PPDT_r05
August 10, 1999

P1394.*n*
Draft Standard for a
High Performance Serial Bus
Peer-to-Peer Data Transfer Protocol (PPDT)

Sponsor

1394 Printer Working Group

Not yet Approved by

(an accredited standards organization)

Abstract:

Keywords:

Contents

Page

1 Scope and purpose 1
1.1 Scope 1
1.2 Purpose 1

2 Normative references 3
2.1 Approved references 3
2.2 References under development 3

3 Definitions and notation 5
3.1 Definitions 5
3.1.1 Conformance 5
3.1.2 Glossary 5
3.1.3 Abbreviations 8
3.2 Notation 8
3.2.1 Numeric values 8
3.2.2 Bit, byte and quadlet ordering 8

4 Model (informative) 11
4.1 Protocol stack and service model 11
4.2 Independent data paths for each service 12
4.3 Connection management 13
4.4 Data transfer between initiator and target 14
4.5 Control requests and responses 14
4.6 Unsolicited status 14

5 Data structures 15
5.1 Transport flow ORBs 15
5.2 Status block 16
5.3 Control information 18
5.4 Queue information 21

6 Control operations 23
6.1 Login and queue zero 23
6.2 Autonomous response information 24
6.3 Service discovery 25
6.4 Connection management 25
6.4.1 Connection establishment 25
6.4.2 Queue shutdown 26
6.4.3 Resetting a queue 29
6.5 Queue status information 29

7 Transport flow operations 31
7.1 Data transfer to a target 32
7.2 Data transfer to an initiator 33
7.3 Completion status 33
7.4 Execution context for active ORBs 34
7.5 Error recovery 34
7.5.1 Resetting a connection 36
7.5.2 Resynchronizing a connection 36

8 Configuration ROM 39
8.1 Root directory 40
8.2 Instance directories 41

8.3 Feature directories 41
8.4 Keyword leaves 42
8.5 Unit directories 42

Tables

Table 1 – Parameter ID values 20
Table 2 – Connection type encoded by queue ID parameters 25
Table 3 – Root directory entries 40
Table 4 – Feature directory entries 41
Table 5 – Recommended keywords 42
Table 6 – Unit directory entries 42

Figures

Figure 1 – Bit ordering within a byte 8
Figure 2 – Byte ordering within a quadlet 9
Figure 3 – Quadlet ordering within an octlet 9
Figure 4 – Protocol stack (service at target) 11
Figure 5 – Protocol stack (service at initiator) 11
Figure 6 – Multiplexed queues in an SBP-2 task set 12
Figure 7 – Independent queues (logical model) 13
Figure 8 – Transport flow ORB 15
Figure 9 – Status block format 17
Figure 10 – Control information format 18
Figure 11 – Immediate parameter format 20
Figure 12 – Variable-length parameter format 21
Figure 13 – Queue information format 21
Figure 14 – Queue information byte format 22
Figure 15 – Transport flow (datagram model) 31
Figure 16 – Transport flow (stream model) 31
Figure 17 – Transport flow (spanned datagram model) 32
Figure 18 – Excess initiator data (datagram model) 32
Figure 19 – Excess target data (datagram model) 33
Figure 20 – Example configuration ROM hierarchy 39
Figure 21 – First five quadlets of configuration ROM 39
Figure E–1 – Example bus information block and root directory 53
Figure E–2 – Feature directory with service ID and device ID leaves 54
Figure E–3 – Unit directory for peer-to-peer data transfer (PPDT) protocol target 55
Figure E–4 – Instance directory and keyword leaf for a scanner 56
Figure E–5 – Instance directory and keyword leaf for a multiple protocol printer 57

Annexes

Annex A (normative) Minimum Serial Bus node capabilities 45

Annex B (normative) Compliance with ANSI NCITS 325-1998 47

Annex C (normative) Control request and response parameters 49

Annex D (normative) Control and status registers 51

Annex E (informative) Configuration ROM 53

1 Scope and purpose

1.1 Scope

This is a full-use standard whose scope is the definition of a peer-to-peer data transfer (PPDT) protocol between Serial Bus devices that implement ANSI NCITS 325-1998, Serial Bus Protocol 2. The facilities specified include, but are not limited to, the following:

- Device and service discovery. PPDT devices may use uniform discovery procedures to locate other PPDT devices on the same bus. These procedures are extensible to an interconnected net of buses, when specified by IEEE P1394.1, Draft Standard for Serial Bus to Serial Bus Bridges. Once other PPDT devices are identified, facilities are provided to permit client applications to discover services;
- Self-configurable (plug and play) binding of device drivers to PPDT devices in a dynamic environment where users are free to insert and remove devices at will; and
- Connection management. A PPDT device (either an SBP-2 initiator or target) may establish and manage uni- or bi-directional connections for data transfer with other PPDT devices. The connections may be blocking or nonblocking, dependent upon application requirements, and operate independently of each other.

Although the original impetus for the development of this standard came from participants knowledgeable about printers and printing, the work evolved and became relevant to any application that utilizes a client/server model and requires efficient, peer-to-peer transfer of data between devices.

1.2 Purpose

Experience with SBP-2 has demonstrated its high efficiency for the confirmed transfer of large quantities of data between two devices. For historical reasons, SBP-2 is optimally tailored to an environment where one device is the initiator (client) and the other the target (server); this is not necessarily the most natural approach when client applications and their associated servers may be located within initiator, target or both.

The standard creates a new layer of protocol services based upon SBP-2 but that provides building blocks more suited to a peer-to-peer environment. Because SBP-2 is already widely implemented in operating systems, this standard leverages that effort in order to enhance the value of Serial Bus to devices in a wider range of operational circumstances. These include printers, facsimile devices, scanners (or multifunction devices that present some combination of these functions) when a computer is present—but is also intended to address their peer-to-peer needs to communicate with each other in the absence of a computer.

2 Normative references

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

Copies of the following documents can be obtained from ANSI:

Approved ANSI standards;

Approved and draft regional and international standards (ISO, IEC, CEN/CENELEC and ITUT); and

Approved and draft foreign standards (including BIS, JIS and DIN).

For further information, contact the ANSI Customer Service Department by telephone at (212) 642-4900, by FAX at (212) 302-1286 or *via* the world wide web at http://www.ansi.org.

Additional contact information for document availability is provided below as needed.

2.1 Approved references

The following approved ANSI, international and regional standards (ISO, IEC, CEN/CENELEC and ITUT) may be obtained from the international and regional organizations that control them.

ANSI NCITS 325-1998, American National Standard for Information Systems—Serial Bus Protocol 2 (SBP-2)

IEEE Std 1284-1994, Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers

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

ISO/IEC 9899:1990, Programming Languages—C

2.2 References under development

At the time of publication, the following referenced standards were still under development.

IEEE P1212r, Draft Standard for a Control and Status Register (CSR) Architecture for Microcomputer Buses (Revision)

IEEE P1394a, Draft Standard for a High Performance Serial Bus (Supplement)

3 Definitions and notation

3.1 Definitions

3.1.1 Conformance

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

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

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

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

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

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

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

3.1.2 Glossary

The following terms are used in this standard:

3.1.2.1 active ORB: .

3.1.2.2 byte: Eight bits of data.

3.1.2.3 connection: A queue or a pair of queue(s) that affords access to a service. A connection may be unidirectional or bi-directional; in the latter case, a connection may be blocking or nonblocking. Two queues are necessary to implement a bi-directional, nonblocking connection.

3.1.2.4 control information: Information exchanged between initiator and target whose format is defined by this standard. The format of control information is independent of the format of application data exchanged by client applications and services—but both control information and application data are transferred by the same ORB methods.

3.1.2.5 control ORB: A transport flow whose *queue* field is zero; it is used to transfer request or response control information between initiator and target.

3.1.2.6 final ORB: A transport flow ORB whose *final* and *notify* bits are both one. An initiator uses a final ORB to indicate to a target that no subsequent ORBs with the same *queue* value will be signaled unless the queue number is reissued by the target in a future CONNECT control request or response.

3.1.2.7 function: A capability of the device expressed as a unit architecture (unit directory) with a single logical unit (LU zero).

3.1.2.8 initiator: A node that originates SBP-2 management requests, control operation and transport flow ORBs and signals them to a target for processing.

3.1.2.9 logical unit: The part of the unit architecture that provides access to one or more services. Devices compliant with this standard implement one logical unit with a LUN of zero.

3.1.2.10 login: The process by which an initiator obtains access to a target fetch agent. The target fetch agent and its CSRs provide a mechanism for an initiator to signal ORBs to the target.

3.1.2.11 management service: A mandatory service provided for each function; it executes control requests to establish or terminate connections to the other services of the function. The connection to this service is implicitly established as the result of an SBP-2 login.

3.1.2.12 node: An addressable device attached to Serial Bus.

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

3.1.2.14 operation request block: A data structure fetched from system memory by a target in order to execute the request encapsulated within it.

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

3.1.2.16 queue: An ordered set of ORBs within a task set that does not block with respect to other queues that are part of the same task set.

3.1.2.17 receive: When any form of this verb is used in the context of Serial Bus primary packets, it indicates that the packet is made available to the transaction or application layers, *i.e.*, layers above the link layer. Neither a packet repeated by the PHY nor a packet examined by the link is "received" by the node unless the preceding is also true.

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

3.1.2.19 request subaction: A packet transmitted by a node (the requester) that communicates a transaction code and optional data to another node (the responder) or nodes.

3.1.2.20 response subaction: A packet transmitted by a node (the responder) that communicates a response code and optional data to another node (the requester). A response subaction may consist of either an acknowledge packet or a response packet.

3.1.2.21 service: A protocol used to control an independently operable component of a function.

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

3.1.2.23 status block: A data structure which may be written to system memory by a target when an operation request block has been completed.

3.1.2.24 store: When any form of this verb is used in the context of data transferred by the target to the system memory of either an initiator or other device, it indicates both the use of Serial Bus write request subaction(s), quadlet or block, to place the data in system memory and the corresponding response subaction(s) that complete the write(s).

3.1.2.25 system memory: The portions of any node's memory that are directly addressable by a Serial Bus address and which accepts, at a minimum, quadlet read and write access. Computers are the most common example of nodes that might make system memory addressable from Serial Bus, but any node, including those usually thought of as peripheral devices, may have system memory.

3.1.2.26 target: A node that fetches SBP-2 management requests, control operation and transport flow ORBs from an initiator. In the case of control operation or transport flow requests, the ORBs are directed to the target's logical unit zero to be executed. A CSR Architecture unit is synonymous with a target.

3.1.2.27 task: A task is an organizing concept that represents the work to be done by a target to carry out a command encapsulated by an ORB. In order to perform a task, a target maintains context information for the task, which includes (but is not limited to) the command, parameters such as data transfer addresses and lengths, completion status and ordering relationships to other tasks. A task has a lifetime, which commences when the task is entered into the target's task set, proceeds through a period of execution by the target and finishes either when completion status is stored at the initiator or when completion may be deduced from other information. While a task is active, it makes use of both target resources and initiator resources.

3.1.2.28 task set: A group of tasks available for execution by a logical unit of a target. ANSI NCITS 325-1998 specifies some dependencies between individual tasks within the task set and this standard mandates others.

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

3.1.2.30 unit: A component of a Serial Bus node that provides processing, memory, I/O or some other functionality. Once the node is initialized, the unit provides a CSR interface that is typically accessed by device driver software at an initiator. A node may have multiple units, which normally operate independently of each other. Within this standard, a unit is equivalent to a target.

3.1.2.31 unit architecture: The specification of the interface to and the services provided by a unit implemented within a Serial Bus node. This standard is a unit architecture for image devices (*e.g.*, printers, scanners or multifunction peripherals) intended to be used with the unit architecture for SBP-2 targets.

3.1.2.32 unit attention: A state that a logical unit maintains while it has unsolicited status information to report to one or more logged-in initiators. A unit attention condition shall be created as described elsewhere in this standard or in the applicable command set- and device-dependent documents. A unit attention condition shall persist for a logged-in initiator until a) unsolicited status that reports the unit attention condition is successfully stored at the initiator or b) the initiator's login becomes invalid or is released. Logical units may queue unit attention conditions; after the first unit attention condition is cleared, another unit attention condition may exist.

3.1.2.33 unsolicited status block: A status block whose *src* field is two; the meaning of the *ORB_offset_hi* and *ORB_offset_lo* fields is unspecified and the status block does not pertain to any particular ORB.

3.1.2.34 working set: The part of a task set that has been fetched from the initiator by the target and is available to the target in its local storage.

3.1.3 Abbreviations

The following are abbreviations that are used in this standard:

CSR	Control and status register
CRC	Cyclical redundancy checksum
EUI-64	Extended Unique Identifier, 64-bits
LUN	Logical unit number
ORB	Operation request block
SBP-2	ANSI NCITS 325-1998

3.2 Notation

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

3.2.1 Numeric values

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

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

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

3.2.2 Bit, byte and quadlet ordering

Devices compliant with this standard use the facilities of Serial Bus, IEEE Std 1394-1995; therefore this standard uses the ordering conventions of Serial Bus in the representation of data structures. In order to promote interoperability with memory buses that may have different ordering conventions, this standard defines the order and significance of bits within bytes, bytes within quadlets and quadlets within octlets in terms of their relative position and not their physically addressed position.

Within a byte, the most significant bit, *msb*, is that which is transmitted first and the least significant bit, *lsb*, is that which is transmitted last on Serial Bus, as illustrated below. The significance of the interior bits uniformly decreases in progression from *msb* to *lsb*.

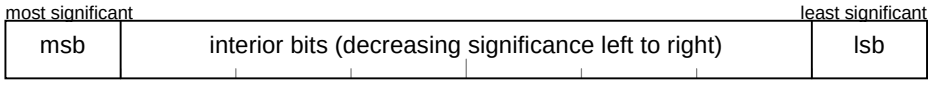

Figure 1 – Bit ordering within a byte

Within a quadlet, the most significant byte is that which is transmitted first and the least significant byte is that which is transmitted last on Serial Bus, as shown below.

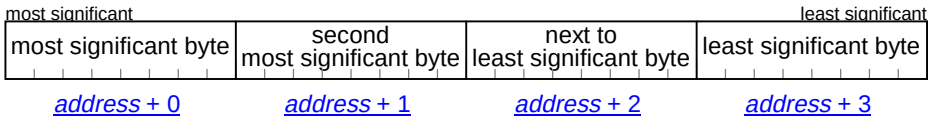

Figure 2 – Byte ordering within a quadlet

Within an octlet, which is frequently used to contain 64-bit Serial Bus addresses, the most significant quadlet is that which is transmitted first and the least significant quadlet is that which is transmitted last on Serial Bus, as the figure below indicates.

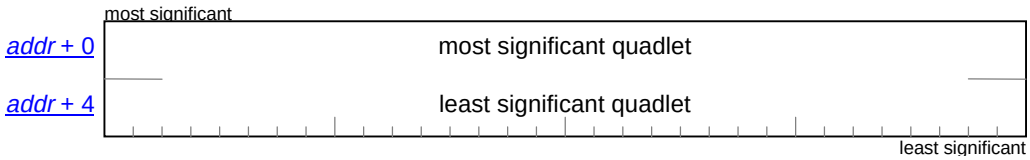

Figure 3 – Quadlet ordering within an octlet

When block transfers take place that are not quadlet aligned or not an integral number of quadlets. No assumptions can be made about the ordering (significance within a quadlet) of bytes at the unaligned beginning or fractional quadlet end of such a block transfer, unless an application has knowledge (outside of the scope of this standard) of the ordering conventions of the other bus.

4 Model (informative)

This section is informative and describes ~~imaging~~ devices that conform to this document and its normative references. It is intended to enhance the usefulness of the other, normative parts of the document. In addition to the information in this clause, users of this document should also be familiar with the CSR architecture, Serial Bus standards and the ANSI NCITS 325-1998.

Examples of ~~imaging~~ devices that come within the scope of this profile include (but are not limited to) copiers, printers, facsimile machines, scanners and multi-function peripherals that combine two or more of these capabilities. These devices are characterized by high-volume transfers of application data; modest amounts of control information may be communicated in parallel with the application data transfers. These devices are used with diverse operating systems and application protocols; consequently any standard for their use with Serial Bus needs to hide many of the transport protocol details from the user applications. For example, a print driver that supports Postscript data formats should not be concerned with how data and control information are transported between it and the printer. This document resolves those concerns.

4.1 Protocol stack and service model

The relationship between the initiator and target may be modeled as a software stack present in both devices, as shown by Figure 4 and Figure 5 below. The physical interconnection, *via* Serial Bus, exists at the lowest protocol level. Logical connections (shown by dashed lines) exist at the other protocol levels: an SBP-2 LOGIN between the initiator and target multiplexes *queues* (defined by this document) that in turn support end-to-end *connections* (also defined by this document) between client applications and services. This document defines the data structures and methods necessary to implement the shaded levels in the protocol stacks, a peer-to-peer data transport (PPDT) based upon SBP-2. Note that client application(s) may reside at either the initiator or the target (they are commonly found at the initiator) and the service(s) at the corresponding SBP-2 functional role, target or initiator.

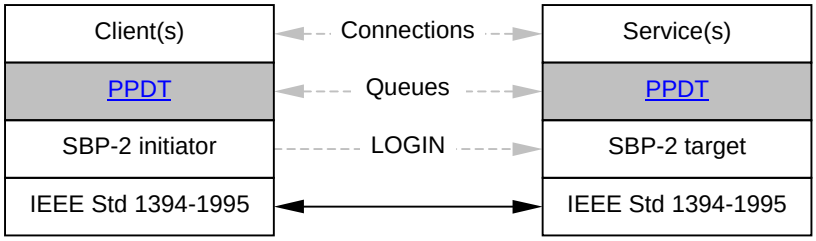

Figure 4 – Protocol stack (service at target)

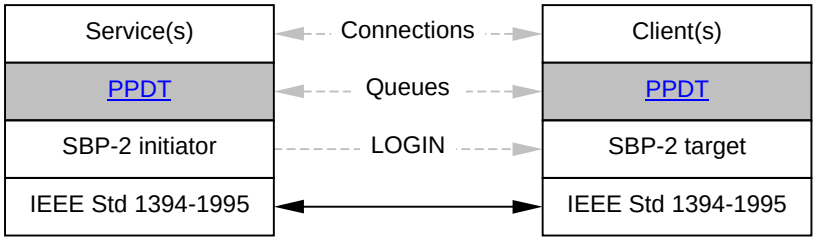

Figure 5 – Protocol stack (service at initiator)

In order for the application(s) and service(s) to communicate in a peer-to-peer, transport-independent manner, this document defines how SBP-2 may be used to implement uni- and bi-directional transport

flows for both control information and application data. Key concepts introduced below are used to explain the details of the transport flow model:

function: A capability of the device expressed as a unit architecture (unit directory) that contains a single logical unit (LU zero);

service: A protocol used to control an independently operable component of a function;

management service: A mandatory service provided for each function; it executes control requests to establish or terminate connections to the other services of the function. The connection to this function is implicitly established as the result of an SBP-2 login;

queue: An ordered set of ORBs within a task set that does not block with respect to other queues that are part of the same task set; and

connection: A queue or a pair of queues that affords access to a service. A connection may be unidirectional or bi-directional; in the latter case, a connection may be blocking or nonblocking. Two queues are necessary to implement a bi-directional, nonblocking connection.

4.2 Independent data paths for each service

ANSI NCITS 325-1998 describes all the work to be performed by a particular logical unit as a *task set*, a collection of ORBs linked together as shown by Figure 6.

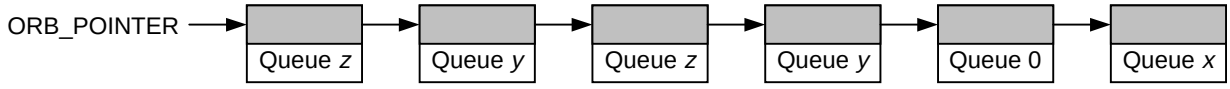

Figure 6 – Multiplexed queues in an SBP-2 task set

Because a single device function (logical unit) may be implemented as one or more ***services*** (protocols used to control independently operable components of a function), each of which may require an independent uni- or bi-directional transport flow, this document augments SBP-2 to permit multiplexed *queues* within a single task set, as illustrated by Figure 7. A ***queue*** is an ordered set of ORBs within a task set that does not block with respect to other queues that are part of the same task set; each ORB in the task set is labeled to identify the logical queue to which it belongs. Although the target may in general reorder the execution of ORBs within the task set, all of the ORBs within a particular queue are executed in order. Within this framework, both the initiator and the target manage the single task set illustrated above as the collection of logically independent queues illustrated below. The dashed lines connecting ORBs represent the logical ordering of ORBs within each queue, not the actual pointers that link ORBs in the task set.

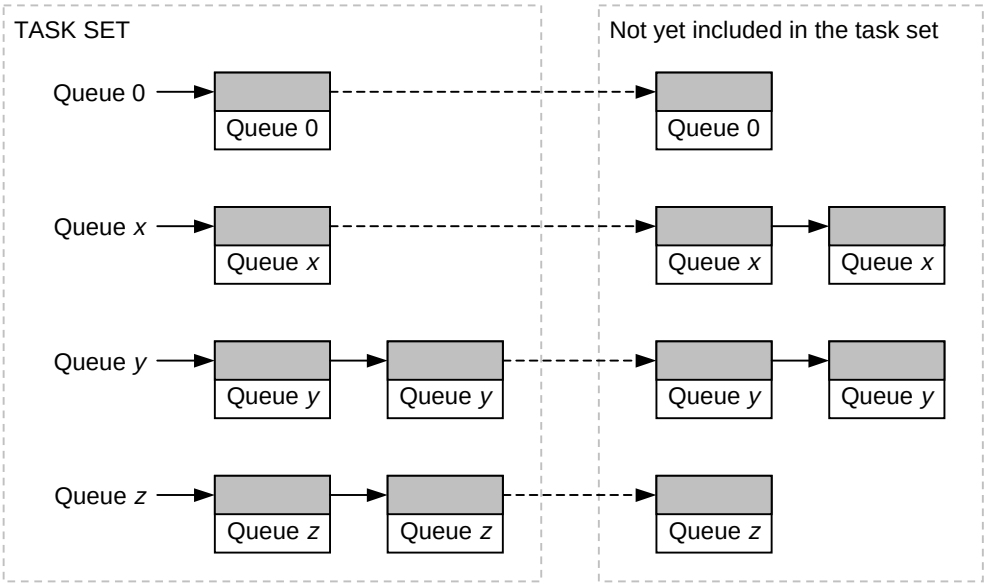

Figure 7 – Independent queues (logical model)

In theory the size of an SBP-2 task set is bounded only by the amount of memory available to the initiator to store ORBs; in practice targets have sufficient memory to fetch only a subset of the task set, the *working set*. Nonblocking behavior between the separate queues is achieved by restricting the size of the task set to that of the target's working set. If the initiator never places more ORBs in the task set than the target can accommodate in its working set, all outstanding ORBs may be fetched by the target and made available for execution. The initiator restricts the number of outstanding ORBs on a queue by queue basis so that a ***task slot*** in the working set is always available for each queue. Since the client application or service may initiate more data transfer requests than can be simultaneously active in the task set, the initiator marshals ORBs by queue number outside of the task set and enters them into the task set as task slots become available.

Because queues do not block with respect to each other, nonblocking bi-directional data transfer between initiator and target may be accomplished through the use of two queues, one for each direction.

4.3 Connection management

The multiplexed queue management scheme just described requires the allocation of target resources (queue numbers and task slots) before it may be used. Collectively these resources constitute a *connection* between a client and a service. This document defines methods by which connection(s) are established and subsequently terminated and their resources freed.

Connections may be established by either an initiator or a target. Because of asymmetries in SBP-2, the connection parameters differ dependent upon the source of the connection request—but at the transport-independent level perceived by clients and services the connection mechanisms are peer-to-peer and symmetric. When a client wishes to establish a connection with a particular service in the other device, it provides a *service ID*, a unique string that specifies the desired service. Service IDs are maintained in a separate registry and are assumed by this document to be well-known identifiers. If the specified service exists in the other device (along with sufficient resources for the connection), the connection is created and subsequently identified by the queue number(s) assigned to the connection.

Connections may be one of three different types:

- Unidirectional; the application data flow is one direction, either from the initiator to the target or *vice versa*;
- Bi-directional (nonblocking); the application data flows in both directions with one queue used for each of the directions; or
- Bi-directional (blocking); the data flows in both directions *via* a single queue which has the potential to block. Nonblocking behavior is not guaranteed by the transport but must be a property of the application itself. The queue used for management services is an example of a bi-directional, blocking queue, but because it is restricted to single-threaded, serialized use it cannot block.

Once a connection is established it persists across bus reset(s) until explicitly terminated or abandoned as a consequence of a logout.

Just as either initiator or target may establish a connection, either may terminate the connection regardless of which one created the connection. A disconnect may be synchronized with the transport flow in order to gracefully end the connection or it may preempt the transport flow if necessary. Once the disconnect is complete, the target resources (queue numbers and task slots) are available for reuse.

EDITOR's NOTE – All of the remaining (informative) clauses in this section are awaiting stabilization of the (normative) technical details in other sections of this document.

4.4 Data transfer between initiator and target

4.5 Control requests and responses

4.6 Unsolicited status

5 Data structures

This document defines the format of those parts of the SBP-2 ORB and status block reserved by ANSI NCITS 325-1998 for specification by command set standards. It also defines a format for control information transferred between initiator and target. All data structures defined in the following clauses shall be aligned on quadlet boundaries.

5.1 Transport flow ORBs

ANSI NCITS 325-1998 defines command block ORBs for SBP-2 devices; these have a common 20-byte header and leave the definition of the subsequent quadlets to individual command set standards. ~~Image~~ Devices compliant with this standard shall use 32-byte command block ORBs (renamed transport flow ORBs to emphasize their function) whose format is illustrated by Figure 8. Transport flow ORBs are used to regulate the transfer of application data or control information between initiator and target.

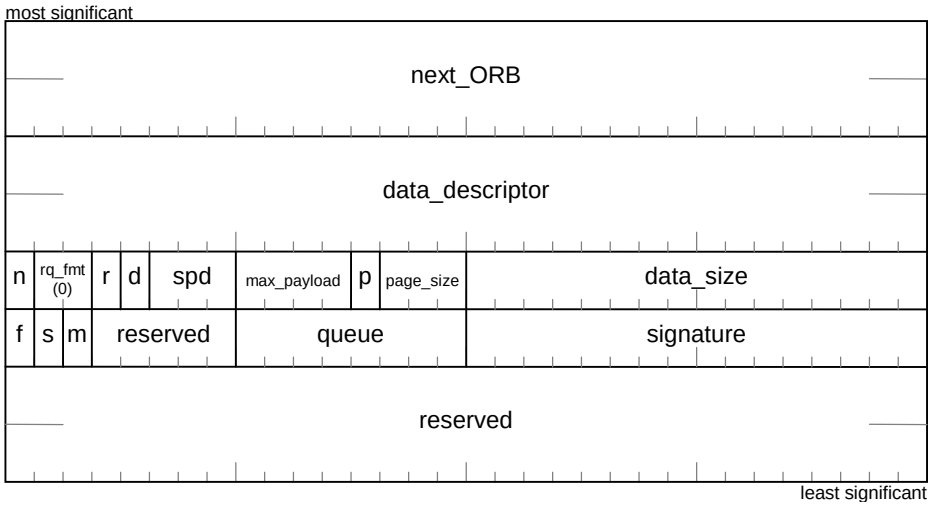

Figure 8 – Transport flow ORB

The usage of the *next_ORB*, *data_descriptor*, *rq_fmt*, *spd*, *max_payload*, *page_size* and *data_size* fields and the *notify* and *page_table_present* bits (abbreviated as *n* and *p*, respectively, in the figure above) is defined by ANSI NCITS 325-1998. The *rq_fmt* field shall be zero.

NOTE – For most SBP-2 implementations the *notify* bit should always be one so that the SBP-2 initiator software may accurately determine completion status for each ORB; this is a consequence of the unordered execution model. Other implementations that do not require completion status notification for each ORB may be possible if information is shared between the SBP-2 initiator and its client application(s) but the implementation details are beyond the scope of this document.

The *direction* bit (abbreviated as *d* in the figure above) shall specify the direction of data transfer for the buffer described by *data_descriptor*. If the *direction* bit is zero, the target shall use Serial Bus read transactions to fetch data from the buffer (the flow direction is from the initiator to the target). Otherwise, when the *direction* bit is one, the target shall use Serial Bus write transactions to store data in the buffer (the flow direction is from the target to the initiator).

NOTE – The direction of data transfer is determined solely by the *direction* bit without reference to the *queue* number. Unspecified behavior may occur if an ORB's *direction* bit does not match the expected data transfer direction for the queue.

The *final* bit (abbreviated as *f* in the figure above) shall be set to one to indicate that the initiator shall not signal any subsequent ORBs with the same *queue* value as this ORB until the target allocates the queue number in a future CONNECT request or response. Otherwise the value of *final* bit shall be zero and the initiator may continue to signal ORBs for the queue. When the *final* bit is one the *notify* bit shall also be one.

The *special* bit (abbreviated as *s* in the figure above) provides additional information pertinent to application data transferred from the initiator to the target. The meaning of the *special* bit is unspecified when either of the *data_size* or *queue* fields are zero or the *direction* bit is one. Otherwise the meaning and usage of the *special* bit are application-dependent and shall apply to all of the application data contained within the buffer described by the ORB.

NOTE – Stream socket abstractions include the notion of *out of band* data, as some transport protocols allow portions of incoming data to be marked as "special" in some way. These special data blocks may be delivered to the user out of the normal sequence—for example, expedited data in X.25 and other OSI protocols or the use of urgent data in TCP by BSD Unix. The *special* bit enables such usage to be mapped to a transport protocol based on SBP-2.

The *end_of_message* bit (abbreviated as *m* in the figure above) shall indicate whether or not a boundary exists in the application data or control information transferred from the initiator to the target. The meaning of the *end_of_message* bit is unspecified when the *direction* bit is one. Otherwise, when *end_of_message* is one, a boundary exists after the last byte of application data or control information described by the ORB. In the case of application data, the nature of the boundary and its interpretation shall be specified by the service definition. When the *queue* field is zero, the *end_of_message* bit shall also be one; all control information for a single request or response shall be contained within one buffer.

NOTE – When *end_of_message* is one and *data_size* is zero, a boundary exists at the end of application data or control information previously transferred to the target. The target flushes this data to the receiving application client and indicates the *end_of_message* condition.

The *queue* field shall specify either queue zero or a queue number assigned the target in either a CONNECT request or response (see 6.4.1). When the *queue* field is zero, the *final* bit shall be zero and the *notify* bit shall be one.

The *signature* field shall contain an identifying number assigned by the initiator and shall be unique within the context of a queue. Individual data buffers are uniquely identified by the combination of *queue* and *signature*. For a particular queue, an initiator shall not reuse a *signature* value until either the connection has been reset (see 0) or a status block has been received for a subsequent ORB in same the queue. This field is used to facilitate the resumption of data transfer after a bus reset or other transient interruption while minimizing retransmission of data securely stored prior to the interruption (see 7.5).

5.2 Status block

As described by ANSI NCITS 325-1998, a target may store status at an initiator *status_FIFO* address when a request completes (successfully or in error) or because of an unsolicited event (device status change). The *status_FIFO* address is obtained either explicitly from the ORB to which the status pertains or implicitly from the fetch agent context. Whenever the target has status to report and is enabled to do so, it shall store the status block illustrated by Figure 9.

Without regard to the value of the *notify* bit in the ORB to which status pertains, the target shall store status if any of the *dead*, *attention*, *target_data_pending*, *special* and *end_of_message* bits or either of the *status* and *residual* fields are nonzero.

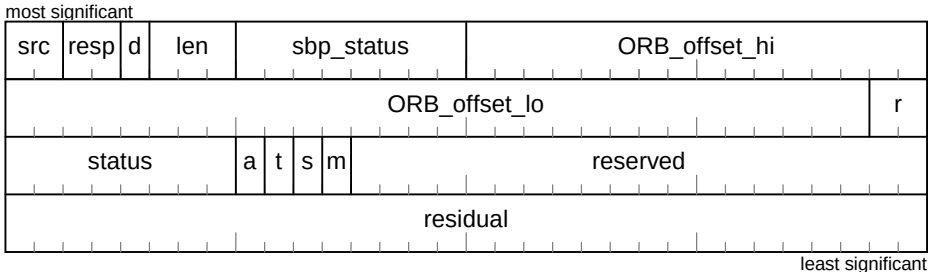

Figure 9 – Status block format

The definition and usage of the *src*, *resp*, *len*, *sbp_status*, *ORB_offset_hi* and *ORB_offset_lo* fields, as well as the *dead* bit (abbreviated as *d* in the figure above), are specified by ANSI NCITS 325-1998.

The *len* field shall have a value of three to indicate that the length of the status block is four quadlets.

The *status* field shall specify the completion status of the transport flow requested by the ORB, as encoded by the table below.

status	**Description**
0	The application data or control information has been successfully transferred; consult the *residual* field for details of the actual transfer length.
1	Invalid queue; the queue identified in the ORB has not been allocated to an active connection.
2	Target reset by another initiator; all tasks aborted.

The *attention* bit (abbreviated as *a* in the figure above) indicates the availability of target control information. When the *attention* bit is one, the initiator should post an ORB for queue zero to retrieve the control information. Once set to one by the target, this bit shall remain set until the initiator successfully retrieves the control information.

The *target_data_pending* bit (abbreviated as *t* in the figure above) indicates the availability of target application data for the queue specified by the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*. When the *target_data_pending* bit is one, the initiator should post an ORB for the specified queue to retrieve the application data. The target shall zero this bit when there is no pending application data awaiting transfer to the initiator. The meaning of *target_data_pending* is unspecified for an unsolicited status block.

The *special* bit (abbreviated as *s* in the figure above) provides additional information pertinent to application data transferred from the target to the initiator. The meaning of the *special* bit is unspecified when the value of the *src* field is two, when (in the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*) any of the *data_size* or *queue* fields or the *direction* bit are zero or if no data has been transferred. The meaning and usage of the *special* bit are application-dependent and shall apply to all of the application data contained within the buffer described by the ORB.

The *end_of_message* bit (abbreviated as *m* in the figure above) shall indicate whether or not a boundary exists in the application data or control information transferred from the target to the initiator. The meaning of the *end_of_message* bit is unspecified when the value of the *src* field is two or when the *direction* bit (in the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*) is zero. Otherwise, when *end_of_message* is one, a boundary exists after the last byte of application data or control information described by the ORB. In the case of application data, the nature of the boundary and its interpretation shall be specified by the service definition. When the *control* bit (in the ORB identified by *ORB_offset_hi* and *ORB_offset_lo*) is

one, the *end_of_message* bit in the associated status block shall also be one; all control information for a single request or response shall be contained within one buffer.

The *residual* field shall specify the difference between the requested and actual data transfer lengths, in bytes.

When *residual* is negative, no data has been transferred because of a mismatch between the size of the buffer and the data transfer length acceptable to the target: either the target's buffer space is too small to accept the data described by the ORB to which the status pertains or else the buffer described by the pertinent ORB is too small to accept the data available from the target. In these cases, the meaning of *residual* depends upon the value of the *direction* bit of the ORB to which the status pertains. When *direction* is zero (the flow direction is from the initiator to the target), the target shall calculate *residual* by subtracting the size of the buffer provided by the initiator from the maximum acceptable data transfer length. Otherwise, when direction is one (the flow direction is from the target to the initiator), the target shall calculate *residual* by subtracting the minimum acceptable data transfer length from the size of the buffer provided by the initiator. Negative values shall be encoded in two's complement notation.

Otherwise, when *residual* is greater than or equal to zero, there is no mismatch between the size of the buffer and the data transfer length to prevent data transfer. The target shall calculate *residual* by subtracting the actual data transfer length from the size of the buffer provided by the initiator. A *residual* value greater than zero is not necessarily indicative of an error.

5.3 Control information

Control information, both requests and their corresponding responses, may be exchanged between initiator and target *via* transport flow ORBs whose *queue* field is zero (control ORBs). This indicates that the data in the buffer (or the data to be stored in the buffer) associated with the ORB is control information rather than application data. Only one control request or response shall be transferred by an ORB; when the *direction* bit is zero the *end_of_message* bit in the ORB shall be one, otherwise the *end_of_message* bit in the status block shall be one. If the initiator has provided more control information than the target can accept or if the buffer is too small to receive all the target's control information, no transfer shall take place and the *residual* field shall indicate the appropriate transfer size. The format of the control information in the buffer is illustrated by Figure 10.

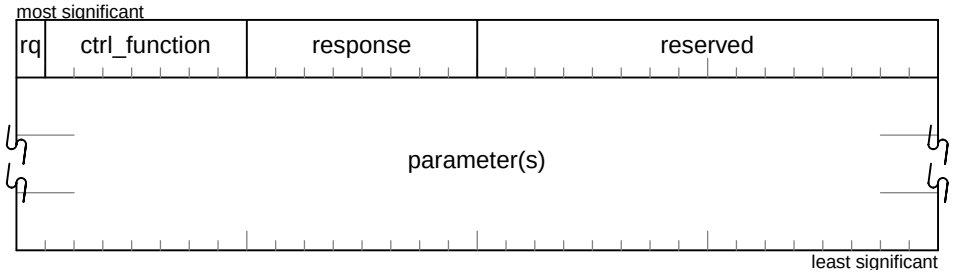

Figure 10 – Control information format

The *rq* bit shall specify whether the control function is a request or a response. A value of one indicates a request.

The *ctrl_function* field shall specify the control function, as defined by the table below.

ctrl_function	Name	Comment
0		Reserved for future standardization
1	CONNECT	Establish a connection with a particular service
2	SHUTDOWN QUEUE	
3	RESET CONNECTION	Resynchronize a connection between a client application and the service
4	RELEASE QUEUE	
5	SERVICE DIRECTORY	Query all the services implemented (identified by a list of service Ids)
$6 – 7F_{16}$		Reserved for future standardization

The *response* field is valid only when the *rq* bit is zero. In this case, it encodes a response indication for the corresponding control function, as defined by the table below.

response	Definition
0	Request completed OK; response parameters are meaningful.
1	Unknown control function.
2	Insufficient resources are available to complete the request; the same request may succeed if resubmitted later.
3	The service identified by the SERVICE_ID parameter does not exist.
4	Mismatch between actual and expected queue number parameter(s).
5	The connection request is refused.
6	The connection identified by the queue number parameter(s) does not exist.
FF_{16}	Unspecified error.

The remainder of the control information, up to the maximum size specified by *data_size* in the ORB that references the control information buffer, shall consist of zero or more parameters identified by a parameter ID (see Table 1). Relative to the start of the buffer, each parameter shall be quadlet-aligned and occupy an integral number of quadlets. The first parameter shall start in the second quadlet of control information and subsequent parameters, if any, shall immediately follow the preceding parameter. The order in which parameters appear is unimportant. Any quadlets that follow the last parameter, up to the end of the control information, shall be cleared to zero.

Table 1 – Parameter ID values

Parameter ID	Parameter Name	Value restrictions	Description
0		0	Indicates end of parameter list in control information (optional).
1	TASK_SLOTS	Minimum 1	For a particular connection, the maximum number of ORBs permitted in the task set. The initiator shall observe the limit established by the target and may optionally provide this parameter to indicate a self-imposed limit. Task slots are allocated *per* connection and may be used for any of the connection's queues.
82_{16}	SERVICE_ID	40 bytes maximum	An ASCII text string (without leading or trailing blank characters) that uniquely identifies a service.
3	I2T_QUEUE	Nonzero; FF_{16} maximum	The queue number assigned to the connection for the transport of application data from the initiator to the target
4	T2I_QUEUE		The queue number assigned to the connection for the transport of application data from the target to the initiator
85_{16}	QUEUE_INFO		A bit map that reports the state of *target_data_pending* for all queues implemented by the target.
6	MODE		Specifies the desired mode, datagram or stream, at the time a connection to a service is established.

The parameter ID shall specify the parameter format, either immediate or variable-length. The most significant bit of the parameter ID determines the format; parameters whose ID values are in the range zero to $7F_{16}$, inclusive, shall conform to the format specified by Figure 11 while those in the range 80_{16} – FF_{16}, inclusive, shall conform to the format specified by Figure 12. All parameter ID values not specified are reserved for future standardization.

The format of immediate parameters is shown below.

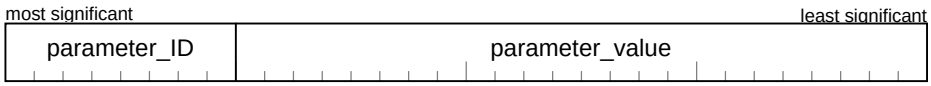

Figure 11 – Immediate parameter format

The *parameter_ID* field shall specify the parameter, as encoded by Table 1.

The *parameter_value* field shall specify the immediate value of the parameter. Unless otherwise specified for a particular value of *parameter_ID*, the *value* field shall contain an unsigned 24-bit number.

The format of variable-length parameters (which are usually ASCII text strings) is shown below.

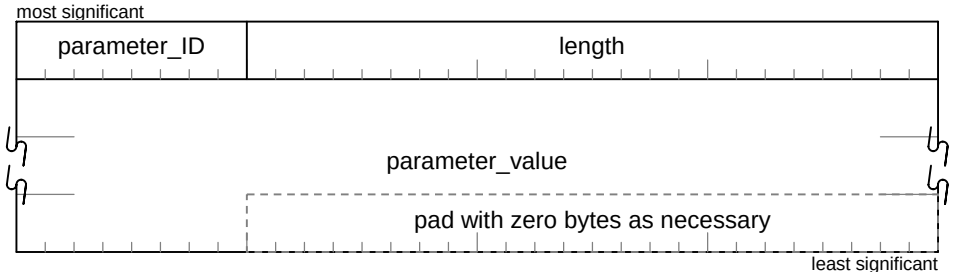

Figure 12 – Variable-length parameter format

The *parameter_ID* field shall specify the parameter, as encoded by Table 1.

The *length* field shall specify the length of the *parameter_value* field, in bytes. Pad bytes, if present, are excluded from the value of *length*.

The *parameter_value* field shall contain the value of the parameter and shall commence with the most significant byte of the parameter value. If the length of the parameter is not a multiple of four, the parameter value shall be padded with trailing bytes of zero.

5.4 Queue information

Queue information is an array; each entry reports status information for the queues implemented by the target. The length of the array is implementation-dependent (determined by the largest queue number supported by the target) and shall conform to the format illustrated by Figure 13.

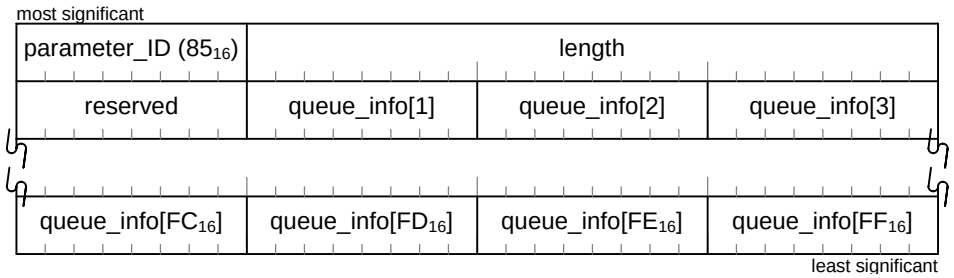

Figure 13 – Queue information format

The value of *parameter_ID* shall be 85_{16}.

The value of *length* shall be greater than the largest queue number implemented by the target and less than or equal to 256.

Each *queue_info* entry provides information for an individual queue; this array of bytes is directly indexed by the queue number. The first entry in the array is reserved.[1] For all other queues, the format of the *queue_info* byte is specified by Figure 14.

[1] Queue zero is always active; the availability of control information is indicated by the status block *attention* bit.

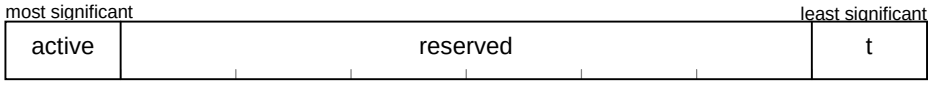

Figure 14 – Queue information byte format

The *active* bit shall be zero if the queue is not implemented by the target or not assigned to an established connection. A value of one indicates that the queue is in use by a connection.

The *target_data_pending* bit (abbreviated as *t* in the figure above) indicates the availability of target application data for the queue. When the *target_data_pending* bit is one, the initiator should post a transport flow ORB for the specified queue to retrieve the application data. The target shall zero this bit when there is no pending application data awaiting transfer to the initiator.

6 Control operations

Before application client(s) and service(s) may exchange data in uni- or bi-directional transport flows (explained in detail in section 7), control operations are necessary to set up the communication paths. This section specifies the methods used by both initiator and target to establish and manage connections for these transport flows.

6.1 Login and queue zero

Access to a target compliant with this standard commences with an SBP-2 login request by the initiator. Upon successful completion of the login request, the target has reserved resources for the use of the initiator:

- SBP-2 registers assigned by the target at the time of the login (the AGENT_STATE, AGENT_RESET, ORB_POINTER, DOORBELL and UNSOLICITED_STATUS_ENABLE registers);
- queue zero, the control operations queue; and
- two task slots for use by queue zero ORBs.

Once queue zero exists, either initiator or target may use it in a peer-to-peer fashion to communicate control information, requests or responses, to the other. At no time shall the task set contain more than two control ORBs.

The completion of a request requires two ORBs, one, which describes the control information buffer that contains the request and a complementary ORB which describes the control information buffer for the response. Although queue zero provides full peer-to-peer functionality between initiator and target, the details of its use are asymmetric and vary according to whether the initiator or the target is the requester.

When an initiator issues a request to a target, they shall perform the following operations:

a) The initiator shall store the request and its associated parameters (if any) in a buffer in system memory and signal to the target fetch agent an ORB, whose *queue* field and *direction* bit are zero and *end_of_message* bit is one, that describes the control information buffer;

b) The target shall fetch the ORB and read the control information buffer. The status block stored by the target to complete the ORB may have its *attention* bit set to one to indicate that the target intends to transfer control information (the response) to the initiator;

c) At any time the initiator receives a status block whose *attention* bit is one and there is no ORB in the task set whose *queue* field is zero and *direction* bit is one, the initiator shall create such an ORB and place it in the task set; and

d) Once the target has executed the indicated request and there is an ORB in the working set whose *queue* field is zero and *direction* bit is one, the target shall store the response data in the buffer described by the ORB and then store completion status for the ORB. So long as the target has pending control information to transfer to the initiator, it shall continue to set the *attention* bit to one in any status block (including unsolicited status) stored into the initiator *status_FIFO*.

NOTE – In order to reduce ORB fetch latency, the initiator may place two control information ORBs in the task set at the same time, the first for the request (with a *direction* bit of zero) and the second for the response (with a *direction* bit of one). Although the algorithm described above works correctly even if the initiator awaits a status block whose *attention* bit is one before signaling a target response ORB to receive the response data, it is more efficient to post both ORBs at the same time.

When a target issues a request to an initiator, they shall perform the following operations:

a) The target shall set the *attention* bit to one in a status block stored into the initiator *status_FIFO*. Either unsolicited status or completion status associated with an ORB may be used. So long as the target has pending control information to transfer to the initiator, it shall continue to set the *attention* bit to one in any status block stored into the initiator *status_FIFO*.

b) At any time the initiator receives a status block whose *attention* bit is one and there is no ORB in the task set whose *queue* field is zero and *direction* bit is one, the initiator shall create such an ORB and place it in the task set;

c) Once there is an ORB in the working set whose *queue* field is zero and *direction* bit is one, the target shall store the control information data (request) in the buffer described by the ORB and then store completion status for the ORB. The *attention* bit shall be zero in the status block associated with the ORB;

d) When the initiator has executed the indicated request, it shall store the response and its associated parameters (if any) in a buffer in system memory and signal to the target fetch agent an ORB that describes the control information buffer. The ORB's *queue* field and *direction* bit shall be zero and the *end_of_message* bit shall be one;

e) The target shall fetch the ORB and read the response from the control information buffer. The status block stored by the target to complete the ORB may have its *attention* bit set to one if the target intends to transfer other control information (request or autonomous response) to the initiator.

It is possible for both initiator and target to initiate requests at roughly the same time. In this case the working set contains an ORB for transfer of the request from initiator to target while the status block *attention* condition is simultaneously asserted by the target. The ordered execution properties of queue zero give a natural precedence to initiator requests over target requests, as follows. When a target fetches an ORB whose *queue* field and *direction* bit are zero and whose *end_of_message* bit is one, the request contained in the control information shall be processed before a request is transferred to the initiator. Consequently, if a target has an uncompleted initiator request when it fetches a control ORB and whose *direction* bit is one it shall not store any control information except the response that completes the request.

When neither initiator nor target have outstanding requests or responses, the control queue (queue zero) is idle and there shall be no ORBs in the task set whose *queue* field is zero.

6.2 Autonomous response information

The preceding clause describes the use of queue zero for request / response pairs between initiator and target. It is also possible for either initiator or target to autonomously transfer response information to the other. Autonomous response information is typically status information and does not necessarily require any additional action on the part of the recipient.

Autonomous response information may be sent for any of the *ctrl_function* values enumerated in the table below.

ctrl_function	**Name**
4	SERVICE DIRECTORY
5	STATUS

The *response* code in autonomous response information shall be zero.

Autonomous response information shall not be transferred while there is an uncompleted control request. A target requests the transfer of autonomous response information by means of the status block *attention* bit. If a target asserts *attention* and subsequently fetches an initiator request ORB, it shall first complete the initiator's control request and transfer the corresponding response information to the initiator before

transferring the autonomous response information. The *attention* bit shall remain asserted in any status blocked stored in the initiator *status_FIFO* while the transfer of the autonomous response information is pending.

6.3 Service discovery

Services implemented by either initiator or target are uniquely identified by their service ID, an ASCII string which may be registered with TBD. A client application that wishes to establish a connection with a particular service may attempt the connection without *a priori* knowledge that the service is implemented or the client application may request service directory information.

A service discovery request shall have a *ctrl_function* code of SERVICE DIRECTORY and no parameters. The response information shall contain zero or more SERVICE_ID parameters that identify services implemented by the target. There is no requirement that all implemented services be listed in the service directory. The order of the SERVICE_ID parameters in the response is unspecified.

TO BE DETERMINED – Some method of “paging” through large quantities of service ID information needs to be agreed.

6.4 Connection management

Table 2 – Connection type encoded by queue ID parameters

Connection type	I2T_QUEUE value	T2I_QUEUE value
Unidirectional	unrestricted	—
	—	unrestricted
Bi-directional (nonblocking)	not equal to T2I_QUEUE	not equal to I2T_QUEUE
Bi-directional (blocking)	equal to T2I_QUEUE	equal to I2T_QUEUE

6.4.1 Connection establishment

Before a client application may communicate with a service, necessary resources shall be allocated and confirmed by means of a control request with a *ctrl_function* code of CONNECT and its corresponding response. The operations are fundamentally similar whether the service resides at the initiator or the target, but because the initiator and target control different resources, the procedures are described separately.

The only initiator resource required for a connection is sufficient system memory to hold the ORBs for that portion of the task set allocated to the connection.

The target resources required for a connection are one or two available queue numbers, local memory to hold active ORBs and their associated context (up to the maximum set by the target *via* the TASK_SLOTS parameter) and an application client to provide the requested service.

Once a successful response has been received for a connection request, the initiator may place ORBs into the task set that use the queue number(s) specified in the target response data. The queue number(s)

remain valid until either a LOGOUT (either explicit on the part of the initiator or implicit as the result of a failure to reconnect after a bus reset), a shutdown of either or both queues or the connection is aborted.

6.4.1.1 Connection established by an initiator

When an application client at an initiator desires to establish a connection with a target service, it shall create a control ORB whose buffer contains a CONNECT control request. The initiator shall specify the SERVICE_ID and MODE parameters. The initiator may specify the TASK_SLOTS parameter, in which case the initiator shall guarantee that the task set never contains more active ORBs for the connection than the value set by TASK_SLOTS.

If the connection is established, the target shall return response data that specifies TASK_SLOTS and one or both of the I2T_QUEUE and the T2I_QUEUE parameters. When the initiator has provided the optional TASK_SLOTS parameter in its request, the target should return a TASK_SLOTS value less than or equal to that specified by the initiator.

Connection requests may fail because of a lack of target resources. This failure mode is probably transient; if the CONNECT control request is retried at some unspecified future time it may succeed. Other failure modes, indicated by the *resp* code, are fatal and should not be retried.

6.4.1.2 Connection established by a target

An application client at a target that desires to establish a connection with an initiator service shall create a buffer that contains a CONNECT control request and signal the initiator to retrieve the control request by asserting the *attention* bit in a status block. The CONNECT control request shall specify the SERVICE_ID, MODE, TASK_SLOTS and one or both of the I2T_QUEUE and T2I_QUEUE parameters. If the connection is established, the initiator shall guarantee that the task set never contains more active ORBs for the connection that the TASK_SLOTS value provided by the target.

If the requested service exists at the initiator and supports the requested transport flow mode, datagram or stream, the connection may be confirmed by a control response from the initiator. No parameters are required in the control response, but if the initiator specifies the TASK_SLOTS parameter it shall guarantee that the task set never contains more active ORBs for the connection than the value provided.

The initiator shall not use the queue number(s) identified by the I2T_QUEUE or T2I_QUEUE parameters until successful completion status has been stored at the initiator's *status_FIFO* for the ORB that transferred the control response to the target.

Just as a target may refuse a connect request because of insufficient resources, so may an initiator. Such a failure is probably transient and may be retried at some unspecified future time.

6.4.2 Queue shutdown

Although connections, which consist of one or two queues, are established by a single control function, there is no corresponding unified function to request disconnection. Instead, each queue may be shutdown individually. When all queues allocated to a connection are shutdown, the connection no longer exists and all resources allocated to it may be released.

Either the client application or the service may request queue shutdown; there are no fundamental differences. A more important consideration is whether it is the queue's data producer or data consumer that initiates the shutdown. Queue shutdown requested by the producer is orderly while shutdown requested by the consumer is not, as summarized by the table below.

Data transfer flow	Queue shutdown requestor	
	Initiator	Target
Initiator to target (I2T_QUEUE)	Orderly	Abortive
Target to initiator (T2I_QUEUE)	Abortive	Orderly
Bi-directional (I2T_QUEUE equals T2I_QUEUE)	Application-dependent	

Whether or not the shutdown of a bi-directional queue is abortive or orderly is dependent upon usage rules agreed by the client application and service and is beyond the scope of this document.

6.4.2.1 Queue shutdown by an initiator

When an application client or service at an initiator desires to shutdown a queue, it shall signal a transport flow ORB to the target whose *final* and *notify* bits are one and whose *queue* field specifies the queue to be shutdown. The initiator shall not signal any other ORBs with the same *queue* value unless the target allocates the queue number upon future establishment of a connection. A data buffer may be associated with a transport flow ORB whose *final* bit is one.

If the direction of data transfer *via* the queue is from target to initiator, the initiator shall signal a control ORB whose buffer contains a SHUTDOWN QUEUE control request whose T2I_QUEUE parameter is equal to the value of the *queue* field in the final ORB. This avoids a deadlock with the target and insures that the final ORB is eventually fetched and processed by the target. If completion status for the control ORB indicates that the SHUTDOWN QUEUE control request was not delivered to the target, the initiator shall either signal the control ORB again or reset the target by a write to its RESET_START register.

A target that receives a SHUTDOWN QUEUE request shall commence execution of all active ORBs for the queue identified by the T2I_QUEUE parameter and shall not defer completion of any ORB for the queue solely because data is not available to transfer to the initiator. If no data or, when the connection was established in stream mode, data insufficient to fill the initiator's buffer is available, transport flow ORBs for the queue shall be completed and the *residual* field in their completion status shall report the actual data transfer.

Once the target completes any indicated data transfer for the final ORB, it shall mark the queue as provisionally shutdown and store completion status at the initiator's *status_FIFO*. The target may not yet release the resources associated with the queue. If the initiator signals any other ORBs whose *queue* field identifies the provisionally shutdown queue, the target shall reject this with a completion status of TBD.

After the target has completed the final ORB, it shall discard any data generated by a client application or service for the provisionally shutdown queue. Because a service or client application at the initiator is unaware of the discarded data, if any, the shutdown of a queue used for data transfer from the target to the initiator may be disorderly (abortive) when requested by the initiator.

When the initiator receives completion status for the final ORB, it shall signal a control ORB whose buffer contains a RELEASE QUEUE control request. For a unidirectional queue, the initiator shall specify the I2T_QUEUE or T2I_QUEUE parameter, as appropriate, to identify the queue to be released. For a bi-directional queue, the initiator shall specify both parameters and their values shall be equal. If completion status for the control ORB indicates that the RELEASE QUEUE control request was not delivered to the target, the initiator shall either signal the control ORB again or reset the target by a write to its RESET_START register. Once completion status indicates successful receipt of the RELEASE QUEUE control request by the target, no additional initiator action is necessary. The RELEASE QUEUE control request has no corresponding response.

A target that receives a RELEASE QUEUE control request shall ignore it unless the queue specified by the I2T_QUEUE or T2I_QUEUE parameters is marked provisionally shutdown. In this case, the target may release all resources allocated to the queue. If no active queues remain for the connection to which the queue was originally allocated, any additional connection resources may be released. The target shall not respond to a RELEASE QUEUE control request.

6.4.2.2 Queue shutdown by a target

When an application client or service at a target desires to shutdown a queue, it shall create a control ORB whose buffer contains a SHUTDOWN QUEUE control request and signal the initiator to retrieve the control request by asserting the *attention* bit in a status block. For a unidirectional queue, the target shall specify the I2T_QUEUE or T2I_QUEUE parameter, as appropriate, to identify the queue to be shutdown. For a bi-directional queue, the target shall specify both parameters and their values shall be equal. After a successful completion response is received for the SHUTDOWN QUEUE request, the target shall not defer completion of any ORB for the queue solely because data is not available to transfer to the initiator. If no data or, when the connection was established in stream mode, data insufficient to fill the initiator's buffer is available, transport flow ORBs for the queue shall be completed and the *residual* field in their completion status shall report the actual data transfer.

NOTE – If the direction of data transfer *via* the queue is from target to initiator and an orderly shutdown, the target's client application or service should cease generation of data and the target should complete all outstanding data transfers before issuing the SHUTDOWN QUEUE request.

An initiator that receives a SHUTDOWN QUEUE request shall signal a transport flow ORB to the target whose *final* and *notify* bits are one and whose *queue* field specifies the queue identified in the control parameters for the request. The initiator shall not signal any other ORBs with the same *queue* value unless the target allocates the queue number upon future establishment of a connection. A data buffer may be associated with a transport flow ORB whose *final* bit is one.

Once the target completes any indicated data transfer for the final ORB, it shall mark the queue as provisionally shutdown and store completion status at the initiator's *status_FIFO*. The target may not yet release the resources associated with the queue. If the initiator signals any other ORBs whose *queue* field identifies the provisionally shutdown queue, the target shall reject this with a completion status of TBD.

After the target has completed the final ORB, it shall discard any data generated by a client application or service for the provisionally shutdown queue. Because a service or client application at the initiator is unaware of the discarded data, if any, the shutdown of a queue used for data transfer from the target to the initiator may be disorderly (abortive) when requested by the initiator.

When the initiator receives completion status for the final ORB, it shall signal a control ORB whose buffer contains a RELEASE QUEUE control request. For a unidirectional queue, the initiator shall specify the I2T_QUEUE or T2I_QUEUE parameter, as appropriate, to identify the queue to be released. For a bi-directional queue, the initiator shall specify both parameters and their values shall be equal. If completion status for the control ORB indicates that the RELEASE QUEUE control request was not delivered to the target, the initiator shall either signal the control ORB again or reset the target by a write to its RESET_START register. Once completion status indicates successful receipt of the RELEASE QUEUE control request by the target, no additional initiator action is necessary. The RELEASE QUEUE control request has no corresponding response.

A target that receives a RELEASE QUEUE control request shall ignore it unless the queue specified by the I2T_QUEUE or T2I_QUEUE parameters is marked provisionally shutdown. In this case, the target may release all resources allocated to the queue. If no active queues remain for the connection to which the queue was originally allocated, any additional connection resources may be released. The target shall not respond to a RELEASE QUEUE control request.

6.4.2.3 Queue shutdown initiated by both initiator and target

Unaware of the other's actions, an initiator and a target might initiate shutdown of the same queue simultaneously. In this case, there is no guarantee of orderly shutdown.

If an initiator receives a SHUTDOWN QUEUE control request that identifies a queue for which a final ORB has already been signaled it shall take no action other than to return a successful completion response to the target.

No special action is required of a target when a initiator signals a final ORB at roughly the same time as the target issued a SHUTDOWN QUEUE request. The unordered, split transaction properties of Serial Bus make it impossible for the target to distinguish this situation from a delay in the return of the completion response for the control request.

~~6.4.3Aborting a connection~~

6.4.3 Resetting a queue~~connection~~

EDITOR's NOTE – Working group to discuss whether or not explicit RESET CONNECTION is necessary. Are there synchronization problems if the queue is active (*i.e.*, the task set is not aborted)? Is it adequate to provide for an implicit connection reset by sending an ORB with a fresh signature after task set abort? Is there a necessity to resynchronize a queue at any other time?

6.5 Queue status information

7 Transport flow operations

Once a connection is established between a client application and a service, work is accomplished by the uni- or bi-directional flow of application-dependent data between the two. This section describes transport flow (and error recovery procedures) from the viewpoint of a queue instead of that of a connection; the reader may generalize from a single queue's operation to two coordinated queues that form a nonblocking bi-directional connection.

Service implementers select a datagram or stream model of transport flow. The datagram model is the simplest: there is a one-to-one relationship between ORBs, buffers and service data units (SDUs), as illustrated by Figure 15. The end of an SDU is demarcated by the *end_of_message* bit, which is always one when the datagram model is used.

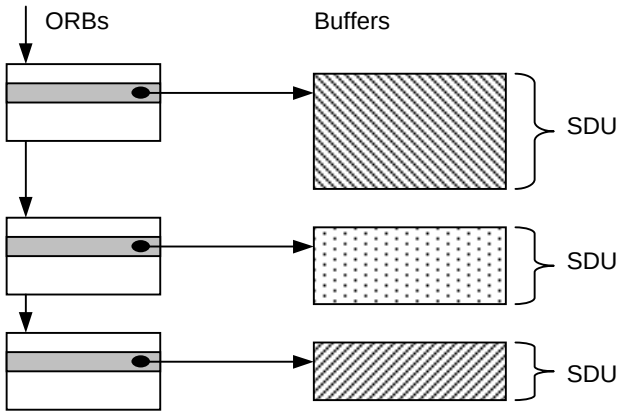

Figure 15 – Transport flow (datagram model)

The stream model permits SDU boundaries (*e.g.*, the separation between pages or print jobs for a printer) to occur without regard for the boundaries between data buffers specified by different ORBs. Figure 16 illustrates the relationship between stream data, the ORBs that describe its buffers and the SDUs.

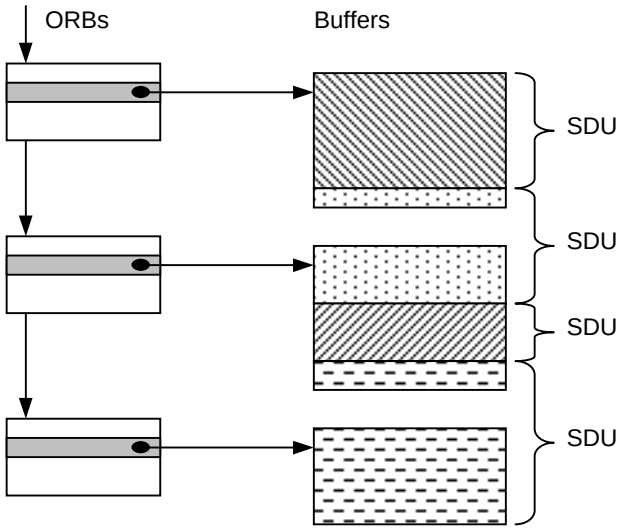

Figure 16 – Transport flow (stream model)

In the preceding figure, the stream data is assumed to be self-descriptive: it may be parsed by its recipient without the necessity for the ORBs to explicitly mark the SDU boundaries. The stream model may also be used in conjunction with SDUs that span more than one buffer, as illustrated by Figure 17.

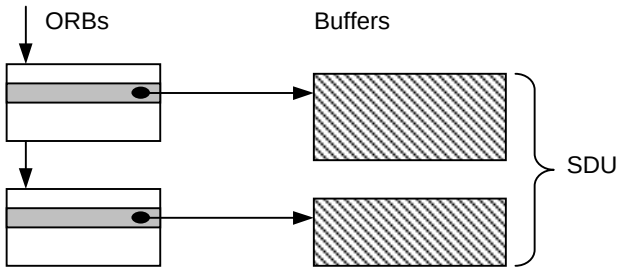

Figure 17 – Transport flow (spanned datagram model)

In the preceding figure, the *end_of_message* bit is zero in all but the last ORB. Spanned datagrams may be freely intermixed with the simple datagram model shown by Figure 15.

7.1 Data transfer to a target

Application data is transferred to a target by means of transport flow ORBs whose *direction* bit is zero. Within the limits of TASK_SLOTS allocated by the target at the time the connection (identified by the *queue* field) was established, the initiator may post more than one such outstanding transport flow ORB to the task set at a time. ORB fetch latency is reduced if the initiator is permitted to have at least two such outstanding ORBs in the task set.

The target transport may use read requests that address the data buffer in arbitrary order so long as none of the data is presented to an application client out of order. Upon successful completion of the data transfer, the *residual* field in the status block shall be zero.

The transport flow mode, datagram or stream, established when the connection was created, governs behavior when the initiator has more data available than the target is capable of processing at one time. For stream mode, this condition cannot arise; the target transfers data within the limits of local memory, delivers the data to the application client and continues to transfer data as local memory is released by the application client. Barring an unrecoverable error in the data transport or application client, all of the data described by the ORB is eventually transferred.

When datagrams are used, the possibility exists that an SDU is larger than the maximum acceptable to the target. In this case, no data shall be transferred and the *residual* field shall indicate the error condition. Figure 18 shows the relationship between the initiator's data buffer, the maximum SDU acceptable to the target and the value of *residual.* For simplicity, the figure assumes that no page table is used.

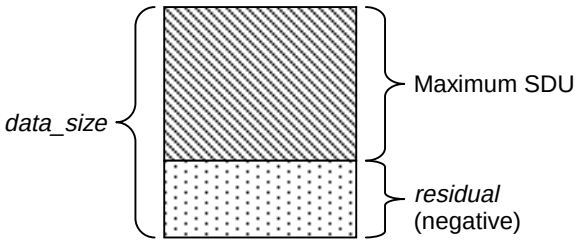

Figure 18 – Excess initiator data (datagram model)

The initiator may calculate the target's maximum acceptable SDU size by adding *residual* to *data_size*.

7.2 Data transfer to an initiator

Application data is transferred to an initiator by means of transport flow ORBs whose *direction* bit is one. Within the limits of TASK_SLOTS allocated by the target at the time the connection (identified by the *queue* field) was established, the initiator may post more than one such outstanding transport flow ORB to the task set at a time. ORB fetch latency is reduced if the initiator is permitted to have at least two such outstanding ORBs in the task set.

A target shall report the availability of data for a queue by setting *target_data_pending* bit to one in all status blocks stored for that queue's ORBs, regardless of the value of their *direction* bit, so long as there is untransferred data. If the initiator has posted no ORBs for a queue, the target may set *attention* to one in the status block for any ORB. This requests the initiator to post a control ORB to transfer queue information from the target (see 6.5) which in turn causes the initiator to post transport flow ORBs whose *direction* bit is one for the queues with available data.

The target transport may use write requests that address the data buffer in arbitrary order so long as successful completion status is not reported to the initiator until all of the data has been transferred. Upon successful completion of the data transfer, the *residual* field in the status block shall be zero.

The transport flow mode, datagram or stream, established when the connection was created, governs behavior when the target has more data available than the initiator is capable of processing at one time. For stream mode, this condition cannot arise; the target transfers data (supplied by its application client) within the limits of the data buffer provided by the initiator; if more data is available from the application client, the *target_data_pending* bit shall be one and the target awaits a subsequent ORB for the same queue whose *direction* bit is one. Unless an unrecoverable error occurs in the data transport, the target continues to fill initiator buffers so long as data is available.

When datagrams are used, the possibility exists that an SDU available at the target is larger than data buffer provided by the initiator. In this case, no data shall be transferred and the *residual* field shall indicate the error condition. Figure 19 shows the relationship between the initiator's data buffer, the SDU available at the target and the value of *residual*. Although the figure assumes that no page table is used, the relationships remain valid if a page table is present—except that the buffer size is summed from the page table elements instead of being directly available as *data-size*.

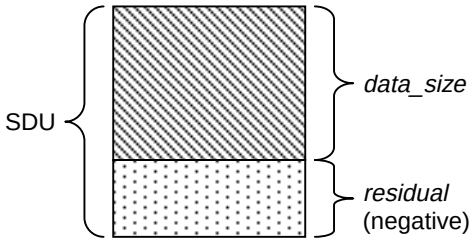

Figure 19 – Excess target data (datagram model)

The initiator may calculate the minimum buffer size necessary to receive the SDU by subtracting *residual* from *data_size*.

7.3 Completion status

The target shall signal completion status for a transport flow ORB by storing a status block to the initiator *status_FIFO* active for the login. Unless the buffer is sized incorrectly or a nonrecoverable error occurs, the target shall transfer all the data specified by the ORB and receive a response subaction of

resp_complete for each data transfer request subaction before it stores completion status to the initiator *status_FIFO*. All pending request subactions for the data transfer specified by the ORB shall be completed before the target stores a status block for the ORB to the initiator *status_FIFO*.

Although ANSI NCITS 325-1998 does not constrain targets which implement the unordered execution model to store completion status in order, this standard requires compliant targets to store completion status for each queue's ORBs in order as they are completed. In addition, the initiator shall deliver status to client applications or services in the same order indicated by the target. If a single event, such as an aborted task set, causes the initiator to simultaneously complete more than one ORB, the completion status shall be reported to the client in the same order as the ORBs were signaled to the target.

7.4 Execution context for active ORBs

This document specifies a data transport between services and their application clients that is reliable across interruptions, such as a bus reset, that cause target task set(s) to be aborted. The data transport is not only robust in these circumstances, but efficient. Data transfer may be quickly resumed without the necessity to redundantly move data already stored in or retrieved from an initiator's buffers. This is accomplished by cooperation between initiator and target in the use of the *signature* information field in transport flow ORBs. The *signature* field provides a method for the target to recognize an ORB ~~already in progress~~ previously active in an aborted task set ~~if it is signaled after a bus reset~~.

In order to recognize and correctly resume execution for ~~duplicate~~ previously active ORBs, the target shall maintain context information (a history log) for each active ORB. An ORB is active from the time the target fetches it and commences data transfer[2] up until the time completion status for the ORB is stored at the initiator's *status_FIFO* and either an *ack_pending* with a subsequent response of *resp_complete* or an *ack_OK* are received by the target.

The exact details of context information maintained by a target are implementation-dependent, but the context shall be sufficient to correctly resume execution of a ~~duplicate~~ previously active ORB if signaled by the initiator after a task set abort ~~bus reset~~. At a minimum, context information consists of the *direction*, *special* and *end_of_message* bits and the *queue* and *signature* fields for each active ORB as well as the status of data transfer—in progress or completed successfully or in error. Context information probably includes the original buffer size (derived from page table entries if a page table is associated with the ORB), the amount of data already transferred or remaining to be transferred and a pointer to the current location within the data buffer.

~~Under normal circumstances,~~ Context information for an active ORB ~~may~~ shall be discarded when the target receives a successful completion response after storing the status block for that ORB at the initiator's *status_FIFO*. If the target fails to receive successful completion response, context information for an active ORB shall be discarded when a successful completion response is received after storing a status block for a subsequent ORB for the same queue. An ORB is subsequent to another if it was signaled by the initiator after the first ORB.

7.5 Error recovery

All of the events in the following table cause one or more of the target's task sets to be aborted; see ANSI NCITS 326-1998 for details. Unless otherwise notice, a target shall preserve execution context for active ORBs across these events.

[2] The exact point in time at which an ORB becomes active is implementation-dependent and consequently difficult to define. An ORB is not yet active if the same ORB, signaled by an initiator after a bus reset, does not require context information in order for the target behavior to be essentially the same as if no bus reset had occurred. Whether or not data has been transferred is often an unreliable measure of an ORB's active status. For example, if a printer is designed to accumulate some minimum quantity of data before commencing image transfer to the medium, and ORB might not be active until print engine started.

Event	AGENT_STATE.*st*	Comment
Unrecoverable transaction errors	DEAD	Store status block CHECK CONDITION for faulted ORB, if possible.
ABORT TASK SET		
LOGICAL UNIT RESET		Target support for LOGICAL UNIT RESET is optional
Fetch agent reset (write to AGENT_RESET)	RESET	
TARGET RESET	DEAD	
Bus reset	RESET	For each login, a target shall retain, for at least *reconnect_hold* + 1 seconds after the bus reset, sufficient information to permit initiators to reconnect their logins. After this time, a target shall discard execution context for the task set of any initiator that failed to reconnect.
Command reset (write to RESET_START)		These events are equivalent; execution context for all ORBs in all task sets shall be discarded. Device operations should be halted and the device restored to an idle condition.
Power reset		

Unrecoverable transaction errors may be caused by a missing acknowledgement packet, a split transaction timeout, a data error or a retry limit exceeded. A missing acknowledgment by itself is not necessarily an error; the target shall wait a split timeout period before further action. If a transaction response is received within the split timeout period, there is no error. Otherwise, a split transaction timeout has occurred. In the case of a data error or a split transaction timeout, if the request was not addressed to an initiator *status_FIFO,* target may retry the transaction up to some implementation-dependent limit. Once the target deems a transaction error unrecoverable, it shall create a CHECK CONDITION for the ORB and transition the fetch agent to the dead state.

When an unrecoverable transaction errors occur for a request that does not address an initiator *status_FIFO*, the target should attempt to store status for the faulted ORB before transitioning the fetch agent to the dead state. This notifies the initiator that error recovery is necessary. If an unrecoverable transaction error occurs for a write request addressed to an initiator's *status_FIFO*, the target shall take no additional action. It is the initiator's responsibility to detect such an error, usually by means of a timeout.

~~**8.6Bus reset**~~

~~Upon a bus reset, the target aborts all task sets and awaits reconnection from initiator(s) active prior to the reset. Once an initiator successfully completes a RECONNECT with the target, its~~ After a task set has been aborted, an initiator's client application(s) and service(s) may resume data transfer with the target's service(s) and client application(s) on a connection by connection basis.

Data transfer between a client application and a service may have caused device operations to commence even if not all the data had been transferred before the task set was aborted ~~bus reset~~. For this reason, it is essential for each connection to be ~~resynchronized~~ resumed by one of two methods. The simplest ~~case~~ is to abandon any operations in progress, flush initiator and target buffers as necessary and return both endpoints of the connection to a known state—at which point the abandoned operation(s) may be reinitiated. The other approach is more efficient and uses execution context for active ORBs to permit resumption of data transfer at the point at which it was interrupted.

NOTE – A prerequisite to the resumption of data transfer is the existence of a login (the initiator reconnects to the target if there was a bus reset) and a reset fetch agent (the initiator writes to AGENT_RESET if the target's fetch agent had been left in the dead state after the task set was aborted).

7.5.1 Resetting a connection

An initiator may implement simple recovery by issuing a RESET CONNECTION control operation to the target. The connection to be reset shall be identified by the same I2T_QUEUE and T2I_QUEUE parameters provided by the target when the connection was established. Until the RESET CONNECTION control operation completes successfully, the initiator shall not signal any ORBs to the target whose *queue* field is equal to either the I2T_QUEUE or T2I_QUEUE parameter for the connection. Once a connection has been reset, ORBs may be signaled on the connection's queue(s) independent of the status of other connections for the same login.

7.5.2 Resynchronizing a connection

Although ~~this method of recovery from a bus reset~~ resetting a connection is robust, it may be improved upon. If the client application and service can reliably resume data transfer from the point it was interrupted, it may be unnecessary to cancel operations and flush buffers. In order for this method to work, the transport must be able to recognize resumption of an ORB ~~in progress~~active at the time the task set was aborted ~~of the bus reset~~. The *signature* field in a transport flow ORB (see 5.1) provides a method by which ~~identical~~previously active ORBs may be recognized if they are resubmitted after a task set abort ~~reset~~.

If the initiator elects not to reset the connection as specified by 7.5.1, data transfer may be safely resumed if initiator and target can identify, for each queue, the ORB's active at the time the task set was aborted ~~of the bus reset~~. For a particular queue, an initiator considers an ORB to be active if no status has been received from the target while a target considers an ORB active until positive acknowledgment of the receipt of status is signaled by the initiator. When an initiator does not reset a connection, it shall perform the following steps for each queue that forms the connection:

a) If there were no ~~uncompleted~~active ORBs in the task set whose *queue* field identifies one of the queue(s) that form the connection, no action is necessary and the initiator may resume data transfer for the connection;

b) Otherwise, for each active ORB for each of the connection's queues, the initiator shall signal an equivalent ORB to the target fetch agent. ~~For each queue, the ORBs shall be signaled in the same relative order as they had been prior to the bus reset.~~ Certain parts of the ORB shall remain unchanged: the *direction*, *special* and *end_of_message* bits and the *queue* and *signature* fields shall have the same values both before and after the task set abort ~~bus reset~~. The *data_descriptor*, and *data_size* fields and the *page_table_present* may have different values but they shall describe a buffer of the same size and whose contents are identical to the buffer described by the ORB at the time it was aborted ~~by the bus reset~~. The *spd* and *max_payload* bits may differ as a result of a different topology between the initiator and target ~~after the~~ if a bus reset caused the task set to be aborted.

NOTE – An initiator might signal equivalent ORBs in the same relative order within a queue as they had been prior to the task set abort. This straightforward strategy is known to work, but other correct implementations may be possible.

c) Once all the previously active ORBs for a particular queue have been signaled, the initiator may signal new ORBs in any order; these shall be interpreted by the target as if they are new; there are no restrictions on their field values.

When the target fetches an ORB, the action taken depends upon the value of the *queue* and *signature* field, which together uniquely identify an execution context for the initiator. If the value of *signature* is equal to the signature of an ORB active for the queue at the time the task set was aborted ~~of the bus reset~~, the target shall discard execution context information for any older, previously active ORBs for the same queue ~~that preceded the current ORB~~. An ORB is older than another ORB if it was signaled before the other ORB. If the value of the *signature* field is not equal to any previously active ORB for the queue, the target shall discard all execution context information for ~~all active ORBs for the same~~ that queue.

When the *signature* field identifies execution context for a previously active ORB, the target operations are determined by the data transfer state at the time the task set was aborted. If the data transfer had completed, successfully or in error, and completion status had been written to the initiator'' *status_FIFO* (but no response had been received from the initiator), the target simply stores the same completion status again. The ORB remains active until the conditions specified by X are met. Otherwise, when data transfer had been in progress, the target shall resume data transfer from the point specified by the execution context for the ORB.

8 Configuration ROM

All devices compliant with this standard shall implement general format configuration ROM in accordance with IEEE Std 1394-1995, draft standards IEEE P1394a and IEEE P1212r and the additional requirements of this document. Targets compliant with this standard shall also conform to the configuration ROM requirements of ANSI NCITS 325-1998, except as specifically exempted by this document. General format configuration ROM is a self-descriptive structure; an example appropriate to a target is illustrated below.

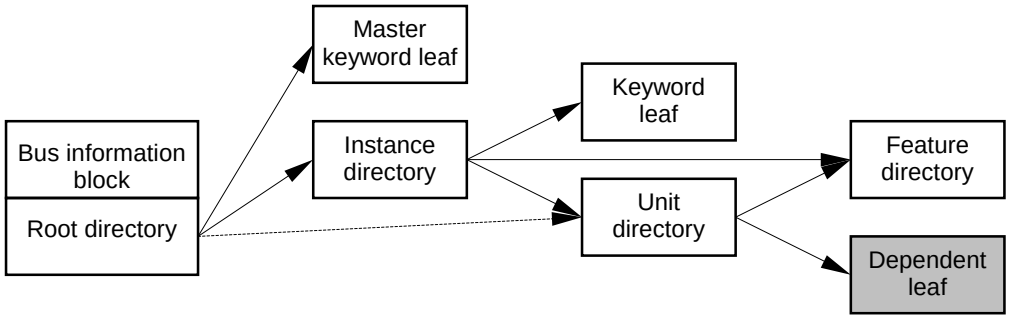

Figure 20 – Example configuration ROM hierarchy

With the exception of the dependent leaf (shown shaded), all of the configuration ROM components shown above are required for targets compliant with this standard. The connection from the root directory to the unit directory (shown by a dashed line) is optional; instance directories are the preferred access routes for unit directories.[3]

In addition to the requirements of the referenced standards and draft standards, the first five quadlets of configuration shall conform to the format illustrated by Figure 21.

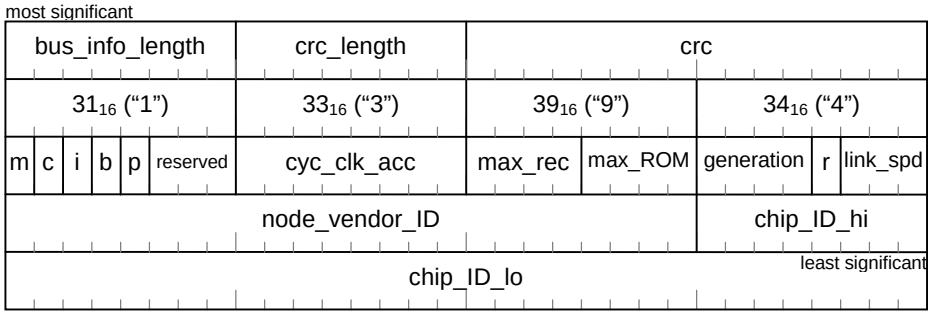

Figure 21 – First five quadlets of configuration ROM

The *bus_info_length* field shall have a value of four.

The *crc_length* field shall have a value of four plus the size, in quadlets, of the root directory. This indicates that the *crc* field is calculated for both the bus information block and the root directory—but not for any of the other configuration ROM data structures. The value of the *crc* field shall be calculated in accordance with draft standard IEEE P1212r.

[3] ANSI NCITS 325-1998 mandates a Unit_Directory entry in the root directory; this document is noncompliant in that respect and adheres to the more contemporary recommendations of draft standard IEEE P1212r.

The second quadlet shall contain the string “1394” in ASCII characters as specified by draft standard IEEE P1394a.

The meaning and usage of the *irmc*, *cmc*, *isc*, *bmc* and *pmc* bits (abbreviated as *m*, *c*, *i*, *b* and *p*, respectively, in the figure above) and the *cyc_clk_acc*, *max_rec*, *max_ROM*, *generation*, *link_spd*, *node_vendor_ID*, *chip_ID_hi* and *chip_ID_lo* fields are specified by draft standard IEEE P1394a.

The *max_rec* field shall have a minimum value of five.

The *max_ROM* field shall have a minimum value of one.

8.1 Root directory

Configuration ROM for devices compliant with this standard shall contain a root directory. The root directory immediately follows the bus information block and has an address of FFFF F000 0414_{16}. Relevant mandatory and optional entries for the root directory are summarized by the table below; unless explicitly excluded, any optional root directory entries permitted by draft standard IEEE P1212r are also permitted by this document.

Table 3 – Root directory entries

Directory entry			
Name	**Type**	**Mandatory**	**Description**
Vendor_ID	I	Y	24-bit RAC ID of the vendor that manufactured the device. This entry shall be immediately followed by a Textual_Descriptor entry. The addressed textual descriptor leaf (or leaves, if an intermediate textual descriptor directory exists) should contain an informal form of the vendor name easily recognizable by users.
Node_Capabilities	I	Y	Identifies which options of the CSR architecture are implemented.
Node_Unique_ID	L		Although permitted by draft standard IEEE P1212r, devices compliant with this standard shall not include a Node_Unique_ID entry in the root directory.
Keyword_Leaf	L	Y	"Thumbnail" description of the characteristics of the all instances implemented by the device.
Instance_Directory	D	Y	The instance directories provide a method to group unit architectures (software protocols) to identify shared physical components.
Unit_Directory	D		Unit_Directory entries are applicable only for targets. The use of Unit_Directory entries in the root directory is discouraged; designers should consult draft standard IEEE P1212r for more information.

The Vendor_ID entry shall contain the RAC ID of the vendor that manufactured the device and shall be immediately followed by a Textual_Descriptor entry that specifies the location of either a textual descriptor directory or leaf. The referenced textual descriptor leaf or leaves should contain an informal (short) form of the company name of the vendor.

A Keyword_Leaf entry is optional within the root directory and, if present, shall specify the location of a keyword leaf in configuration ROM. The keywords included in the keyword leaf shall be the union of all keywords from all keyword leaves in the device's configuration ROM. Simple devices that implement only one instance may reuse its keyword leaf as the master keyword leaf.

At least one Instance_Directory entry is required in the root directory; each shall specify the location of an instance directory in configuration ROM.

8.2 Instance directories

Configuration ROM for devices compliant with this standard shall contain one or more instance directories, each of which describes the function(s) implemented by a particular instantiation within the device. The mandatory and optional directory entries for an instance directory are specified by draft standard IEEE P1212r.

All instance directories shall contain a Keyword_Leaf entry.

8.3 Feature directories

All unit directories compliant with the requirements of clause 8.4 shall contain a Feature_Directory entry that specifies the location of a feature directory whose content and meaning are compliant with this clause. Configuration ROM may contain feature directories whose content and meaning are specified either by this standard, another organization or vendor. Relevant mandatory and optional entries for feature directories compliant with this document are summarized by the table below; unless explicitly excluded, any optional feature directory entries permitted by draft standard IEEE P1212r are also permitted by this document.

Table 4 – Feature directory entries

Directory entry Name	Type	Mandatory	Description
Specifier_ID	I	Y	24-bit RAC ID of the directory specifier, 00 5029_{16}.
Version	I	Y	In combination with the directory specifier ID, it identifies the software interface for the unit.
Service_ID	L		Collection of service ID text strings for all services implemented for the instance or unit.
Device_ID	L		Device identifier commonly used for plug and play device enumeration.
Initiator_Capabilities	I		Indicates that the instance can function as an SBP-2 initiator.

The Specifier_ID entry, whose 24-bit immediate value shall be 00 5029_{16}, and the Version entry, whose 24-bit immediate value shall be zero, identify this document as the specification of the feature directory.

The Service_ID entry, if present, shall specify the location of a leaf in configuration ROM that contains text strings, each of which is the service ID of a service implemented by the instance or unit. The format of the leaf shall be identical to that specified by draft standard IEEE P1212r for keyword leaves.

The Device_ID entry shall specify the location of a textual descriptor leaf in configuration ROM that contains a device identifying string in the format specified by IEEE Std 1284-1994 clause 6.6.

8.4 Keyword leaves

Each instance directory shall be characterized by a set of appropriate keywords selected from Table 5 and placed in a keyword leaf referenced by a Keyword_Leaf entry in the instance directory. Additional keywords may be present in any keyword leaf, but their meaning and usage are beyond the scope of this standard. Instances that share exactly the same set of keywords may reference the same keyword leaf.

Table 5 – Recommended keywords

Keyword	Recommended usage
CAMERA	Captures digital images.
COLOR	More than grayscale capabilities are supported, but the device may operate in a grayscale only mode.
DISK	Nonvolatile storage, often rotating.
FAX	Implements facsimile protocols commonly used over public switched telephone networks (PSTN) or integrated services digital networks (ISDN).
IMAGE	Applicable to ~~all~~ devices that capture, manipulate or transduce images~~described by this standard~~.
INITIATOR	Identifies the presence of initiator capabilities independently of target capabilities.
MULTIFUNCTION	Indicates the grouping of separate functions (*e.g.*, fax, printer and scanner) into a single controllable entity. Superior control may be available if the device is used as an MFP instead of as its separate functions.
MODEM	Data transmission protocols; may be dedicated or public switched telephone networks (PSTN).
PHOTO	Suited to the processing of photographic images.
PRINTER	Output device that marks removable media (hardcopy).
SBP-2	Applicable to all devices described by this standard.
SCANNER	Captures digital images, usually by means of relative motion between a sensor and a document.

8.5 Unit directories

Configuration ROM for targets compliant with this standard shall contain one or more unit directories, each of which specifies a software interface (unit architecture) for a device function. Relevant mandatory and optional entries for unit directories are summarized by the table below; unless explicitly excluded, any optional unit directory entries permitted by draft standard IEEE P1212r or ANSI NCITS 325-1998 are also permitted by this document.

Table 6 – Unit directory entries

Directory entry			
Name	**Type**	**Mandatory**	**Description**
Specifier_ID	I	Y	24-bit RAC ID of the directory specifier.
Version	I	Y	In combination with the directory specifier ID, it identifies the software interface for the unit.

Directory entry Name	Type	Mandatory	Description
Command_Set_Spec_ID	I	Y	24-bit RAC ID of the command set specifier, $00\ 5029_{16}$.
Command_Set	I	Y	In combination with the command set specifier ID, it identifies the command set for the unit.
Management_Agent	I	Y	Provides the address of the SBP-2 MANAGEMENT_AGENT register for login to the device.
Unit_Characteristics	I	Y	
Logical_Unit_Number	I	Y	
Reconnect_Timeout	I		Describes the maximum reconnect timeout supported by a logical unit; a minimum value of ten seconds is recommended
Feature_Directory	D	Y	Additional information that describes features (usually independent of the software interface and command set) of the unit. At least one of the feature directories shall be specified by this standard.

The Specifier_ID entry, whose 24-bit immediate value shall be $00\ 609E_{16}$, and the Version entry, whose 24-bit immediate value shall be $01\ 0483_{16}$, identify the device as compliant with ANSI NCITS 325-1998, SBP-2.[4]

The Command_Set_Spec_ID entry, whose 24-bit immediate value shall be $00\ 5029_{16}$, and the Command_Set entry, whose 24-bit immediate value shall be zero, identify the device as compliant with this document. The optional Command_Set_Revision entry, if present, shall have a 24-bit immediate value of zero.

The Unit_Characteristics entry shall specify a vendor-dependent *mgt_ORB_timeout* and an ORB size of eight quadlets (32 bytes). Consult ANSI NCITS 325-1998 for details.

Devices compliant with this standard shall contain a single Logical_Unit_Number entry for logical unit zero in each unit directory. The entry shall specify an unordered execution model (the *ordered* bit shall be zero). The *device_type* field shall be $1F_{16}$, unspecified device type.

The Reconnect_Timeout entry is optional, but because connections between client(s) and service(s) are terminated by a reconnection failure, designers should give careful consideration to a value for *max_reconnect_hold*. Omission of this entry sets the default value to one second, which may not be appropriate for the intended application.

There shall be at least one Feature_Directory entry that specifies the location of a feature directory whose content and meaning are specified by this standard. There may be additional Feature_Directory entries that reference feature directories whose content and meaning are specified either by this standard, another organization or vendor.

[4] The names given are those used by draft standard IEEE P1212r; they correspond to the names Unit_Spec_ID and Unit_SW_Version, respectively, in both ISO/IEC 13213:1994 and ANSI NCITS 325-1998.

Annex A
(normative)

Minimum Serial Bus node capabilities

In addition to the minimum capabilities defined by IEEE Std 1394-1995, ANSI NCITS 325-1998 and draft standard IEEE P1394a, this annex specifies other capabilities or restrictions mandated by this standard.

A.1 Initiator capabilities

TO BE DETERMINED – Review all of these (from SBP-2) and determine if this profile requires any GREATER capabilities.

With the exception of configuration ROM and control and status registers, an initiator shall be capable of responding to block read or write requests with a *data_length* less than or equal to ~~32~~64 bytes.

An initiator shall also be capable of responding to block read requests with a *data_length* less than or equal to 4 * *ORB_size*, where *ORB_size* is obtained from the Unit_Characteristics entry in the target's configuration ROM.

For the largest value of *max_payload* specified in any command block ORB it signals to the target, an initiator shall be capable of responding to block read and write requests with a *data_length* less than or equal to $2^{max\_payload + 2}$ bytes.

An initiator shall report the largest of these possible *data_length* values by setting the value of the *max_rec* field in the bus information block in its configuration ROM to a value equal to or greater than ($\log_2$ *data_length*) - 1.

A.2 Target capabilities

TO BE DETERMINED – Review all of these (from SBP-2) and determine if this profile requires any GREATER capabilities.

A target shall be capable of responding to block read or write requests with a *data_length* equal to eight bytes if the *destination_offset* specifies either the MANAGEMENT_AGENT or the ORB_POINTER register.

A target shall be capable of initiating write requests and shall report this by setting the *drq* bit in the Node_Capabilities entry in configuration ROM to one. Consequently, targets shall implement the *dreq* bit in the STATE_CLEAR and STATE_SET registers. The value of STATE_CLEAR.*dreq* shall be unaffected by a Serial Bus reset. Targets may automatically set *dreq* to zero (request initiation enabled) upon a power reset or a command reset.

A target shall be capable of initiating block write requests with a *data_length* of at least 16 bytes.

While initializing after a power reset, a target shall respond to quadlet read requests addressed to FFFF F000 0400_{16} with either a response data value of zero or acknowledge the request subaction with *ack_tardy*, as specified by draft standard IEEE P1394a. This indicates that the remainder of configuration ROM, as well as other target CSRs, are not accessible.

Targets shall support management request functions addressed to the MANAGEMENT_AGENT register as specified by the table below.

function	Support	Description
0	Mandatory	LOGIN
1	Mandatory	QUERY LOGINS
2	—	Reserved for future standardization
3	Mandatory	RECONNECT
4	Optional	SET PASSWORD (see ANSI NCITS 325-1998 Annex C)
5 – 6	—	Reserved for future standardization
7	Mandatory	LOGOUT
8 – A_{16}	—	Reserved for future standardization
B_{16}	Not supported	ABORT TASK
C_{16}	Mandatory	ABORT TASK SET
D_{16}	—	Reserved for future standardization
E_{16}	Not supported	LOGICAL UNIT RESET
F_{16}	Mandatory	TARGET RESET

Annex B
(normative)

Compliance with ANSI NCITS 325-1998

Subsequent to the approval of SBP-2 as an American National Standard, the IEEE P1212r working group commenced a revision of the CSR Architecture. The image data transport protocol, based upon SBP-2, conforms to the more recent recommendations and requirements of draft standard IEEE P1212r, some of which conflict with normative requirements of ANSI NCITS 325-1998. This annex lists the points of divergence as well as areas for which this standard specifies SBP-2 implementation constraints not required by ANSI NCITS 325-1998.

B.1 Divergences from ANSI NCITS 325-1998

EDITOR's NOTE – This is a group action item; it awaits careful review of both this draft and SBP-2 to identify points of divergence. This review is necessary before this draft goes to ballot.

B.2 Implementation constraints in addition to ANSI NCITS 325-1998

An initiator shall report completion status to its application clients and services in the same order as the status block(s) are written to the initiator's *status FIFO*. This additional requirement is necessary to permit the ordered execution of ORBs within a single queue even though the target reports that it implements the SBP-2 unordered execution model in its configuration ROM.

If an event, such as the abortion of a task set, causes more than one ORB to simultaneously complete, an initiator shall report completion status to its application clients and services in the same order as which the ORBs were signaled to the target.

An initiator shall either permit its application clients and services to control the value of the *notify* bit for individual ORBs or shall set the *notify* bit to one for all ORBs.

Annex C
(normative)

Control request and response parameters

The table below provides a quick reference to the parameters associated with particular control requests and responses; consult section 6 for details for a particular request or response. Optional parameters are shown by parentheses; the last column indicates whether or not the response information may be sent autonomously.

ctrl_function	**Name**	**Requester**	**Request parameters**	**Response parameters**	**Autonomous response**
1	CONNECT	Initiator	SERVICE_ID MODE (TASK_SLOTS)	Queue ID(s)[5] TASK_SLOTS	No
		Target	Queue ID(s)[3] SERVICE_ID MODE TASK_SLOTS	(TASK_SLOTS)	No
2	SHUTDOWN QUEUE	—	Queue ID	No response allowed	
3	RESET CONNECTION	Initiator	Queue ID(s)[6]	—	No
4	SERVICE DIRECTORY	—	—	SERVICE_ID(s)	Permitted
5	STATUS	Initiator	—	QUEUE_INFO	Target only

[5] At least one queue ID parameter shall be present, either I2T_QUEUE or T2I_QUEUE, and both may be present. In the latter case the two queue ID parameters may identify different queues or the same queue.

[6] The queue ID parameter(s) shall be the same originally provided by the target when the connection was established.

Annex D
(normative)

Control and status registers

The control and status registers (CSRs) implemented by a target shall conform to the requirements defined by this standard and its normative references. The CSRs may be arranged in three principal categories:

- core registers defined by draft standard IEEE P1212r and required by either that standard or this document;
- bus-dependent registers required by IEEE Std 1394-1995; and
- unit architecture registers required by ANSI NCITS 325.1998.

The relevant standard shall be consulted for details of register definition and usage; the table below provides a quick reference that summarizes all CSRs used by this document. Except for the optional MESSAGE_REQUEST and MESSAGE_RESPONSE registers, all of the CSRs are mandatory.

Offset	Register name	Description
0	STATE_CLEAR	State and control information
4	STATE_SET	Sets STATE_CLEAR bits
8	NODE_IDS	Contains the 16-bit node_ID value used to address the node
C_{16}	RESET_START	Resets the node's state
18_{16} – $1C_{16}$	SPLIT_TIMEOUT	Time limit for split transactions
80_{16} – BC_{16}	MESSAGE_REQUEST	Message area for target requests when no login exists
$C0_{16}$ – FC_{16}	MESSAGE_RESPONSE	Message area for initiator responses to target requests addressed to MESSAGE_REQUEST.
210_{16}	BUSY_TIMEOUT	Controls transaction layer retry protocols
specified by configuration ROM	MANAGEMENT_AGENT	Login and other SBP-2 task management requests
specified by login response data	AGENT_STATE	Reports SBP-2 fetch agent state
	AGENT_RESET	Resets SBP-2 fetch agent
	ORB_POINTER	Address of current ORB
	DOORBELL	Signals SBP-2 fetch agent to refetch an address pointer
	UNSOLICITED_STATUS_ENABLE	Acknowledges the SBP-2 initiator's receipt of unsolicited status

Annex E
(informative)

Configuration ROM

Configuration ROM is located at a base address of FFFF F000 0400_{16} within a node's address space. The requirements for general format configuration ROM for devices compliant with this standard are specified in section 0. This annex contains illustrations of typical configuration ROM for a variety of devices.

E.1 Bus information block and root directory

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

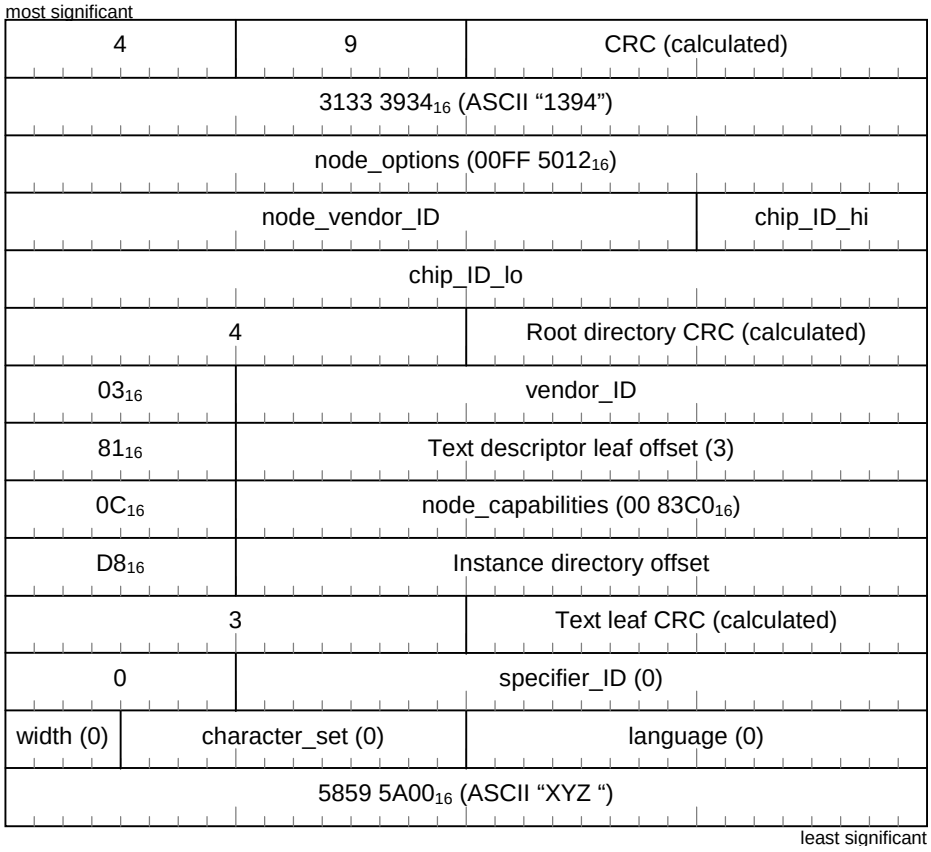

Figure E–1 – Example bus information block and root directory

The CRC in the first quadlet is calculated on following nine quadlets of configuration ROM, the bus information block and the root directory. Devices should not include all of configuration ROM within the coverage provided by this CRC; the other directories and leaves each contain their own CRC.

The *node_options* field represents a collection of bits and fields specified draft standard IEEE P1212r. The value shown, 00FF 2000_{16}, represents basic characteristics of a device that is not isochronous capable.

This value is composed of a *cyc_clk_acc* field with a value of FF_{16}, a *max_rec* value of five, a *max_ROM* value of one and a *link_spd* value of two. The *max_rec* field encodes a maximum payload of 32 bytes in block write requests addressed to the target.

The Node_Capabilities entry in the root directory, with a *key* field of $0C_{16}$, has a value where the *spt*, *64*, *fix*, *lst* and *drq* bits are all one. This is a minimum requirement for devices compliant with this standard.

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

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

E.2 Feature directory

Devices compliant with this standard implement a feature directory for each instance. An example of a feature directory, with its associated service ID and device ID leaves is illustrated by Figure E–2. Except for these generic features, additional content of the feature directory is device-dependent; see 8.3 for more details on the other directory entries that may be present.

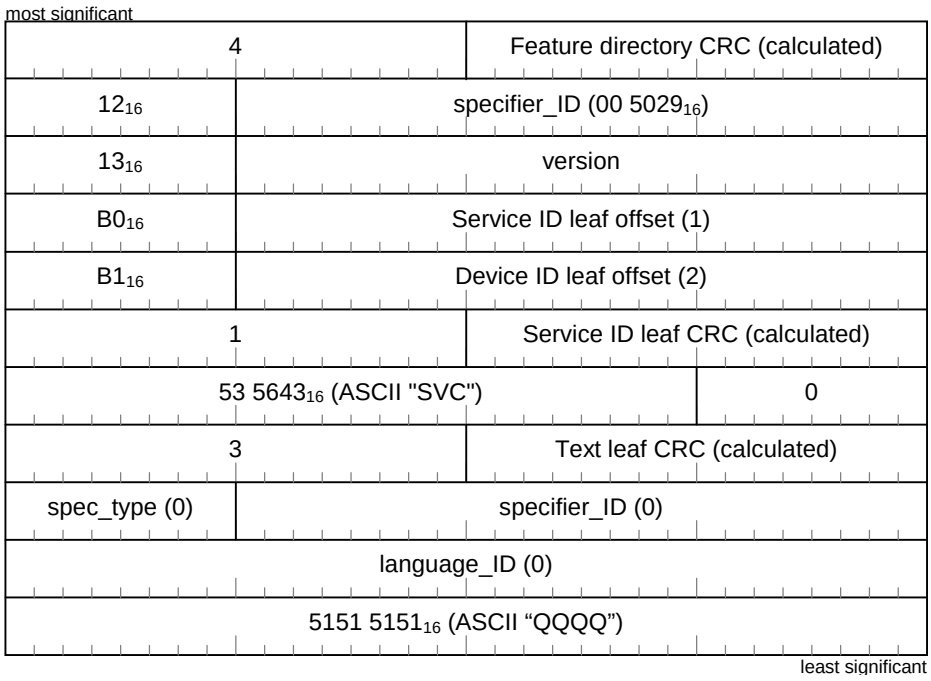

Figure E–2 – Feature directory with service ID and device ID leaves

The Specifier_ID and Version entries, with a *key* field of 12_{16} and 13_{16}, respectively, indicate that the format of the unit directory is specified by this document.

The Service_ID entry, with a *key* field of $B0_{16}$, points to the service ID leaf, which is in the same format as keyword leaves. The service ID leaf specifies a hypothetical service identified as "SVC".

The Device_ID entry, with a *key* field of $B1_{16}$, points to a textual descriptor leaf that contains an ASCII string that identifies the device to bus enumeration software.

E.3 Unit directory

Targets compliant with this standard implement at least one unit directory in the format illustrated by Figure E–3. Devices that support more than one software protocol (unit architecture) may implement additional unit directories whose format is specified by other documents.

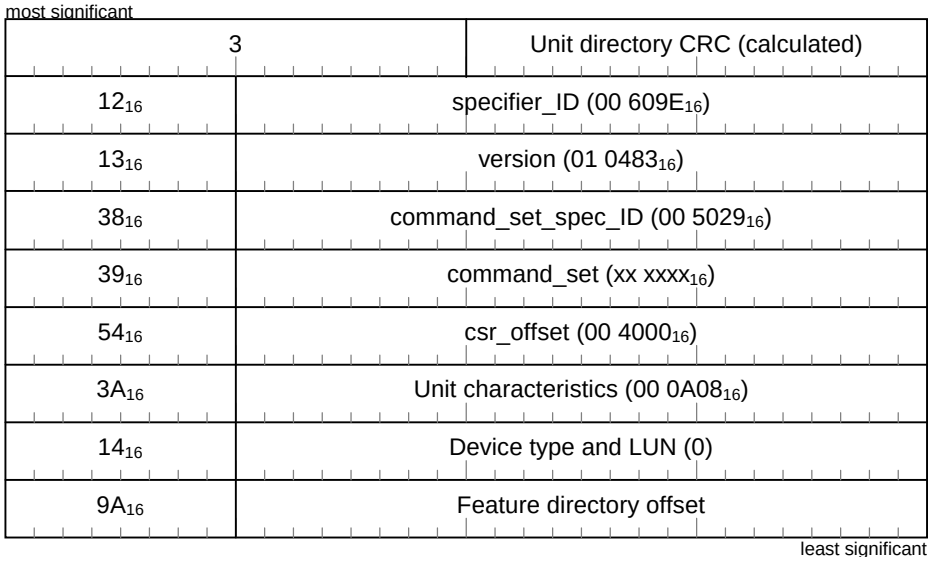

Figure E–3 – Unit directory for peer-to-peer data transfer (PPDT) protocol target

The Specifier_ID and Version entries, with a *key* field of 12_{16} and 13_{16}, respectively, indicate that the format of the unit directory is specified by ANSI NCITS 325-1998.

The Command_Set_Spec_ID and Command_Set entries, with *key* fields of 38_{16} and 39_{16}, respectively, indicate that the target use the peer-to-peer data transfer (PPDT) protocol specified by this document..

The Management_Agent entry in the unit directory, with a *key* field of 54_{16}, has a *csr_offset* value of 00 4000_{16} that indicates that the management agent CSR has a base address of FFFF F001 0000_{16} within the node's memory space.

The Unit_Characteristics entry in the unit directory, with a *key* field of $3A_{16}$, has an immediate value of 00 $0A08_{16}$. This indicates a target that is expected to complete task management requests (including login) within five seconds and fetches 32-byte ORB's.

The Logical_Unit_Number entry in the unit directory, with a *key* field of 14_{16}, has an immediate value of zero that indicates a device that may reorder tasks without restriction and has a logical unit number of zero.

At least one Feature_Directory entry is present; it addresses the same feature directory as the parent instance directory for this unit. See Figure E–2 for an example of a typical feature directory.

E.4 Scanner with a single unit architecture

The configuration ROM for a simple device, such as a scanner, that implements only one software protocol (unit architecture) utilizes the bus information block and root directory structure already described in Figure E–1. An example instance directory and its associated keyword leaf is illustrated by Figure E–4.

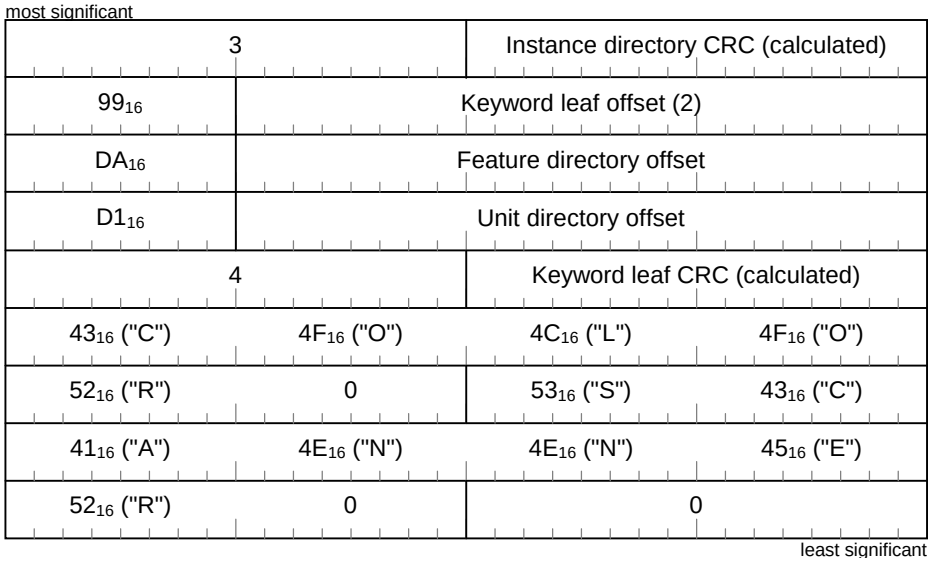

Figure E–4 – Instance directory and keyword leaf for a scanner

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

Since this is a simple device that supports a single software protocol (unit architecture), there is only one Unit_Directory entry, with a *key* value of $D1_{16}$, in the instance directory.

E.5 Printer with multiple unit architectures

The configuration ROM for a more complex device, such as a printer that implements more than one software protocol (unit architecture) also utilizes the bus information block and root directory structure already described in Figure E–1. An example instance directory and its associated keyword leaf is illustrated by Figure E–5.

most significant

4		Instance directory CRC (calculated)	
99_{16}	Keyword leaf offset		
DA_{16}	Feature directory offset		
$D1_{16}$	Unit directory offset (PPDT protocol)		
$D1_{16}$	Unit directory offset (DPP)		
6		Keyword leaf CRC (calculated)	
50_{16} ("P")	48_{16} ("H")	$4F_{16}$ ("O")	54_{16} ("T")
$4F_{16}$ ("O")	0	50_{16} ("P")	52_{16} ("R")
49_{16} ("I")	$4E_{16}$ ("N")	54_{16} ("T")	45_{16} ("E")
52_{16} ("R")	0	44_{16} ("D")	50_{16} ("P")
50_{16} ("P")	0	53_{16} ("S")	42_{16} ("B")
50_{16} ("P")	$2D_{16}$ ("-")	32_{16} ("2")	0

least significant

Figure E–5 – Instance directory and keyword leaf for a multiple protocol printer

The Keyword_Leaf entry, with a *key* value of 99_{16}, points to a keyword leaf that contains the keywords PHOTO, PRINTER, DPP and SBP-2.

Since this is target supports multiple software protocols (unit architectures) for the same physical instance of the print engine, there are two Unit_Directory entries in the instance directory, one that references a unit directory compliant with this standard and one that references a Direct Print Protocol (DPP) unit directory.

E.6 Multifunction device with uniform unit architectures

EDITOR's NOTE – Has the 1394 PWG reached consensus as to what this example should show—or even if it will be intelligible to the average reader?