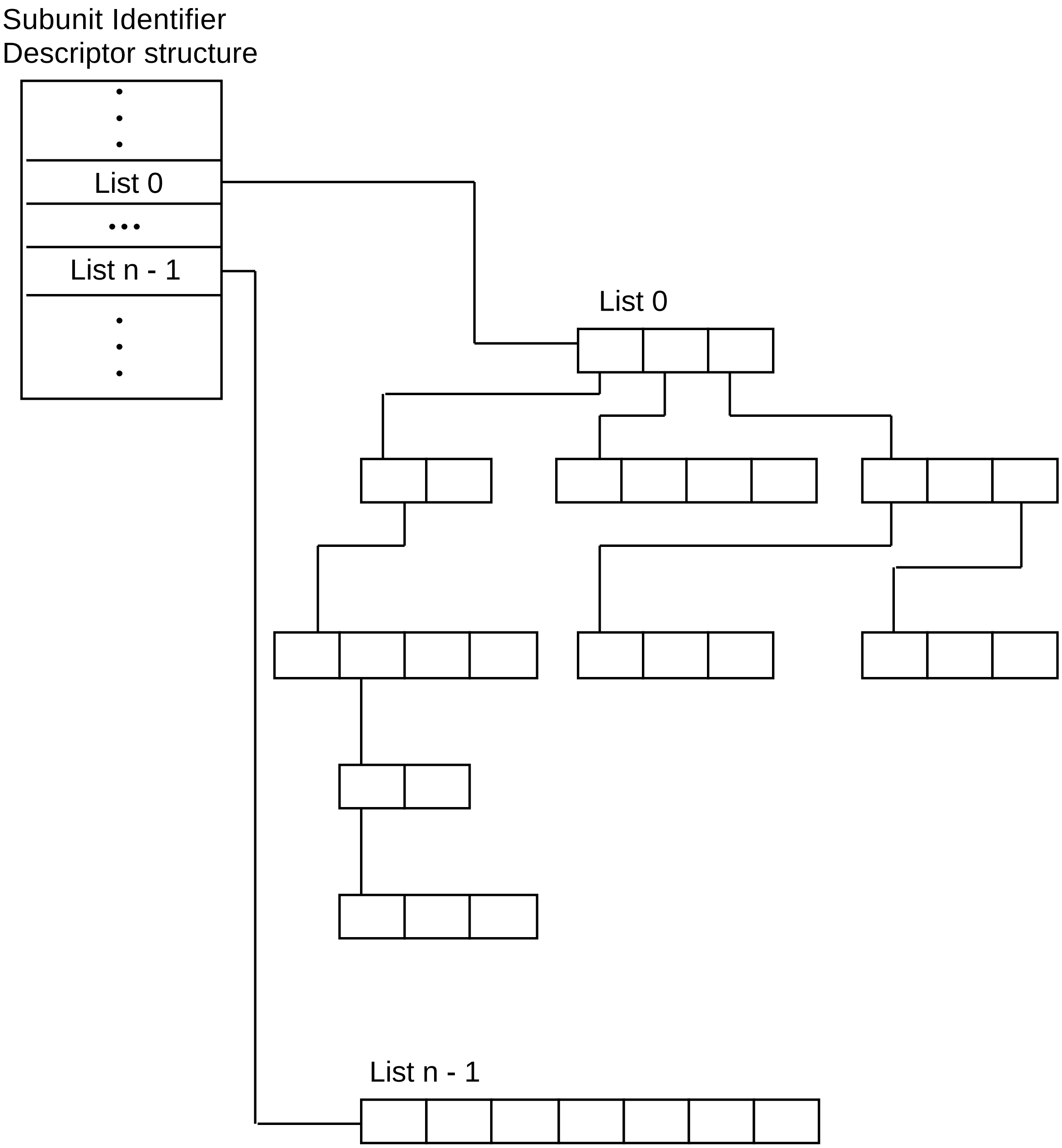
When an object entry has object lists associated with it, we say that the object entry is a ***parent***. The object list that it refers to is a ***child***.

So, object lists which are referred to by the subunit identifier descriptor are root lists; all other lists, which must be referred to by objects within other lists, are child lists. Object lists and object entries are defined per subunit type (tuner, etc.). There may or may not be crossover usage in other subunit types, depending on the definitions.

The unit and subunit identifier descriptors mentioned in this document are examples of descriptor structures; objects and object lists are also descriptors. When a controller wants to access a descriptor, it will use the descriptor commands to specify which descriptor it wants to deal with.

One very important fact to understand is that the structures of the various descriptors defined here are for interfacing purposes, where two separate AV/C entities (a target and a controller) must exchange information. The internal storage strategy used by a particular entity is completely transparent to these interface definitions. Data can be stored in any manner or location as required by the entity. Only when it comes time to present it to a controller will it be necessary to use these structure formats.

The following diagram illustrates the general relationship between the subunit identifier descriptor, object lists and object entries:



List 0 is the root of a hierarchy, and list n-1 is the root of another hierarchy.

8.1 The General Subunit Identifier Descriptor

The subunit identifier descriptor is a data structure that contains attribute information about the subunit that it refers to; the descriptor content and format can vary based on the type of subunit that it describes. All subunit identifier descriptors share some common information. The basic structure of a subunit identifier descriptor is as follows:

The General Subunit Identifier Descriptor	
address	contents
00 00$_{16}$	descriptor_length
00 01$_{16}$	
00 02$_{16}$	generation_ID
00 03$_{16}$	size_of_list_ID
00 04$_{16}$	size_of_object_ID
00 05$_{16}$	size_of_object_position
00 06$_{16}$	number_of_root_object_lists (n)
00 07$_{16}$	
00 08$_{16}$	root_object_list_id_0
:	
:	:
:	root_object_list_id_n-1
:	
:	subunit_dependent_length
:	
:	subunit_dependent_information
:	
:	
:	manufacturer_dependent_length
:	
:	manufacturer_dependent_information
:	
:	

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

generation_ID values	
generation_ID	meaning
00$_{16}$	Data structures and command sets as specified in the AV/C General Specification, version 3.0
all others	reserved for future specification

The *size_of_list_ID* field indicates the number of bytes used to indicate a list ID for this subunit. All lists maintained within the scope of this subunit shall use this number of bytes for their ID values.

The *size_of_object_ID* field indicates the number of bytes used to indicate an object ID for this subunit. All objects maintained within the scope of the subunit which have an ID shall use this number of bytes for their ID. It is possible for some objects within the scope of a subunit to have ID values, and for some to not have ID values.

The *size_of_object_position* field indicates the number of bytes used when referring to an object by its position in a list. All such references used with the subunit shall use this number of bytes for the position reference.

The *number_of_root_object_lists* field contains the number of root object lists directly associated with this subunit. This field is 2 bytes in size.

The *root_object_list_id_x* fields are the ID values for each of the root object lists associated with this subunit (all of the lists that are referenced from the subunit identifier are root lists). The *number_of_root_object_lists* field indicates how many of these ID values are present. Each of these lists forms the top of an individual hierarchy; this concept will be explained in more detail in the section titled Object Lists.

The *subunit_dependent_length* and *subunit_dependent_information* fields contain information whose format and contents will depend on the type of subunit this is describing. The length field specifies the number of bytes in the information field.

The *manufacturer_dependent_length* and *manufacturer_dependent_information* fields are used for vendor-specific data. The format and contents are completely up to the manufacturer. The length field specifies the number of bytes in the information field. If there is no manufacturer-dependent information in the descriptor, then the length field shall be zero and the *manufacturer_dependent_information* field shall not exist.

8.2 Object Lists

An object list contains entries which we call objects. The meaning and format of an object will be defined for the particular kind of information it represents. Object lists will be pre-defined as part of the model for a unit or subunit.

Object lists are uniquely identified within the scope of the entire subunit by their unique list_ID field, the size of which is specified in the subunit identifier descriptor as mentioned above. The following table illustrates the general list ID value range assignments:

List ID Value Assignment Ranges	
range of values	list definition
0000_{16} - $0FFF_{16}$	reserved
1000_{16} - $3FFF_{16}$	subunit-type dependent
4000_{16} - $FFFF_{16}$	reserved
1 0000_{16} - max list ID value	subunit-type dependent

Depending on the type of subunit for which a set of lists are defined, some lists may have fixed ID values, while others will vary based on how many of the lists are present at any given time. Note that the size of the maximum list ID value is *theoretically* unbounded, since the subunit can specify the number of bytes used.

All object lists will have the same basic layout, which includes some standard fields at the beginning, and then a collection of object entries. The descriptor defines the template for this basic layout. The object list descriptor has the following format:

The General Object List Descriptor	
address	contents
00 00_{16}	descriptor_length
00 01_{16}	
00 02_{16}	list_type
00 03_{16}	attributes
00 04_{16}	size_of_list_specific_information
00 05_{16}	
00 06_{16}	list_specific_information
:	
:	
:	number_of_entries (n)
:	
:	object_entry[0]
:	
:	
:	:
:	object_entry[n-1]
:	
:	

The *descriptor_length* field contains the number of bytes which follow in this object list structure. The value of this field does *not* include the length field itself.

The *list_type* field indicates what kind of list this is. There are two ranges of values defined for list_type: general and subunit-type-specific. Within the general range, ID values will have the same meaning for all types of subunits which use them. Within the subunit-specific range, the meaning of a particular list_type value will depend on the subunit type for which that list is defined.

Because the *list_type* definitions within the subunit range are unique to a given type of subunit (the combination of subunit type value and list_type are unique), different subunit types may define list_types with the same value in this range.

The following table shows the range definitions for *list_type*:

list_type Value Assignment Ranges	
range of values	list definition
00_{16} - $7F_{16}$	general definitions
80_{16} - FF_{16}	subunit-type dependent

For details on subunit-dependent *list_type* definitions, please refer to the appropriate subunit specification.

The *attributes* field contains bit flags which indicate attributes that pertain to the entire list structure. These values are defined as general attributes, not specific to any particular kind of list in any particular type of subunit.

The following table illustrates the attributes that are defined for both object lists and object entries. Some of these attributes are common to both objects and object lists, and others are unique to one type or the other. Unless otherwise noted, the attributes in this table are common to both:

Attribute value	Name	Meaning
1xxx xxxx	has_more_attributes	If this bit is set to 1, then the next byte is also an attributes byte. If this bit is 0, then the next byte is as defined for this structure.
x1xx xxxx	skip	This bit indicates the *presence* of data. If this bit is set to 1, the controller must skip the list or object entry. When this bit is set to 0, then the information in the list or object entry may be read by the controller. The subunit can use this bit to perform a "delayed memory clean up" operation. By setting this bit, the subunit can defer the actual deletion and reclamation of this memory until a convenient time.
xx1x xxxx	has_child_ID	VALID ONLY FOR THE OBJECT ENTRY DESCRIPTOR If this bit is set to 1, then the object entry has a child_list_ID field. If it is 0, then this field does not exist in the structure.
xxx1 xxxx	has_object_ID	VALID ONLY FOR THE OBJECT LIST DESCRIPTOR If this bit is set to 1, then all objects in this list have an ID. If it is zero, then none of the objects in this list have an ID.
xxxx 1xxx	up_to_date	This bit indicates the *validity* of data. If this bit is set to 1, then the descriptor data is known by the subunit to be up to date. If this bit is set to 0, then the descriptor data *might* be stale (the subunit may or may not be sure of this), and the controller should take this into consideration when dealing with the data.
all others		Reserved for future specification.

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

The *list_specific_information* field contains information that is specific to a particular *list_type*. Refer to the specific *list_type* definitions for details on this field.

The *number_of_entries* field contains the number of entries in this list.

The *object_entry[x]* field is an object entry, the basic format of which is presented next.

8.3 Object Entries

All object entries share a basic common format, but there is room for technology-specific information within the entry. The basic object entry descriptor has the following structure:

The General Object Entry Descriptor	
address offset	contents
00_{16}	descriptor_length
01_{16}	
02_{16}	entry_type
03_{16}	attributes
04_{16}	child_list_ID
:	
:	object_ID
:	
:	
:	
:	
:	size_of_entry_specific_information
:	
:	entry_specific_information
:	
:	

NOTE: The child_list_ID and object_ID fields exist only if specified in the attributes field.

The *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. If this object entry is empty, then the *entry_specific_information* field does not exist; the length field shall be set to the appropriate size, which may or may not include the *child_list_ID* and *object_ID* fields.

The *entry_type* field indicates what kind of entry this is. The values for *entry_type* are defined in the same manner as the *list_type* fields for object lists; there is a range for general definitions, and a range for subunit-type-specific definitions.

entry_type Value Assignment Ranges	
range of values	entry definition
00_{16} - $7F_{16}$	general definitions
80_{16} - FF_{16}	subunit-type dependent

For details on subunit-dependent *entry_type* definitions, please refer to the appropriate subunit specification.

The *attributes* field is the same as defined for the object list attributes above.

The *object_ID* field contains a value which uniquely identifies this object within a certain scope, using the following general rules:

a) Object ID values must be unique within their list.

b) Additional conditions on object ID uniqueness may be defined by a particular type of subunit. For details, please refer to the appropriate subunit specification.

c) The object ID does NOT necessarily indicate the position of this object in the object list. In some lists, the object number *may* be the same as the object position, but this is not guaranteed or required by the model.

The list owner (unit or subunit) which manages the list shall be responsible for ensuring object ID uniqueness according to these rules. If a controller attempts to set an object ID to a value which conflicts with an existing object ID within this scope, the list owner shall REJECT the operation. If an object ID assignment is accepted, then the list owner shall not change it.

The number of bytes used to specify the *object_ID* is defined in the subunit identifier descriptor. All objects within a list shall have the same number of bytes for their ID values.

The *child_list_ID* field holds the *list_ID* of the child list associated with this entry. If the object does not have a child list, then the *has_child_ID* bit of the attributes will be set to 0, and this field shall not exist in the structure. The general model defines an object as having at most one child_list_ID field. Normally, this means that the object has a single child list, which indicates a relationship between the object and all of the items in the list.

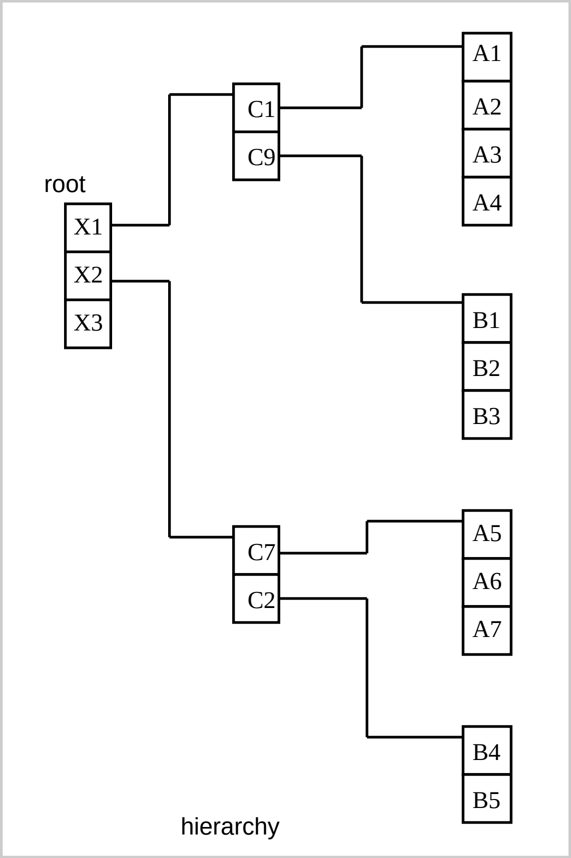
If a particular technology requires that an object be able to have more than one child list directly associated with it, then a subunit-specific list type can be defined to satisfy this requirement. Each entry in the list will represent a child of the object which owns the list.

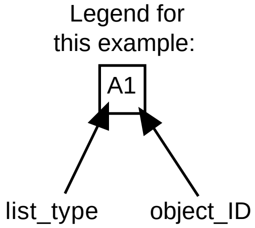
For an example of this concept, please refer to the diagram presented below.

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

The *entry_specific_information* area will have a format and contents specific to the type of object being referenced.

The following diagram illustrates the rules regarding object ID assignments and the support of multiple child lists from one object:

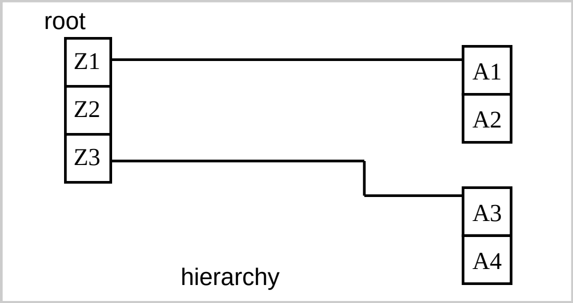




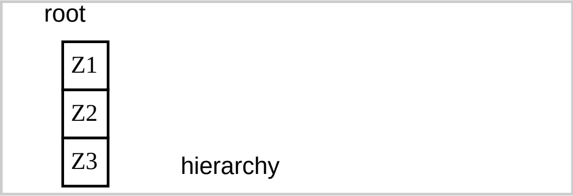
The object ID must be unique within a list.

The objects in these root lists may have multiple child lists, so they use subunit-specific lists to hold the child list ID values. In this example, object X2 has two child lists. It uses a list to hold two objects (C7 and C2), each of which point to one of the child lists.

The object ID does not necessarily indicate the position of the object in the list.



The objects in this root list have only zero or one child list.



A lone root list is a hierarchy.

8.4 Object References

When working with objects and object lists, controllers will need to specify the particular object(s) they are interested in accessing. In some situations, the only way a controller can indicate which object it wants is by specifying its position in a list. This is the normal way of stepping through a list to examine objects that the controller has not seen before.

Because we specify that object ID values are not necessarily the same as their position in a list, the object ID is another useful way to refer to an object. This would be necessary in those cases where a list is dynamic, which would result in some objects changing their position.

To address these two types of situations, we define two ways of referring to an object in a list: the Object Position Reference and the Object ID Reference.

8.4.1 Object Position Reference

The object position reference indicates the position of the desired object. The first entry in a list has position value zero. The format is as follows:

address offset	contents
00_{16}	object_position (MSB)
:	
:	object_position (LSB)

8.4.2 Object ID Reference

The object ID reference indicates the object ID of the desired object. The format is as follows:

address offset	contents
00_{16}	object_ID (MSB)
:	
:	
:	object_ID (LSB)

The number of bytes used to specify the *object_ID* is defined in the subunit identifier descriptor. Refer to the subunit-specific list descriptions for details. All objects maintained by the subunit shall have the same number of bytes for their ID values.

8.5 Parsing and Navigating the General AV/C Descriptor Structures (Informative)

This section provides some background information for controllers, to illustrate the design strategies behind the AV/C descriptor mechanism. Understanding these strategies helps controllers to parse and navigate through structures even though the controllers may not have full knowledge of their contents.

8.5.1 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 in the structure or command frame. The most significant bit (msb) of a field is at the highest bit position.

8.5.2 Variable Length Fields Within Data Structures

Most descriptor structures (subunit identifier descriptor, object lists, object entries within lists, etc.) contain multi-byte fields inside of them, and many of these fields can have a variable length. All of these variable-length fields will be preceded by a field which indicates their length. Note that the length field specifies the number of bytes for the following field; the length value does NOT include itself in this calculation.

An example of this concept is the *manufacturer_dependent_information* field in the subunit identifier descriptor. This field is preceded by the *manufacturer_dependent_length* field, which specifies the number of bytes used for the *manufacturer_dependent_information* field which follows.

8.5.2.1 Indicating Variable vs. Fixed Number of Bytes in Diagrams

When the structure diagram for a multi-byte field contains bytes whose entry is noted as ":", then the field contains a variable number of bytes. The specification of the field will indicate how many bytes.

When the structure diagram of a multi-byte field has a value specified for each byte, then this field has exactly that number of bytes. The address or address offset of these fields within the operand may be specified as ":", because the addresses will depend on prior variable-length operands in the structure.

The following diagram illustrates this concept:

	msb							lsb
opcode	COMMAND OPCODE (XX_{16})							
operand[0]	field A							
operand[1]	field B							
operand[2]	:							
operand[3]	field C							
:								

This example illustrates a command frame, with the following attributes:

field A - fixed length (1 byte)
field B - fixed length (2 bytes)
field C - variable length, where field C would be further documented as:

field C description	
address offset	contents
00_{16}	field C-1
01_{16}	
02_{16}	field C-2
03_{16}	length_of_field_C-3
04_{16}	field C-3
:	

In field C:

field C-1 - fixed length (2 bytes)
field C-2 - fixed length (1 byte)
field C-3 - variable length, determined by length_of_field_C-3

The boundaries of operands in command frames, or fields in data structures, are denoted by solid lines.

Descriptor data structures follow this same model, but instead of specifying "opcode" and "operand[x]", the labels for fields are "address" and "address offset".

8.5.3 The Length of Object Entry Descriptors

The normal mechanism for indicating the size of a variable length field, as described above, is to **precede** the field with a length indicator (usually one byte for strings, and two bytes for most other fields). The one exception to this rule is for object entry descriptors.

Object descriptors are embedded within list structures, as shown in the general object list structure described above. The first field **inside** of each object entry descriptor is its length (the length field is two bytes). The reason for this exception has to do with the READ DESCRIPTOR command.

The READ DESCRIPTOR command allows the controller to read an entire descriptor in one action (assuming that both the controller and target have the capacity to transfer the required number of bytes in one operation). Object descriptors are treated as complete entities which can be read by specifying the object, such as by its position in a list, or by its ID, etc.

When a controller reads an object descriptor structure, the controller must be able to begin navigating through the structure. If the length field is NOT included in the structure, then the controller has no way of knowing how to begin parsing.

This model is consistent with all descriptor structures (subunit identifier, object list, etc.). All descriptors begin with their length field as the first entry in the structure. Object descriptors are unique because they are embedded within object list descriptors.

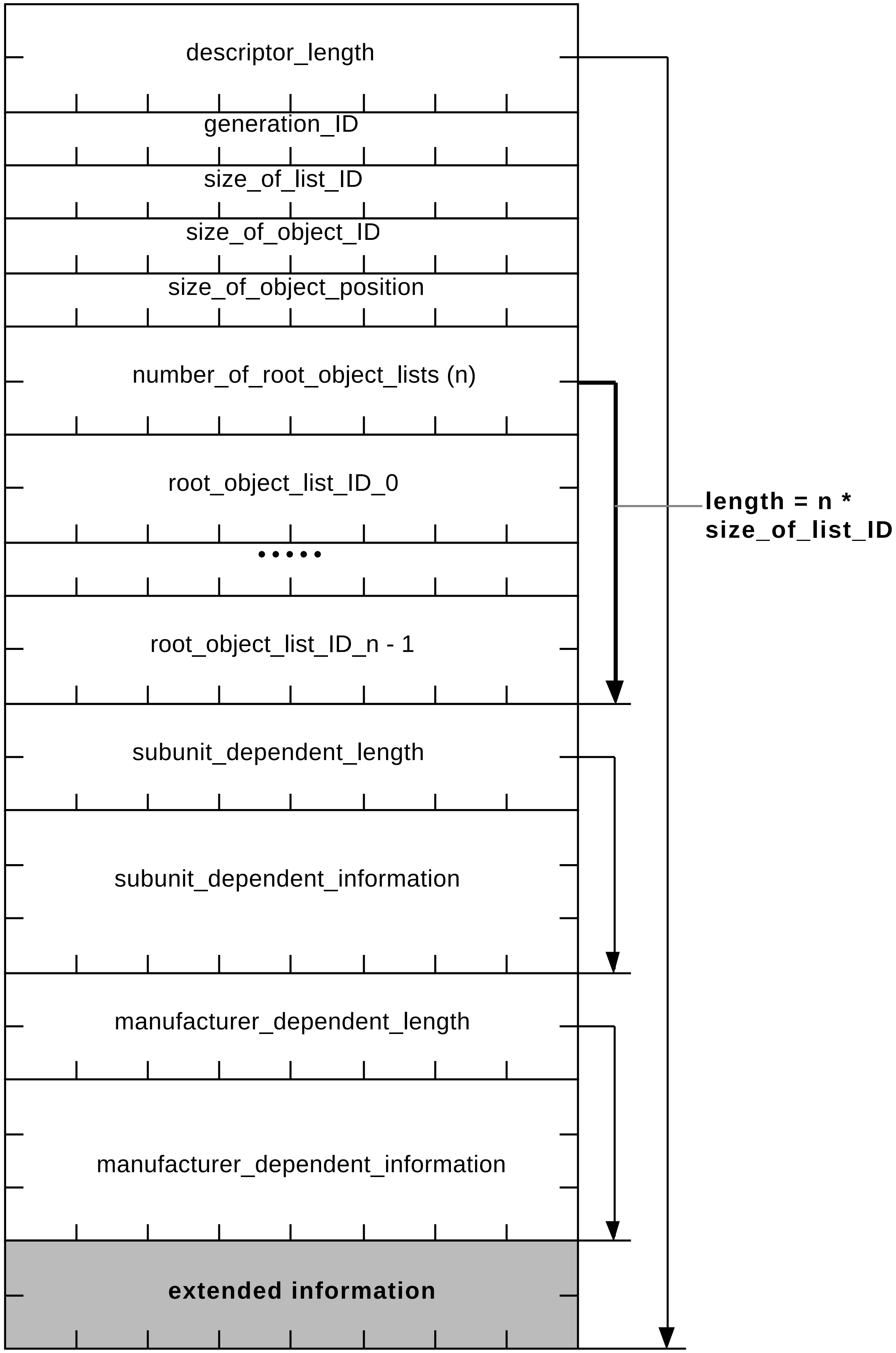
Note that the length field inside of all descriptor structures indicates the number of bytes which follow in the rest of the structure; the length field does NOT include itself in this calculation.

8.5.4 Extended Data Structures

The set of data structures which conform to a certain version of the AV/C descriptor mechanism are identified by their generation_ID. This value is defined in the subunit identifier descriptor, and specifies that all structures maintained by the subunit conform to the descriptor version indicated.

First generation descriptors shall only be extended in a backward-compatible manner, by adding new fields to the end of previously-existing fields. When controllers which understand only first-generation descriptors are parsing second- and later generation descriptors, they must be aware that the *descriptor_length* field MAY indicate a value which is larger than the descriptor are they can decode. In these circumstances, controllers must not assume that an error condition has occurred; rather, they should assume that the descriptor has been extended.

The following diagram illustrates the navigation rules described in 8.5.3 and 8.5.4. In this example, the subunit identifier descriptor structure is illustrated:


descriptor_length
generation_ID
size_of_list_ID
size_of_object_ID
size_of_object_position
number_of_root_object_lists (n)
root_object_list_ID_0
•••••
root_object_list_ID_n - 1
length = n *
size_of_list_ID
subunit_dependent_length
subunit_dependent_information
manufacturer_dependent_length
manufacturer_dependent_information
extended information

8.6 Rules for Reserved Fields

This section clarifies the rules which have always been in effect regarding how reserved fields shall be treated in command parameters and data structures.

Unless otherwise specified (see note below), command parameters and data structure fields marked as “reserved” or “reserved for future specification” shall be set to zero by controllers on input to a target, and by targets on output to controllers.

For input operands of commands, targets shall NOT ignore fields that were reserved when the target was implemented. Rather, the target shall examine the reserved fields; if any of them are specified, then the target shall reject the command with a NOT IMPLEMENTED response.

On output data structures or parameters of commands, controllers shall ignore fields that were reserved when the controller was implemented. These rules exist to allow future extension of the specification while retaining compatibility with existing products.

> **NOTE:** In some instances, reserved command operands or data structure fields may be specified as non-zero values. These cases will be clearly indicated in the specification. Controllers and targets shall deal with them in the same manner as defined above.

9. Unit commands

Unit commands are those that are addressed to an AV device implemented as a unit architecture at a Serial Bus node. Unit commands are identified by a *subunit_type* value of $1F_{16}$ and a *subunit_ID* value of seven. Table 8.6-1 below summarizes the AV/C unit commands.

Table 8.6-1 — Unit commands

		Support level (by *ctype*)			
Opcode	**Value**	**C**	**S**	**N**	**Comments**
CHANNEL USAGE	12_{16}	–	R	R	Report information on IEEE 1394 isochronous channel usage
CONNECT	24_{16}	O	O	R	Establish connections for unspecified streams between plugs and subunits
CONNECT AV	20_{16}	O	O	O	Establish AV connections between plugs and subunits
CONNECTIONS	22_{16}	–	O	–	Report connection status
DIGITAL INPUT	11_{16}	O	O	–	Make or break broadcast Serial Bus connections
DIGITAL OUTPUT	10_{16}	O	O	–	
DISCONNECT	25_{16}	O	–	–	Break unspecified stream connections between plugs and subunits
DISCONNECT AV	21_{16}	O	–	–	Break AV connections between plugs and subunits
INPUT PLUG SIGNAL FORMAT	19_{16}	O	R	O	Set or report signal formats for Serial Bus plugs
OUTPUT PLUG SIGNAL FORMAT	18_{16}	O	R	O	
SUBUNIT INFO	31_{16}	–	M	–	Report subunit information
UNIT INFO	30_{16}	–	M	–	Report unit information

A dash in the support level column indicates that the command is not defined for the *ctype* value CONTROL, STATUS or NOTIFY, indicated. The specific operand formats and corresponding response frame formats are described for each of the commands in the clauses that follow.

9.1 CHANNEL USAGE command

The CHANNEL USAGE status command can be used to find out which AV unit, if any, is using a particular IEEE 1394 isochronous channel.

Using a channel means that one of the AV unit's oPCRs indicates that there exists a connection which uses this channel.

For the CHANNEL USAGE status command, it is permissible to use the broadcasting node_ID.

> **NOTE:** When using the broadcasting node-ID, this command shall only generate a broadcast on one particular bus. Pending the definition of the addressing scheme in a bridged environment, a controller shall use the enumerated bus-ID value of the bus for which the command is intended as part of the

broadcasting node-ID. This also holds in case the command is intended for the bus to which the controller is attached. Only in case no bus-ID has been assigned, it is allowed to use the bus-ID value $3FF_{16}$ as part of the broadcasting node-ID.

The CHANNEL USAGE status command has the format as illustrated in Figure 9-1 below.

	msb							lsb
opcode	CHANNEL USAGE (12_{16})							
operand[0]	IEEE 1394 isochronous channel							
operand[1]	FF_{16}							
operand[2]								
operand[3]								

Figure 9-1 — CHANNEL USAGE status command format

Operand[0] denotes the isochronous channel which the target must check, to see if it is using that channel.

The CHANNEL USAGE status response has the format as illustrated in Figure 9-2 below.

	msb							lsb
opcode	CHANNEL USAGE (12_{16})							
operand[0]	IEEE 1394 isochronous channel							
operand[1]	node_ID							
operand[2]								
operand[3]	oPCR number							

Figure 9-2 — CHANNEL USAGE status response format

If in the STATUS response frame operand[1] through operand[3] are NOT all FF_{16}, it indicates that the target identified by operand[1] and operand[2] is using the channel indicated in operand[0], through the oPCR identified by operand[3]. Operand[1] contains the most significant byte of the node_ID.

If in the STATUS response frame operand[1] through operand[3] ARE all FF_{16}, it indicates that the target that returned the response is not using the channel indicated in operand[0].

In case the CHANNEL USAGE status command was broadcast (as opposed to unicast), the response obligation on this command exists only for those targets that use the channel. Because at most one target can meet this condition, at most one response frame will be returned and that response shall have operand[1] through operand[3] NOT all equal to FF_{16}.

In case the CHANNEL USAGE status command was unicast, the target shall return a response with operand[1] through operand[3] indicating whether or not the target is using the channel.

The CHANNEL USAGE command may also be used as a notify command. The notify command has the same syntax and meaning for its operands as the CHANNEL USAGE status command.

For the CHANNEL USAGE notify command, it is permissible to use the broadcasting node_ID.

In case the CHANNEL USAGE notify command was unicast and the target is not using the channel, it shall return a REJECTED response. Otherwise, it shall return an INTERIM response with operand[1] through operand[3] NOT all equal to FF_{16}. If an INTERIM response has been returned, a CHANGED response shall be returned with operand[1] through operand[3] all equal to FF_{16} once the target stops using the specified channel.

In case the CHANNEL USAGE notify command was broadcast, the response obligation on this command exists only for those targets that use the channel. Because at most one target can meet this condition, at most one INTERIM response frame will be returned with operand[1] through operand[3] NOT all equal to FF_{16}. If an INTERIM response has been returned, a CHANGED response shall be returned with operand[1] through operand[3] all equal to FF_{16} once the target stops using the specified channel.

9.2 CONNECT command

An AV *subunit* has *source* and *destination* plugs. A source plug outputs one stream from the AV subunit and a destination plug inputs one stream into the AV subunit.

An AV *unit* has *Serial Bus input* and *output* plugs to model the Serial Bus interface of the AV unit. An AV unit can have at most one Serial Bus interface and thereby at most one node ID on the Serial Bus. A Serial Bus input plug inputs one stream from the Serial Bus interface into the AV unit and a Serial Bus output plug outputs one stream from the AV unit to the Serial Bus interface.

An AV unit also has *external input* and *output* plugs to model external interfaces of the AV unit other than Serial Bus. An external input plug inputs one stream from an external interface into the AV unit and an external output plug outputs one stream from the AV unit to one external interface.
The CONNECT control command establishes a connection within an AV Unit between:

a source plug of an AV subunit and a destination plug of an AV subunit to carry a stream that flows inside the AV unit.

a source plug of an AV subunit and a Serial Bus or external output plug to carry a stream that flows from the AV subunit to the Serial Bus or external interface.

a Serial Bus or external input plug and a destination plug of an AV subunit to carry a stream that flows from the Serial Bus or external interface to the AV subunit.

These connections are independent from the type of data (audio, video, data, ...) inside the stream which they carry. These streams are named “unspecified streams.”

	msb							lsb
opcode	CONNECT (24_{16})							
operand[0]	$3F_{16}$						lock	perm
operand[1]	source_subunit_type					source_subunit_ID		
operand[2]	source_plug							
operand[3]	destination_subunit_type					destination_subunit_ID		
operand[4]	destination_plug							

Figure 9-3 — CONNECT control command

The subunit_type and subunit_ID fields for both the source and destination plugs have the same syntax and meaning as an AV/C address (see section 5.3.3).

In case the value of source and destination *subunit_type* and *subunit_ID* are extended in the above control command, the frame will look as follows:

	msb							lsb
opcode	CONNECT (24_{16})							
operand[0]	3F$_{16}$						lock	perm
operand[1]	source_subunit_type = IE$_{16}$					source_subunit_ID = 5$_{16}$		
operand[2]	extended_source_subunit_type							
operand[3]	extended_source_subunit_ID							
operand[4]	source_plug							
operand[5]	destination_subunit_type = IE$_{16}$					destination_subunit_ID = 5$_{16}$		
operand[6]	extended_destination_subunit_type							
operand[7]	extended_destination_subunit_ID							
operand[8]	destination_plug							

Figure 9-4 — CONNECT control command with extended subunit_type and extended subunit_ID

For the example above, the source and destination *subunit_type* and *subunit_ID* values have been extended only once.

The source_plug and destination_plug fields are defined by the table below:

value	source plug	destination plug
0 - 1E$_{16}$	Source plug 0 - 30	Destination plug 0 - 30
1F$_{16}$ - FC$_{16}$	Reserved for future specification	Reserved for future specification
FD$_{16}$	Reserved for future specification	Multiple plugs
FE$_{16}$	Invalid	Invalid
FF$_{16}$	Any available source plug	Any available destination plug

When the stream flows from or to one of the AV unit's Serial Bus or external plugs, the *subunit_type* field shall have a value of 1F$_{16}$ (AV unit) and the *subunit_ID* field shall have a value of 7. In this case, the *source_plug* and *destination_plug* fields identify either a Serial Bus or an external plug according to Table 9.2-1 below:

Table 9.2-1 — Serial Bus and external plug numbers

value	source plug	destination plug
0 - 1E$_{16}$	Serial Bus iPCR[0] - iPCR[30]	Serial Bus oPCR[0] - oPCR[30]
1F$_{16}$ - 7E$_{16}$	Reserved for future specification	Reserved for future specification
7F$_{16}$	Any available Serial Bus plug iPCR[x]	Any available Serial Bus plug oPCR[x]
80$_{16}$ - 9E$_{16}$	External input plug 0 - 30	External output plug 0 - 30
9F$_{16}$ - FC$_{16}$	Reserved for future specification	Reserved for future specification
FD$_{16}$	Reserved for future specification	Multiple plugs
FE$_{16}$	Invalid	Invalid
FF$_{16}$	Any available External input plug	Any available External output plug

The PLUG INFO status command may be used to determine the number of Serial Bus and external plugs of an AV unit.

Note that overlaying a connection with another connection between the same source plug and another destination plug resulting in a one-to-many flow of the same stream may or may not be allowed, depending on the capabilities of the target.

The *lock* bit pertains to the connection between the source and destination plugs as indicated in the CONNECT command. If the lock bit in the CONNECT command is set to one to establish a connection between a source and a destination plug, any subsequent CONNECT command that would result in a disruption of the stream flowing between these plugs shall return a REJECTED response. This rule shall remain valid until a subsequent DISCONNECT command has been received by the target for that source plug.

The *perm* bit is ignored in a CONNECT control command.

The CONNECT command may also be used as a status command to determine the current state of the connections within an AV unit. The CONNECT status command is used to request the identity of the source plug that is connected to a given destination plug, or the identity of the destination plug for a given source plug. The two formats for the corresponding CONNECT status commands are shown in Figure 9-5 and Figure 9-6 below, and have the same meaning as the corresponding fields of the CONNECT control command.

	msb							lsb
opcode	CONNECT (24_{16})							
operand[0]	FF_{16}							
operand[1]	source_subunit_type					source_subunit_ID		
operand[2]	source_plug							
operand[3]	FF_{16}							
operand[4]	FE_{16}							

Figure 9-5 — CONNECT status command format for a source plug

	msb							lsb
opcode	CONNECT (24_{16})							
operand[0]	FF_{16}							
operand[1]	FF_{16}							
operand[2]	FE_{16}							
operand[3]	destination_subunit_type					destination_subunit_ID		
operand[4]	destination_plug							

Figure 9-6 — CONNECT status command format for a destination plug

The CONNECT status response frame has the same format and the same meaning for all fields as the CONNECT control command except for the *perm* field.

Except for the *perm* bit, the CONNECT status response frame contains exact copies of the CONNECT operands that were used to establish the connection. This includes the extended source and destination *subunit_type* and *subunit_ID* if they were used.

The *perm* bit in a CONNECT status response frame indicates whether a connection is permanent (value 1) or not (value 0). Permanent connections within an AV unit are connections that cannot be altered by the CONNECT control command or deleted by the DISCONNECT command, in which case a REJECTED response shall be returned.

In case there is no source plug connected to a destination plug, the *source_plug* field of the CONNECT status response frame shall indicate FE$_{16}$ (invalid).

In case there is no destination plug connected to a source plug, the *destination_plug* field of the CONNECT status response frame shall indicate FE$_{16}$ (invalid).

In case there are multiple destination plugs connected to a source plug, the *destination_plug* field of the CONNECT status response frame shall indicate FD$_{16}$ (multiple plugs).

The CONNECT command may also be used as a notify command. The notify command has the same syntax as the CONNECT status command. A notification shall be returned by the target to the controller that issued the notify command in case a connection involving the plug, as indicated in the notify command, changes. These changes shall include establishing a connection to the plug, deleting a connection from the plug, and connecting the plug to another plug.

The notify responses (INTERIM and CHANGED) have the same format as the CONNECT status response frame and indicate the current status of the plug for which the notification was requested. If the plug is still connected, the plug to which it is connected shall be indicated. If the plug is no longer connected, the source or destination plug field shall be indicated as invalid (plug field value FE$_{16}$).

9.3 CONNECT AV command

The CONNECT AV control command is used to establish audio/video connections between subunits and plugs.

	msb							lsb
opcode	CONNECT AV (20$_{16}$)							
operand[0]	video_source_type		audio_source_type		video_dest_type		audio_dest_type	
operand[1]	video_source							
operand[2]	audio_source							
operand[3]	video_destination							
operand[4]	audio_destination							

Figure 9-7 — CONNECT AV control command format for audio/video stream

The four fields of operand[0], video_source_type, audio_source_type, video_dest_type and audio_dest_type, encode the meaning of the four following source and destination identifying fields, as described in the table below.

Value	Source or destination type
0	Subunit
1	Ignore
2	Serial Bus or external plug
3	Reserved

If the source or destination type is zero, the corresponding source or destination operand is a subunit address, encoded as described in 5.3.3. The value of the source or destination type may be extended, and one or more bytes will be added accordingly. For an example, refer to the CONNECT control command. A source or destination value of FF_{16} is a special case and indicates that the AV device may select any appropriate, available subunit.

If the source or destination type is one, the corresponding source or destination operand is ignored. This value may be used to leave existing connections unchanged or it may be used if the AV unit does not implement the connection type. For example, in a CONNECT AV control command sent to an AV unit that had only audio recording capabilities, it would be appropriate to specify a value of one for both *video_source_type* and *video_dest_type*.

If the source or destination type is two, the corresponding source or destination operand represents a Serial Bus or an external plug, as encoded by the table below:

Value	Plug
0 — $1E_{16}$	Serial Bus plug zero — 30
$1F_{16}$ — $7E_{16}$	Reserved for future specification
$7F_{16}$	Any available Serial Bus plug
80_{16} — $9E_{16}$	External plug zero — 30
$9F_{16}$ — FE_{16}	Reserved for future specification
FF_{16}	Any available external plug

NOTE: In the preceding, some of the encoded values permit the AV device to select, at its option, an available subunit, Serial Bus or external plug. The set of plugs from which the device may choose is further limited by what is appropriate. For example, a dual-deck VCR might have one deck capable of recording SD signals and another capable of recording both HD and SD signals. If a Serial Bus input plug is active and configured for HD signals, a CONNECT AV control command for an audio/video stream that specified "any available" subunit would result in the natural connection to the deck capable of recording HD signals. On the other hand, if a Serial Bus input plug is active and configured for SD signals, an arbitrary connection could be established with either deck. In cases where more than one choice is possible, it is expected that the determination will be vendor-dependent.

In addition to its use as a control command, the CONNECT AV command may also be used as a status command to determine the current state of internal A/V connections for a unit or subunit. The form is shown in Figure 9-8 below.

	msb							lsb
opcode	CONNECT AV (20_{16})							
operand[0]	F_{16}				video_dest_type		audio_dest_type	
operand[1]	FF_{16}							
operand[2]	FF_{16}							
operand[3]	video_destination							
operand[4]	audio_destination							

Figure 9-8 — CONNECT AV status command format for audio/video stream

The fields video_dest_type, audio_dest_type, video_destination and audio_destination are used as previously described for the CONNECT AV command.

The response frame returned for the CONNECT AV status command has the same format as described in Figure 9-7. This includes fields for extended subunit type if used.

In case there is no source plug connected to the destination plug indicated in the CONNECT AV status command, the video_source and audio_source fields shall have the value FF16 (invalid), and the video_source_type and audio_source_type fields shall both have the value 1 (ignore).

The CONNECT AV command may also be used as a notify command. The notify command has the same syntax as the CONNECT AV status command. A notification shall be returned by the target to the controller that issued the notify command in case a connection involving the destination as indicated in the notify command changes. These changes shall include establishing a connection to the destination, deleting a connection from the destination, and connecting the destination to another source. The notify response has the same format as the CONNECT AV response frame.

9.4 CONNECTIONS command

The CONNECTIONS status command is used to inquire the state of all connections for unspecified streams. The format of the CONNECTIONS status command is illustrated by Figure 9-9 below.

	msb							lsb
opcode	CONNECTIONS (22_{16})							
operand[0]	FF_{16}							

Figure 9-9 — CONNECTIONS status command format

The response frame returned after a CONNECTIONS status command is variable in length and depends upon the number of connections established. The response frame has the format defined by Figure 9-10 below.

	msb							lsb
opcode	CONNECTIONS (22_{16})							
operand[0]	total_connections							
operand[1]	$3F_{16}$						lock	perm
operand[2]	connection[0].source							
operand[3]								
operand[4]	connection[0].destination							
operand[5]								
…	connection[1] — connection[total_connections - 2]							
operand[*n*-4]	$3F_{16}$						lock	perm
operand[*n*-3]	connection[total_connections - 1].source							
operand[*n*-2]								
operand[*n*-1]	connection[total_connections - 1].destination							
operand[*n*]								

Figure 9-10 — CONNECTIONS response format

The *total_connections* field specifies the number of five-byte connection descriptors returned in the operands that follow. The value of *n* is determined by 5 * *total_connections*.

The format of each connection descriptor is identical to operand[1] through operand[4] of the CONNECT control command. For a connection that includes an extended subunit_type or subunit_ID, these addresses may change depending on the number of extended fields.

9.5 DIGITAL INPUT command

The DIGITAL INPUT control command permits an AV unit to establish a broadcast input connection according to its own preferences. Figure 9-11 below illustrates the format of the command.

	msb							lsb
opcode	DIGITAL INPUT (11_{16})							
operand[0]	connection_state							

Figure 9-11 — DIGITAL INPUT command format

When the DIGITAL INPUT command is issued with a *ctype* value of CONTROL, the *connection_state* field specifies whether the AV unit is expected to establish (70_{16}) or break (60_{16}) a broadcast input connection.

The DIGITAL INPUT command, with a *ctype* value of STATUS, may also be used to determine the current input broadcast connection state of the unit. In this case, *operand[0]* is set to FF_{16} when the status command is issued and is updated to the current *connection_state* when the STABLE response frame is returned.

9.6 DIGITAL OUTPUT command

The DIGITAL OUTPUT control command permits an AV unit to establish a broadcast output connection according to its own preferences. Figure 9-12 below illustrates the format of the command.

	msb							lsb
opcode	DIGITAL OUTPUT (10_{16})							
operand[0]	connection_state							

Figure 9-12 — DIGITAL OUTPUT command format

When the DIGITAL OUTPUT command is issued with a *ctype* value of CONTROL, the *connection_state* field specifies whether the AV unit is expected to establish (70_{16}) or break (60_{16}) a broadcast output connection. The AV unit shall be responsible to allocate or deallocate the necessary isochronous resources, *e.g.*, bandwidth and channel number, and to program an output PCR as appropriate.

The DIGITAL OUTPUT command, with a *ctype* value of STATUS, may also be used to determine the current output broadcast connection state of the unit. In this case, *operand[0]* is set to FF_{16} when the status command is issued and is updated to the current *connection_state* when the STABLE response frame is returned.

9.7 DISCONNECT command

The DISCONNECT control command removes a connection between a destination and a source plug for an unspecified stream as described in the CONNECT control command, even if the connection was established with the lock bit set to one. In the case where multiple connections are overlaid on the same source plug, all connections will be deleted.

The format of the DISCONNECT control command is illustrated by Figure 9-13 below.

	msb							lsb
opcode	DISCONNECT (25_{16})							
operand[0]	FF_{16}							
operand[1]	source_subunit_type					source_subunit_ID		
operand[2]	source_plug							
operand[3]	destination_subunit_type					destination_subunit_ID		
operand[4]	destination_plug							

Figure 9-13 — DISCONNECT command format

The meaning of all fields are identical to the fields as described in the CONNECT control command. This includes the extended source and destination *subunit_type* and *subunit_ID* if they are used.

9.8 DISCONNECT AV command

The DISCONNECT AV control command is used to remove audio/video connections between subunits and plugs. The value of *operand[0]* is other than FF_{16} and the syntax is shown in Figure 9-14 below.

	msb							lsb
opcode	DISCONNECT AV (21_{16})							
operand[0]	video_source_type		audio_source_type		video_dest_type		audio_dest_type	
operand[1]	video_source							
operand[2]	audio_source							
operand[3]	video_destination							
operand[4]	audio_destination							

Figure 9-14 — DISCONNECT AV command format

The field definitions and their uses for DISCONNECT AV are identical to the field definitions given in Figure 9-7 for the CONNECT AV command. This includes the extended source and destination *subunit_type* and *subunit_ID* if they are used.

9.9 INPUT PLUG SIGNAL FORMAT command

The INPUT PLUG SIGNAL FORMAT control command is used to configure a specified Serial Bus input plug to receive data in the designated signal format. The syntax of the INPUT PLUG SIGNAL FORMAT control command is shown in Figure 9-15 below.

	msb							lsb
opcode	INPUT PLUG SIGNAL FORMAT (19_{16})							
operand[0]	plug							
operand[1]	2		fmt					
operand[2]	(most significant byte)							
operand[3]	fdf							
operand[4]	(least significant byte)							

Figure 9-15 — INPUT PLUG SIGNAL FORMAT control command format

The fields *fmt* and *fdf* are as defined in IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment. Together they specify the desired signal format for the Serial Bus input plug identified by *plug*.

The INPUT PLUG SIGNAL FORMAT status command is used to inquire which signal format a specified Serial Bus input plug is configured to receive. The syntax of the INPUT PLUG SIGNAL FORMAT status command is shown in Figure 9-16 below.

	msb							lsb
opcode	INPUT PLUG SIGNAL FORMAT (19_{16})							
operand[0]	plug							
operand[1]	FF_{16}							
…								
operand[4]								

Figure 9-16 — INPUT PLUG SIGNAL FORMAT status command format

The *plug* field specifies which one of the 31 Serial Bus input plugs, zero through $1E_{16}$, is referenced.

If the status command is accepted, the response frame has the same format as the INPUT PLUG SIGNAL FORMAT control command illustrated by Figure 9-15 above. The fields *fmt* and *fdf* are as previously defined and together they specify the signal format that the Serial Bus input plug identified by *plug* is configured to receive.

The INPUT PLUG SIGNAL FORMAT command may also be used as a notify command. The notify command has the same syntax as the status command. A notification shall be returned by the target to the controller that issued the notify command in case the format of the data that the Serial Bus input plug is receiving changes. The notify response has the same format as the status response frame.

9.10 OUTPUT PLUG SIGNAL FORMAT command

The OUTPUT PLUG SIGNAL FORMAT control command is used to configure a specified Serial Bus output plug to transmit data in the designated signal format. The syntax of the OUTPUT PLUG SIGNAL FORMAT control command is shown in Figure 9-17 below.

	msb							lsb
opcode	OUTPUT PLUG SIGNAL FORMAT (18_{16})							
operand[0]	plug							
operand[1]	2		fmt					
operand[2]	(most significant byte)							
operand[3]	fdf							
operand[4]	(least significant byte)							

Figure 9-17 — OUTPUT PLUG SIGNAL FORMAT control command format

The fields *fmt* and *fdf* are as defined in IEC 61883, Digital Interface for Consumer Electronic Audio/Video Equipment. Together they specify the desired signal format for the Serial Bus output plug identified by *plug*.

The OUTPUT PLUG SIGNAL FORMAT status command is used to inquire which signal format a specified Serial Bus output plug is configured to transmit. The format of the OUTPUT PLUG SIGNAL FORMAT command is illustrated by Figure 9-18 below.

	msb							lsb
opcode	OUTPUT PLUG SIGNAL FORMAT (18_{16})							
operand[0]	plug							
operand[1]	FF_{16}							
…								
operand[4]								

Figure 9-18 — OUTPUT PLUG SIGNAL FORMAT status command format

The *plug* field specifies which of the 31 Serial Bus output plugs, zero through $1E_{16}$, is referenced.

If the status command is accepted, the response frame has the same format as the OUTPUT PLUG SIGNAL FORMAT control command illustrated by Figure 9-17 above. The fields *fmt* and *fdf* are as previously defined and together they specify the signal format that the Serial Bus output plug identified by *plug* is configured to transmit.

The OUTPUT PLUG SIGNAL FORMAT command may also be used as a notify command. The notify command has the same syntax as the status command. A notification shall be returned by the target to the controller that issued the notify command in case the format of the data that the Serial Bus output plug is transmitting changes. The notify response has the same format as the status response frame.

9.11 SUBUNIT INFO command

The SUBUNIT INFO status command is used to obtain information about the subunit(s) of an AV unit. The format of the SUBUNIT INFO status command is illustrated by Figure 9-19 below.

	msb							lsb
opcode	SUBUNIT INFO (31_{16})							
operand[0]	0	page			0	extension_code		
operand[1]	FF_{16}							
…								
operand[4]								

Figure 9-19 — SUBUNIT INFO status command format

The *page* field value specifies which part of the subunit table is to be returned. An AV unit may implement up to 32 bytes of information in eight pages.

The *extension_code* field shall have a value of seven.

If the status command is accepted, the response frame returned has the structure illustrated by Figure 9-20 below.

	msb							lsb
opcode	SUBUNIT INFO (31_{16})							
operand[0]	0	page			0	extension_code		
operand[1]	page_data							
…								
operand[n]								

Figure 9-20 — SUBUNIT INFO response format

The *page_data* returned is the four entries from the subunit table for the page requested. The subunit table is an array of byte entries; each entry has the format defined by Figure 9-21 below:

msb							lsb
subunit_type					max_subunit_ID		

Figure 9-21 — Subunit table entry

The *subunit_type* field of each entry is as defined in Table 5.3-1. The extension of subunit_type and max_subunit_ID follows the example explained in the CONNECT control command; so, a subunit table entry may be more than one byte in length.

The *max_subunit_ID* field is the count of subunits of *subunit_type* implemented by the AV unit, less one.

The subunit entries are not required to be in any particular order but are required to be uniquely identified by *subunit_type*. If fewer than 32 entries are present in the subunit table, they are terminated by a byte with the value FF_{16}. The value of entries past the terminating FF_{16} is indeterminate and should be ignored by any controller that requests subunit information.

9.12 UNIT INFO command

The UNIT INFO status command is used to obtain information that pertains to the AV unit as a whole (distinct from subunit information, see 9.11). The format of the UNIT INFO status command is illustrated by Figure 9-22 below:

	msb							lsb
opcode	UNIT INFO (30_{16})							
operand[0]	FF_{16}							
…								
operand[4]								

Figure 9-22 — UNIT INFO status command format

If the status command is accepted by the target, a response frame with the format shown by Figure 9-23 below is returned.

	msb							lsb
opcode	UNIT INFO (30_{16})							
operand[0]	07_{16}							
operand[1]	unit_type					unit		
operand[2]	(most significant byte)							
operand[3]	company_ID							
operand[4]								(least significant byte)

Figure 9-23 — UNIT INFO response format

The *unit_type* field contains a value whose meaning is identical to those defined for subunit_type in Table 5.3-1. Value $1C_{16}$ (vendor unique) should be returned in case none of the other values are considered to be appropriate. The *unit_type* field may take value $1E_{16}$, which means that the field is extended to the following byte. In that case, an additional byte for *extended_unit_type* will be added immediately following operand[1], as shown in the example presented in the CONNECT control command. Further extension is possible when the value of *extended_unit_type* is FF_{16}, in which case another byte will be added.

The meaning of the *unit* field is not defined by this specification.

The *company_ID* field shall contain the 24-bit unique ID obtained from the IEEE Registration Authority Committee (RAC). It is expected that the value of company_ID returned by the UNIT INFO status command is the same as the vendor ID in the Node Unique ID leaf in the AV unit's configuration ROM. The most significant part of the company_ID is stored in *operand[2]* and the least significant part in *operand[4].*

10. Common unit and subunit commands

This section defines commands that are applicable to an AV unit as well as a subunit independent of the functionality that these subunits represent indicated by their subunit_type. Table 9.12-1 below summarizes the common unit and subunit commands.

Table 9.12-1 — Common unit and subunit commands

		Support level (by *ctype*)			
Opcode	**Value**	**C**	**S**	**N**	**Comments**
OPEN DESCRIPTOR	08_{16}	O	O	O	Gains the right to access the descriptor
READ DESCRIPTOR	09_{16}	O	–	–	Reads data from the descriptor
WRITE DESCRIPTOR	$0A_{16}$	O	O	–	Writes data into the descriptor
SEARCH DESCRIPTOR	$0B_{16}$	O	–	–	Search for a specified data pattern within the descriptor data space
OBJECT NUMBER SELECT	$0D_{16}$	O	O	O	Selects one or more objects using an object ID and list ID
POWER	$B2_{16}$	O	O	R	Control power state
RESERVE	01_{16}	O	O	R	Acquire or release exclusive control of a target
PLUG INFO	02_{16}	–	O	–	Information about Serial Bus and External plugs
VENDOR-DEPENDENT	00_{16}	V	V	V	Vendor-dependent commands

In the preceding table, a dash in the support level column indicates that the command is not defined for the *ctype* value, CONTROL, STATUS or NOTIFY, indicated. The specific command formats and corresponding response frame formats are described for each of the common subunit commands in the clauses that follow.

10.1 OPEN DESCRIPTOR command

The OPEN DESCRIPTOR control command is used to **gain access** to a certain address space on the target. The format of the OPEN DESCRIPTOR control command is illustrated by the figure below:

	msb							lsb
opcode	OPEN DESCRIPTOR (08_{16})							
operand[0]	descriptor_identifier (MSB)							
operand[1]	:							
:	:							
:	(LSB)							
:	subfunction							
:	reserved							

The *descriptor_identifier* describes which data structure is being accessed. The exact format of this identifier will vary based on the kind of descriptor, the method of specifying the desired descriptor, and the type of subunit which is managing that descriptor.

For example, an object can be referenced by its position in a certain list, or by its unique object ID. All of these criteria are involved in defining how a piece of information is specified for the various DESCRIPTOR commands.

The general AV/C model defines the subunit identifier descriptor, objects and object lists as the kinds of descriptors that are accessed through these commands. It is possible that other types of descriptors may be defined exclusively for a particular type of subunit. Such descriptors would also be accessed by these DESCRIPTOR commands.

The following diagrams illustrate the common *descriptor_identifier* types:

general descriptor_identifier	
address offset	contents
00_{16}	descriptor_type
01_{16}	descriptor_type_specific_reference
:	
:	
:	

The *descriptor_type* field indicates what kind of descriptor is specified in the rest of the identifier structure. It is encoded by the following table:

descriptor_type	meaning
00_{16}	Subunit identifier descriptor
10_{16}	Object list descriptor - specified by list ID
11_{16}	Object list descriptor - specified by list_type
20_{16}	Object entry descriptor - specified by object position
21_{16}	Object entry descriptor - specified by an object ID
80_{16} - BF_{16}	Subunit dependent descriptor
all others	reserved for future specification

Each of the *descriptor_type* values in the table indicate the format and contents of the *descriptor_type_specific_reference* field. For each of the descriptor types, the structure is as follows:

10.1.1 Subunit Identifier

descriptor_identifier for a subunit identifier descriptor	
address offset	contents
00_{16}	00_{16}

The *descriptor_type_specific_reference* field does not exist (the *descriptor_identifier* consists of only the *descriptor_type* field) because there can be only one subunit identifier descriptor for a subunit.

10.1.2 Object List

The *descriptor_identifier* for an object list specified by its ID appears as follows:

descriptor_identifier for an object list specified by ID	
address offset	contents
00_{16}	descriptor_type = 10_{16}
01_{16}	list_ID
:	
:	

As shown, the *descriptor_type_specific* contents for an object list consists of the desired *list_ID*. All lists within the scope of a subunit shall have unique *list_ID* values, so there is no need to resolve the scope any further.

The *descriptor_identifier* for an object list specified by its *list_type* is as follows:

descriptor_identifier for an object list specified by list_type	
address offset	contents
00_{16}	descriptor_type = 11_{16}
01_{16}	list_type

10.1.3 Object Entry

Objects can be referenced either by their **position** in a specified list, or by their unique **object ID**. The *descriptor_identifier* for an object entry depends on which of these reference methods are being used:

descriptor_identifier for an object entry position reference	
address offset	contents
00_{16}	descriptor_type = 20_{16}
01_{16}	list_ID (MSB)
:	
:	list_ID (LSB)
:	object_position (MSB)
:	
:	object_position (LSB)

The *object_position* field indicates the position of the target object, within the list which was specified by the *list_ID* field. The value of all FF_{16} bytes for the *object_position* is reserved, and has a special meaning when used with the WRITE DESCRIPTOR command. Please refer to the definition of that command for details.

For the object ID reference, the *descriptor_identifier* is as follows:

descriptor_identifier for an object ID reference	
address offset	contents
00_{16}	descriptor_type = 21_{16}
01_{16}	root_list_ID (MSB)
:	
:	root_list_ID (LSB)
:	list_type
:	object_ID (MSB)
:	:
:	:
:	object_ID (LSB)

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 which share the same *list_type* value, within the hierarchy indicated by *root_list_ID*.

WARNING: In some cases, this reference may not uniquely identify a specific object within a hierarchy. In case one or more objects in the hierarchy indicated by the root list have the same ID and belong to lists with the same list type, then an arbitrary object will be addressed. This will depend on the particular technology being represented by the objects and object lists. For one example, please refer to the DAB broadcasting system described in Rules and Guidelines for Tuner Subunit Objects and Object Lists in the tuner subunit specification.

10.1.4 Descriptor Access Support

While it is mandatory for a subunit which has descriptors to support the OPEN DESCRIPTOR command, it is not mandatory for it to support that command down to the individual OBJECT ENTRY level. In other words, it is not required that a subunit be sophisticated enough to allow one controller to have read/write access to object 2, and a separate controller to simultaneously have read/write access to object 7 in the same list. It is sufficient to allow access control only at the OBJECT LIST level. If the subunit does not support access control at the individual object entry level, then it shall return a value of 05_{16} in the last operand of the status command, as shown in the table of response values in the section titled The OPEN DESCRIPTOR Status Command which begins on page 53.

For those object entries which can be modified by a controller, it is mandatory for the target to support READ and WRITE operations of individual object entries within the object list, once **access** to the entire list has been established by a controller.

If a controller intends to read or modify a single object entry, and it is able to gain access control to that individual object entry, then it shall relinquish control of that same object entry when it is finished. If a controller intends to add or delete an entire object entry, then it shall first gain write access control to the entire object list.

The *descriptor_identifier* operand specifies which, among possibly many, of the items that the controller wants to access. There shall be only zero or one (SUB)UNIT IDENTIFIER for a given (sub)unit, but there may be many OBJECT LISTS, and many OBJECT ENTRIES within each OBJECT LIST.

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

Subfunction	Value	Action
CLOSE	00_{16}	Relinquish use of the descriptor
READ OPEN	01_{16}	Open the descriptor for read-only access
WRITE OPEN	03_{16}	Open the descriptor for read or write access

If the subunit which owns the descriptor must change the descriptor while it is open for read or read/write access, then the subunit shall force the descriptor to be closed for all controllers who have access. One example of this would be for a tuner subunit, which may have to update a list of services that are currently available on a multiplex. If that list happens to be open for one or more controllers, then the tuner subunit shall close the list for all of them before it updates the list.

10.1.5 The OPEN DESCRIPTOR Status Command

OPEN DESCRIPTOR may also be used as a status command to inquire about the current condition of the descriptor. The format of the OPEN DESCRIPTOR status command is shown below:

	msb							lsb
opcode	OPEN DESCRIPTOR (08_{16})							
operand[0]	descriptor_identifier(MSB)							
operand[1]	:							
:	:							
:	(LSB)							
operand[x]	FF_{16}							
:	reserved							
operand[y]	FF_{16}							
:	FF_{16}							

The STABLE response frame returned by the subunit updates the operand indicated by the label “operand[x]” to reflect the current access state for the descriptor specified by the *descriptor_identifier* operand, as summarized below:

Value of “operand[x]”	Meaning
00_{16}	The descriptor specified by descriptor_identifier is closed for access. No controllers currently have access (either read or write).
01_{16}	The descriptor is open for read-only access to the data by one or more controllers, and is able to accept additional read-only open requests.
04_{16}	No descriptor specified by descriptor_identifier exists.
05_{16}	Access control for individual object entries in an object list is not supported - the controller must obtain access to the entire list.
11_{16}	The descriptor is open for read-only access to the data, and is unable to accept any additional read-only open requests.
33_{16}	The descriptor is open for write access to the data. No access (read only or read/write) by other controllers is allowed.

It is important to control access to the descriptor contents even for read-only operations, because access to the descriptor data may require coordination between the actions of a controller that is writing to the descriptor, and any controllers that want to read the data.

Also in the STABLE response frame, the subunit shall update the operand indicated by the label “operand[y]” based on the value returned for “operand[x]”, as indicated below:

Value of “operand[x]”	Value of “operand[y]”
33_{16}	When the descriptor is open for READ/WRITE access, then the subunit shall update “operand[y]” to contain the node_ID of the controller with read/write access.
all other values	The value of “operand[y]” is not changed in the response frame; it remains as FF FF_{16}.

With the exception of bus resets, write-access rules and the time out rules described below, the descriptor shall only be closed by the controller which opened it or by the target itself. Thus, it will be necessary for the target to maintain the necessary information about each controller with access. When a close request is made, the target shall verify the controller node ID before permitting the action to proceed.

The target shall also verify that the controller that issues a READ DESCRIPTOR or WRITE DESCRIPTOR command has the access rights to that descriptor.

The OPEN DESCRIPTOR command has write-access, time out and bus reset requirements as described here:

> After a power reset, command reset or Serial Bus reset the descriptor of any subunit shall be in a closed state.
>
> If a descriptor is closed or has only been opened for read access, a subunit may accept any number of OPEN DESCRIPTOR requests with a *subfunction* of READ OPEN as long as the subunit is able to accommodate additional read-only access controllers.
>
> If a descriptor is closed or has only been opened for read access, a subunit may accept a single OPEN DESCRIPTOR control command with a *subfunction* of WRITE OPEN. If accepted, this OPEN DESCRIPTOR operation for write access forces any existing read only opens to be closed.
>
> If a descriptor is open for write access, a subunit shall reject any OPEN DESCRIPTOR control commands except for a command with a *subfunction* of CLOSE sent by the controller that opened the descriptor for write access.
>
> A subunit shall implement a time-out period, recommended to be longer than one minute since the last accepted OPEN DESCRIPTOR (for read-only and read/write access), READ DESCRIPTOR or WRITE DESCRIPTOR control commands. If this time-out period expires, then the subunit shall close the descriptor immediately. The descriptor is then available to be opened again, by any controller.
>
> When the descriptor has been opened for read/write access, the time out period shall be measured between the subunit’s response to the initial OPEN DESCRIPTOR (read/write subfunction) and the first ensuing READ DESCRIPTOR or WRITE DESCRIPTOR command issued by the controller. If the READ DESCRIPTOR or WRITE DESCRIPTOR command is not issued before the time out period, then the descriptor shall be closed for read/write access. The subunit shall also measure the time out between its response to a READ DESCRIPTOR or WRITE DESCRIPTOR command, and the subsequent READ DESCRIPTOR or WRITE DESCRIPTOR command issued from the controller. If the controller fails to either close the descriptor or issue another READ DESCRIPTOR or WRITE DESCRIPTOR command within the time out period, then the subunit shall close the descriptor for read/write access. The descriptor is then available to be opened again, by any controller.
>
> Similar rules apply for read-only access, concerning the time between the initial OPEN DESCRIPTOR (read only) response and the subsequent READ DESCRIPTOR command from the controller, and the time between the subunit’s response to a READ DESCRIPTOR command and the next READ DESCRIPTOR command issued by the controller. When the subunit closes the descriptor for read access, it shall do so ONLY for that controller which failed to respond within the time out period. All other controllers that have read access and are still within their time out limits shall retain read access.
>
> Note that all of these time out measurements involve the same 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.

The rules described above are intended to help the target maintain a fair and stable environment for descriptor access by controllers. Controllers are strongly recommended to keep descriptors open (for read only or read/write access) only for the duration that access is needed, and to relinquish access as soon as possible.

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 (when the state of the descriptor changes based on the status values shown in the table above). The format of the notify command is the same as status command, but with a ctype value of NOTIFY. Controllers that want to know about changes to a descriptor should ask for notification on the OPEN DESCRIPTOR command; when they are notified that it has been closed after having been opened for write access, then they must determine if the data they care about has changed.

10.2 READ DESCRIPTOR command

The READ DESCRIPTOR control command is used to read the data specified by the *descriptor_identifier* from the descriptor. The format of the READ DESCRIPTOR control command is illustrated by the figure below:

	msb							lsb
opcode	READ DESCRIPTOR (09_{16})							
operand[0]	descriptor_identifier (MSB)							
operand[1]	:							
:	:							
:	descriptor_identifier (LSB)							
:	read_result_status							
:	reserved							
:	data_length (MSB)							
:	(LSB)							
:	address (MSB)							
:	(LSB)							

The *descriptor_identifier* describes which descriptor structure is being accessed. The exact format of this identifier will vary based on the kind of descriptor (e.g., subunit identifier descriptor, object, object list), the method of specifying this descriptor, and on the type of subunit for which the descriptor is defined.

The *read_result_status* operand is set by the target in the response frame. For the control command frame, the controller shall set this field to FF_{16}. For details on what this field means, please refer to the description of the possible values returned for this field, below.

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 entire descriptor is to be read. In the response frame, this field will be updated to contain the actual number of bytes that were read.

IMPORTANT: Reading beyond the end of a data boundary is not permitted. If a target receives a read request with a *data_length* that would result in reading past the end of the data indicated by the *descriptor_identifier* operand, then it shall return only the legitimate data, and update the *data_length* field to reflect the size of this data. It shall also update the *read_result_status* field as described in the table below.

The *address* field specifies the address of the starting point to be read. It is an offset from the beginning of the particular entity described by *descriptor_identifier*. When *data_length* = 0, then the *address* field shall be ignored and the target shall return the entire descriptor.

When *data_length* is not 0, if there is no data at the specified address or if the address is not valid, then the target shall REJECT the command.

If an ACCEPTED response frame is returned by the target after a READ DESCRIPTOR control command, the response data consists of additional operands, 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 operand 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 the table below.

The following table summarizes the values that shall be set by the target for the *read_result_status* field in the response frame, and what actions the controller should take as a result:

When this occurs..	...return this value in the read_result_status operand...	...return this value in the data_length operand...	...and the controller will understand this:
The READ request can be handled with no problem	10_{16}	The original length value passed as input to the command	The returned data is complete
The READ request was only partially satisfied due to data transfer capacity limitations	11_{16}	The actual number of bytes read (will be different from the input value)	The controller must issue additional READ command(s) to get all of the desired data
The READ request started in valid data space, but went beyond the end of valid data space	12_{16}	The actual number of bytes read (will be different from the input value)	Only the actual number of bytes read were available with the specified command operands
The READ request began in invalid data space, or no data was at the specified address	N/A (return a REJECTED response frame)	N/A	The operands specified an invalid condition, or some other abnormal situation occurred

An INTERIM response frame without data may be returned in advance of the ACCEPTED or REJECTED response.

10.3 WRITE DESCRIPTOR command

The WRITE DESCRIPTOR control command is used to store variable-length data in the descriptor of the target. The format of the WRITE DESCRIPTOR control command is illustrated by the figure below:

	msb							lsb
opcode	WRITE DESCRIPTOR ($0A_{16}$)							
operand[0]	descriptor_identifier (MSB)							
operand[1]	:							
:	:							
:	(LSB)							
:	subfunction							
:	group_tag							
:	data_length (MSB)							
:	(LSB)							
:	address (MSB)							
:	(LSB)							
:	data							
:								
:								

The *descriptor_identifier* describes which descriptor structure is being accessed. The exact format of this identifier will vary based on the kind of descriptor (subunit identifier descriptor, object, object list), the method of specifying this descriptor, and on the type of subunit for which the descriptor is defined. For more information, please refer to the descriptor identifiers described in the OPEN DESCRIPTOR command.

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:

subfunction	value	action
change	10_{16}	Overwrite the descriptor part indicated by the *address* and *length* operands. The controller is responsible for making sure that exactly the same number of bytes are being used to replace existing data.
replace	20_{16}	Overwrite the 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.
insert	30_{16}	Perform an "insert descriptor" operation, inserting the new descriptor before the one specified by the operand specified by *descriptor_identifier*. The *address* operand is ignored for this subfunction. This subfunction applies only to inserting object entries into a list. See NOTE* below.
delete	40_{16}	Perform a "delete descriptor" operation, deleting the descriptor specified by *descriptor_identifier*. For this subfunction, *address* and *data_length* are ignored. The target is responsible for adjusting the size of descriptor storage to accommodate this operation.
partial_replace	50_{16}	Perform a "partial insert" or "partial delete" operation. The subunit updates the descriptor structure to deal with the change in size (bytes removed or added).
X	all others	reserved for future definition

IMPORTANT: Uncontrolled writing beyond the end of a data boundary is not permitted. For the "change" subfunction, 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_identifier* operand, then it shall REJECT the command. This applies to the subunit identifier, object list and individual entries within the object list. For fields within an object entry, the controller is responsible for making sure not to overwrite the field being modified.

NOTE*: To use the subfunction *insert* for placing items at the end of a list, the controller must specify a certain value to indicate where the insert shall take place. When using an object entry position reference for the *descriptor_identifier*, specifying a value of all FF_{16} bytes for the object position field means "add to the end of the list". When using an object ID reference, specifying a value of all FF_{16} bytes for the object ID field means "add to the end of the list".

The *group_tag* operand is used for indivisible update operations on a descriptor, in which several WRITE DESCRIPTOR commands must be issued. The controller may use this field to specify an arbitrary number of update operations which must be performed on an "all or nothing" basis. This field may take one of the following values:

group_tag	value	action
immediate	00_{16}	Immediately write the data to the descriptor.
first	01_{16}	Begin an "indivisible 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 can be preserved should the sequence not finish normally.
continue	02_{16}	Any number of subsequent commands may be issued using this tag, after the "first" has been issued.
last	03_{16}	This command signals the last in the sequence of indivisible update operations. When the target subunit receives this *group_tag*, it commits all of the commands in the sequence, including this one, to the specified descriptor structure. If the subunit 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 commands in this sequence and leave the specified descriptor unmodified.
X	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 subunit. Supporting the *group_tag* is optional; if a 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. Controllers should *always* verify the results of any WRITE DESCRIPTOR actions, whether individual commands or a sequence of indivisible update operations.

The *data_length* operand specifies the number of bytes 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 either ACCEPT or REJECT the command. However, if it does drop excess data and ACCEPT the command, then this must be indicated to the controller so that it can respond accordingly.

If the combination of *data_length* and *address* reference invalid memory, then the target shall REJECT the command.

The result shall be indicated by having the target modify the low nibble of the subfunction field in the returned response (either the ACCEPTED or REJECTED response) according to these rules:

response type	returned subfunction value	action
ACCEPTED	$x0_{16}$	The specified subfunction was performed with no problem.
ACCEPTED	$x1_{16}$	The data write operation was only partially satisfied, so some data was dropped. The controller will need to issue more commands
REJECTED	$x2_{16}$	The target supports the specified descriptor_type but not the specified descriptor_type_specific_reference in the descriptor_identifier, **or** the target supports the specified descriptor_identifier but the address and data_length fields specified an invalid address, so the write operation was not performed.
NOT IMPLEMENTED	same as control command	The target does not support the specified descriptor_type.
-	all others	reserved for future definition

In the case that $x1_{16}$ is returned, then the target shall also update the *data_length* field in the ACCEPTED response frame to indicate how many bytes were actually written. The controller will then know that it must issue more WRITE DESCRIPTOR commands to completely write the data.

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 *data_identifier*.

The *data* field contains the data bytes to be stored, the length of which is indicated by *data_length*.

An INTERIM response frame may be returned in advance of the ACCEPTED or REJECTED response.

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

10.3.1 Partial Replace Operations

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

	msb							lsb
opcode	WRITE DESCRIPTOR (0A$_{16}$)							
operand[0]	descriptor_identifier (MSB)							
operand[1]	:							
:	:							
:	(LSB)							
:	subfunction “partial_replace” = 50$_{16}$							
:	group_tag							
:	replacement_data_length (MSB)							
:	(LSB)							
:	address (MSB)							
:	(LSB)							
:	original_data_length							
:								
:	replacement_data							
:								
:								

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

The *replacement_data_length* field specifies the number of bytes in the *replacement_data* operand. 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_lengh* operand shall be greater than 0, indicating the number of bytes to be deleted.

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.

The combination of *replacement_data_length* = 0 and *original_data_length* = 0 is illegal.

The *address* operand specifies where the operation (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.

10.3.2 WRITE DESCRIPTOR Status

In addition to the use of the WRITE DESCRIPTOR control command, WRITE DESCRIPTOR with a ctype of STATUS may also be used to determine the subunit’s capability to accept data in its descriptor in a single operation. In this case, the format shown by the figure below is used:

	msb							lsb
opcode	WRITE DESCRIPTOR (0A$_{16}$)							
operand[0]	descriptor_identifier (MSB)							
operand[1]	:							
:	:							
:	(LSB)							
:	FF$_{16}$							
:	FF$_{16}$							
:	FF$_{16}$							
:	FF$_{16}$							

The *descriptor_identifier* field is as described above. The remaining 4 bytes shall be set to FF_{16} on input.

The response frame for the WRITE DESCRIPTOR status command is the same as the control command frame, except that the variable sized *data* operand and the *address* operand are not included. In the STABLE response frame returned by the subunit, the operands after the *descriptor_identifier* will be replaced with the *subfunction, group_tag* and *data_length* as shown in the control command frame.

The *data_length* operand shall indicate 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 this means that there is no write access possible for the descriptor because it can't be modified.

The *subfunction* operand shall be set to FF_{16} in the status command response frame, but will have no meaning.

The *group_tag* operand shall be set to 00_{16} in the response frame.

10.4 SEARCH DESCRIPTOR command

The SEARCH DESCRIPTOR command allows a controller to request the 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 which identifies the first candidate which was found; multiple specifiers are not returned by the search operation. The controller must specify additional searches to find additional instances which match the search criteria.

The control command has the following format:

	msb							lsb
opcode	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	search_for							
:								
:								
:	search_in							
:								
:								
:	start_point							
:								
:								
:	direction							
:	response_format							
:	status							

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

address offset	msb						lsb
	search_for operand of SEARCH DESCRIPTOR						
00_{16}	length						
01_{16}	search_data						
:							
:							

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

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 subunit shall search through all descriptors which match the *search_in* operand, even if they are open for modification by a controller.

This operand has the following format:

address offset	msb						lsb
	search_in operand of SEARCH DESCRIPTOR						
00_{16}	length						
01_{16}	type						
02_{16}	type_specific_info						
:							
:							

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

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

type value for *search_in*	meaning
10_{16}	Object Lists
20_{16}	Objects in Object List Descriptors
30_{16}	Other Descriptors
50_{16}	Fields specified by an offset address and length in Object List Descriptors
52_{16}	*list_type* fields in Object List Descriptors
60_{16}	Fields specified by an offset address and length in Object Descriptors
62_{16}	*entry_type* fields in Object Descriptors
64_{16}	*child_list_ID* fields in Object Descriptors
66_{16}	*object_ID* fields in Object Descriptors
70_{16}	Fields specified by an offset address and length in Other Descriptors
all others	reserved for future specification

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 the section titled type_specific_info for the search_in operand on page 66.

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

address offset	msb							lsb
	start_point operand of SEARCH DESCRIPTOR							
00_{16}	length							
01_{16}	type							
02_{16}	type_specific_info							
:								
:								

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

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

type value for *start_point*	starting point for the search
00_{16}	The controller does not care where the start point is - the subunit chooses where to start the search operation.
02_{16}	At the “current location”, where the current location is defined by the currently selected object 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 Object List Descriptor, where the list is specified by its *list_ID*.
11_{16}	At the *list_type* field in the specified Object List Descriptor, where the list is specified by its *list_ID*.
20_{16}	At the point specified by an offset address in the specified Object Entry Descriptor, where the object is specified by *object_position*.
21_{16}	At the point specified by an offset address in the specified Object Entry Descriptor, where the object is specified by *object_ID*.
22_{16}	At the *entry_type* field in the specified Object Entry Descriptor, where the object is specified by *object_position*.
23_{16}	At the *entry_type* field in the specified Object Entry Descriptor, where the object is specified by *object_ID*.
24_{16}	At the *child_list_ID* field in the specified Object Entry Descriptor, where the object is specified by *object_position*.
25_{16}	At the *child_list_ID* field in the specified Object Entry Descriptor, where the object is specified by *object_ID*.
26_{16}	At the *object_ID* field in the specified Object Entry Descriptor, where the object is specified by *object_position*.
27_{16}	At the *object_ID* field in the specified Object Entry Descriptor, where the object 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_identifier* structure.
all others	reserved for future specification

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 the section titled *type_specific_info* for the *start_point* operand on page 69.

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

direction	meaning
00_{16}	The controller does not care where about the direction of the search - the subunit 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 *object_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 *object_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:

search_in	rules for search *direction*
a field	The address value within the field (increasing -10 - or decreasing - 20 - addresses)
an Object Entry Descriptor	
an Object List Descriptor	
fields in objects	The *object_entry_position* value (increasing -12 - or decreasing - 22 -)
objects	The *object_id* value (increasing - 13 - or decreasing - 23 -)
fields in lists	The *list_id* value (increasing - 10 - or decreasing - 20 -)
lists	

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

response_format	meaning
00_{16}	Not specified - the subunit may choose how to present the data.
10_{16}	By descriptor_type 10_{16} (specified by list_ID)
11_{16}	By descriptor_type 11_{16} (specified by list_type)
20_{16}	By descriptor_type 20_{16} (object_position)
21_{16}	By descriptor_type 21_{16} (object_ID)
all others	reserved for future specification

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.

The ACCEPTED response frame has the following format:

	msb							lsb
opcode	SEARCH DESCRIPTOR ($0B_{16}$)							
operand[0]	search_for							
:								
:								
:	search_in							
:								
:								
:	start_point							
:								
:	direction							
:	response_format							
:	status							
:	descriptor_identifier							
:								
:								
:	address							
:								

The operands from *search_for* through *direction* are as described above, and will be returned with the same values as were passed in the control frame.

The *response_format* operand will either contain the originally specified value (in the case where a specific format was requested by the controller), or it will contain the format chosen by the subunit (in the case where the controller specified a “don’t care” format).

The *status* operand specifies the status of the search after completion. The status can be successful, meaning that either the specified data or any data matching the search criteria was found, or unsuccessful, meaning that no data matching the search criteria was found. Note that while the SEARCH DESCRIPTOR command may be accepted by the subunit, the search might not be successful. The following table specifies the values which may be returned for the *status* operand:

response frame	*status*	meaning
ACCEPTED	10_{16}	Successful - the specified data was found.
REJECTED	20_{16}	The specified data was not found.
	21_{16}	The length of the *search_for* operand exceeds the capability of the subunit. See NOTE*.
---------------	all others	reserved for future specification

NOTE: In the case of returned status 21_{16}, the *length* field of the *search_for* operand shall be updated to indicate the maximum length of the *search_for* specification supported by the subunit.

The *descriptor_identifier* operand identifies the data which is being returned. It will be in the format indicated by the *response_format* operand.

The *address* operand specifies where the returned data can be found, in the case of a search for a data field. In the case of descriptor structures such as lists, objects or any other descriptor, the *descriptor_identifier* will contain enough information for the controller to access the data. In these cases, the address field shall be ignored by the controller. This operand is 2 bytes in length.

10.4.1 *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.

10.4.1.1 Supporting Data Structures

Many of the *type_specific_info* structures for the *search_in* operand make use of the following two structures:

10.4.1.1.1 object_entry_descriptor_specifier

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

address offset	msb							lsb
	object_entry_descriptor_specifier							
00_{16}	type							
:	type_specific							
:								

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

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

type	meaning	*type_specific* field	size of *type_specific* field
20_{16}	a specified object (by position)	object_position	k bytes (see NOTE below)
21_{16}	a specified object (by object_ID)	object_ID	k bytes (see NOTE below)
22_{16}	any objects with the specified entry_type field	entry_type	1 byte
$2F_{16}$	any objects	none	zero bytes
all others	reserved for future specification	--------------	--------------

NOTE: When an object is specified by its *object_ID*, the size of the *type_specific* field is indicated by the *size_of_object_ID* field of the subunit identifier descriptor. When an object is specified by its *object_position*, the size of the *type_specific* field is indicated by the *size_of_object_position* field of the subunit identifier descriptor.

10.4.1.1.2 object_list_descriptor_specifier

The *object_list_descriptor_specifier* is a data structure which identifies an object list (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	msb							lsb
	object_list_descriptor_specifier							
00_{16}	type							
:	type_specific							
:								

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

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

type	meaning	*type_specific* field	size of *type_specific* field
10$_{16}$	a specified list (by list_ID)	list_ID	n bytes *
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 subunit identifier descriptor.

10.4.1.1.3 descriptor_identifier

The general *descriptor_identifier* structure is already defined in the OPEN DESCRIPTOR command. For the SEARCH DESCRIPTOR command, it is used to specify one of the (non-object and non-list) descriptor structures (such as the Subunit Identifier Descriptor) in which the search operation is to be performed.

10.4.1.1.4 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.

10.4.1.2 The type_specific_info Data Structures

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 10$_{16}$							
00$_{16}$	object_list_descriptor_specifier							
:								
:								

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 20$_{16}$							
00$_{16}$	object_list_descriptor_specifier							
:								
:								
:	object_entry_descriptor_specifier							
:								
:								

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 30$_{16}$							
00$_{16}$	descriptor_identifier							
:								
:								

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 50_{16}							
00_{16}	object_list_descriptor_specifier							
:								
:								
:	offset_address							
:								
:	length							

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 52_{16}							
00_{16}	object_list_descriptor_specifier							
:								
:								

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 60_{16}							
00_{16}	object_list_descriptor_specifier							
:								
:								
:	object_entry_descriptor_specifier							
:								
:								
:	offset_address							
:								
:	length							

address offset	msb							lsb
	type_specific_info for the *search_in* operand, *type* 62_{16}							
00_{16}	object_list_descriptor_specifier							
:								
:								
:	object_entry_descriptor_specifier							
:								
:								

address offset	msb						lsb
	type_specific_info for the *search_in* operand, *type* 64_{16}						
00_{16}	object_list_descriptor_specifier						
:							
:							
:	object_entry_descriptor_specifier						
:							
:							

address offset	msb						lsb
	type_specific_info for the *search_in* operand, *type* 66_{16}						
00_{16}	object_list_descriptor_specifier						
:							
:							
:	object_entry_descriptor_specifier						
:							
:							

address offset	msb						lsb
	type_specific_info for the *search_in* operand, *type* 70_{16}						
00_{16}	descriptor_identifier						
:							
:							
:	offset_address						
:							
:	length						

10.4.2 *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.

10.4.2.1 Supporting Data Structures

The *type_specific_info* structures of the *start_point* operand make use of the *descriptor_identifier* specifier structures defined by the OPEN DESCRIPTOR command, INCLUDING those for the object entry and object list descriptors. Note that this is different from the SEARCH DESCRIPTOR command, which by necessity had to define its own object and object 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 object entry, as defined by the *entry_type* field of the object descriptor structures.

10.4.2.2 The type_specific_info Data Structures

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 00_{16}							
------	There is no type_specific_info for type 00_{16}							

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 02_{16}							
------	There is no type_specific_info for type 02_{16}							

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 03_{16}							
------	There is no type_specific_info for type 03_{16}							

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 10_{16}							
00_{16}	descriptor_identifier for an object list specified by list_ID							
:								
:								
:	offset_address (2 bytes)							
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 11_{16}							
00_{16}	descriptor_identifier for an object list specified by list_ID							
:								
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 20_{16}							
address offset 00_{16}	msb							lsb
:	descriptor_identifier for an object entry position reference							
	type_specific_info for the *start_point* operand, *type* 21_{16}							
address offset 00_{16}	msb							lsb
:	descriptor_identifier for an object ID reference offset_address							
	type_specific_info for the *start_point* operand, *type* 22_{16}							
address offset 00_{16}	msb							lsb
:	descriptor_identifier for an object entry position reference offset_address							
	type_specific_info for the *start_point* operand, *type* 23_{16}							
address offset 00_{16}	msb							lsb
:	descriptor_identifier for an object ID reference entry_type							
	type_specific_info for the *start_point* operand, *type* 24_{16}							
00_{16}								
:	descriptor_identifier for an object entry position reference entry_type							
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 25_{16}							
00_{16}	descriptor_identifier for an object ID reference							
:								
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 26_{16}							
00_{16}	descriptor_identifier for an object entry position reference							
:								
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 27_{16}							
00_{16}	descriptor_identifier for an object ID reference							
:								
:								

address offset	msb							lsb
	type_specific_info for the *start_point* operand, *type* 30_{16}							
00_{16}	descriptor_identifier “other” descriptor							
:								
:								
:	offset_address							
:								

10.4.3 Examples of the SEARCH DESCRIPTOR Command (Informative)

This section presents some examples of how to use the SEARCH DESCRIPTOR command for various types of searches. For those examples which refer to a specific type of subunit, supporting information may be found in the specification document for that subunit.

10.4.3.1 Example 1: Search for the *service_name* “NHK” in all DVB service lists (tuner subunit)

This example demonstrates how an arbitrary field of an object 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 objects 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$_{16}$)						

10.4.3.2 Example 2: Search for the next service object in the service lists (tuner subunit)

This example shows how to search on one of the basic fields defined for all object 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 objects in the lists						
operand[6]	**start_point** (02$_{16}$) start at the currently selected object						
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$_{16}$)						

10.4.3.3 Example 3: Search for the parent object 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 objects 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 object position reference							
operand[11]	**status** (FF_{16})							

10.5 OBJECT NUMBER SELECT command

The OBJECT NUMBER SELECT command performs a selection of an object (or many objects). This is achieved by specifying, for each desired object, a particular list and an object in that list. The nature of what it means to “select” an object will be defined by the type of subunit receiving the command. For details on subunit-specific functionality, please refer to the OBJECT NUMBER SELECT reference in the appropriate subunit-specific documents.

The general operation of ONS is to allow the controller to specify which object(s) to select, a source plug that should receive the data from the object(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:

	msb							lsb
opcode	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	source_plug							
operand[1]	subfunction							
operand[2]	status							
operand[3]	number_of_ons_selection_specifications (n)							
operand[4]	ons_selection_specification[0]							
:								
:								
:								
:	:							
:	ons_selection_specification[n - 1]							
:								
:								
:								

The *source_plug* operand indicates the subunit source plug number which shall output the specified object(s). The plug value FE_{16} is a special case; it means that the specified item(s) should be “selected”, but not output to any source plug of the subunit. 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.

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 subunit receiving this command. Different types of subunits may not be able to support all of the general subfunctions, and they may define subunit type-specific subfunctions. Please refer to the appropriate subunit 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:

subfunction	value	action
clear	$C0_{16}$	Stop the output of all selections on the specified plug. No selection specifiers shall be included in the command frame for this subfunction.
remove	$D0_{16}$	Remove the specified selection from the output stream on the specified plug.
append	$D1_{16}$	Add (multiplex) the specified selection to the current output.
replace	$D2_{16}$	Remove the current selection from the specified plug, and output or multiplex the specified selection.
new	$D3_{16}$	Output the specified selection on the specified plug if the plug is currently unused; otherwise, REJECT the selection command.
X	all others	reserved for future specification

IMPORTANT NOTE: The values defined for the subfunction indicate that the two most significant bits are set to 1. These two bits are reserved for future specification, and are set to 1 so that they are compatible in polarity with the existing CONNECT commands. Controllers should be aware of this when examining the values of subfunctions for all selection commands. Please refer to the section titled Rules for Reserved Fields on page 34 for details on handling reserved fields.

The *status* field shall be set to FF_{16} on input to the control command. In the ACCEPTED response frame, this field shall be updated with the appropriate value as defined here:

status value	meaning
00_{16}	the selection specification indicated a unique item, which was selected
01_{16}	the selection specification was ambiguous, so the subunit selected one (please refer to the “don’t care” path specification description below)
all others	reserved for future specification

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 *number_of_ons_selection_specifications* operand shows the number of ONS selection specifiers that are provided in the parameter block.

The *ons_selection_specification[x]* operands each define a single object to be selected. The ons_selection_specification structure has two basic forms: a full path specification from the root of a hierarchy down to the object being selected; and a “don’t care” version of this specifier which does not indicate a path to an object. These are illustrated in the following sections.

Some of the contents of this specification will vary based on the type of subunit (target) which is receiving this command. For details on subunit-specific descriptors, please refer to the OBJECT NUMBER SELECT reference in the appropriate subunit-specific sections.

10.5.1 Subfunction Implementation Rules

The following rules shall be adhered to when a subunit implements the OBJECT NUMBER SELECT command subfunctions. Note that some types of subunits may add further rules, but they shall not conflict with these general rules:

> The remove subfunction shall be REJECTED if the specified information instances are not present on the specified output plug.
>
> 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.

10.5.2 The General ons_selection_specification Structure

The general ons_selection_specification is as follows:

General ons_selection_specification (full path specification)	
address offset	contents
00_{16}	root_list_ID
:	
:	selection_indicator
:	target_depth (m)
:	path_specifier[0]
:	
:	
:	
:	:
:	path_specifier[m - 1]
:	
:	
:	
:	target
:	
:	
:	

The fields of this *ons_selection_specification* define exactly one path, among possibly many, to the desired object being selected. This is necessary because in general, an object may have more than one parent and hence more than one path specification.

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 hierarchy (e.g., it must be referenced from the subunit identifier descriptor). The root lists will have fixed or well-known ID values.

The *selection_indicator* field indicates how the object 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:

value	selection_indicator for the full path specification
1xxx xxxx	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 object 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 object. When this flag is zero, then the entire object is to be selected (no specific CHILDREN are specified). Please see the example selections for details.
all others	reserved for future specification

The *target_depth* field indicates how deep in the hierarchy to go in order to find the list which holds the target object(s) to be selected. The root of the hierarchy has a depth of zero.

The *path_specifier[x]* fields appear in order from the top of the hierarchy down. A level corresponds to a list, and each *path_specifier* indicates an object in the list at that level of the 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.

The *target* field will indicate which object is to be selected from the target level list indicated by the path specification. The general format of the target is as follows:

format of the *target* field (full path specification)	
address offset	contents
00_{16}	target_object_reference
:	
:	
:	
:	number_of_children (m)
:	child_object_reference [0]
:	
:	
:	:
:	child_object_reference [m - 1]
:	
:	

The format of the *target_object_reference* and *child_object_reference[x]* fields is specified by the *target_format_flag* of the *selection_indicator* field described above. If the *selection_indicator* indicates that no specific child objects are to be selected, then the *number_of_children* and *child_object_reference[x]* fields are not present.

The format of the *target* field may also be defined in subunit-specific ways if required.

10.5.3 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 list 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 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 object to be selected. The format of the "don't care" ons_selection_specification is as follows:

ons_selection_specification ("don't care" specification)	
address offset	contents
00_{16}	root_list_ID
:	
:	selection_indicator
:	target_depth (m) = FF_{16}
:	target
:	
:	
:	

The *root_list_ID* specifies the root list of the hierarchy from which the objects will be selected. This narrows down the scope of where an object with a given object ID is located.

The *selection_indicator* field is described in the table below. For the don't care specification, the *target_depth* field shall be FF_{16}.

value	selection_indicator for the "don't care" specification
1xxx xxxx	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 object 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 object. When this flag is zero, then the entire object is to be selected (no specific CHILDREN are specified). Please see the example selections for details.
all others	reserved for future specification

When the *specifier_type_flag* is zero, then the *target* field shall have the following format:

target field ("don't care" specification) when specifier_type_flag = 0	
address offset	contents
00_{16}	list_ID
:	
:	target_object_reference (object_position)
:	
:	number_of_children (m)
:	child_object_reference[0] (object_position)
:	
:	
:	:
:	child_object_reference[m - 1] (object_position)
:	
:	

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:

target field (“don’t care” specification) when specifier_type_flag = 1	
address offset	contents
00_{16}	list_type
01_{16}	target_object_reference (object_ID)
:	
:	
:	number_of_children (m)
:	child_object_reference[0] (object_ID)
:	
:	
:	:
:	child_object_reference[m - 1] (object_ID)
:	
:	

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.

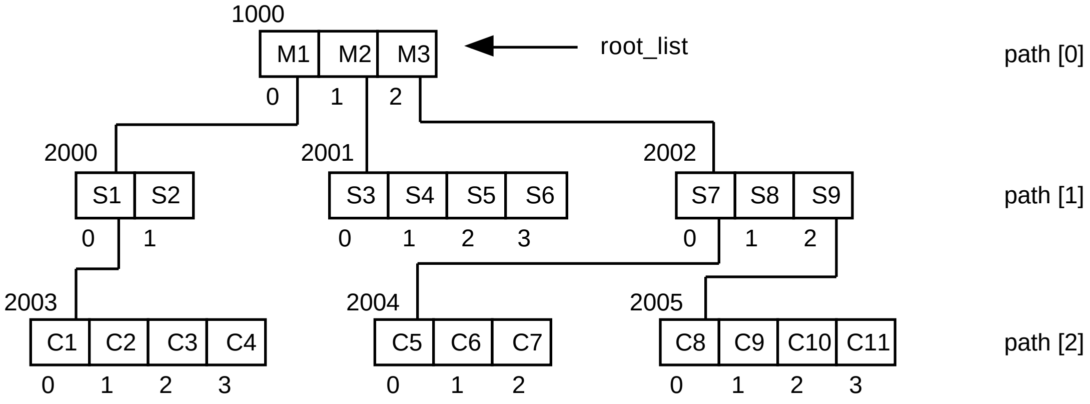
10.5.4 Object Selection Examples

The following diagrams illustrate various types of object selection, as indicated by the *selection_indicator* described above.

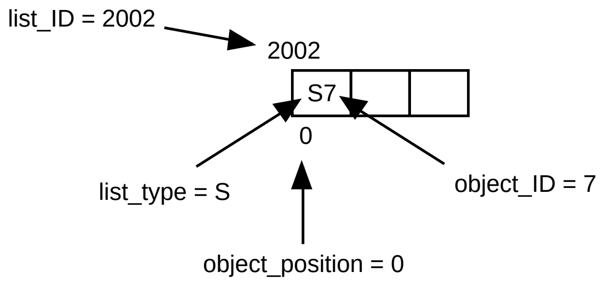
Example 1 shows the selection of an entire object (S7) with the object reference being the object ID. The results of the selection will be object S7 composed of all three of its children (C5, C6 and C7).

Example 2 shows the selection of the same object (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 object “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: ons_selection_specification for the object “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 object by specifying a subset of its child objects (the objects from its child list). Note that when specifying the path, we stop at the target object, and do not go down to the child list level. Because there is only one child list for the target object, there is no ambiguity; we then just specify which child objects to use from the implied child list. In this example, all references (including the child references) were defined using object_ID.

10.5.5 Object Selection Semantics

The semantics of selecting an object 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 *object_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.

10.5.6 The OBJECT NUMBER SELECT Status Command

OBJECT NUMBER SELECT 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 illustrated by the figure below:

	msb							lsb
opcode	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	source_plug							
operand[1]	FF_{16}							

The *source_plug* operand indicates the subunit plug number for which status is being requested.

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:

	msb							lsb
opcode	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	source_plug							
operand[1]	status							
operand[2]	number_of_ons_selection_specifications (n)							
operand[3]	ons_selection_specification[0]							
:								
:								
:								
:	:							
:	ons_selection_specification[n - 1]							
:								
:								
:								

The *status* field describes the current situation of the specified object(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.

Only objects which have an entry in the object 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 this data will 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.

In addition, OBJECT NUMBER SELECT with a ctype of NOTIFY 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 is illustrated by the figure below:

	msb							lsb
opcode	OBJECT NUMBER SELECT ($0D_{16}$)							
operand[0]	source_plug							
operand[1]	FF_{16}							

The format of associated INTERIM and CHANGED responses is the same as the OBJECT NUMBER SELECT status response described above.

10.6 POWER command

The POWER control command is used to control the power status of the AV unit or one of its subunits determined by the AV/C address that is contained in the AV/C frame. The format of the POWER command is illustrated by Figure 10-1 below.

	msb							lsb
opcode	POWER ($B2_{16}$)							
operand[0]	power_state							

Figure 10-1 — POWER command format

When the POWER command is issued with a *ctype* value of CONTROL, the *power_state* field specifies the desired power state of the unit. Power on is encoded as 70_{16} and power off as 60_{16}.

Setting the power status of the AV unit to on or off shall cause the power of all of its subunits to be set in the same way. Setting the power status of a subunit shall not affect the power status of the AV unit or any of the other subunits.

The POWER command with a *ctype* value of STATUS may be used to determine the current power state of the AV unit or one of its subunits. In this case, *operand[0]* is set to $7F_{16}$ when the command is issued and is updated to the current power state when the STABLE response is returned.

The POWER command may also be used as a notify command. The notify command has the same syntax as the status command. A notification shall be returned by the target to the controller that issued the notify command in case the power state of the addressed unit or subunit changes. The notify response has the same format as the response frame.

10.7 RESERVE command

The RESERVE control command permits a controller to acquire or release exclusive control of the AV unit or one of its subunits determined by the AV/C address that is contained in the AV/C frame. The format of the command is illustrated by Figure 10-2 below.

	msb							lsb
opcode	RESERVE (01_{16})							
operand[0]	priority							
operand[1]	text							
…								
operand[12]								

Figure 10-2 — RESERVE control command format

The *priority* field shall specify the relative priority associated with the command. Zero has special meaning and indicates that no controller has reserved the AV (sub)unit. The other values, between one and 0F$_{16}$, indicate that the target holds a reservation for a controller. A *priority* value of four is, by convention, the standard priority that controllers are expected to use in the absence of other reasons for choosing a higher or lower priority.

The *text* field provides for up to 12 bytes of ASCII characters. If no *text* string is present, the bytes are expected to have a value of FF$_{16}$.

An AV (sub)unit accepts RESERVE control commands according to the following rules:

after a power-on reset or a command reset, the AV (sub)unit is in a free state and reports a *priority* value of zero in response to any RESERVE inquiries (see the discussion of RESERVE when *ctype* has a value of STATUS, below).

a) an AV (sub)unit that is in the free state may be reserved by any controller that issues a RESERVE control command. The target shall internally record the *priority* at which the reservation is made, the *text* string that accompanies the reservation and the 16-bit node ID of the controller. An ACCEPTED response guarantees to the controller that the reservation has succeeded.

NOTE: When a priority value is accepted by an AV (sub)unit and a reservation is established, the stored value is transformed according to the following table.

Command priority	Stored priority
0 — 1	priority
02$_{16}$ — 0E$_{16}$	priority & 0E$_{16}$
0F$_{16}$	priority

This has the effect of rounding most odd priorities down to a smaller even value.

b) while an AV (sub)unit holds a reservation for a controller, it shall reject any control commands other than RESERVE with a *ctype* of CONTROL that are issued by any other controller. The 16-bit node ID stored by the AV (sub)unit upon receipt of the RESERVE control command is the basis for accepting or rejecting control commands for controllers.

c) if a RESERVE control command is received from the same controller that holds the reservation, it shall be accepted. This permits the original controller to raise or lower the *priority* associated with the reservation.

d) if a RESERVE control command is received from a different controller than that which made the reservation, the AV (sub)unit shall reject the command unless the *priority* is greater than the current reservation priority. In the case where the new priority is greater than the current priority, the existing reservation is preempted and a reservation is established for the new controller according to the procedures already described in b).

e) If a RESERVE control command is addressed to the AV unit but that AV unit contains a subunit that already holds a reservation with an equal or higher priority, the RESERVE control command shall return a REJECTED response.

f) If a RESERVE control command is addressed to the AV unit and that AV unit contains no subunits that are already reserved with an equal or higher priority, then each existing reservation of a subunit shall be preempted and a reservation of the AV unit is established for the new controller according to the procedures already described in b).

g) Any control command that is addressed to a subunit within an AV unit that is reserved by a different controller than the one which issued the control command, shall be rejected.

When an AV (sub)unit detects a Serial Bus reset, it shall reset its reservation priority to zero (free) and set both the reservation node ID and the reservation text to values of all ones. Then, until the reservation has been reestablished, or until a period of ten seconds has elapsed, it shall reject all commands with a *ctype* of CONTROL except for RESERVE commands. This procedure permits the original holder of the reservation to reestablish the reservation with its reassigned node ID after the bus reset.

> **NOTE:** Controllers shall not issue RESERVE control commands within ten seconds of a bus reset unless they had established a reservation with the target AV (sub)unit prior to the bus reset. Because the node ID of the AV unit may have changed after the bus reset, a controller that wishes to reestablish (sub)unit reservations is expected to examine the unique identifier, EUI-64, in configuration ROM to locate the AV (sub)unit previously reserved.

Because of this restriction, the target can assume that a RESERVE command received within 10 seconds of a bus reset is legitimate, and shall therefore accept the reservation. Any controller may request the current reservation status of an AV (sub)unit by issuing a RESERVE command with a *ctype* field of STATUS, in the format shown in Figure 10-3.

	msb							lsb
opcode	RESERVE (01_{16})							
operand[0]	FF_{16}							
…								
operand[12]								

Figure 10-3 — RESERVE status command format

If a response frame is returned that indicates STABLE, *operand[0]* holds the current reservation priority and *operand[1]* through *operand[12]* hold the text string stored at the time the reservation was established. There is no way to determine the identity of the controller that holds the reservation.

Controllers that wish to be advised of a possible change of status of their own reservations, for example preemption by another controller by means of a higher priority reservation, should issue a RESERVE command in the format shown in Figure 10-3 but with a *ctype* value of NOTIFY. If a new reservation is established, the original reservation holder is notified by an AV/C response frame with CHANGED status and operand values that reflect the new reservation.

> **NOTE:** Any new reservation results in CHANGED status, even a reservation made by the same controller that already holds a reservation. A response frame is returned to any outstanding notify command in all of these cases.

10.8 PLUG INFO command

The PLUG INFO status command is used to inquire about the number of plugs on the AV unit or one of its subunits determined by the AV/C address contained in the AV/C frame. The format of the PLUG INFO status command is illustrated by Figure 10-4 below.

	msb							lsb
opcode	PLUG INFO (02_{16})							
operand[0]	00_{16}							
operand[1]	FF_{16}							
operand[3]								
operand[4]								

Figure 10-4 — PLUG INFO status command format

If the PLUG INFO status command was addressed to an AV subunit, the format of the response frame is shown in Figure 10-5 below.

	msb							lsb
opcode	PLUG INFO (02_{16})							
operand[0]	00_{16}							
operand[1]	destination_plugs							
operand[2]	source_plugs							
operand[3]	FF_{16}							
operand[4]	FF_{16}							

Figure 10-5 — PLUG INFO status response format from an AV subunit

For the AV subunit response frame, operand[1] and operand[2] shall indicate the number of destination and source plugs of that AV subunit, and operand[3] and operand[4] shall have the value FF_{16}.

If the PLUG INFO status command was addressed to an AV unit, the response frame returned is illustrated by Figure 10-6 below.

	msb							lsb
opcode	PLUG INFO (02_{16})							
operand[0]	00_{16}							
operand[1]	Serial_Bus_input_plugs							
operand[2]	Serial_Bus_output_plugs							
operand[3]	External_input_plugs							
operand[4]	External_output_plugs							

Figure 10-6 — PLUG INFO response format from an AV unit

If the PLUG INFO status command is addressed to the AV unit, operand[1] and operand[2] shall indicate the number of Serial Bus input and output plugs, respectively, while operand[3] and operand[4] shall indicate the number of external input and output plugs, respectively.

10.9 VENDOR-DEPENDENT commands

The VENDOR-DEPENDENT command permits module vendors to specify their own set of commands and responses for AV units or subunits determined by the AV/C address that is contained in the AV/C frame. The structure of the command is illustrated by Figure 10-7 below.

	msb							lsb
opcode	VENDOR-DEPENDENT (00_{16})							
operand[0]	(most significant byte)							
operand[1]	company_ID							
operand[2]	(least significant byte)							
operand[3]	vendor_dependent_data							
…								
operand[*n*]								

Figure 10-7 — VENDOR-DEPENDENT command format

The *company_ID* field shall contain the 24-bit unique ID obtained from the IEEE Registration Authority Committee (RAC). It is expected that the value of company_ID provided in the operands of vendor-dependent commands be the same as the vendor ID in the Node Unique ID leaf in configuration ROM of the AV unit to which the command is addressed. The most significant part of the company_ID is stored in *operand[0]* and the least significant part in *operand[2].*

The format and meaning of the *vendor_dependent_data* field are specified by the vendor identified by company_ID.

Although the behavior of vendor-dependent commands is beyond the scope of this specification, it is recommended that vendor-dependent be defined in the same five command types, CONTROL, SPECIFIC INQUIRY, STATUS, NOTIFY and GENERAL INQUIRY, specified by the *ctype* field described in 5.3.1.

A. AV/C commands in numerical order (normative)

The table below lists all the AV/C commands, in numerical order by *opcode*. Commands that pertain to subunits in addition to units are indicated by an X in the Subunit commands column. The legend for the subunit types follows the table.

		Unit command	Subunit commands	Support level (by *ctype*)		
Value	**Opcode**			**C**	**S**	**N**
00_{16}	VENDOR-DEPENDENT	X	X	V	V	V
01_{16}	RESERVE	X	X	O	O	R
02_{16}	PLUG INFO	X	X	–	O	–
08_{16}	OPEN DESCRIPTOR	X	X	O	O	O
09_{16}	READ DESCRIPTOR	X	X	O	–	–
$0A_{16}$	WRITE DESCRIPTOR	X	X	O	O	–
$0B_{16}$	SEARCH DESCRIPTOR	X	X	O	–	–
$0D_{16}$	OBJECT NUMBER SELECT	X	X	O	O	O
10_{16}	DIGITAL OUTPUT	X		O	O	–
11_{16}	DIGITAL INPUT	X		O	O	–
12_{16}	CHANNEL USAGE	X		–	R	R
18_{16}	OUTPUT PLUG SIGNAL FORMAT	X		O	R	O
19_{16}	INPUT PLUG SIGNAL FORMAT	X		O	R	O
20_{16}	CONNECT AV	X		O	O	O
21_{16}	DISCONNECT AV	X		O	–	–
22_{16}	CONNECTIONS	X		–	O	–
24_{16}	CONNECT	X		O	O	R
25_{16}	DISCONNECT	X		O	–	–
30_{16}	UNIT INFO	X		–	M	–
31_{16}	SUBUNIT INFO	X		–	M	–
$B2_{16}$	POWER	X	X	O	O	R

In the preceding table, an asterisk in the support level column indicates that the command operands or the type of subunit determine whether the command is mandatory (M), recommended (R), optional (O), or vendor-dependent (V).

B. Unresolved issues (informative)

This annex describes areas of the AV/C Digital Interface Command Set that are not yet fully resolved or subject to ambiguous interpretations by implementors.

It is recommended that this informative annex remain a part of the specification until the 1394 Trade Association is able to resolve the ambiguities.

B.1 Command execution model

There is no well articulated model for how commands are to be executed, to what degree command queuing is possible, whether or not response frames are required to be returned in the same order as the corresponding commands were initially issued, how (precisely) response frames are to be correlated with their command frames and so on.

The document editors have been unable to describe consistent behavior for AV devices in this area. The document editors are concerned that these ambiguities will permit varying implementations of both peripherals and host software such that interoperability is compromised.

The matter of response frame identification is troublesome because there is no way to uniquely guarantee that a particular response frame can be matched with its command frame.

B.2 Remote bus resets

In section 6, the behavior of an AV device when it detects a Serial Bus reset on the local bus is described. The AV device shall discard any in progress transactions without the return of a response frame. This is intended to prevent the return of a response frame to the incorrect controller, since the 6-bit physical ID of the controller may have changed as the result of a bus reset.

The same considerations apply if the controller is located on a remote bus that experiences a bus reset: the physical ID’s on that bus may change. This in turn implies that an AV device, if it is to be able to be controlled by a remote controller must have some way to detect the occurrence of a bus reset on the remote bus, in order to be able to discard an in progress transactions.

B.3 Notification support

It is desirable for a controller to have the opportunity to receive notification for any change of state in a target. This implies that all commands which cause state changes need to have notification specifications. There are some commands currently defined without notification support as an option, such as DIGITAL INPUT, DIGITAL OUTPUT, and a few others. Future work should focus on defining them.

B.4 Identifying the Specific Type of Subunit

In the situation where given subunit type has variations, it would be useful for a controller to be able to distinguish what variant it is talking to. For example, the VCR subunit type now has DVCR and D-VHS variants (or subtypes?). We need a method of classifying this new relationship model, and for providing this information to controllers.