TA Document 2001009

AV/C Compatible Asynchronous Serial Bus Connections 2.1

July 23, 2001

Sponsored by:
1394 Trade Association

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

Abstract:
This specification defines a set of lightweight protocols for efficiently streaming asynchronous data between nodes. Data is transported by writing into a consumer-resident segment buffer; updates of space-available and data-available counts provide bi-directional flow control. For simplicity all data structures are quadlet aligned and segment buffer addresses are power-of-two aligned.

Keywords:
asynchronous, lightweight connections, 1394, streams, segment buffer, flow control.

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

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

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

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

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

Editor, 1394 Trade Association
Regency Plaza Suite 350
2350 Mission College Blvd.
Santa Clara, Calif. 95054, USA

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

Table of contents

List of figures

List of tables

Change history

The following table shows the change history for this specification.

Version 2.0 (October 24, 2000)

The original version.

Version 2.1 (July 23, 2001)

*Content change for version 2.1

Category	Description
Editorial	The latest document was included in the references.
Editorial	Some of the descriptions for the size of the segment buffer were corrected.
Technical	The operation to set the sc value of the FREE confirmation was added in the producer state machine.
Technical	The conditions to check the sc values of the FREE and SUSPENDED confirmations were added in the consumer state machine.
Editorial	Some descriptions for the operations to set the APR.run values after a bus reset were corrected in the consumer state machine.
Editorial	More proper descriptions for the transitions to the INACTIVE state were given in the consumer state machine.

1. Overview

This clause "1. Overview" contains introductory material and is informative in nature. Other clauses (before the annexes) are normative and define the specification details. Annexes are informative information.

1.1 Purpose and scope

This document describes asynchronous connections, an asynchronous data-transfer service that efficiently transfers data between simple Serial Bus compliant devices. Standardization of asynchronous connections is intended to simplify device hardware and software, by providing a common transport for transferring (potentially) different forms of application-specific data.

This document replaces Reference **[R13]**

1.1.1 Purpose

The purpose of asynchronous connections is to provide efficient robust data-transfer services for Serial Bus devices including, but not limited to, AV devices that are controlled by AV/C commands. The same protocols can be used for many devices, although the content of the transferred data is expected to be application dependent.

1.1.2 Scope

Asynchronous connections provide a lightweight mechanism for robustly transferring large amounts of data between Serial Bus nodes. Bi-directional flow control limits the data transfer rate to the slower of the rates supported by the producer or consumer. The architecture is compatible with software managed interfaces (such as Open Host Controller Interface (OHCI)) and allows for hardware optimizations.

1.2 Design objectives

Asynchronous connections were designed to be simple and provide an asynchronous equivalent of isochronous data-transfer services. Specific design objectives include the following:

1) Lightweight. Simple asynchronous unidirectional data-transfer protocols should allow large data frames to be efficiently transferred between producer and consumer nodes.
2) Robust. The protocols should allow recovery from any number of bus resets and data transmission errors.
3) Leveraged. Existing AV/C resources may be used for connection management purposes.
4) Extendable. Variations of the basic producer-to-consumer data-transfer protocols should be available to support multicast, parallel path, and duplex connections.

1.3 Asynchronous connections highlights

An asynchronous connection is comparable to an isochronous stream: in an isochronous stream, data flows between talker and listener nodes; in an asynchronous connection, data flows between a producer node and one or more consumer nodes. Asynchronous connections do not have real-time delivery guarantees that characterize isochronous streams, but are more robust and are flow-controlled. Sequences of asynchronous

write transactions, rather than isochronous data packets, are used to transfer data, providing reliable data delivery. Highlights of asynchronous connections include:

1) Simple and efficient. The producer writes into a consumer-resident segment buffer and updates the consumer-resident data-available count. The consumer reads from this segment buffer and updates the producer-resident space-available count.
2) Flow controlled. The simple count-update protocol efficiently limits the data transfer rate to the slower of the producer and consumer nodes.
3) Robust. Complete recovery from bus errors or bus resets is supported. The idempotent nature of the data and count-register writes allows the data transfers to continue despite one or many disruptions.
4) Scalable. Asynchronous connections allow nodes to provide a wide range of physical buffer sizes (64 bytes to 16 megabytes), based on the capabilities of the consumer node.
5) Data push. Data is pushed from the producer to the consumer, using sequences of write transactions. These write transactions (that can be ack_complete'd) are expected to be more efficient than read transactions.
6) Directed and 1-to-N multicast. Directed and multicast 1-to-N asynchronous connections are supported.
7) Data frames. Data bytes are grouped into variable length sequences of data bytes called frames. Frame boundaries provide synchronization points for application-specific data descriptors.

1.4 Enhanced behavior

This version of asynchronous connections describes a feature not included in previous versions of asynchronous connections. This feature is sometimes referred to as *enhanced behavior* and the feature is enhanced segment buffer types.

Previously, the segment buffer was required to be located in contiguous memory with the plug registers. Also, this was the only buffering scheme for asynchronous connections. This document describes how the producer and consumer may select among multiple segment buffer types. It also describes one buffering scheme that allows the segment buffer to be located in memory that is not contiguous with the plug registers.

1.5 Functional overview

1.5.1 Data transfers

Asynchronous connections are optimized for copying data from a producer node to one or more consumer nodes. With the "push" data transfer model, the producer writes into the consumer's data buffer and less efficient read transactions are never used for data transfer.

The consumer starts the cycle by indicating how much data the producer can safely write into the buffer by writing a limit-count value into the producer port's control register (oAPR[n]). Data can then be written into a segment buffer until the end of the buffer has been reached. The data consumption is triggered by the producer that signals, through a control register update (iAPR), when the segment writes have completed. After processing this data, the consumer re-enables writing to the segment buffer by indicating how much data the producer can safely write.

A register containing a limit count (located in the producer (oAPR[n])) and a register containing a delivered count (located in the consumer (iAPR)), provide bi-directional flow control. This allows the data transfer rate to be paced by the slower of the producer and consumer nodes. These registers are updated using the CompareSwap4 (a 4-byte lock transaction called compare-and-swap) transaction, which provides a consistency check and efficiently supports existing Serial Bus host interfaces, such as the open host controller interface (OHCI).

A standard method for grouping data into (potentially large) *frames* is specified. The register update at the end of each segment provides distinctive end-of-frame indications. As an option, data within each frame can be further decomposed into a sequence of subframes, where the length of each subframe is specified by its header (see annex C for details).

1.5.2 Asynchronous plugs

On asynchronous connections, data transfers involve writes into the address space of the consumer node.

This address space and affiliated internal state are called an asynchronous plug. The plug address is minimally 64-byte aligned, i.e. the plug address shall be a multiple of 2^n, where *n* is 6 or larger.

The controller is responsible for establishing an asynchronous connection, by connecting producer and consumer components. The plug1 and plug2 addresses, located on producer and consumer nodes, need not be related. After the connection process completes, each plug has parameters that specify its counterpart plug location and capabilities, as illustrated in Figure 1.1.

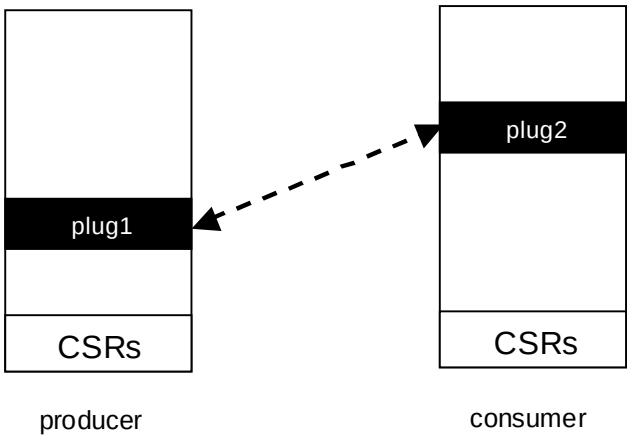

Figure 1.1 – Coupled producer and consumer components

Establishing an asynchronous connection associates one node (the producer node) with another node (the consumer node). In its simplest form, the producer node produces frames and the consumer node consumes frames.

The resources associated with an asynchronous connection depend on the properties of the connected device. The data-transfer protocols are optimized for basic simplex connection (see section 1.5.3).

1.5.3 Basic data transfers

A basic connection transfers data between the producer and consumer components, as illustrated in Figure 1.2.

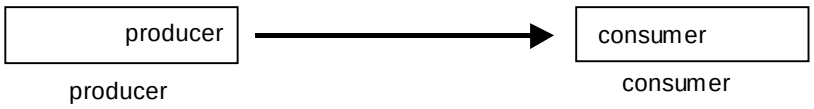

Figure 1.2 – Basic simplex connection

The basic connections are appropriate for devices that use AV/C commands to control their operations, but desires to transfer significant amounts of asynchronous data.

1.6 Data frames

Framing facilitates support for unknown-length transfers, or unexpected termination of normally fixed-length transfers, as could occur in a receiving-FAX application. Each frame starts at a zero-valued segment buffer offset. The use of out-of-band end-of-frame signaling (updates of the *iAPR* (input Asynchronous Port Register) register) allows each frame to contain an arbitrary number of data bytes. A frame may be composed of one or more segments. The grouping of data bytes into frames is graphically illustrated in Figure 1.3.

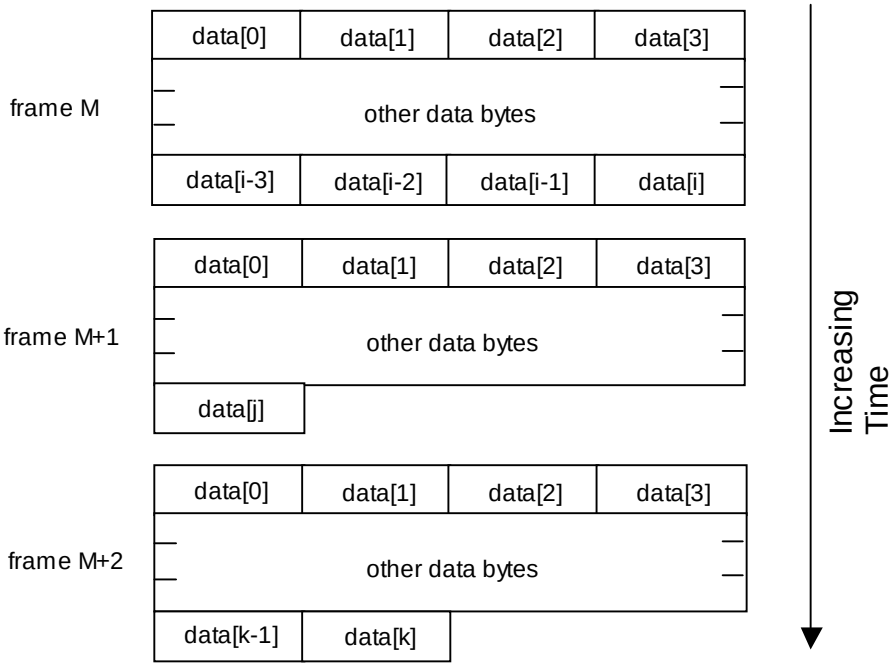

Figure 1.3 – Frame sequences

Each frame has affiliated status information to distinguish between correct and corrupted frames.

1.7 Rate limiting options

1.7.1 Delayed write-request processing

A consumer may place all incoming requests into a receive buffer (illustrated as black), as illustrated in Figure 1.4, before interrupting the microprocessor. While the microprocessor is copying the previously queued write-request into local buffers, further segment buffer writes cannot be accepted.

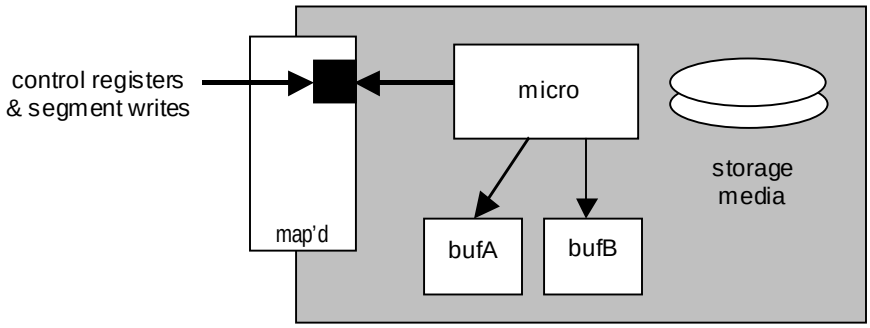

Figure 1.4 – Emulated segment buffer addresses

If further write requests were to be received before the queued write request had been saved, these writes (and their retries) would be busied. These retries are inefficient and (if allowed to continue) would reduce the Serial Bus bandwidth available to other nodes.

When back-to-back segment buffer writes cannot normally be processed without delay (which would force the second write to be busied), the node shall throttle the generation rate of write requests to match its write-request processing rate, as follows:

1) Sequential writes. When the asynchronous connection is established, the consumer shall provide the producer with a zero-valued *ct* (concurrent transaction) consumer-plug address modifier (see 4.2.4).

2) Responsive writes. When a write request is accepted, the consumer node shall provide an ack_pending acknowledge (an ack_complete acknowledge shall not be used) and the write response shall not be returned until the write buffer has been emptied.

Constraint (1) forces the producer's segment buffer writes to be performed in a sequential fashion, so the next write transaction will not start until the previous write transaction has completed. Constraint (2) forces the consumer to empty its write buffer before the completion of each write transaction. In the absence of writes from other nodes, these two constraints will ensure that write requests are never busied.

1.7.2 Immediate write-transaction processing

A higher performance hardware-assisted consumer could provide distinct write-request buffers for holding control-register and segment buffer writes. Although the microprocessor processes control register writes, the segment buffer writes are directly transferred into memory, as illustrated in Figure 1.5.

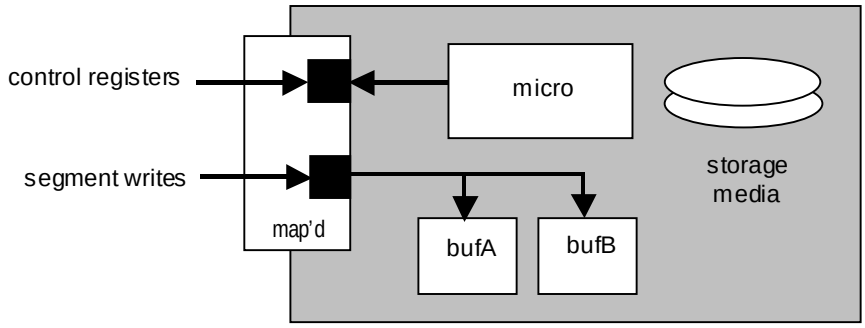

Figure 1.5 – Emulated segment buffer addresses

If such nodes have sufficient write-transaction processing bandwidth, there is no need to throttle the producer. In the absence of this throttling requirement, either or both of the following performance-enhancement features are allowed:

1) Concurrent writes. When the asynchronous connection is established, the consumer may provide the producer with a one-value *ct* (concurrent transactions allowed) consumer-plug address modifier (see section 4.2.4).

2) Posted writes. When a write request is accepted, the consumer node may provide an ack_complete or ack_pending acknowledge.

However, these options may not be implemented for other reasons: 1) extra hardware is required to support concurrent writes and 2) safe posting of writes requires an early address decode to distinguish between valid and invalid address offset values.

1.8 Asynchronous plug management

1.8.1 Managing asynchronous plugs

Multiple tasks are involved in establishing and utilizing asynchronous connections, as listed below and described in the remainder of this subsection.

1) Connection. The controller determines which sets of target nodes should be connected and establishes those connections.

2) Object selection. Object selection protocols identify the desired data-transfer objects. On a camera, for example, object selection determines which image is to be sent.

3) Data transfer. Data-transfer protocols allow one or more frames to be sent in an asynchronous (bi-directional flow controlled fashion) between producer and consumer nodes.

4) Resumption. After a bus reset, the controller is responsible for quickly resuming asynchronous connection traffic, before these connections are broken by a post-reset time out.

5) Disconnection. There are two possible methods of disconnection. First, the controller determines which pair of nodes should be disconnected and, second, a target node that is part of a connection may initiate disconnection.

Establishing connections (1) requires the most intelligence and is delegated to a distinct component called the controller. Sophisticated controllers are expected to be distinct physical components (such as digital TVs or set top boxes). Simpler embedded controllers are expected to be located within the producer or

consumer components. Embedded controllers may only be capable of connecting to bus-local devices; more sophisticated controllers could be used to connect between non-local devices.

Object selection (2) may be done by a controller (the home computer selects objects to be played from disk) or a device (the camera selects the pictures to be printed). Although the data is transferred over an asynchronous connection, a properly configured asynchronous connection (particularly a dual-duplex connection) could also be used for object selection. The definition of object selection protocols is beyond the scope of this standard.

The data transfer protocols (3) are the focus of this document. Transfers over asynchronous connections involve block write transactions (that transfer the data) and CompareSwap4 transactions (that update flow-control registers, see section 4.3). For simplicity and efficiency, these tasks are delegated to the producer and consumer components; the controller is not involved in the lower-level flow-control protocols.

Resumption of asynchronous plugs (4) is currently the responsibility of the controller that established the connection. If that controller is no longer present or functioning, a disconnection timeout causes the connection to be broken. Extensions to this standard may be designed to support communications across bus bridges; these extensions are expected to relieve the controller of this burden, placing the responsibilities to maintain cross-bus connections on the producer and consumer nodes.

Disconnection (5) has two methods: controller initiated or target node (producer or consumer) initiated. The controller-initiated disconnection is a normal controller action, similar to connection. The second case allows one of the target nodes to initiate the tear-down of the connection. This method requires that the controller that established the connection know that a controller-initiated disconnection is not required. The target nodes communicate via the plug's flow control registers and effect a complete disconnection without controller intervention. This method is typically used when the controller is resident in a target node.

1.8.2 Basic point-to-point connections

The basic connection transfers data between producer and consumer plugs, as illustrated in Figure 1.6. In this figure, each plug is represented by a large rectangle within the node. Each plug has one or more attached ports, shown as smaller adjacent rectangles. The double arrow between plugs represents the logical connectivity (each of the plug's ports has a pointer to the other port) and the larger arrowhead indicates the data-transfer direction.

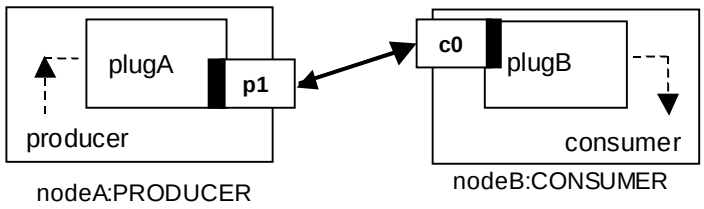

Figure 1.6 – Basic connection

The small black rectangles represent the *PortContext* information, which contains the address of the remote plug.

1.8.3 Multicast connections

Simple extensions of the basic connections are sufficient to support radial multicasts. Radial 1-to-N multicast involves multiple producer plugs, one for each of the attached consumers, as illustrated in Figure 1.7. In this figure, the larger arrowheads illustrate the data-transfer direction.

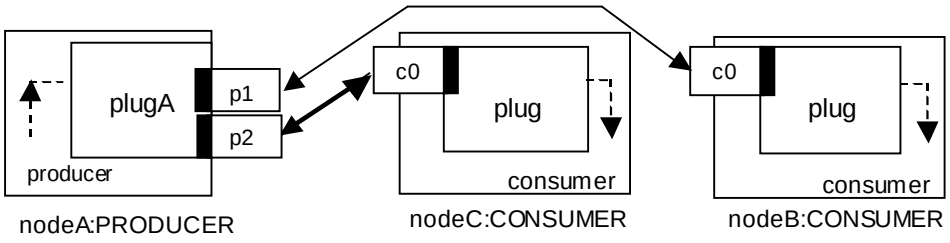

Figure 1.7 – Basic multicast connections

As before, the small solid rectangles represent the *PortContext* information, which specifies the address of the remote plug. For multicast connections, the producer may have multiple (up to 14) contexts, one for each of the attached consumers.

NOTE —The 14 contexts correspond to a subset of the 16 port addresses, where one of these port addresses is reserved for use by a consumer plug's port and another is reserved and its number is used to communicate a desire to "select any" port address.

The same data can be written to each of the consumers, using separate sets of interleaved write transactions.

When the most restrictive consumer-specified *oAPR* count is reached, these writes cease until the most restrictive *oAPR* value changes. Note that the consumer with the most restrictive *oAPR* may change over time, due to dynamically changing data consumption rates.

Because each consumer may have a different segment buffer size or behavior, the multicast-producer plugs are assumed to have independent *oAPR* and *iAPR* update sequencers. To simplify the multicast producer designs, the size of segment buffers on multicast capable consumers is constrained to be a multiple of 2k bytes in size.

1.9 File-type transfers

Asynchronous connections have a bi-directional flow control mechanism and can be used with file-type or stream-type applications. In file-type transfers, all of the selected frame data in the producer is transferred to the consumer. If the consumption rate is less than the production rate, the producer transfers are delayed while previous segment data of the frame are being processed by the consumer.

1.10 Stream-type transfers

1.10.1 Stream-type producers

In stream-type applications, the producer may be receiving real-time data; examples are data coming from a video tape or broadcast signal source. Since most producers are expected to have storage for only a few frames, a congested interconnect may force some frames to be discarded at the producer, as illustrated in Figure 1.8. In this illustration, shading of the producer frame corresponds to the portion of the frame that

has been transferred to the consumer; shading of the consumer frame corresponds to the portion of the frame that hasn't been received from the producer.

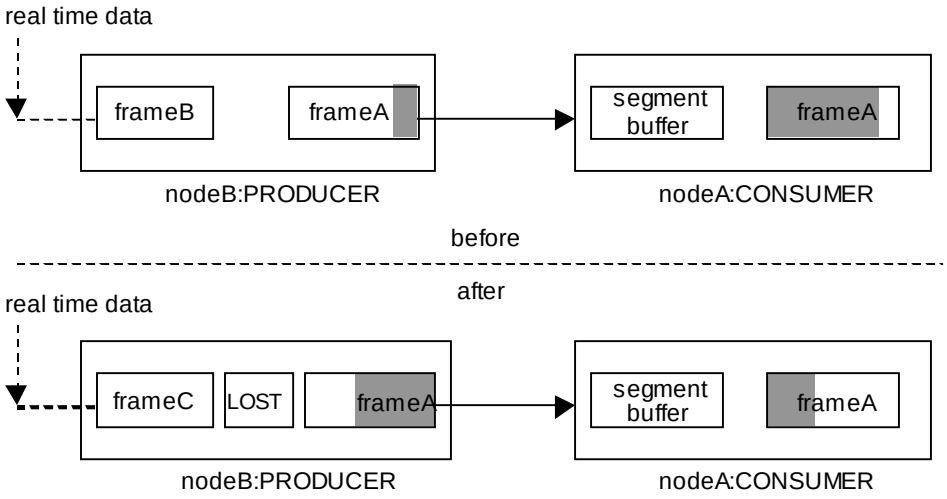

Figure 1.8 – Discarding of stream-type data at the producer

When one or more frames (such as frameB and frameC) are dropped in preference for incoming frames, a pseudo-frame (containing zero data bytes and a LOST indication) is inserted in its place. Partial frames are not discarded by the producer, but selected frames (as required) are discarded. The intent is to maintain some useful frames, rather than corrupting significant portions of all frames.

1.10.2 Stream-type consumers

A slow consumer may be incapable of processing stream-type data at the rate of the producer. In this case, the consumer's segment buffer is tossed if it cannot be quickly emptied (typically because a previously accepted frame hasn't been processed), as illustrated in Figure 1.9. Signaling protocols allow the producer to efficiently discard the not-yet-transferred portion of the lost frame.

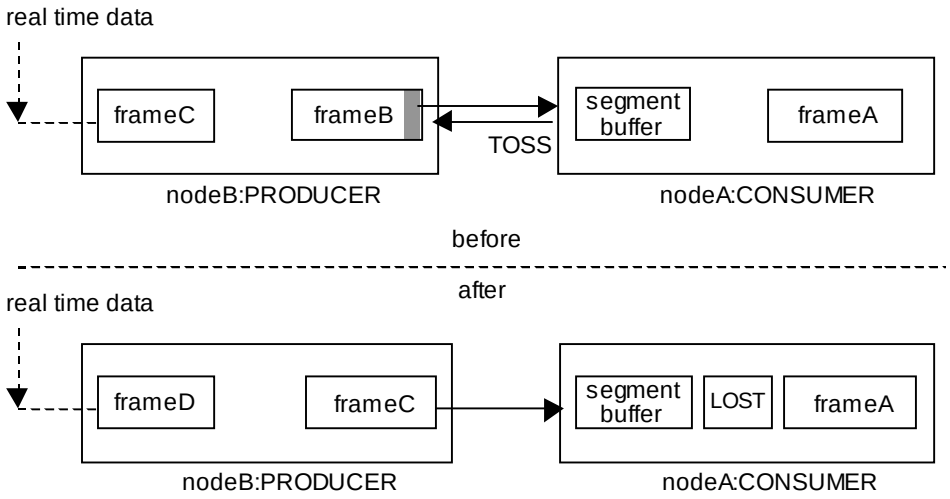

Figure 1.9 – Discarding of stream-type data at the consumer

In this case, the consumer inserts the pseudo-frame (containing zero data bytes and a LOST indication) in place of the lost frame.

1.10.3 Stream-type multicasts

In normal operations, the consumer's discarding of stream-type data prevents a low-bandwidth consumer from blocking the progress of multicast data transfers. To recover from consumer failures (a broken consumer or communication failure), the producer of multicast stream-type data may discard the remainder of a frame, if that frame is not quickly accepted by the consumer.

1.11 Streaming operation

Several forms of abnormal frame labels are provided, listed below as well as illustrated in Figure 1.10. In these illustrations, normal data is shaded white, missing data is shaded black, and corrupted data is shaded gray.

1) LAST. The contents of this frame are thought to be correct.
2) LESS. The transmitted frame is correct, but was truncated early.
3) JUNK. The contents of the transmitted frame have been corrupted and should be discarded.
4) LOST. One or more frames were discarded, typically due to the real-time delivery constraints of stream-type transfers.

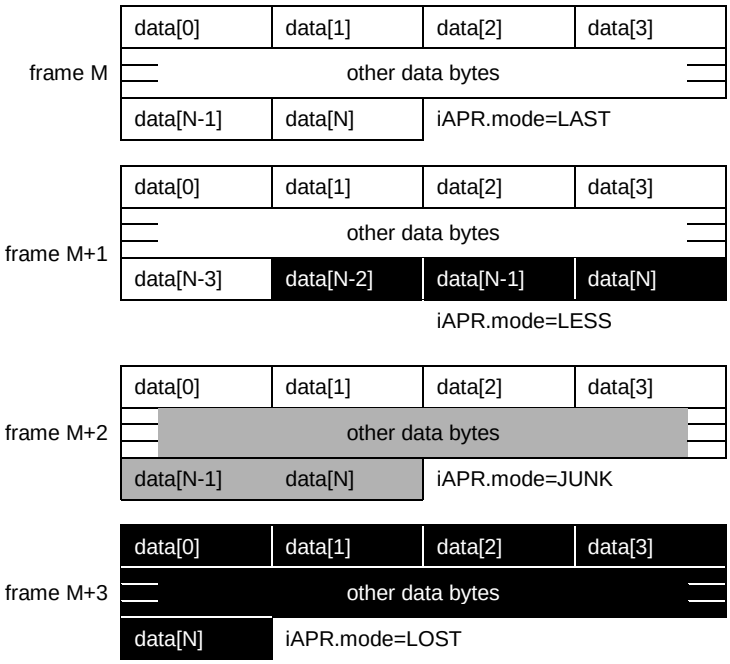

Figure 1.10 – Frame sequences

The state of the frame is identified by bits in the iAPR, which is updated once during each segment and after the final portion of each frame is transferred.

A LAST indication is normally used to complete the frame; the frame contained valid data and the data was correctly sized.

A LESS indication indicates the frame contained valid but an insufficient amount of data. A multicast stream-type producer could provide a LESS indication to a slow consumer (stopping its data transfer), if the other consumers are ready to consume additional data.

A JUNK indication indicates the frame contained invalid data; the size of this frame may or may not be correct.

A LOST indication indicates that one or more frames have been lost. Since no data is available, this indication is affiliated with a zero-length frame transfer.

1.12 Heartbeat indications

The heartbeat protocol allows one of the connected applications to delay an expected transaction, either a control register access or segment buffer access. Heartbeats involve conjugated updates of the *iAPR* and *oAPR* and any other transactions are blocked until a heartbeat handshake is completed.

The heartbeat mechanism may be used when a producer or consumer is unable to generate the transactions normally expected by the protocol. These heartbeat transactions allow the nodes to maintain knowledge of the operational state of the connection.

A heartbeat is initiated by complementing the *hb* (heartbeat) bit value in the flow-control register of the initiating port. The receiving port is responsible for updating the *hb* (heartbeat) bit in the initiating port, to

reflect its most recently observed *hb*-bit value. The same protocol is used by the producer and consumer nodes. See section **5.6** for details.

1.13 Suspended consumer ports

When the application at the consumer plug becomes inactive, this condition can be signaled to the producer plug. Returning this application-inactive status eliminates the sending of unnecessary frames while the application-inactive condition persists.

When the application of the consumer plug becomes active, this information is also communicated to the producer plug. This allows the communication to continue, at the next available boundary. See section **5.4** for details.

1.14 Asynchronous Connections management commands

Asynchronous connections are expected to be established using AV/C commands, although other command mechanisms can be used. To minimize the dependency of asynchronous connections on AV/C commands, AV/C management commands are defined in companion documents. With this partitioning, the asynchronous connection transport protocols can remain stable when the AV/C commands are extended to deal with future bus bridge architectures.

As background, the expected operation of the basic management commands is described in this subsection. The connection commands are provided by the controller, and are sent to the producer and consumer nodes. For simplicity, the controller, producer, and consumer are assumed to be on the same bus. The controller may be an independent node, or may be collocated on the producer or consumer nodes.

1.14.1 Connecting asynchronous plugs

An asynchronous connection is established by a controller, which sends an ALLOCATE command to the consumer node, as illustrated by Figure 1.11. To maintain consistency, the consumer plug shall reject other accesses between the processing of the initial ALLOCATE and the final ATTACH commands.

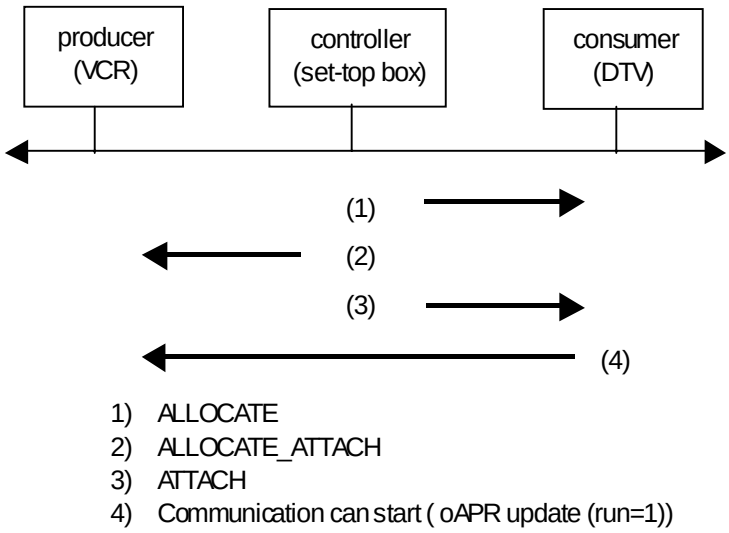

Figure 1.11 – Connecting asynchronous plugs

The ALLOCATE command (1) allocates the consumer plug resources and returns the consumer plug address to the controller.

The following ALLOCATE_ATTACH command (2) allocates the producer plug resources, initializes the producer plug with the address of the connected consumer plug, and returns the producer plug address to the controller.

The final ATTACH command initializes the previously allocated consumer plug, with the address of the connected producer, and transitions the consumer plug to an active operational state.

The producer remains inactive until its *oAPR.run* bit is set to one by a CompareAndSwap update from the consumer.

1.14.2 Disconnecting asynchronous plugs

The disconnection of asynchronous plugs is initiated by a controller, which sends a DETACH command to the consumer node, as illustrated by Figure 1.12.

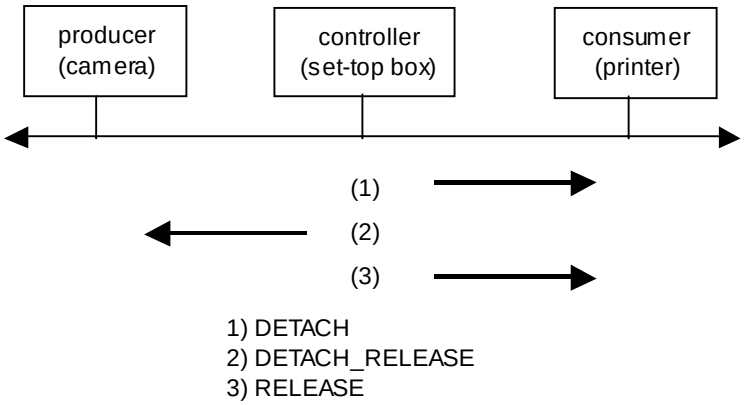

Figure 1.12 – Disconnecting asynchronous plugs

The DETACH command (1) leaves the consumer-plug resource in an inactive state. While inactive, the consumer-plug resource accepts register updates and segment buffer writes from the producer, but inhibits the generation of producer-port updates.

The following DETACH_RELEASE command (2) disconnects and releases the producer port. The final RELEASE command (3) releases the currently inactive consumer plug resource.

1.14.3 Resuming asynchronous communications

After a bus reset, the controller is responsible for resuming communications between previously connected plugs. Distinct RESTORE_PORT commands provide the producer and consumer with the (possibly changed) node ID addresses of each other, as illustrated by Figure 1.13. The producer remains inactive until its *oAPR.run* bit is set to one by a CompareAndSwap update from the consumer.

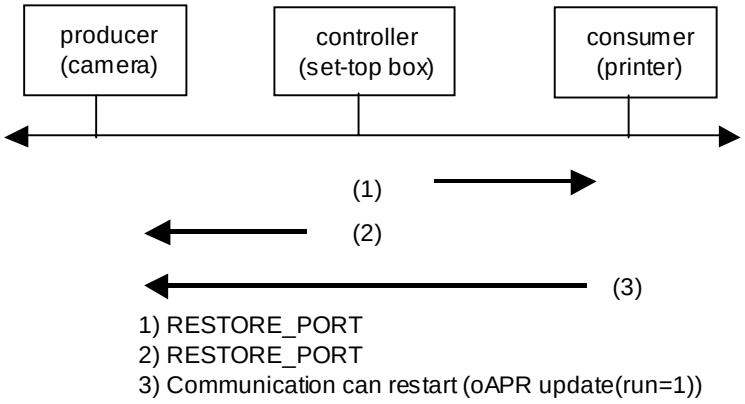

Figure 1.13 – Resuming asynchronous communications

After the bus reset, previously connected plugs remain in a distinct RESTORE-expected state for a short (10 second) interval. If the expected RESTORE_PORT command is not received within this interval, the plug resource is released. Since the controller must actively participate in resuming communications, this post-bus-reset recovery protocol has the desirable side effect of releasing no-longer-managed plug connections.

1.15 Bus transaction errors

Asynchronous connections are robust, in that they can recover from a nearly arbitrary number of packet-transmission failures. The communication protocols are designed to be idempotent (two updates have the same effects as one), so that segment buffer writes and control-register updates can be safely retried after an error is detected. See section **5.7** for details.

1.16 Options

To support low-cost and higher-performance designs, several basic interoperable options are supported, as follows:

1) Segment buffer options. The simplest consumer port design provides a fixed-sized segment buffer. If the producer also supports the option, the consumer port can be allowed to change its segment buffer size (when more resources become available), as described in 4.2.5 and summarized below:
   a) Segment change. The segment buffer size changes between segments.
2) Consumer-port options. The simplest consumer port design assumes that writes are delivered in a fixed sequential order and very small 64-byte-sizes segment buffers are allowed. As options, the consumer port can support the following, as described in 4.2.4:
   a) Concurrent writes. Concurrent writes that may arrive in a non-sequential order are allowed.
   b) Multicast capable. A sufficient larger (integer multiple of 2k bytes) segment buffer allows the consumer port to be connected to a multicast enabled producer.

2. References

The following standards contain provisions that, through reference in this document, constitute provisions of this standard. All the standards listed are normative references. Informative references are given in Annex A. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

[R1] IEEE Std 1394-1995, Standard for a High Performance Serial Bus.

[R2] IEC 61883-1, Consumer audio/video equipment – Digital interface – Part 1: General.

[R3] ISO/IEC 13213: 1994 [ANSI/IEEE Std 1212, 1994 Edition], Information Technology— Microprocessor systems— Control and Status Register (CSR) Architecture for Microcomputer Buses.

[R4] AV/C Digital Interface Command Set General Specification, Version 3.0. TA document number 1998003.

[R5] 1394 Open Host Controller Interface Specification (release 1.00). October 20, 1997.

[R6] AV/C Compatible Asynchronous Serial Bus Connections, Version 1.0. TA Document number 1998016.

[R7] AV/C management commands for management of Asynchronous Serial Bus Connections. Version 1.0, TA Document number 1998011.

[R8] Editorial corrections to the Asynchronous Serial Bus Connections, Version 1.0. TA Document number 1999010.

[R9] AV/C management commands for management of Asynchronous Serial Bus Connections. Version 1.1. TA Document number 2000006.

[R10] AV/C Command for Management of Enhanced Asynchronous Serial Bus Connections. Version 1.0. TA Document number 1999037.

[R11] IEEE Std 1394a-2000, IEEE Standard for a High Performance Serial Bus -- Amendment 1

[R12] Connection and Compatibility Management Specification 1.0. TA Document number 1999031.

[R13] AV/C Compatible Asynchronous Serial Bus Connections, Version 2.0. TA Document number 2000005.

3. Definitions

3.1 Document definitions

3.1.1 Conformance levels

Several key words are used to differentiate between different levels of requirements and options, as follows:

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

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

3.1.1.3 shall: A key word indicating a mandatory requirement. Designers are *required* to implement all such mandatory requirements to ensure interoperability with SCI and VME.

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

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

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

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

3.1.2 Glossary of terms

Many bus and interconnect-related technical terms are used in this document. These terms are described below:

3.1.2.1 asynchronous connection: A logical point-to-point communication path established between producer and consumer nodes, that supports robust high-bandwidth, flow-controlled transfers of one or more data frames.

3.1.2.2 asynchronous connection consumer (abbreviated as **consumer**): The component of a node that consumes data frames provided by the asynchronous connection producer.

3.1.2.3 asynchronous connection producer (abbreviated as **producer**): The component of a node that produces data frames for consumption by the asynchronous connection consumer.

3.1.2.4 basic segment buffer: A segment buffer located in memory immediately following the plug registers.

3.1.2.5 byte: Eight bits of data, used as a synonym for octet.

3.1.2.6 connection: The attachment of a producer plug to a consumer plug for the purpose of sending an asynchronous stream of data frames.

3.1.2.7 consumer: (see asynchronous connection consumer).

3.1.2.8 CompareSwap4: A bus transaction that conditionally stores a *next* value to a specified address and returns the previous data value from that address. The store occurs when the addressed memory value and a second *test* value are equal. In the CSR Architecture, this is called a 4-byte compare_swap transaction.

3.1.2.9 compound plug: A collection of plugs that can be simultaneously connected to matching set of plugs using one sequence of connection-establishment commands.

3.1.2.10 consumer port: A port that is the sink of data frames and is flow controlled by updates of its externally visible *iAPR* control register.

3.1.2.11 CSR Architecture: A abbreviation of the following reference (see clause 2): ISO/IEC 13213 : 1994 [ANSI/IEEE Std 1212, 1994 Edition], Information Technology—Microprocessor systems—Control and Status Register (CSR) Architecture for Microcomputer Buses.

3.1.2.12 data frame (abbreviated as **frame**): A contiguous group of data bytes sent between producer and consumer.

3.1.2.13 data segment (abbreviated as **segment**): A largest portion of a data frame that can be written into the segment buffer before updating the consumer's *iAPR*.

3.1.2.14 dual-duplex connection: A form of asynchronous connection that supports bi-directional data-frame transfers. Request frames are sent from producer plug to the consumer plug and response frames are returned from the consumer plug to the producer plug. Two bi-directional data paths are provided, so that control and data-transfer frames can be processed independently.

3.1.2.15 enhanced segment buffer: A segment buffer that is further described by a segment buffer descriptor. While only one enhanced segment buffer type (a contiguous block of memory located at any address) is supported by this specification, additional types may be defined in future versions of this specification.

3.1.2.16 file-type transfers: An asynchronous connection data transfer that has no real-time delivery constraints that could force discarding of selected frames (as distinguished from stream-type transfers).

3.1.2.17 frame: (see data frame).

3.1.2.18 input Asynchronous Port Register (abbreviated as ***iAPR***): A consumer-resident register affiliated with a consumer port, that is updated by the producer to indicate how much of data has been written to the segment buffer. This register also has other bits that are used for demarcation of variable-length frames, and to support the connection disconnect sequence.

3.1.2.19 output Asynchronous Port Register (abbreviated as ***oAPR***): A producer-resident register affiliated with a producer port on a plug, that is updated by the consumer to indicate how much data can be safely written by the producer. This register also has other bits that are used for demarcation of variable-length frames, and to support the connection disconnect sequence.

3.1.2.20 payload: The portion of a request or response packet that contains data defined by an application layer.

3.1.2.21 plug: A collection of externally visible components (called ports) that can be connected to a subunit for the purposes of sending sequences of variable-length frames.

3.1.2.22 port: A subcomponent of a plug that supports unidirectional data transfers.

3.1.2.23 producer: (see asynchronous connection producer).

3.1.2.24 producer port: A port that is the source of data frames and is flow controlled by updates of its externally visible *oAPR* control register.

3.1.2.25 quadlet: Four bytes of data.

3.1.2.26 segment: (see data segment).

3.1.2.27 segment buffer: An externally visible address space on a consumer into which data is written by the connected producer.

3.1.2.28 segment buffer descriptor: A block of information used to specify a segment buffer scheme in use and its associated buffer space.

3.1.2.29 stream-type transfers. An asynchronous connection data transfer that has real-time delivery constraints, where these delivery constraints can force discarding of selected frames (as distinguished from file-type transfers).

3.2 Numerical notation

Decimal, hexadecimal, and binary numbers are used within this document. For clarity, decimal numbers are generally used to represent counts, hexadecimal numbers are used to represent addresses, and binary numbers are used to describe bit patterns within binary fields.

Decimal numbers are represented in their usual 0, 1, 2, … format. Hexadecimal numbers are represented by a string of one or more hexadecimal (0-9,A-F) digits followed by the subscript 16, except in C++ code contexts, where they are written as 0x123EF2 etc. Binary numbers are represented by a string of one or more binary (0,1) digits, followed by the subscript 2. Thus the decimal number “26” may also be represented as “$1A_{16}$” or “11010_{2}”.

3.3 C++ code notation

The conditions and actions of the state machines are formally defined by C++ code. Since many C++ code operators are non-obvious to the casual reader, their meanings are summarized in Table 3.1.

Table 3.1 – Specific expression summary

Expression	Description
`~I`	Bitwise complement of integer I
`++I`	Pre-increment of integer I (I is incremented, then used in the expression)
`--I`	Pre-decrement of integer I (I is decremented, then used in the expression)
`I ^ J`	Bitwise XOR of integers I and J
`I & J`	Bitwise AND of integer values I and J
`I | J`	Bitwise OR of integer values I and J
`I << J`	Value of I, shifted left by J bits, zero fill
`I >> J`	Value of I, shifted right by J bits, zero fill if I is an unsigned number, sign extension if I is signed
`I == J`	Equality test, true if I is equal to J
`I != J`	Inequality test, true if I is not equal to J
`!B`	Logical negation of boolean variable B
`A && B`	Logical AND of boolean values A and B
`A || B`	Logical OR of boolean values A and B

3.4 State machine notation

All state machines in this standard use the style shown in Figure 3.1. This is similar to the notation used in the Serial Bus standard, with modifications to more compactly and consistently illustrates state transition actions. To illustrative the functionality of transition-destination labels, the equivalent state machines without transition-destination labels is also illustrated in Figure 3.2. Labels of the form Sxx are state transition labels that signify a destination state. Labels of the form Tnn signify a transition to a tag that performs an action prior to entering the state to which it is attached.

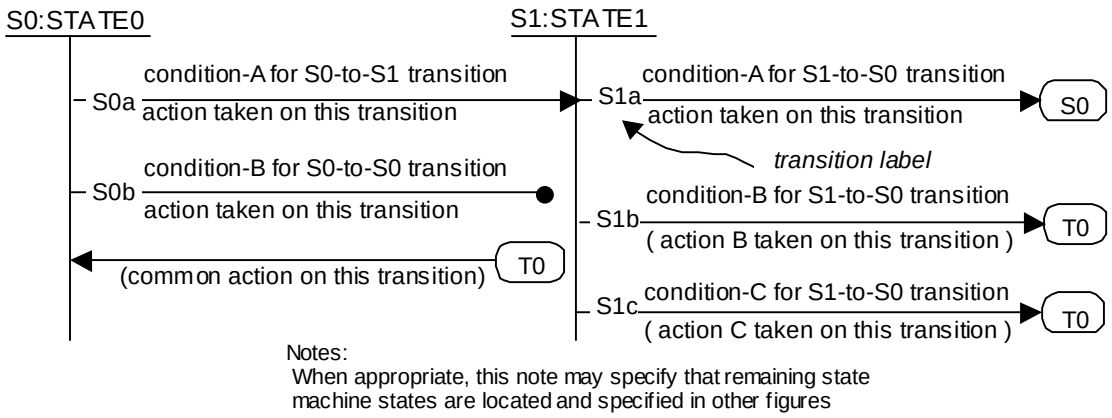

Figure 3.1 – State machine notation

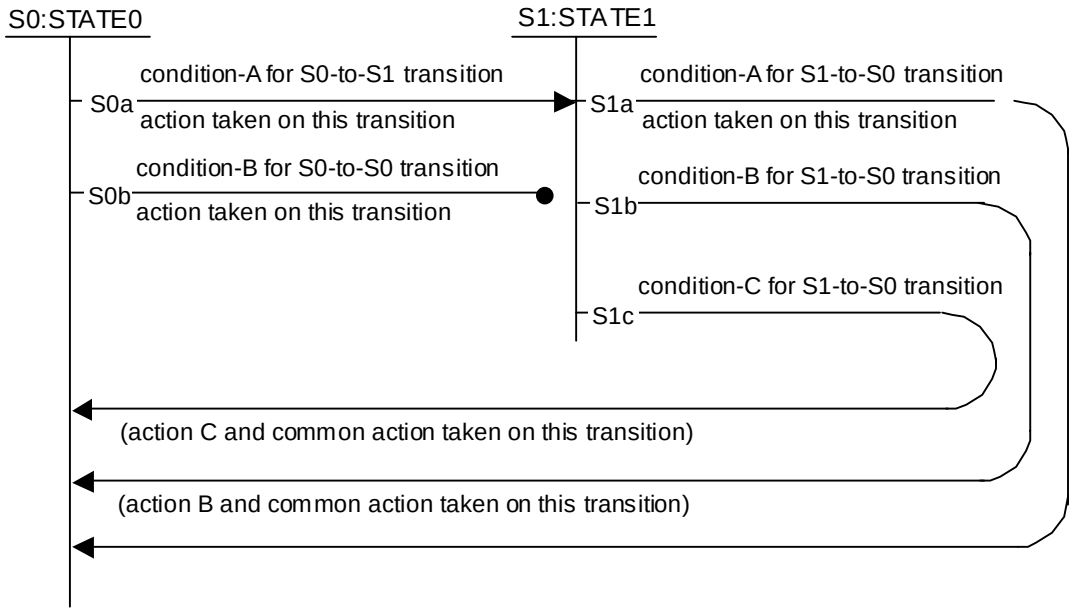

Figure 3.2 – Equivalent state machine

3.5 Unimplemented locations

The capabilities of reserved, ignored, and unused values are exactly defined to minimize conflicts between current implementations and future definitions. This notation shall apply to all fields unless otherwise noted.

3.5.1 Reserved locations

Some locations or portions of locations are not implemented and are defined as *reserved* (abbreviated as *r*). When a reserved value is written, a zero values shall be provided; when read, the returned value shall be ignored.

3.5.2 Ignored locations

Selected locations or portions of locations are partially implemented and are defined as *ignored* (abbreviated as *ign* or *i*). An ignored value has an affiliated storage element, however the value in this storage element has no side effect. When an *ignored* value is written, a zero value shall be provided; when read, the returned value shall be ignored.

3.5.3 Unused locations

Selected locations or portions of locations may be not implemented or partially implemented and are defined as *unused* (abbreviated as *un* or *u*). For *unused* locations, the selection between *reserved* and *ignored* behaviors is implementation dependent.

4. Plug resources

4.1 Asynchronous plug structures

4.1.1 Basic plug structures

A basic asynchronous connection efficiently streams data between a producer (such as a camera) and a consumer (such as a printer), as illustrated by Figure 4.1. In this example, the camera is the producer of the data and the printer is the consumer. The consumer provides a buffer (called the segment buffer) into which the producer writes its data.

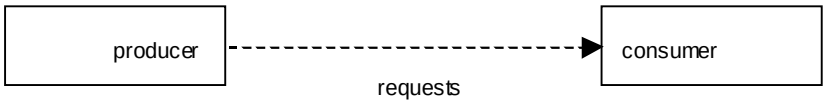

Figure 4.1 – Basic data transfers

4.2 Asynchronous plug components

4.2.1 Asynchronous plug spaces

The IEEE 1394 64-bit address space is partitioned into 2^{16} (64k) nodes and an asynchronous plug occupies a portion of its node's address space, as illustrated by Figure 4.2. An asynchronous plug contains control registers (shaded black) and segment info (shaded gray). In this version of the specification, the segment info is either a segment descriptor or the segment buffer itself. The segment info immediately follows the control registers.

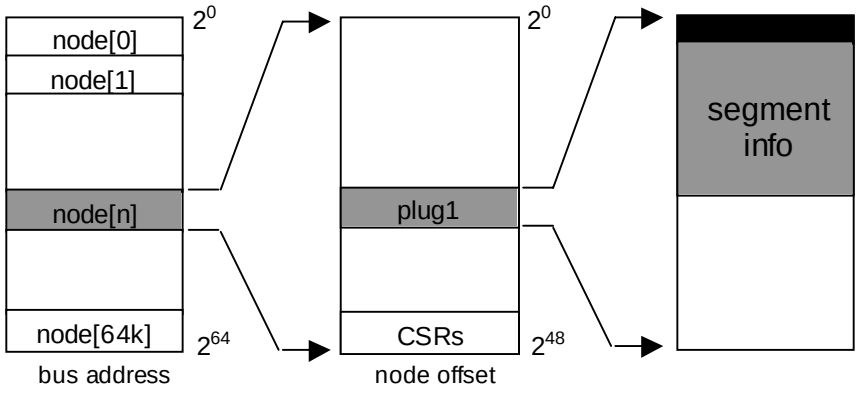

Figure 4.2 – Locations of plug address spaces

4.2.2 Asynchronous plug organization

As noted above and illustrated in Figure 4.3, an asynchronous plug contains control registers (shaded black) and segment info (shaded gray). The plug's control registers, which correspond to distinct port resources, are located at the lowest address for the plug (starting at *plugAddress*) and the segment info is located at a fixed 64-byte offset from the start of the plug. The segment info may be either a segment buffer or a segment buffer descriptor, which is information that describes the segment buffer.

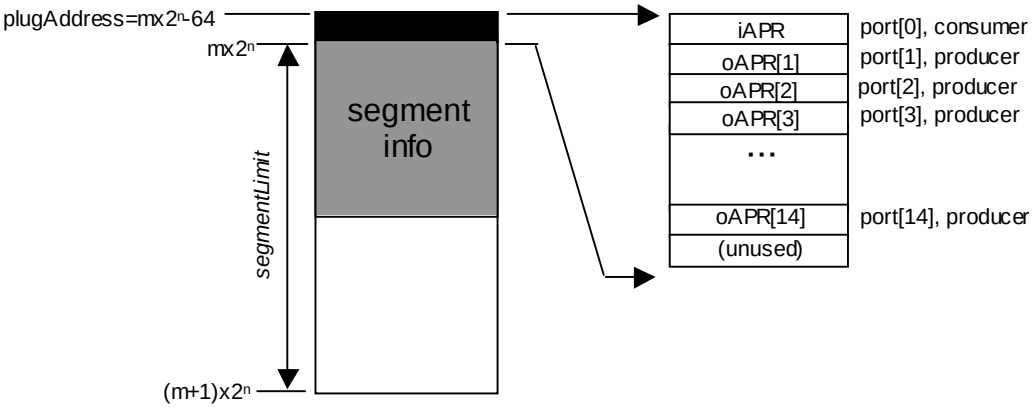

Figure 4.3 – Plug address space components

If the segment info is a segment buffer, the following rules applies: The size of the segment buffer's address space is called the *segment limit*. The segment limit is expressed as 2^n in Figure 4.3, where *n* is an integer greater than or equal to 6. The starting address of the segment buffer shall be a multiple of the segment limit and expressed as $m \times 2^n$, where *m* is an integer greater than 0. The size of the segment buffer shall be multiple of the size specified by the *maxLoad* value (see section 4.3.3) and no less than 64 bytes and no more than the segment limit.

If the segment info is a segment buffer descriptor, the information is as described in section 4.2.6.2.

For simplicity, the plug-offset addresses of ports never change, although their presence or absence depends on the type of connection that is opened. For example, a simple multicast-by-2 producer that has two *oAPR* and a simple consumer that has one *iAPR* is illustrated in Figure 4.4.

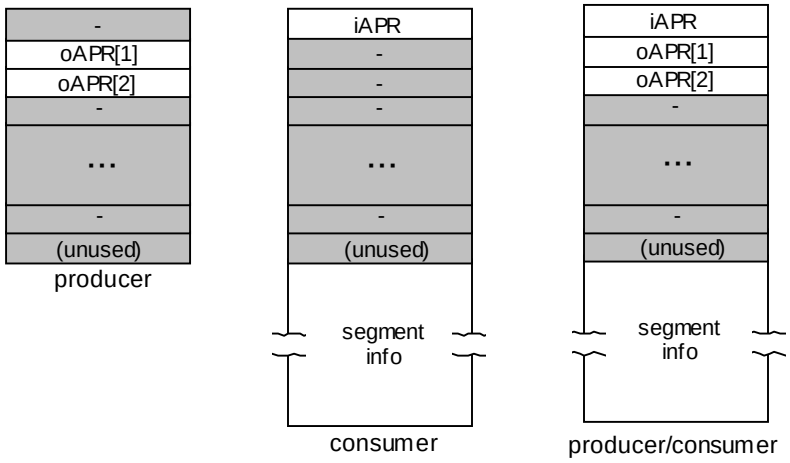

Figure 4.4 – Producer and consumer plug components

The plug's resources are specified by their offset addresses from the plug's address, as listed in Table 4.1.

Table 4.1 – Plug resource addresses

Address offset	Description	Use
0	consumer port	iAPR
4	producer port[1]	oAPR[1]
8, 12, …, 52	producer port[2]-to-port[13]	oAPR[2,...,13]
56	producer port[14]	oAPR[14]
60	reserved	not used
64	segment info	data segment writes (segment buffer) or data segment info reads (segment buffer descriptor)

4.2.3 Asynchronous plug components

A plug resource is directly mapped to a contiguous range of Serial Bus addresses.

The single 32-bit ***iAPR*** (that is written by the producer) is provided for flow-control purposes. See section 4.3.2 for details.

The fourteen 32-bit ***oAPR[n]*** (that are written by the consumers) are provided for flow-control purposes. See section 4.3.3 for details.

The segment info is either a segment buffer descriptor or a segment buffer. See section 4.2.6 for details. If the segment info is a segment buffer, then it shall be a multiple of the size specified by the *maxLoad* value (see section 4.3.3) and be minimally 64 bytes in size. The address space allocated to the segment buffer may be larger than the actual segment buffer size. For example, a 192-kbyte physical buffer could be mapped into the initial portion of 256-kbyte-aligned segment buffer address.

4.2.4 Consumer port address

A consumer port address (iAPR) is a 64-byte aligned 64-bit Serial Bus address, as illustrated by Figure 4.5. When passed in a management command or response, the following address information is used, with the two least-significant bits of the address specifying the options that are supported by the addressed port.

Figure 4.5 – Consumer-port address format

The *reserved* field is reserved and may be used to specify optional behaviors of the consumer port in future revisions of this standard.

The ***m*** (multicast) bit indicates whether this consumer can support multicast connections. If *m*=1, the segment buffer shall be an integer multiple of 2048 bytes (2k bytes), a basic requirement for support of multicast-enabled producers.

The 1-bit ***ct*** bit shall be zero if the segment buffer writes are required to be sent in a sequential order; otherwise, this value should be one.

When the ***ct*** bit==0, there are restrictions on the use of write requests to the consumer. First, the producer shall not issue another write request until the prior write transaction is complete. That is, the prior write request should have generated (1) ack_complete, or (2) an ack_pending followed by a write response. Furthermore, the addresses of write requests shall be sequential: the destination address of any request shall be the destination address of the preceding request plus the data length of the previous request, except for the first request**.**

Some segment buffer designs (such as those on the OHCI processor interface) may map incoming writes into memory. Such interfaces can support more efficient data transfers, because the producer has the option of generating concurrent writes, i.e. more than one write request can be sent before previous write responses have returned. Concurrent writes cannot be safely generated if the segment buffer mandates a sequential ordering, since the relative ordering of requests may change as they pass through the interconnect.

4.2.5 Producer port addresses

A producer-port address (oAPR[n]), which provides access to the port's externally visible flow-control register, is minimally 4-byte aligned. When passed in a management command, the following address information is used, with the two least-significant bits of the address specifying the options that are supported by the addressed port, as illustrated by Figure 4.6.

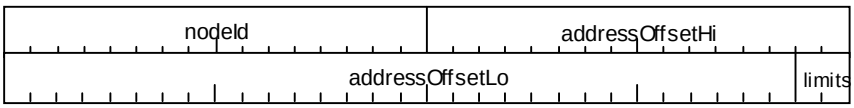

Figure 4.6 – Producer-port address format

The 2-bit ***limits*** (*oAPR* constraints) field specifies constraints on the use of *oAPR.count* values, which specifies the effective segment buffer size, as specified by Table 4.2.

Table 4.2 – Producer port's address.limits field

Value	Name	Description
0	FIXED	Segment buffer size shall never change
1	FRAME	Segment buffer size may change at the next frame boundary
2	SEGMENT	Segment buffer size may change at the next segment boundary
3	—	Reserved

4.2.6 Segment info

Segment info may be either the segment buffer itself or a segment buffer descriptor. When the segment info is the segment buffer itself, this scheme is referred to as a *basic segment buffer*. When the segment info is a segment buffer descriptor, this scheme is referred to as an *enhanced segment buffer*.

4.2.6.1 Segment buffer address

An asynchronous segment buffer address is minimally 64-byte aligned, as illustrated by Figure 4.7. Multicast capable plugs are required to have larger 2-kbyte aligned segment buffers. The 11 least significant bits of the segment buffer address (shown as a shaded box) are also zero on multicast-capable consumer plugs. The segment buffer address for the basic segment buffer is derived from the iAPR or oAPR[n] by using the definition in Table 4.1.

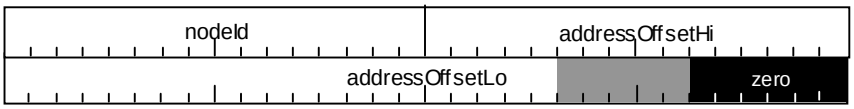

Figure 4.7 – Segment buffer address format

The size of the segment buffer shall be a multiple of 2^n and its address shall be aligned to its size, although the consumer's *oAPR.count* value may restrict the producer to use less than the full segment buffer size.

4.2.6.2 Segment buffer descriptor

The segment buffer descriptor contains the type of the segment and further type-dependent information, which defines how the buffer is accessed. The segment buffer descriptor has the format shown in Figure 4.8.

Address offset	Contents			
00_{16}	segment type	segment sub-type		
04_{16}	seg_info[0]	seg_info[1]	seg_info[2]	seg_info[3]
:	:			
:	:			seg_info[n]

Figure 4.8 – Segment buffer descriptor

The segment type field in the segment buffer descriptor is a bit field with one bit assigned to each specified segment type, as shown in Table 4.3.

Table 4.3 – Segment type field values

segment type	Description
01_{16}	contiguous segment buffer
02_{16}	reserved
:	
80_{16}	

The meaning of the *segment sub-type* and *seg_info* fields is dependent on the segment type.

This version of the specification defines one segment type, the contiguous segment buffer. Future versions of this specification may specify additional segment types.

4.2.6.3 Contiguous segment buffer

For a segment type of 01_{16} (contiguous segment buffer), the *segment sub-type* field is always zero and *seg_info[0]* to *seg_info[5]* contains the start address of the segment buffer, as shown in Table 4.4.

Table 4.4 – Segment info for contiguous segment buffer

seg_info field	meaning
seg_info[0]	Segment offset (48 bit address of the start of the segment buffer)
seg_info[1]	
seg_info[2]	
seg_info[3]	
seg_info[4]	
seg_info[5]	

The contiguous segment buffer is located in the initial portion of its address space. The size of this address space is referred to as the *segment limit*. This segment limit is the same as the segment limit expressed in Figure 4.3. The starting address of the address space shall be a multiple of the segment limit and expressed as $m * 2^n$, where m is an integer greater than zero. The size of the segment buffer shall be a multiple of the size specified by the *maxLoad* value (see section 4.3.3) and no less than 64 bytes and no more than the segment limit.

4.2.6.4 Segment buffer type negotiation

The segment buffer type is negotiated when the asynchronous connection is set-up. See section 1.14.1 for details on how a connection is set-up. The following describes how the segment buffer type is negotiated:

1) The controller sends the ALLOCATE command to the consumer. The consumer will include in its response a segment buffer type value. That value should be a segment buffer type value that indicates the segment buffer type(s) that it supports. However, it may be the value zero in order to force usage of the basic segment buffer scheme.
2) The controller sends the ALLOCATE_ATTACH command to the producer. The segment buffer type value from the consumer is included in the command. The producer will include in its response a segment buffer type response value. That value should represent the segment buffer types supported by both the producer and the consumer. This is generated by performing a bitwise AND between the segment buffer type value from the consumer and the segment buffer type value that represents the segment buffer types supported by the producer. However, the producer may also respond with the value zero in order to force usage of the basic segment buffer scheme.
3) The controller sends the ATTACH command to the consumer. The segment buffer value from the producer is included in the command.
4) The consumer selects a segment buffer scheme based on the segment buffer type value from the ATTACH command. If the segment buffer type value is zero, the consumer will select a basic segment buffer. If the value is not zero, the consumer will select the indicated enhanced segment buffer. If more than one enhanced segment buffer type is indicated, the consumer will choose one by unspecified criteria.
5) The consumer sets up an appropriate segment buffer, either locating it contiguous to the plug registers (basic segment buffer) or locating it elsewhere in memory and creating an appropriate segment buffer descriptor immediately following the plug registers.
6) When the producer selects the segment buffer type value other than zero, the producer reads the segment buffer descriptor to get the segment buffer information after the consumer updated the *oAPR* register (run=1).

4.3 Flow control registers

4.3.1 Register properties

4.3.1.1 Command-reset values

All fields shall be unaffected by a command reset (which is defined in section 8.3.2.1 of reference [R1]).

4.3.1.2 Unimplemented registers

On a consumer plug, the *oAPR* is not implemented. On a producer plug, the *iAPR* is not implemented and one or more of the *oAPRs* are implemented.

When these unimplemented registers are accessed, their behaviors are listed below. This definition is intended to be compatible with existing hardware interfaces, such as OHCI:

1) CompareAndSwap. A 4-byte CompareAndSwap lock transaction shall complete with a type_error status (ack_type_error or resp_type_error).

2) Other transactions. Other transactions are not supported and should return a type_error. For compatibility with existing hardware interfaces (such as OHCI), the writes may update the addressed location and reads may return previously written data.

4.3.1.3 Implemented registers

When implemented registers are accessed, their behaviors are listed below. This definition is intended to be compatible with existing hardware interfaces, such as OHCI:

1) CompareAndSwap. A 4-byte CompareAndSwap lock transaction shall complete with an ack_pending and a resp_complete status.

2) Other transactions. Other transactions are supported as defined in section 4.4.2.

NOTE — The requester is responsible for checking whether the CompareAndSwap operation has been effected, by comparing the returned response-data value with the previously sent request-packet argument value. Distinct completion codes or data-bit values are not provided for this purpose.

4.3.2 iAPR (input Asynchronous Port Register)

4.3.2.1 iAPR format

iAPR is written by the producer and is located on the consumer. This register contains multiple control bits and a flow controlling *count* value, as illustrated in Figure 4.9.

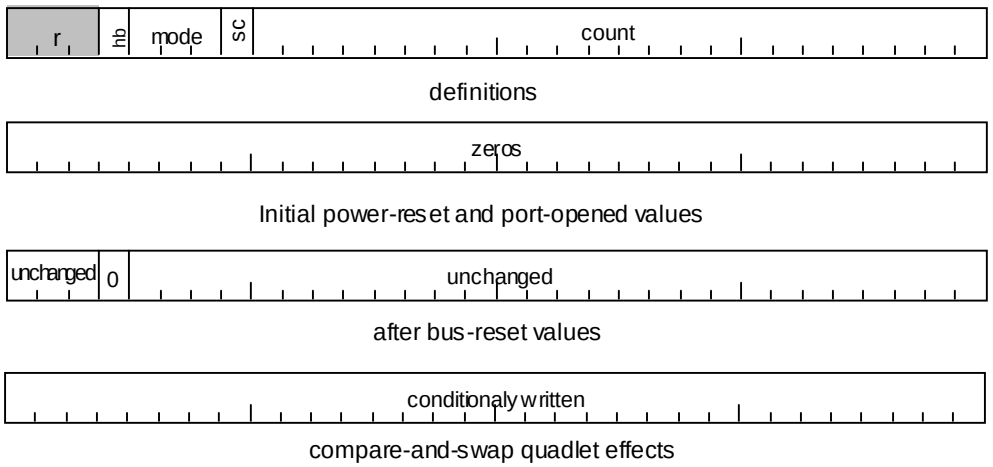

Figure 4.9 – iAPR formats

The ***r*** bits shall be reserved.

The ***hb*** (heartbeat) bit is set to indicate a heartbeat handshake or confirm a heartbeat handshake, as specified in section **5.6**.

The 3-bit ***mode*** field provides frame completion and disconnection state information, as specified in 4.3.2.2.

The ***sc*** (segment count) bit is normally set to the *oAPR.sc* value after each segment has been transferred, to indicate when that segment has been sent.

The 24-bit ***count*** identifies how many of this segment's data bytes have been written by the producer, and is used for flow-control purposes.

4.3.2.2 iAPR.mode values

The 3-bit ***mode*** field provides frame-completion and disconnection-state information, as specified in Table 4.5.

Table 4.5 – iAPR.mode encoded values

mode	Name	Description
0	FREE	Producer initiated disconnection sequence (Initial state)
1	MORE	The frame has not yet ended
2	SUSPENDED	Suspend confirmation; suspended frame transfers
3	—	reserved
4	LAST	A successful frame transfer
5	LESS	A truncated-length frame was transferred
6	JUNK	A corrupted frame was transferred
7	LOST	A zero-length pseudo frame, marking a discarded frame location

The **FREE** code is used for two purposes:

1) This is the initial state of a port, before a connection has been made.
2) This is the code value used to communicate a shutdown request from the producer to the consumer.

Upon receiving this indication, the consumer ceases operation and decommits all resources allocated to the plug.

The **MORE** indication labels the leading (not end-of-frame) segments within a frame.

The **SUSPENDED** indication is a confirmation that the producer is in the suspend mode based on the request from the consumer, as described in section **5.4**. In the suspend mode the producer is not sending any frames, and data may be discarded in the producer.

The **LAST** indication labels the final segment within a successfully transferred frame.

The **LESS** indication labels the last segment in abnormal frame, when the abnormal frame contains valid data but the frame was truncated early.

The **JUNK** indication labels the last segment in an abnormal frame, when the transferred portion of the frame contains corrupted data.

The **LOST** indication labels a zero length frame, marking where one or more frames have been lost.

4.3.3 oAPR (output Asynchronous Port Register)

4.3.3.1 oAPR format

The producer-resident *oAPR* is written by the consumer. Multiple control bits and a flow-controlling *count* value are provided, as illustrated in Figure 4.10. The initial value of this register (after a power reset or when the port enters the free state) shall be zero.

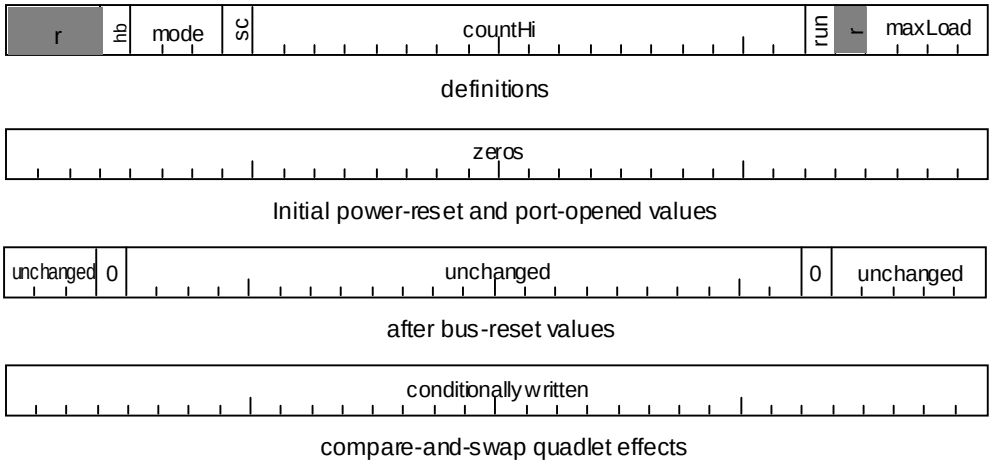

Figure 4.10 – oAPR formats

The ***r*** bits shall be reserved.

The ***hb*** (heartbeat) bit is set to a value that indicates a heartbeat handshake or confirms a heartbeat handshake, as specified in section **5.6**.

The 3-bit ***mode*** field provides segment buffer state information, as specified in section 4.3.3.2.

The ***sc*** (segment count) bit is normally toggled after each segment has been consumed, to distinctively label the sequential segments.

The 18 bit ***countHi*** value shall be concatenated with a 6 bit zero value to generate the 24 bit *count* value.

The *count* value indicates how many bytes can be safely written by the producer and is used to limit the range of the producer's data-write transactions. This value shall be a multiple of the *maxLoad*-specified write-transaction payload size (a minimum payload size is 64 bytes).

On multicast-capable consumers, the effective count value shall be an integer multiple of 2kbytes.

NOTE —The intent is to reduce the complexity of multicast-capable producers, by allowing ports to efficiently transfer relatively large 2-kbyte contiguous blocks before checking for the next *oAPR*-constrained data-transfer boundary.

A zero-valued *run* bit inhibits the producer-port from generating segment buffer writes and control register updates. A one-valued ***run*** bit enables the generation of these writes and control register updates. The initial value of the *run* bit shall be zero and the *run* bit shall be cleared by a bus reset.

NOTE —The *run* bit is intended to delay the producer's generation of write and update transactions until the consumer's state has been properly initialized. At this time, the consumer is expected to update the producer's *oAPR*, setting *oAPR.run* to one.

The 4-bit *maxLoad* field specifies the data-payload size limitations for segment buffer writes, as specified in equation 1. In other words, the amount of data in the write request cannot exceed the *payloadSizeInBytes* value. The *maxLoad* value shall be equal-to or larger-than 5 (which corresponds to 64-byte writes) and shall not exceed the size of the node's ROM-specified *max_rec* value, as defined by IEEE Std 1394-1995.

$$payloadSizeInBytes = 2^{(maxLoad+1)} \quad (1)$$

NOTE —The connection's *maxLoad* field may differ from the node's *max_rec* field, because the size of receive-FIFO queues may depend on where the incoming request is routed.

If the *maxLoad* is set to a size that is larger than supported by the producer, the producer should use the largest of its supported value.

4.3.3.2 oAPR.mode values

The 3-bit ***mode*** field provides segment buffer status information, as specified in Table 4.6.

Table 4.6 – oAPR.mode encoded values

mode	Name	Description
0	FREE	Consumer initiated disconnection sequence (Initial state)
1	—	reserved
2	SUSPEND	Suspend frame transfers
3	—	reserved
4	RESUME	Resume frame transfers
5	SEND	Segment space available, transfer the segment
6	—	reserved
7	TOSS	Next segments will not be used, should be discarded

The **FREE** code is used for two purposes.

1) This is the initial state of a port, before a connection has been made.
2) This is the code value used to communicate a shutdown request from the consumer to the producer.

Upon receiving this indication, the producer ceases operation and decommits all resources allocated to the port, and if there are no remaining active ports, decommits the resources allocated to the plug.

The **SUSPEND** indication means that segments should be discarded (rather than sent) and is asserted when consumer-local connections to the plug have been suspended, as described in section **5.4**.

The **RESUME** indication means that segments may be sent and is asserted when consumer-local connections to the plug have been re-attached, as described in section **5.4**.

The **SEND** indication means that segment buffer space is available, so the next segment can be safely sent.

The **TOSS** indication means that the current frame is being discarded and (for efficiency) the remainder of the frame shall not be written by the producer. The intent is to allow termination of large data transfers when an analysis of the initial data (or a changed device state) determines that the remaining data will not be used and therefore shall not be sent.

4.4 Data-access constraints

4.4.1 Segment buffer accesses

The segment buffer shall respond to read and write, either quadlet or block, requests as follows. This applies to both basic and enhanced segment buffers.

1) Block or quadlet write checks. The block or quadlet write transactions may be checked in several ways, as listed below:
   a) Robust. A write with an expected source_ID value (as specified by the plug's context) shall be accepted; others shall be terminated with a type_error (ack_type_error or resp_type_error).
   b) Minimal. All writes shall be accepted, without source_ID checking.
2) Block write sizes. The size of the normal block write transaction shall be one fixed size, which is the largest power of two, subject to the following constraints:
   a) Speed. The size of the write request shall not exceed the size supported by the effective (S100, S200, ...) data transfer rate.
   b) Consumer limitations. The size of the write request shall not exceed the capabilities of the consumer, as indicated by its *oAPR.maxLoad* field (see section 4.3.3).
   c) Producer limitations. This is one of the producer's supported write request sizes. The size of the final block within each frame does not obey these rules, but is truncated to generate a write transaction with the number of data bytes being sent. In the case that the final block is 4 bytes in size, according to the reference [R1] (Section 7.3.3.1.2), a quadlet write transaction shall be used.
3) Block or quadlet reads. A block or quadlet read transaction may have one of the following two behaviors, listed in order of preference:
   a) A type_error status (ack_type_error or resp_type_error) is returned.
   b) Previously written data is returned.

In some implementations, the segment buffer writes can be serviced by hardware, without microprocessor intervention. On a basic consumer, the segment buffer writes should be done in a monotonically increasing order, so that hardware can stream incoming data into affiliated segment buffer storage. A more sophisticated consumer may allow these writes to be done in any order, which can improve data-transfer efficiencies by allowing the producer to have more than one concurrently outstanding write transaction.

4.4.2 Control register accesses

For the *iAPR* and *oAPR*, only aligned 4-byte CompareAndSwap lock transaction shall be supported. When such implemented registers are accessed, their behaviors are listed below. This definition is intended to be compatible with existing hardware interfaces, such as OHCI:

1) CompareAndSwap. A 4-byte CompareAndSwap lock transactions shall be supported. Other locks. Lock transactions other than CompareAndSwap shall return a type_error status (ack_type_error or resp_type_error).
2) Lock checks. The 4-byte CompareAndSwap lock transactions may be checked in several ways, as listed below.
    a) Robust. A 4-byte CompareAndSwap lock transaction with an expected source_ID value shall be accepted and others shall be terminated with a type_error (ack_type_error or resp_type_error).
    b) Minimal. All 4-byte CompareAndSwap lock transactions shall be accepted without checking source_ID.
3) Illegal request. When there is an unsupported field or bit value in the data field of the request packet, the response may be returned with a response code of resp_data_error.
4) Read/Write. Read and write transactions are not supported and may have either of the following behaviors, listed in order of preference:
    a) A type_error status (ack_type_error or resp_type_error) is returned.
    b) Reads return unspecified data; writes have unspecified effects.

Although the CompareSwap4 transaction can be serviced by hardware, a microprocessor or higher level sequencer shall be signaled when the transaction completes. This signal allows the affiliated segment data and status information to be processed.

4.4.3 Segment buffer descriptor accesses

The segment buffer descriptor shall respond to quadlet read requests only, as follows.

1) Quadlet read checks. The quadlet read transactions may be checked in several ways, as listed below:
    a) Robust. A read with an expected source_ID value (as specified by the plug's context) shall be accepted; others shall be terminated with a type_error (ack_type_error or resp_type_error).
    b) Minimal. All reads shall be accepted, without source_ID checking.
2) Write. Write transactions are not supported and may have either of the following behaviors, listed in order of preference:
    a) A type_error status (ack_type_error or resp_type_error) is returned.
    b) Unspecified effects occur.

NOTE — Although returning a type_error is the most desirable behavior, existing CPU hardware interfaces (such as OHCI) may be unable to return a type_error because read and write transactions are directly mapped to memory locations, rather than processed by microprocessor firmware.

5. Asynchronous communications

5.1 Frame transfer sequences

The flow control for the asynchronous connection can be phrased in terms of the producer and consumer behavior constraints. Frame transfers are decomposed into a sequence of segment transfers, which are normally performed as follows:

1) Space indication. The consumer updates the *oAPR*, to indicate the segment buffer size.
2) Segment writes. The producer writes into the consumer's segment buffer, until the end of the consumer's segment buffer or the end of the producer's frame (whichever comes first) is reached. (The protocols do not rely on these writes being detected by the consumer's flow-control sequencer.)
3) Segment indication. The producer updates the *iAPR*, to inform the consumer when the segment buffer writes have completed. The *iAPR.mode* field communicates the data-transfer status, as follows:
   a) *iAPR.mode*=MORE. One segment of the frame has been transferred; more segments are expected.
   b) *iAPR.mode*=LAST. The final segment of the frame has been transferred successfully.
   c) *iAPR.mode*=LESS. A portion of the frame has been transferred successfully, but the frame was truncated early.
   d) *iAPR.mode*=JUNK. A portion (or all) of the frame has been transferred successfully, but the frame was corrupted and should not be used.
   e) *iAPR.mode*=LOST. One or more frames have been lost.

   For these segment indications, the asserted *iAPR.sc* value shall equal the previously observed *oAPR.sc* value.
4) Space indication. The consumer shall update the *oAPR* to inform the producer of the updated segment buffer status. The *oAPR.sc* bit shall be toggled and the *oAPR.mode* field communicates the segment buffer status, as follows:
   a) *oAPR.mode*=SEND. The segment buffer has been emptied; the next segment may be sent.
   b) *oAPR.mode*=TOSS. The segment buffer is unavailable; the remainder of the frame shall be discarded.
5) Next transfer. The next segment transfer depends on the current *oAPR.mode* value, as follows:
   a) If *oAPR.mode*=SEND in step 4), the next segment can be written by continuing from step (2).
   b) If *oAPR.mode*=TOSS in step 4), the producer's remaining same-frame segments are tossed. Further transfers continue from step (3e).

5.2 Segment transfer constraints

Transfers of a data segment involve writes to the segment buffer and updates of the *oAPR* and *iAPRs'* locations, which are further restricted as follows:

1) Register updates. Updates of the *oAPR* and *iAPR*s shall be performed using the 4-byte compare-and-swap transaction.
2) Segment writes. Segment buffer writes shall be power-of-two, their addresses shall be an integer multiple of their size, and the write shall be no larger than the value specified by the producer-

port's *oAPR.maxLoad* field. The last write within each segment is an exception; its length corresponds to the number of remaining data bytes.

3) Segment sizes. The consumer supplied *oAPR.count* value shall be an integer multiple of the transfer size specified by the *oAPR.maxLoad* field.

5.3 Frame transfer illustrations

Several of these frame-transfer sequences are illustrated in the following subsections. All illustrations assume the same size frames are transferred, but illustrate a range of segment buffer sizes and frame-transfer failure conditions.

5.3.1 Flow control illustration: 256-kbyte consumer buffer

As a simple example, consider the transfer of two frames (one 34k byte and the other 3k byte) into a 256k byte size consumer buffer. For each frame, the data transfer starts with a zero buffer-offset location, and data is transferred to successive buffer-offset addresses, as illustrated in Figure 5.1. In this illustration, the write(arg1, arg2) label corresponds to a segment buffer write: the *arg1* value corresponds to the remote segment buffer address; the *arg2* value corresponds to the producer's data buffer address.

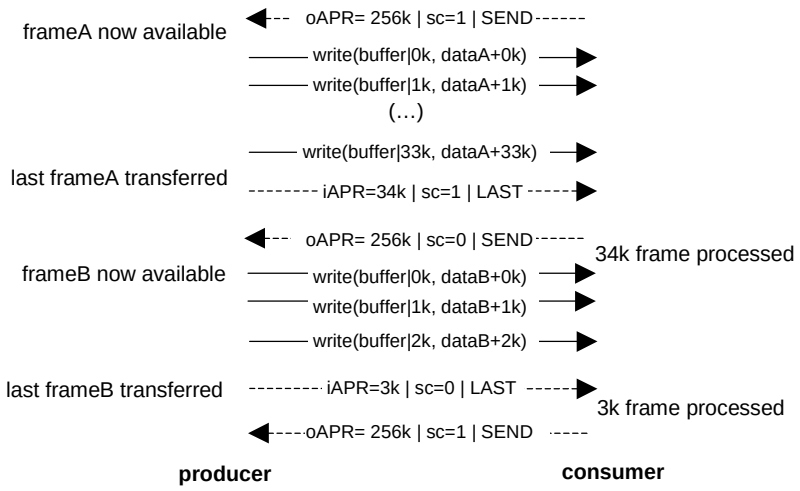

Figure 5.1 – 34k/3k data frame illustration, 256k hardware mapped buffer

A normal buffer-transfer sequence consists of an *oAPR* update, buffer-data writes, and an *iAPR* update. For clarity, the next *oAPR* update, that is logically associated with the following frame transfer, is shown at the end of this transfer sequence.

Signaling of the end-frame condition involves updates of the consumer-resident *iAPR* and the producer-resident *oAPR*s, as follows:

1) Producer indication. After writing the remainder of the frame into the consumer-resident buffer, the producer sets *iAPR.mode* to a nonzero value, informing the consumer of the end-of-frame condition.

2) Consumer confirmation. After servicing its segment buffer, the consumer updates its *oAPR.sc* (segment count) bit to confirm the frame reception. If allowed by the producer port (see 4.2.5), the *oAPR.count* value is simultaneously updated to reflect the next segment properties.

3) Next-frame indications. When completing its next-frame transfers, the producer sets the *iAPR.sc* to equal the returned *oAPR.sc* value. The use of distinct *iAPR.sc* values avoids possible confusion between redundant and successive end-frame indications.

5.3.2 Flow control: fixed 32-kbyte consumer buffer

As another example, consider the transfer of two frames (one 34k byte and the other 3k byte) into a 32k byte consumer buffer. This would typically force additional segment buffer restarts, as illustrated in Figure 5.2. In this illustration, the write(arg1, arg2) label corresponds to a segment buffer write: the *arg1* value corresponds to the remote segment buffer address; the *arg2* value corresponds to the producer's data buffer address.

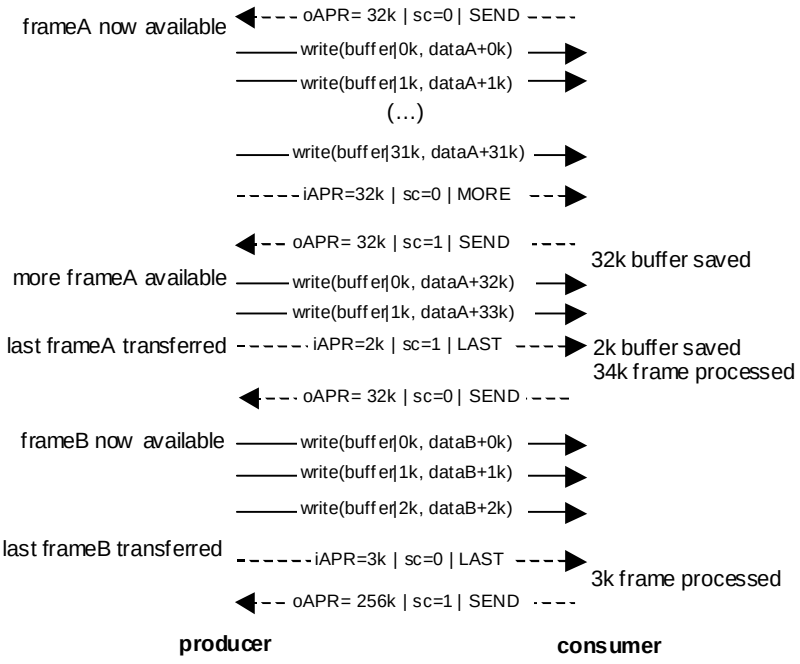

Figure 5.2 – 34k/3k data frame illustration, 32-kbyte consumer buffer

Each buffer-transfer sequence consists of an *oAPR* update, buffer-data writes, and an *iAPR* update. For clarity, the next *oAPR* update, that is logically associated with the following frame transfer, is shown at the end of this transfer sequence.

5.3.3 Flow control: discarding one frame

As another example, consider the transfer of two frames (one 34k byte and the other 3k byte) into a 32k byte segment buffer. In this example, a streaming consumer discards the first frame after the first segment has been received, so the data can be transferred quickly to the other multicast consumers. For clarity, the multiple segment buffer writes are represented by a more compact representation, as illustrated in Figure 5.3.

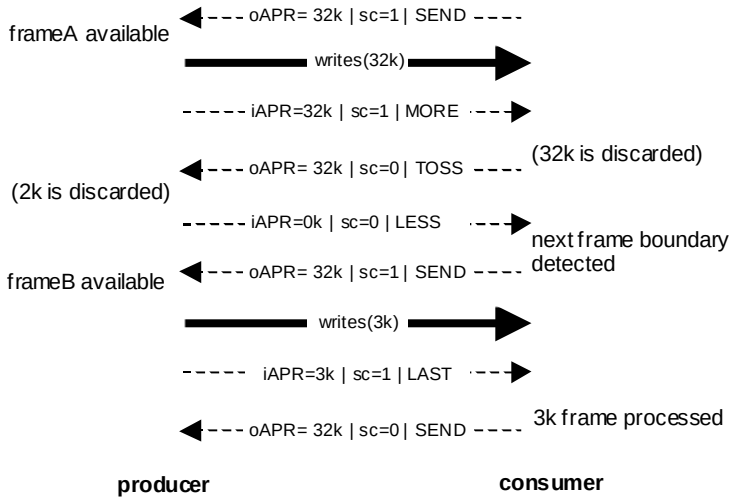

Figure 5.3 – 34k/3k data frame illustration, first frame discarded

The consumer returns a TOSS indication, acknowledging the *iAPR* update but indicating the segment buffer was discarded due to a congested consumer condition. In this example, the producer commits to discarding this frame and provides the consumer with a LESS indication.

5.3.4 Flow control: discarding two frames

For example, a streaming consumer discards the first two frames after one segment of the first frame has been received, so both of these frames can be transferred quickly to the other multicast consumers, as illustrated in Figure 5.4.

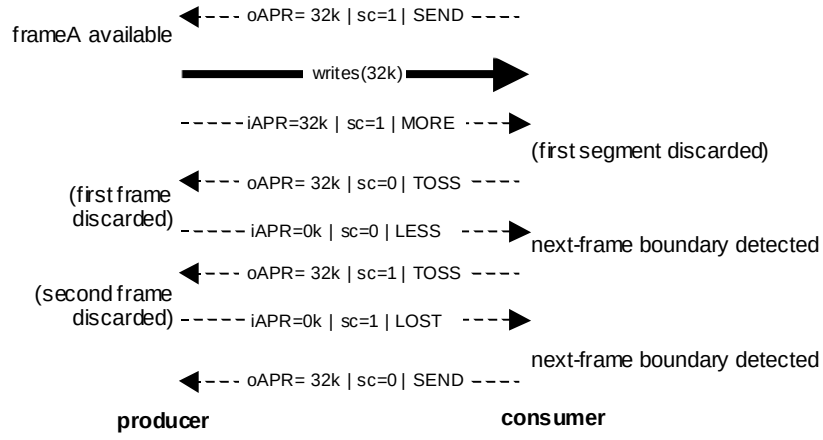

Figure 5.4 – 34k/3k data frame illustration, two frames discarded

The consumer returns a TOSS indication, acknowledging the *iAPR* update but indicating the segment buffer was discarded due to a congested consumer condition. After the producer has provided a confirming LESS indication, the consumer may discard the second segment by providing a TOSS indication.

5.3.5 Flow control: frame truncation

For example, in Figure 5.5, assume that the producer terminates the transfer after only a portion of the frame has been transferred. The producer provides an *iAPR.mode*=LESS indication, so that the truncated frame will not be incorrectly interpreted.

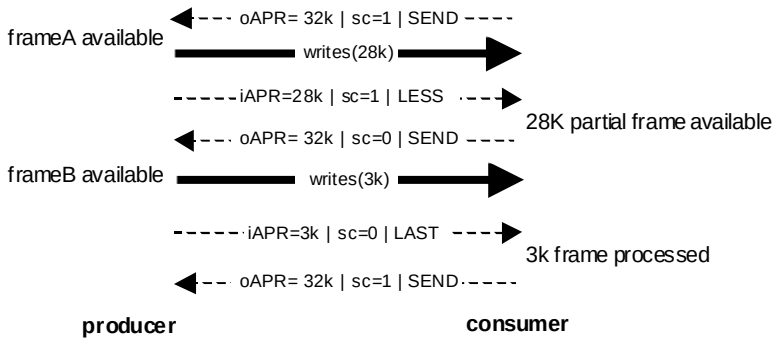

Figure 5.5 – 34k/3k data frame illustration, truncated frame

5.3.6 Flow control: corrupted frame

For example, in Figure 5.6, assume that the producer terminates the transfer after discovering a portion of the previously transferred frame has been corrupted. The producer provides an *iAPR.mode*=JUNK indication, so that the corrupted frame will not be incorrectly interpreted.

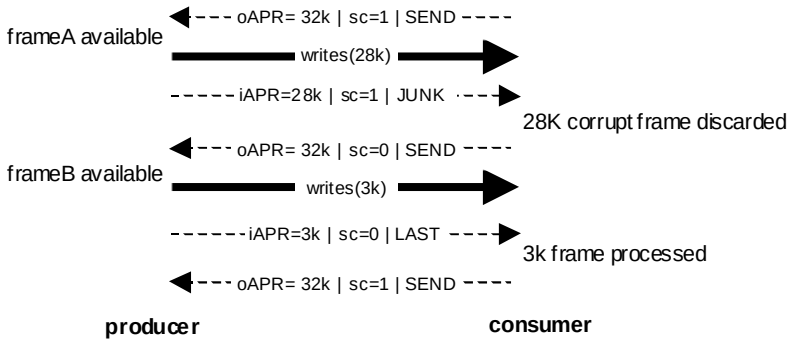

Figure 5.6 – 34k/3k data frame illustration, corrupted frame

5.4 Suspended consumer ports

When the consumer plug is dissociated from its node's resources, the plugs remain connected but the producer is informed that the consumer plug is in a suspended state. The consumer signals that it is suspended (1) by setting *oAPR.mode*=SUSPEND, as illustrated in Figure 5.7. The suspend is confirmed (2) by the producer port by setting the consumer-resident *iAPR.mode* field to SUSPENDED.

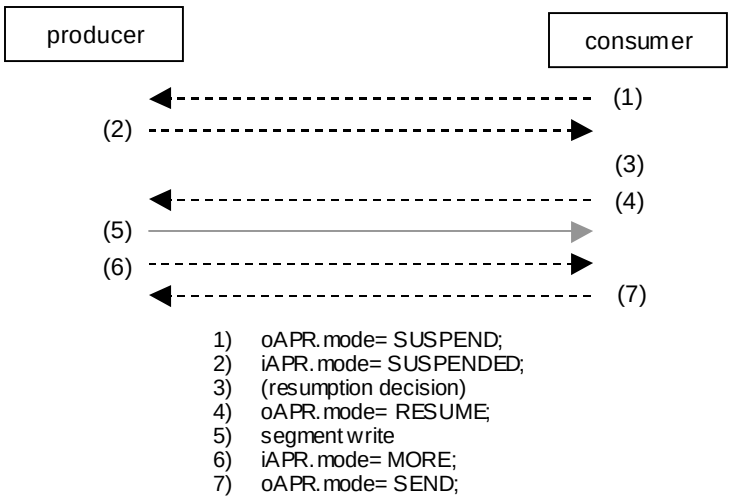

Figure 5.7 – Transient suspend signaling

After the suspend confirmation has been provided, the node with the consumer node can remain idle (3) and not respond to observed producer updates.

When the consumer plug is resumed, the asserted indication (4) of RESUME signals to the producer that data transfer may be resumed. This re-enables the producer and the next frame is transferred in the normal manner, by writing the first segment (5), subsequently setting the consumer-resident *iAPR.mode* indication (5) to MORE (or LAST, LESS, or JUNK). The consumer port then changes *oAPR.mode* from RESUME to SEND (or TOSS), completing the current segment transfer and allowing following frames to be transferred.

5.4.1 Flow control: suspended transfers

Below, in Figure 5.8, is an example of the case when the consumer suspends the connection when the application is no longer active. The consumer provides an *oAPR.mode*=SUSPEND indication to inhibit the sending of additional frames.

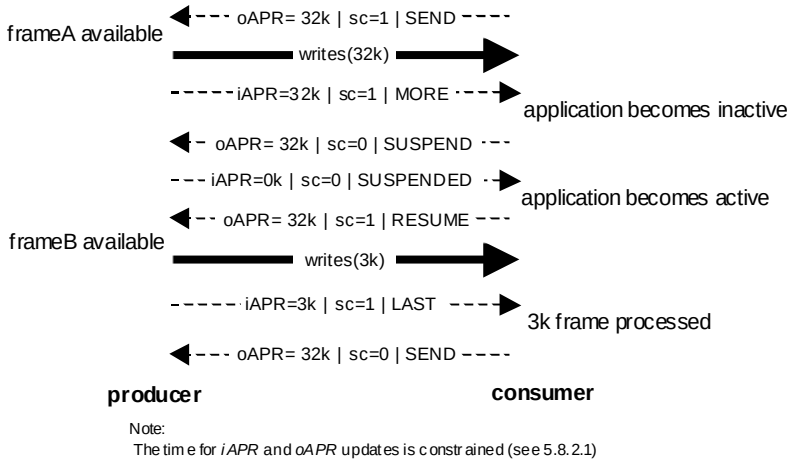

Figure 5.8 – 34k/3k data frame illustration, first frame suspension

When the consumer application becomes active, an *oAPR.mode*=RESUME indication re-activates the producer's data transfers. The next and following frames are transferred in the normal fashion.

5.5 Disconnection indications

The producer or consumer may initiate the disconnection of an asynchronous connection by using an indication to its affiliated plug. The use of this method allows an efficient, controller-free disconnection. The connection is unconditionally terminated, without regard to overlays that may exist at the consumer.

Disconnection may be initiated by either the producer or consumer. Below is an example of a producer-initiated disconnection (see Figure 5.9). In the case of a consumer-initiated disconnection, the signal is sent from the consumer to the producer. The producer releases resources associated with the plug and port and the consumer releases the plugs resources. When a target node has initiated disconnection, it shall perform no further operations on the plug to which the disconnect operation was directed.

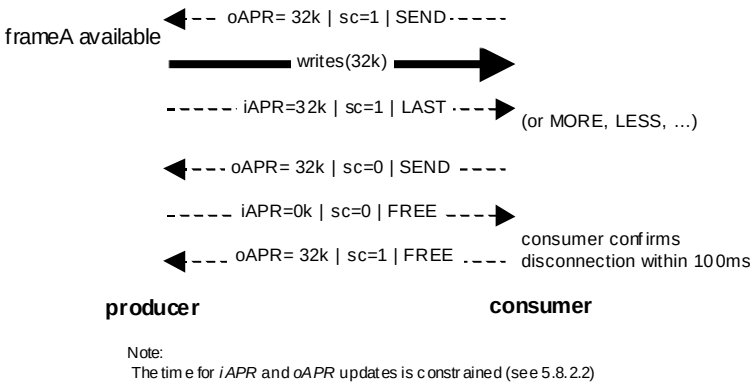

Figure 5.9 – Producer signaled disconnection

5.6 Heartbeat indications

Heartbeats show the presence of a port with a pending protocol transaction when that port is unable to execute the transaction in a timely manner. The port with a pending transaction sends a heartbeat to the connected port in lieu of the protocol transaction. The heartbeat informs the connected port of the delay of the expected transaction (either a control register access or segment buffer access). If the producer or consumer receives a heartbeat indication, the node that receives the indication shall return a heartbeat confirmation.

If an expected transaction or a heartbeat from the pending port has not been received within the specified time-out interval, the port is assumed to be defective. The time-out interval is specified in section 1.1.1.

5.6.1 Producer heartbeat

A producer may initiate the heartbeat by updating the consumer-resident *iAPR*, setting *iAPR.hb* to be the complement of the previously observed *oAPR.hb* value, as illustrated in Figure 5.10. The consumer-plug confirms the heartbeat by updating the producer-resident *oAPR*, setting *oAPR.hb* to be the same as the recently observed *iAPR.hb* value. Other fields than *hb* bit of the *iAPR* and *oAPR* are the same values as the last successful Compare-and-Swap value.

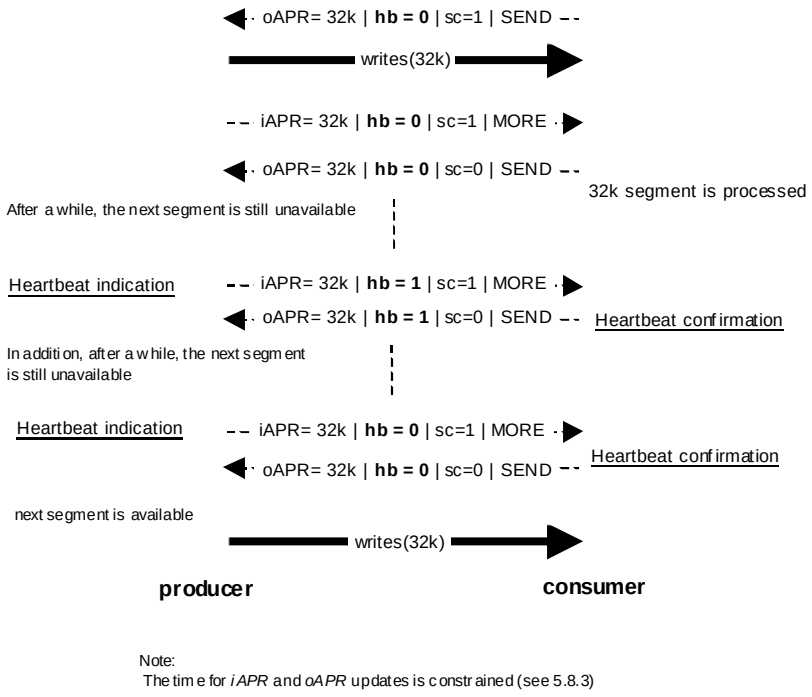

Figure 5.10 – Producer initiated heartbeat

Heartbeats involve updates of the *oAPR* and *iAPR*s and all other transactions shall be blocked until a handshake of heartbeats is completed.

5.6.2 Consumer-plug heartbeat

A consumer-plug may initiate a heartbeat, by updating the producer-resident *oAPR*, setting *oAPR.hb* to be the complement of the previously observed *iAPR.hb* value, as illustrated in Figure 5.11. The producer-plug confirms the heartbeat by updating the consumer-resident *iAPR*, setting *iAPR.hb* to be the same as the recently observed *oAPR.hb* value. Fields other than the *hb* bit of the *iAPR* and *oAPR* are the same values as the last successful Compare-and-Swap value.

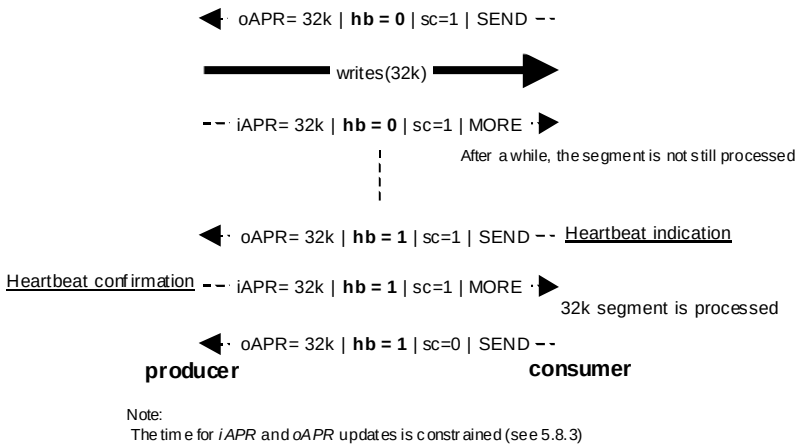

Figure 5.11 – Consumer initiated heartbeat

Heartbeats involve updates of the *oAPR* and *iAPR*s and all other transactions shall be blocked until a handshake of heartbeats is completed.

5.7 Serial Bus transaction errors

The asynchronous connection protocols are designed to allow safe retries of failed Serial Bus transactions. A Serial Bus transaction may fail due to a bus reset (queued requests and responses are discarded) or due to a data-transmission failure. The following design principles are assumed by this standard for the purpose of supporting safe retries:

1) Control registers. Successive updates of control register values always provide distinctive values, due to mandated changes in these registers' *sc* bit or *mode*-field values.

2) Segment buffer addresses. The segment buffer writes are defined to have no effect other than the update of the addressed data value. Processing of duplicate retries can thus be handled as follows:

   a) Ordered transactions. If sequential segment buffer transactions are streamed into memory by a DMA engine that processes write requests, the addresses of these transactions can be checked. Duplicate transactions have the same segment buffer address and no intervening control register updates. These duplicates can be safely discarded.

   b) Unordered transactions. When unordered segment buffer transactions are supported, redundant writes may be accepted because they have no effect on the final segment buffer values. Note that existing OHCI designs have this property.

3) Connection management commands. When desired, the connection-management commands can be designed to be fault tolerant in the following ways:

   a) Duplicate detection. The success of an initial command is determined by the controller before attempting to retry the failed command.

b) Resource recovery. The connection context is discarded unless the connection is quickly reestablished after a bus reset.

Thus, a reliable connection can be maintained because transactions can be safely retried after busy-retry or response-missing timeouts are detected. Of course, the retries are subject to specified transaction label use constraints defined in Reference [R10] and should use the most current (and possibly changed, due to a bus reset) node ID values.

NOTE — Some consumers may have DMA-like hardware that relies on the sequential nature of segment buffer transfers and will return a type_error if the addresses are not received in an increasing (although sometimes replicated) order. Thus, producer-resident data-transfer hardware shall have the capability of continuing by repeating the failed transaction, rather than repeating the entire data transfer.

5.8 Timeout constraints

Some specified timeout values should be required to avoid unnecessary idle state of a plug going on for a long interval. The following subsection describes these timeout constraints.

5.8.1 Transaction timeout

The plug shall issue an expected transaction, which is a segment buffer access or control register access, or an indication of a heartbeat when the transaction is not ready for a while. If there is no expected transaction or a heartbeat from the plug for 5 seconds, the plug is assumed to be defective. Below (Figure 5.12) is an example of transaction time-out of a consumer plug. The same condition applies to the transaction time-out of a producer plug.

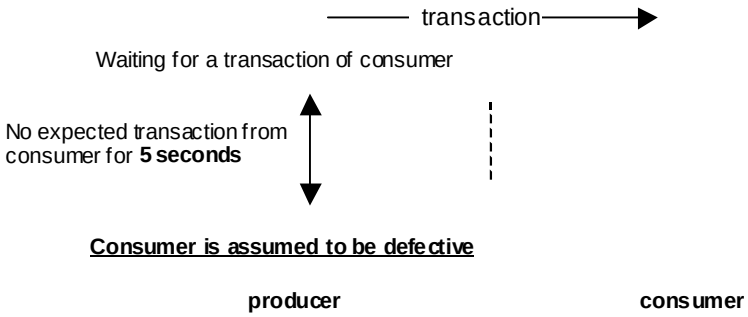

Figure 5.12 – Consumer transaction timeout

5.8.2 Control register timeouts

For several of the flow-control register handshakes, the operation-completion time for the consumer is constrained in order to provide the producer with ways of detecting consumer node failures. These timeouts are described in the following subsections.

These timing limits apply to the control register updates during the assertion and confirmation of heartbeat indications (see 1.1.1).

5.8.2.1 Consumer suspended timeout

The consumer may signal its suspended condition by setting *oAPR.mode* field of the producer-resident *oAPR* to "SUSPEND". The suspended condition is confirmed when the producer sets the *iAPR.mode* field of the consumer-resident *iAPR* to "SUSPENDED". The confirmation of the SUSPEND condition should be provided within 100 ms after the last lock transaction is completed, as illustrated in Figure 5.13.

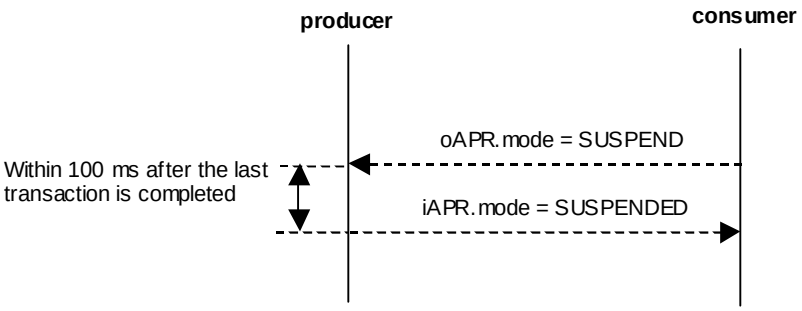

Figure 5.13 – Consumer suspend signaling

5.8.2.2 Disconnection timeout

As described in section 5.5, a producer or consumer may initiate the disconnection of an asynchronous connection by using the indication of *oAPR.mode*=FREE (or *iAPR.mode*=FREE). Time between an indication of the disconnection and its confirmation is constrained and the producer or consumer shall return its confirmation within 100 ms after the last lock transaction is completed. Below (Figure 5.14) is an example of producer initiated disconnection. The same condition applies to the case of the consumer initiated disconnection.

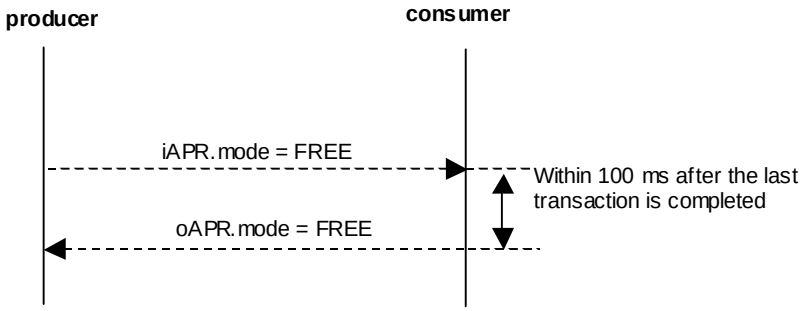

Figure 5.14 – Producer initiated disconnection

5.8.3 Heartbeat timeout

As described in section 5.6, heartbeats involve a handshake of an indication and its confirmation. Time between the indication of a heartbeat and its confirmation is constrained. A producer or consumer receiving an indication of heartbeat shall return its confirmation within 100 ms after the last lock transaction is completed. The heartbeat rate, which is time between successive heartbeat initiations, should

be minimally 2 seconds. Below (Figure 5.15) is an example of producer initiating a heartbeat. The same condition applies to the case of the consumer initiating a heartbeat.

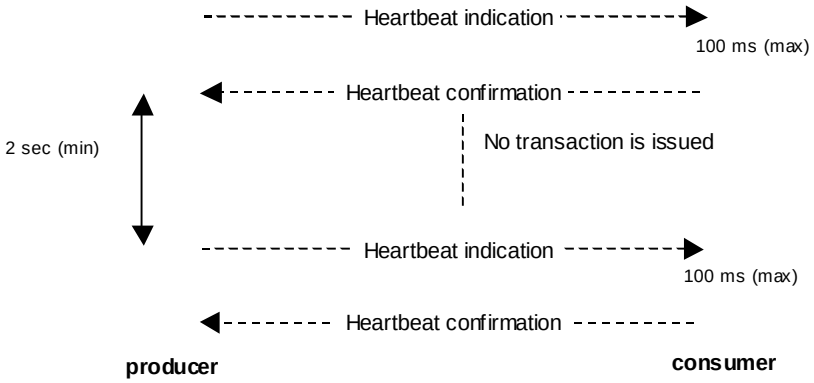

Figure 5.15 – Producer initiate heartbeat

5.9 Producer data-transfer state machines

5.9.1 Producer state machine transitions

The producer's data-transfer state machine is responsible for data movement and flow control register handshaking. For clarity, the application-level interface has been assumed to be a simple polling interface, where the producer port can fetch individual data-transfer blocks by calling the *Fetch()* routine, as illustrated in Figure 5.16, Figure 5.17 and Figure 5.18.

In this state machine, changes to the *oAPR* control register are detected by comparing the current *oAPR* value to its previously observed *oAPRCopy* value. Implementations may provide other mechanisms for detecting the *oAPR* control register updates.

5.9.2 State machine states

The producer state machine states are described below.

The model of the producer in this state machine is as follows:

- support only single consumer
- can only deal with a consumer with $m = 0$ and $ct = 0$ in its consumer-port address

TX0:FREE. The producer remains idle state. No consumer is connected and the plug is inactive. This is the initial state from power-on-reset. All fields of *oAPR* and *oAPRCopy* and *iAPRCopy* are initialized.

TX1:WAIT. The producer is waiting for the expected update of the *oAPR* by the consumer.

TX2:WRITE. Data blocks are written into the consumer's segment buffer until reaching at an end-of-segment or end-of-frame condition.

TX3:SUSPEND. A SUSPEND indication has been received from the consumer and waiting for a RESUME indication by the consumer.

TX4:WAITB. The producer detected a bus reset and wait for a *RestoreEvent* indication from the application layer.

TX5:FREECONF. The producer is waiting for the consumer-plug to confirm the FREE mode value by updating the producer-resident *oAPR*, setting *oAPR.mode* = FREE.

TX6:HBCONF. The producer is waiting for the consumer-plug to confirm the heartbeat by updating the producer-resident *oAPR*, setting *oAPR.hb* to be the same as the recently observed *iAPR.hb* value.

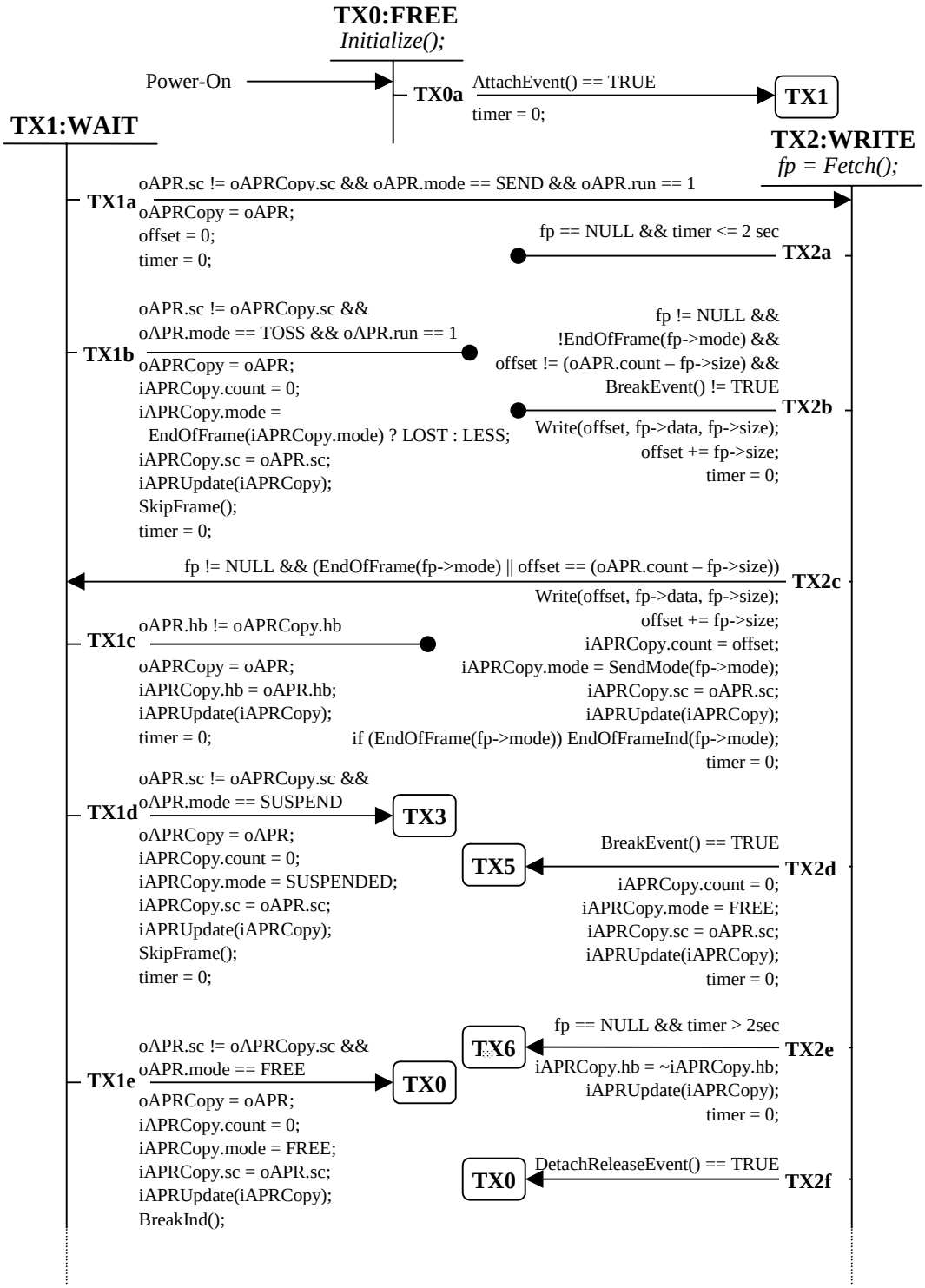

Figure 5.16 – Producer state machine

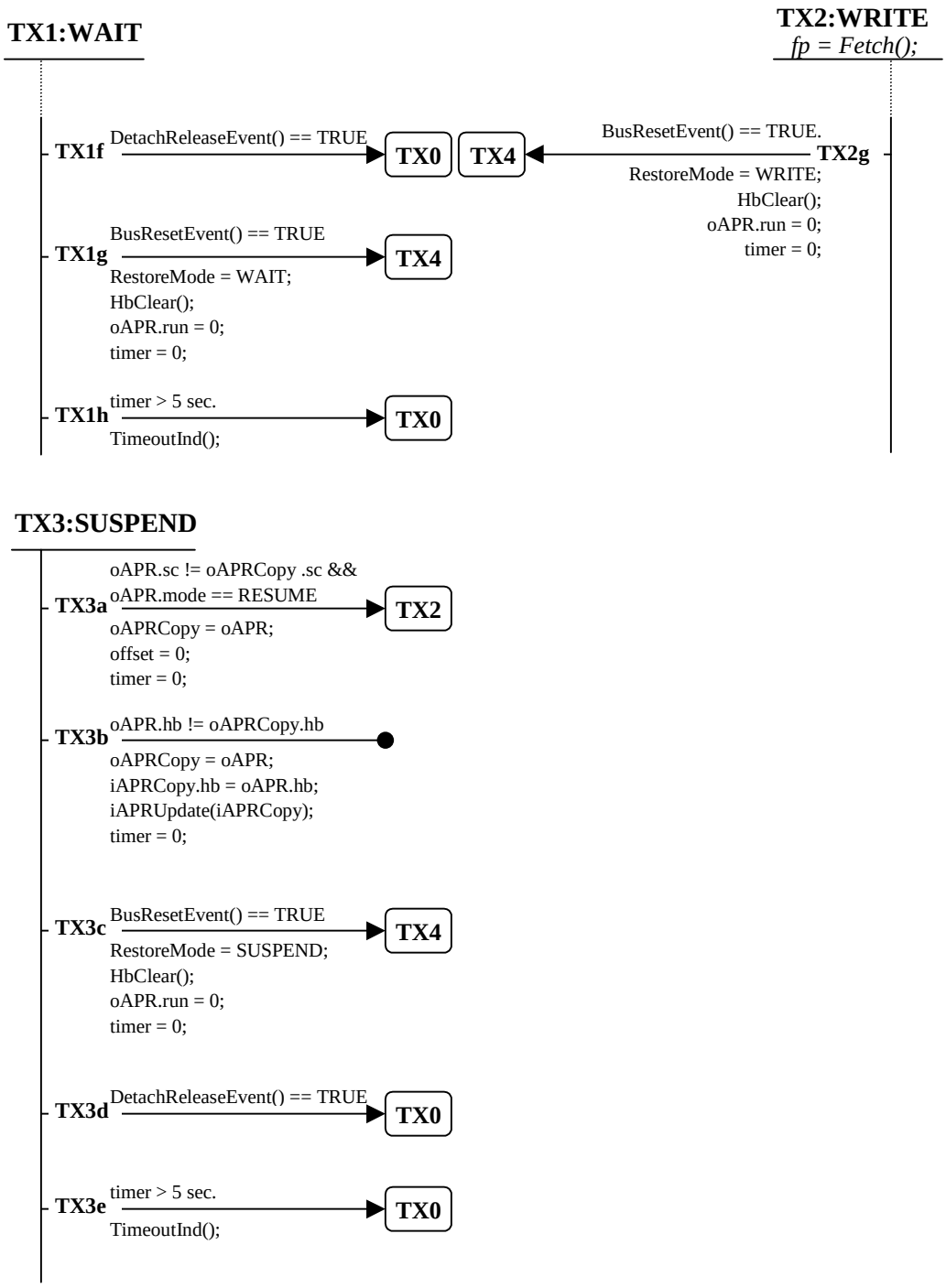

Figure 5.17 – Producer state machine (continued)

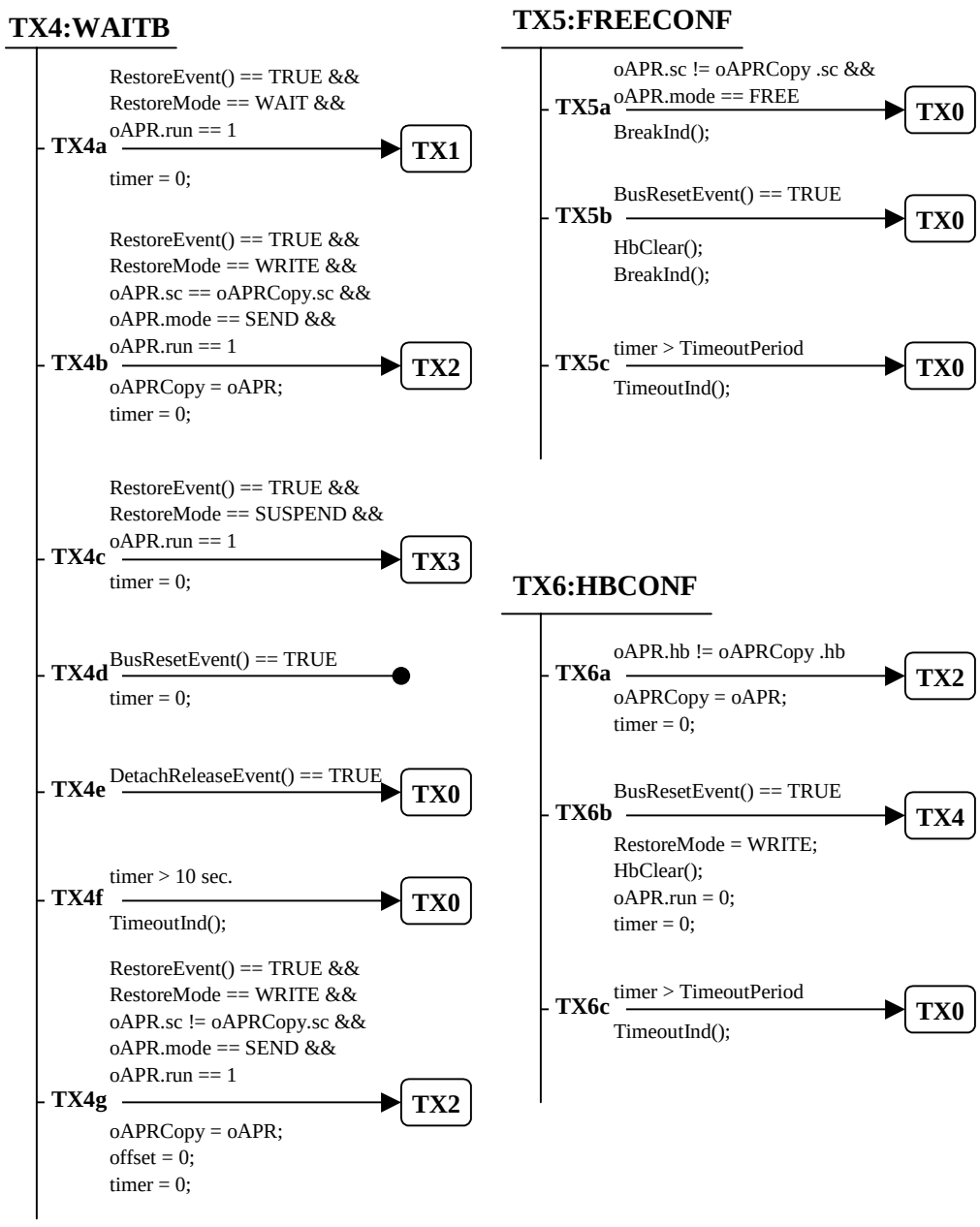

Figure 5.18 – Producer state machine (continued)

NOTE —

a) The conditions and actions of the state machines are defined by C++ code. The code notation is summarized in [B1].

b) The following definitions apply:
```
#define EndOfFrame(mode) (mode==LAST || mode==LESS || mode==JUNK || mode==LOST)
#define SendMode(mode) (EndOfFrame(mode) ? mode:MORE)
```

c) *offset* indicates address offset from top of the consumer's segment buffer and also describes size of the data that is already written in the segment buffer.

d) *timer* is used to check timeout and is counted up automatically.

e) *fp* is the pointer of the structure for pointing the data that is written in the consumer's segment buffer. The structure is:
```
struct{
    Quadlet *data; //Pointer to the data to be written
    int size; // Byte size of the data
    int mode; // Mode of fetched data block
} FramePointer;
```

fp->size is equal to or less than max_payload size.

f) *RestoreMode* is used to indicate which mode the producer should resume after a bus reset.

g) *AttachEvent()* returns TRUE when the application layer requests to make a connection.

h) *DetachReleaseEvent()* returns TRUE when the application layer requests to break the connection.

i) *RestoreEvent()* returns FALSE after a bus reset. It returns TRUE when the application layer requests restoring the connection after the bus reset.

j) *BusResetEvent()* returns TRUE when a bus reset is detected.

k) *Fetch()* fetches application data to write. The size of data to be fetched is equal to payload size of a packet or less when the data is last of the frame. The *Fetch()* returns a null value if no data is ready to be sent. When data becomes available, *Fetch()* returns a pointer to a data-transfer block, size of the data block and a *mode* field that marks the final end of frame block.

l) *Write()* executes write transaction. This function has 3 parameters.
offset:Offset address from the top of the consumer's segment buffer address. The sum of the consumer's segment buffer address and offset is the real address of the write transaction.
data:Pointer to the data to be written.
size:Byte size of the data.

m) *iAPRUpdate()* updates consumer resident iAPR with the value of *iAPRCopy*. This function executes Compare & Swap Lock transaction.

n) *SkipFrame()* skips frame transfer. The current frame data is canceled.

o) *BreakEvent()* returns TRUE if the application wants to stop sending frame data.

p) *EndOfFrameInd(mode)* indicates the mode to the application layer.

q) *Initialize()* initializes all values of the *oAPR*, *oAPRCopy* and *iAPRCopy*. All values become zero.

r) *HbClear()* sets the *hb* bit in all contexts *(oAPR, oAPRCopy, iAPRCopy*) to zero.

s) *TimeoutInd()* indicates the application layer that a timeout occurs.

t) *BreakInd()* indicates the application layer that the connection had been broken.

5.9.3 State machine transitions

TX0a. When the application layer indicates the completion of a connection, the producer activates the producer plug and transitions to RX1:WAIT state.

TX1a. A changed *oAPR* with a SEND indication causes the next segment-buffer transfer to begin. The producer port transitions to TX2:WRITE state.

TX1b. When *iAPRCopy.mode* == MORE, in this case, an updated *oAPR* as a TOSS indication causes the remainder of the frame to be discarded and a LESS indication is provided. In other cases (such as LAST, LESS, JUNK, LOST), an updated oAPR as a TOSS indication causes one or more frame to be discarded and a LOST indication is provided.

TX1c. A changed *oAPR.hb* bit indicates a heartbeat had been requested by the consumer. The producer updates the consumer-resident *iAPR* to confirm the heartbeat and reset the timer value to zero.

TX1d. A changed *oAPR* with a SUSPEND indication causes the producer to make a transition to TX3:SUSPEND state. The remainder of the frame is discarded and a SUSPENDED confirmation is provided.

TX1e. A changed oAPR with FREE indication causes the producer to break the connection. The producer updates the consumer–resident iAPR with setting *iAPR.mode* = FREE to confirm the indication. And the producer requests the application layer to break the connection by *BreakInd()* and transitions to TX0:FREE state.

TX1f. When the application layer requests to make a disconnection, the producer breaks the connection and transitions to TX0:FREE state.

TX1g. When a bus reset is detected, the producer stores the current state value to *RestoreMode*, then transitions to TX4:WAITB state.

TX1h. If there is no expected update of the *oAPR* within 5 seconds, the producer indicates the timeout to the application layer by *TimeoutInd()*, then transitions to TX0:FREE state.

TX2a. If no application data is available, the producer continues to try to fetch until the data becomes ready.

TX2b. A data block is fetched by the producer and written into the consumer's segment buffer, in a sequential fashion. After each segment buffer writes, the offset address is increased by the size of the data payload, then the next data block will be fetched.

TX2c. An indication is provided by *Fetch()*when an end-of-frame or end-of-segment boundary had been detected. In this case, the *iAPR.sc* (segment count) bit is changed to the previously observed *oAPR.sc* value, so that the *iAPR.sc* value is to be changed.

TX2d. The *BreakEvent()* returns TRUE when the application layer requests to break a connection. The producer updates the consumer-resident *iAPR* with a FREE *mode* value and transitions to TX5:FREECONF state.

TX2e. If no application data is available for over 2 seconds, the producer updates the consumer-resident *iAPR* with heartbeat indication flag set (reverse the *iAPR.hb* bit) to inform the presence of the node and transitions to TX6:HBCONF state.

TX2f. When the application layer requests to make a disconnection, the *DetachReleaseEvent()* returns TRUE. Then the producer disconnects the connection and transitions to TX0:FREE state.

TX2g. When a bus reset is detected, the producer stores the current state value to *RestoreMode* and transitions to TX4:WAITB state.

TX3a. A changed *oAPR* with a RESUME *mode* value causes the next segment-buffer transfer to begin. The producer transitions to TX2:WRITE state.

TX3b. A changed *oAPR.hb* bit indicates a heartbeat had been requested by the consumer. The producer updates the consumer-resident *iAPR* to confirm the heartbeat and reset the timer value to zero.

TX3c. When a bus reset is detected, the producer stores current state value to *RestoreMode* and transitions to TX4:WAITB state.

TX3d. When the application layer requests to make a disconnection, the producer disconnects the connection and transitions to TX0:FREE state.

TX3e. If there is no expected update of the *oAPR* within 5 seconds, the producer indicates the timeout to the application layer by *TimeoutInd()*, then transitions to TX0:FREE state.

TX4a. When the application layer requests to restore the connection and *RestoreMode* == WAIT, the producer returns to TX1:WAIT state.

TX4b. When the application layer requests to restore the connection and *RestoreMode* == WRITE and the producer-resident *oAPR* is updated, the producer returns to TX2:WRITE state.

TX4c. When the application layer requests to restore the connection and *RestoreMode* == SUSPEND, the producer returns to TX3:SUSPEND state.

TX4d. When a bus reset is detected, the producer remains in TX4:WAITB state.

TX4e. When the application layer requests to make a disconnection, the *DetachReleaseEvent()* returns TRUE. The producer breaks the connection and moves to TX0:FREE state.

TX4f. If there is no expected update of the *oAPR* within 10 seconds, the producer indicates the timeout to the application layer by *TimeoutInd()*, then transitions to TX0:FREE state.

TX4g. When the application layer requests to restore the connection and *RestoreMode* == WRITE and the producer-resident *oAPR* with toggling *sc* bit of *oAPRCopy* is updated, the producer returns to TX2:WRITE state.

NOTE —TX4g condition is different from TX4b condition. TX4g condition may be met when a bus reset occurs before completing the transaction of updating *iAP*R. For example, the following condition is conceivable. It is that a bus reset occurs between sending a lock request and receiving a lock response from the consumer. In this situation, the producer assumes that the requested lock operation may be failed, whereas the consumer may complete the lock operation. If the request of updating the producer-resident *oAPR* with toggling *sc* bit of *oAPRCopy* is received after a bus reset, the producer may assume that the previous transaction of updating *iAPR* has been completed and start sending a next segment.

TX5a. A changed oAPR with FREE indication causes the producer to break the connection. And the producer indicates the application layer to break the connection by *BreakInd()* and transitions to TX0:FREE state.

TX5b. When a bus reset is detected, the producer indicates the application layer to break the connection by *BreakInd()* and transitions to TX0:FREE state.

TX5c. If there is no expected update of the *oAPR* within a *TimeoutPeriod*, the producer indicates the timeout to the application layer by *TimeoutInd()*, then transitions to TX0:FREE state.

TX6a. A changed *oAP*R.*hb* bit indicates a heartbeat confirmation from the consumer, and transitions to TX2:WRITE state.

TX6b. When a bus reset is detected, the producer sets *RestoreMode* to WRITE, and transitions to TX4:WAITB state.

TX6c. If there is no expected update of the *oAPR* within a *TimeoutPeriod*, the producer indicates the timeout to the application layer by *TimeoutInd()*, then transitions to TX0:FREE state.

NOTE —For detailed definition of TimeoutPeriod, please refer to 5.8.2.

5.10 Consumer data-transfer state machines

5.10.1 Consumer state machine

The consumer's data-transfer state machine is responsible for the processing of received data and flow-control register handshakes. The application-level is assumed to provide a simple data-push interface, where the *Flush()* routine is called to transfer segment data to the application, as illustrated in Figure 5.19, Figure 5.20 and Figure 5.21.

In this state machine, changes to the *iAPR* are detected by comparing the current *iAPR* value to its previously observed *iAPRCopy* value. Implementations may provide other mechanism for detecting the *iAPR* updates.

5.10.2 State machine states

The consumer state machine states are described below:

RX0:FREE. The consumer remains idle state. No producer is connected and the plug is inactive. The initial state from Power-On reset is here. All fields of *iAPR* and *iAPRCopy* and *oAPRCopy* are initialized.

RX1:WAIT. The consumer is waiting for the expected update of the *iAPR* by the producer while segment data is being received from the producer. The consumer remains idle until provided with an indication that the segment data has been transferred.

RX2:FLUSH. The *iAPR* is updated. If the data is written in the segment buffer, the data is being transferred to the application layer. The success of this transfer affects the next state transition.

RX3:SUSPEND. The consumer remains suspended mode until the application layer indicates to resume.

RX4:WAITB. The consumer detected a bus reset and waiting for the indication of restore by the application layer.

RX5:SUSCONF. The consumer is waiting for the producer-plug to confirm the SUSPEND mode by updating the consumer-resident *iAP*R, setting *iAPR.mode* to SUSPENDED.

RX6:FREECONF. The consumer is waiting for the producer-plug to confirm the FREE mode by updating the consumer-resident *iAP*R, setting *iAPR.mode* to FREE.

RX7:INACTIVE. The consumer accepts the update of the control register and segment-buffer writes but inhibits the generation of producer-port updates.

RX8:HBCONFA. The consumer is waiting for the producer-plug to confirm the heartbeat by updating the consumer-resident *iAP*R, setting *iAP*R.*hb* to be the same as the latest observed *oAP*R.*hb* value.

RX9:HBCONFB. The consumer is waiting for the producer-plug to confirm the heartbeat by updating the consumer-resident *iAP*R, setting *iAP*R.*hb* to be the same as the latest observed *oAP*R.*hb* value.

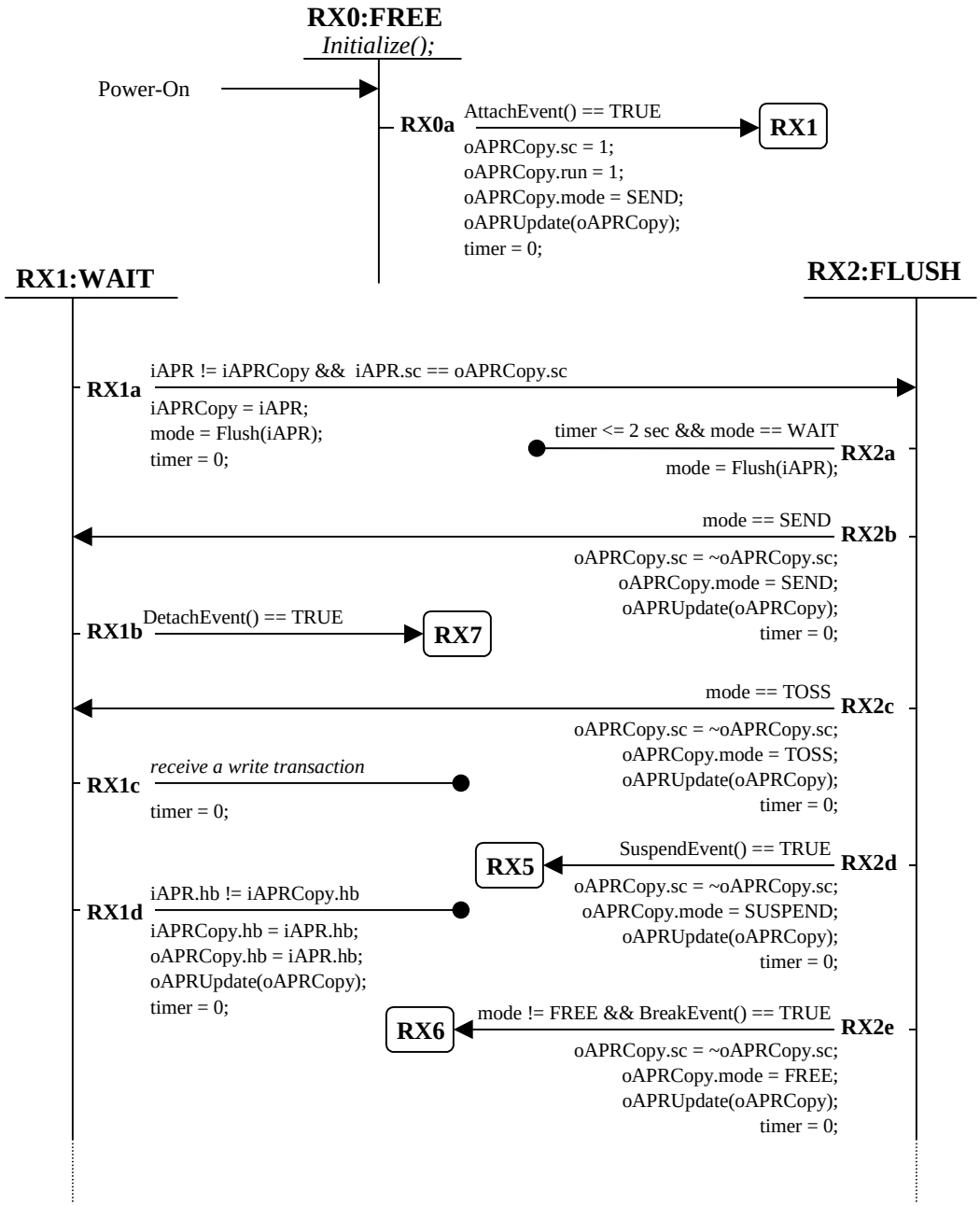

Figure 5.19 – Consumer state machine

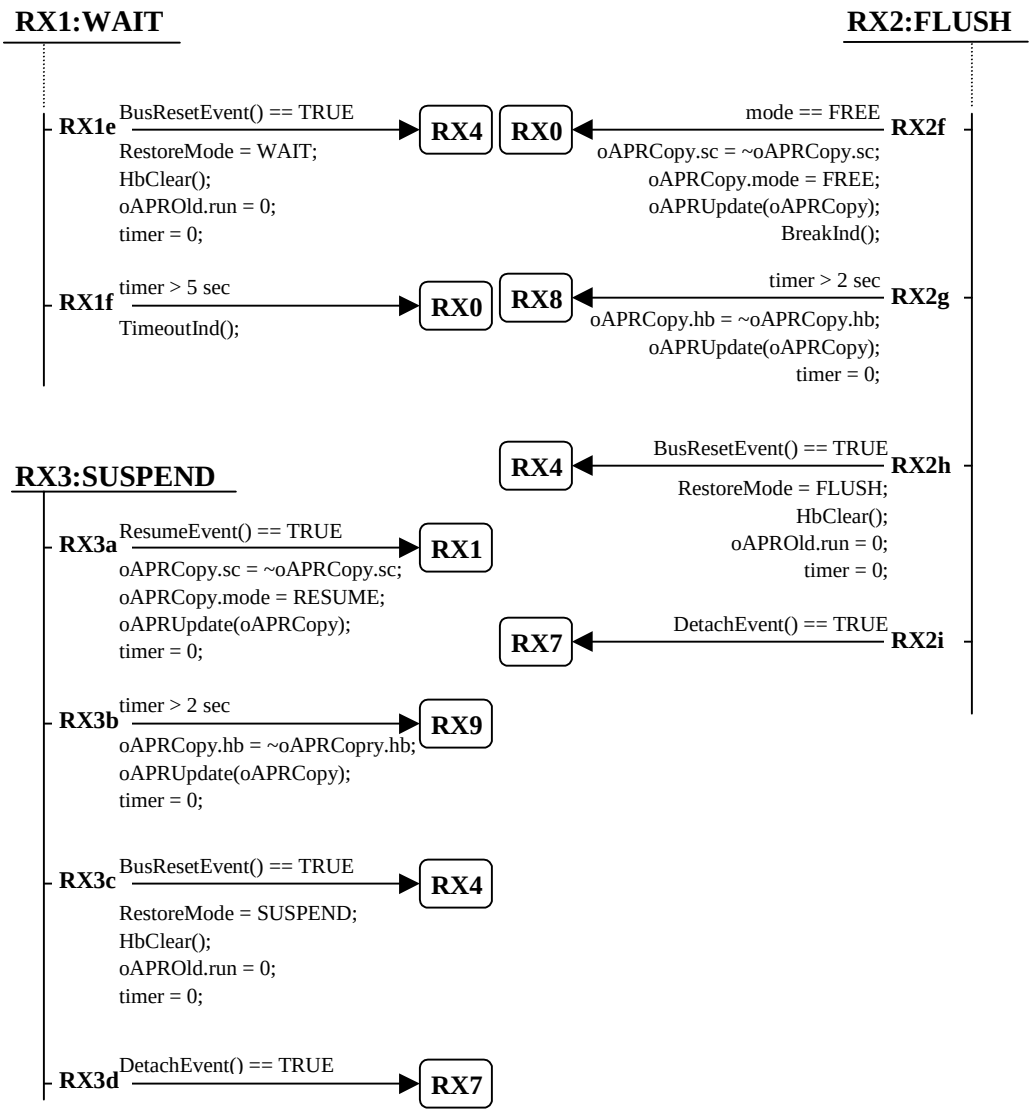

Figure 5.20 – Consumer state machine (continued)

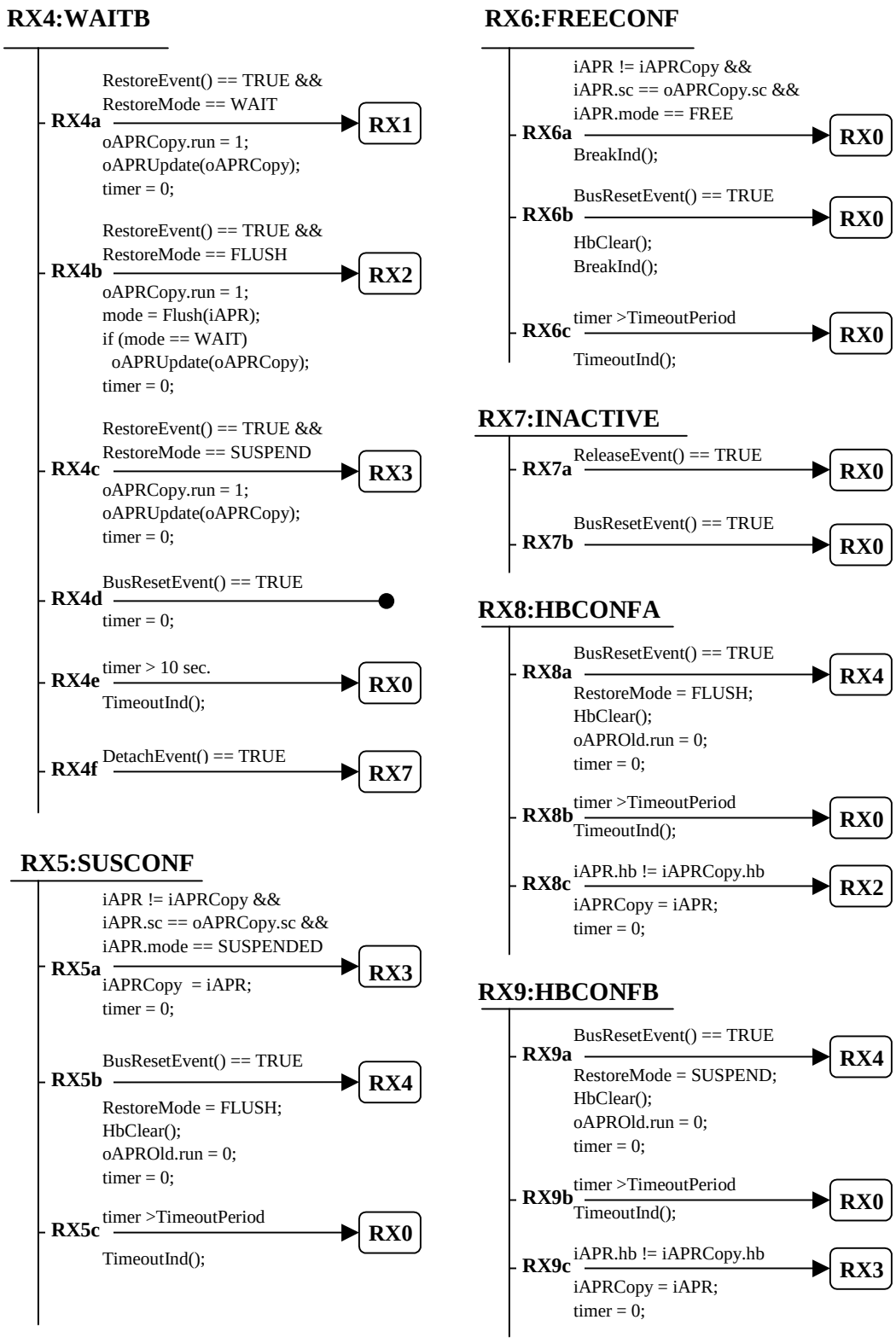

Figure 5.21 – Consumer state machine (continued)

NOTE —

a) The conditions and actions of the state machines are defined by C++ code. The code notation is summarized in [B1].

b) *mode* represents the result of the *Flush()*. The application layer indicates the next action to be taken by the consumer by *mode* value.

c) *timer* is used to check timeout and counted up automatically.

d) *Initialize()* initializes all values of *iAPR*, *iAPRCopy*, *oAPRCopy*, and *oAPROld*. All values become zero.

e) *RestoreMode* is used to indicate which mode the producer should be resume from after a bus reset.

f) *AttachEvent()* returns TRUE when the application layer requests to start communication.

g) *DetachEvent()* returns TRUE when the application layer requests to make a disconnection.

h) *RestoreEvent()* returns TRUE when the application layer requests to restore the connection after a bus reset.

i) *ReleaseEvent()* returns TRUE when the application layer request to release resources.

j) *BreakEvent()* returns TRUE if the application layer wants to break the connection (stop receiving frame data). The initial returned value is FALSE. This value is used to switch FREE indication processing.

k) *SuspendEvent()* returns TRUE when the application layer requests to suspend.

l) *ResumeEvent()* returns TRUE when the application layer requests to resume.

m) *BusResetEvent()* returns TRUE when a bus reset is detected.

n) *Flush()* normally transfers received data from segment buffer to the application buffer.

   1) If iAPR.mode is **LAST** or **MORE**, the consumer transfers the segment data to application buffer.

   2) According to the implementation of this function, data transfer can be divided over more than one process.

   3) When all data transfer is not yet done, this function returns **WAIT** otherwise returns **SEND**.

   4) If *iAPR.mode* is **JUNK**, the consumer discards the segment data and this function returns **SEND**.

   5) If *iAPR.mode* is **LESS**, the consumer transfers or discards the segment data.

   6) It is application dependent that the consumer transfers or discards. When buffer space for the next frame is available, this function returns **SEND** and otherwise returns **WAIT**.

   7) If iAPR.mode is **LAST** or **JUNK** or **LESS** or **LOST**, this function issues the EndOfFrameInd(iAPR.mode) to the application layer.

o) Regardless of the *iAPR.mode*, when the application layer indicates **TOSS**, the consumer discards the segment data and this function returns **TOSS**.

   1) If *iAPR.mode* is **LOST** and the application indicates there is space for storing next frame, this function returns **SEND**.

   2) If *iAPR.mode* is **LOST** and the application indicates there is no space for storing next frame, this function returns **TOSS**.

   3) If *iAPR.mode* is **FREE**, this function returns **FREE**.

   4) When this function returns **SEND** and the consumer supports dynamic buffer allocation, *oAPR.count* field may not be same as the value previously updated.

p) *oAPRUpdate()* updates producer resident *oAPR* with *oAPRCopy*. The function executes Compare & Swap Lock transaction using *oAPRCopy* and *oAPROld*. If this function is succeeded, the return value is kept as *oAPROld*. And *oAPROld* is used as argument value at next *oAPRUpdate()*.

q) *HbClear()* sets the hb bit in all contexts *(iAPR, iAPRCopy, oAPRCopy* and *oAPROld*) to zero.

r) *TimeoutInd()* indicates the application layer that a timeout occurs.

s) *BreakInd()* indicates the application layer that the connection had been broken.

5.10.3 State machine transitions

Descriptions of these state machine transitions are as follows:

RX0a. When the indication of establishing a connection is given by the application layer, the consumer activates the consumer plug and updates the producer-resident *oAPR* with setting *iAPR.mode* to SEND. The consumer transitions to RX1:WAIT state.

RX1a. The consumer transits to RX2:FLUSH state after observing that the *iAPR* has been updated.

RX1b. When the application layer requests to make a transition to the inactive state, the consumer transitions to RX7:INACTIVE state.

RX1c. Whenever the consumer receives a write transaction from the producer, the consumer resets timer. The consumer remains in RX1:WAIT state.

RX1d. A changed *iAPR.hb* bit indicates the request of a heartbeat by the producer. The consumer updates the producer-resident *oAPR* to confirm the indication of the heartbeat and resets the *timer* value to zero. The consumer remains in RX1:WAIT state.

RX1e. When a bus reset is detected, the consumer stores the current state value to *RestoreMode* and transitions to RX4:WAITB state.

RX1f. If there is no expected update of the *iAPR* within 5 seconds, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX2a. The consumer remains in the RX2:FLUSH state while the application is delaying the segment-buffer transfer.

RX2b. After the segment-buffer data has been successfully consumed by the application, the consumer updates the producer-resident *oAPR* with setting *oAPR.mode* to SEND and returns to the RX1:WAIT state.

RX2c. The consumer provides a TOSS indication when the application does not want to receive the current frame. Transferred data is discarded by the application and returns to the RX1:WAIT state.

RX2d. The application layer requests to make the transition to the suspended mode and the consumer provides a SUSPEND indication to the producer and transitions to RX5:SUSCONF state.

RX2e. When the application layer wants to break the connection, *BreakEvent*() returns TRUE, that means the consumer initiated breaking a connection. The consumer updates the producer-resident *oAPR* with setting *oAPR.mode* to FREE and transitions to RX6:FREECONF state.

RX2f. A change *iAPR* with a FREE indication causes the consumer to break the connection (the producer initiated disconnection). The consumer updates the producer-resident *oAPR* with a FREE mode value to confirm the disconnection. Then the consumer indicates the application layer that the connection had been broken by *BreakInd()* and transitions to RX0:FREE state.

RX2g. If no space of the buffer is available within 2 seconds, the consumer updates the producer-resident *oAPR* to indicate a heartbeat, and transitions to RX8:HBCONFA.

RX2h. When a bus reset is detected, the consumer stores the current state value to *RestoreMode* and transitions to RX4:WAITB state.

RX2i. When the application layer requests to make a disconnection, the consumer transitions to RX7:INACTIVE state.

RX3a. The application layer requests to resume communication and the consumer provides a RESUME indication to the producer. After that, the consumer transitions to RX1:WAIT state.

RX3b. When the timer proceeds 2 seconds, the consumer updates the producer-resident *oAPR* with a heartbeat indication to remain the connection, and transitions to RX8:HBCONFB state.

RX3c. When a bus reset is detected, the consumer stores the current state value to *RestoreMode* and transitions to RX4:WAITB state.

RX3d. When the application layer requests to detach, the consumer transitions to RX7:INACTIVE state.

RX4a. When the application layer requests to restore and *RestoreMode* == WAIT, the consumer returns to RX1:WAIT state.

RX4b. When the application layer requests to restore and *RestoreMode* == FLUSH, the consumer returns to RX2:FLUSH state.

RX4c. When the application layer requests to restore and *RestoreMode* == SUSPEND, the consumer returns to RX3: SUSPEND state.

RX4d. When a bus reset is detected, the consumer remains in RX4:WAITB state.

RX4e. If there is no expected update of the *iAPR* within 10 sec, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX4f. When the application layer requests to detach, the consumer transitions to RX7:INACTIVE state.

RX5a. A changed *iAPR* with a SUSPENDED indication causes the consumer to transitions RX3:SUSPEND state.

RX5b. When a bus reset is detected, the consumer sets *RestoreMode* to FLUSH, and transitions to RX4:WAITB state.

RX5c. If there is no expected update of the *iAPR* within a Timeout Period, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX6a. A change *iAPR* with a FREE indication confirms to break the connection (the consumer initiated disconnection). Then the consumer indicates the application layer that the connection had been broken by *BreakInd()* and transitions to RX0:FREE state.

RX6b. When a bus reset is detected, the consumer indicates the application layer that the connection had been broken by BreakInd() and transitions to RX0:FREE state.

RX6c. If there is no expected update of the *iAPR* within a Timeout Period, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX7a. When the application layer requests to release resources, the consumer transitions to RX0:FREE state.

RX7b. When a bus reset is detected, the consumer transitions to RX0:FREE state.

RX8a. When a bus reset is detected, the consumer sets *RestoreMode* to FLUSH, and transitions to RX4:WAITB state.

RX8b. If there is no expected update of the *iAPR* within a Timeout Period, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX8c. A changed *iAPR.hb* bit indicates the confirmation of a heartbeat by the producer, and transitions to RX2:FLUSH state.

RX9a. When a bus reset is detected, the consumer sets *RestoreMode* to SUSPEND and transitions to RX4:WAITB state.

RX9b. If there is no expected update of the *iAPR* within a Timeout Period, the consumer transitions to RX0:FREE state. The consumer indicates the timeout occurred to the application layer by *TimeoutInd()*.

RX9c. A changed *iAPR.hb* bit indicates the confirmation of a heartbeat by the producer.

This page is left intentionally blank

Annexes

These annexes are informative information for other possibilities of this specification and may be subject to change.

Annex A: Bibliography (informative)

A.1 Bibliography

The following document(s) provide useful informative information for the reader:

[B1] ANSI X3.159-1989, Programming Language—C.

Annex B: Applications and design models (informative)

B.1 Asynchronous connection applications

B.1.1 Asynchronous connection applications

A variety of applications are expected to benefit from the asynchronous connection's robust flow-controlled high-bandwidth asynchronous data-transfer services. A few of the possible applications are described in the remainder of this subsection.

B.1.2 Still image transfers

Asynchronous connections are ideal for reliably transferring still-image frames between a camera and a printer, as illustrated in the top left of Figure B.1. When desired, the image may be enhanced as it passes through an intermediate computer, as illustrated in the bottom right of Figure B.1.

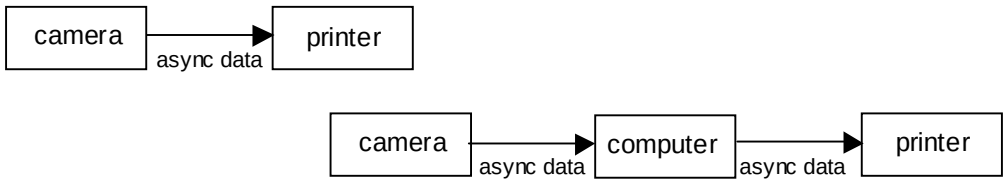

Figure B.1 – Still image transfers

B.1.3 Video overlay transfers

Asynchronous connections can also transfer supplemental information for an isochronous data stream, as illustrated in Figure B.2. For example, a VCR may send an overlay image (such as a program selection guide) to be merged with its video. Alternatively, an audio disk could use an asynchronous connection to send song lyrics.

The advantage of using an asynchronous connection is that the data transfer is robust and flow controlled. A disadvantage is that its delivery can (in the presence of conflicting traffic) be delayed.

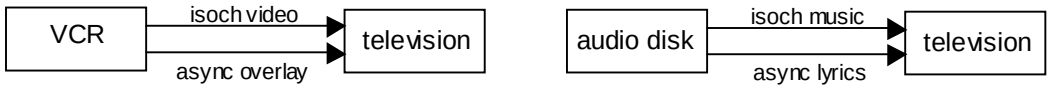

Figure B.2 – Isochronous data supplements

B.1.4 Computer-data transfers

Asynchronous connections supports efficient computer to computer and computer to storage device transfers. Depending on the application, responseless or responsive connections are expected to be used, as illustrated in the left and right of Figure B.3 respectively.

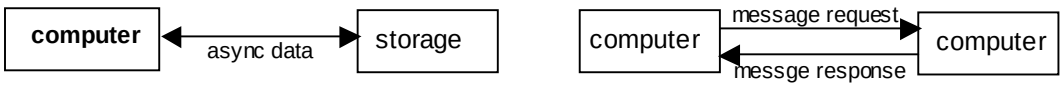

Figure B.3 – Computer-data transfers

B.2 Segment buffer design models

The asynchronous connection protocols were influenced by the desire to support several types of inexpensive segment buffer implementations. As background, several of these are illustrated in the following subsections.

B.2.1 Exported segment buffer addresses

A consumer can be designed to efficiently translate incoming segment buffer data-write transactions into local memory writes, as illustrated in Figure B.4. An open host controller interface (OHCI) selectively routes requests to memory or places the requests into a general receive FIFO (GRF), based on the request's *tcode* (transaction code) and address-offset values.

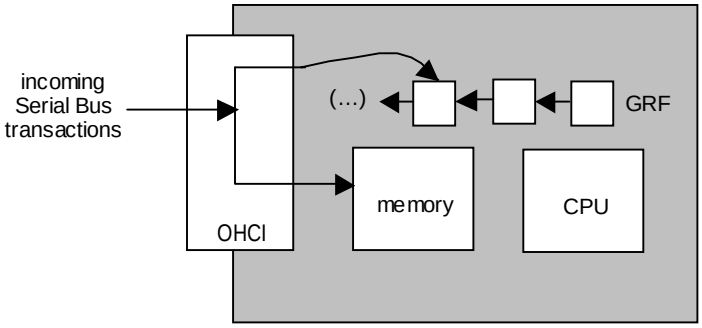

Figure B.4 – Exported segment buffer addresses

The segment buffer writes are handled by hardware and do not interrupt the CPU. The count register updates require processor interrupts so that the segment buffer data can be processed. This asynchronous connection protocol is compatible with either of the following count-register interrupt strategies:

1) Software registers. The count-update requests are saved in a general receive FIFO (GRF) and signals the processor to activate GRF processing.

2) Hardware registers. Hardware processes the count-update requests, updating memory as requested, and signals the CPU when this memory update completes.

The asynchronous connection design allows the segment buffer writes to be efficiently translated to memory writes, while the setup commands and count-register writes queued and interrupt the CPU. Processing of request packets is performed as follows:

1) AV/C commands. AV/C commands are written into a high-memory-address FCP location, which has no direct memory-address equivalent. Therefore, these requests are queued in the GRF.
2) Count updates. Asynchronous connection count updates use CompareSwap4 transactions, which has no generic memory-update equivalent. Therefore, these requests are queued in the GRF.
3) Segment buffer writes. The asynchronous connection plug is normally placed at a low-memory address. Therefore, the segment buffer writes are efficiently translated into memory writes.

B.2.2 Translated segment buffer addresses

A node could also be designed to translate segment buffer writes into data-buffer writes, based on an interface-resident address-translation register. Thus, incoming segment buffer writes are directly mapped into memory-buffer writes and do not interrupt the microprocessor, as illustrated in Figure B.5.

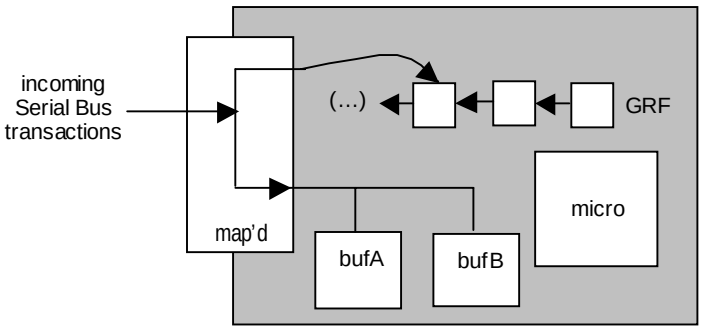

Figure B.5 – Translated segment buffer addresses

The count-register requests are expected to be saved in a general receive FIFO (GRF) and have the side effect of interrupting the microprocessor.

B.2.3 Emulated segment buffer addresses

A software-based consumer may place all incoming requests into a receive buffer (illustrated as black) before interrupting the microprocessor, as illustrated in Figure B.6. The microprocessor parses the incoming packets, updating control state or data buffers (illustrated as bufA and bufB) as required.

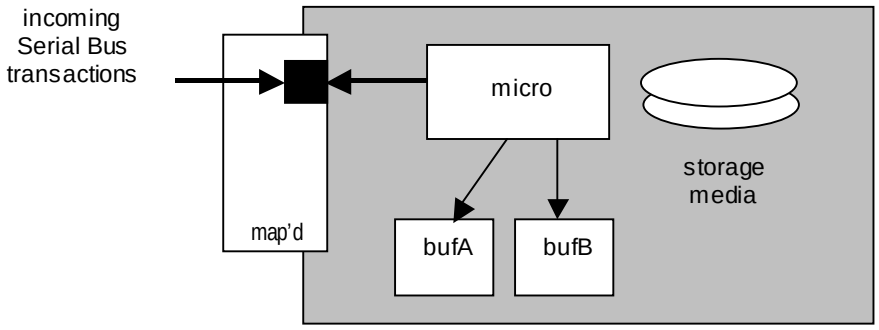

Figure B.6 – Emulated segment buffer addresses

The incoming data is placed into staging buffers for data that is being transferred to (or from) the storage device. Two or more buffers are expected to be provided, so that data writes to bufB can continue while bufA is being transferred to disk, data writes to bufA can continue while bufB data is transferred to disk, etc.

Annex C: Subframe formats (informative)

C.1 Subframe sequences

In many cases, the size of asynchronous connection information block is known when the initial portion of the packet is sent (such as when a still image is being sent). To simplify the consumer (which may have to preallocate storage) and to maximize transfer efficiencies, these information blocks (called subframes) may have a header that specifies their actual length.

A subframe represents a contiguous sequence of bytes, which is processed as a whole, such as a still camera image. Merging multiple subframes into one frame can be more efficient, because all but one or more of the between-frame segment-change handshakes can be avoided.

Only the first few quadlets in these subframes are defined by this specification. Standard length fields simplify the processing of recognized subframes, while allowing unrecognized subframes can be conveniently skipped. An application has the option of extending the subframe header, to include additional application-specific information.

Thus, transfers between asynchronous connection producer and consumer nodes may involve the transfer of data subframes, where more than one data subframes may be placed in each frame. Any of the four subframe formats may be located in any frame, as illustrated in Figure C.1

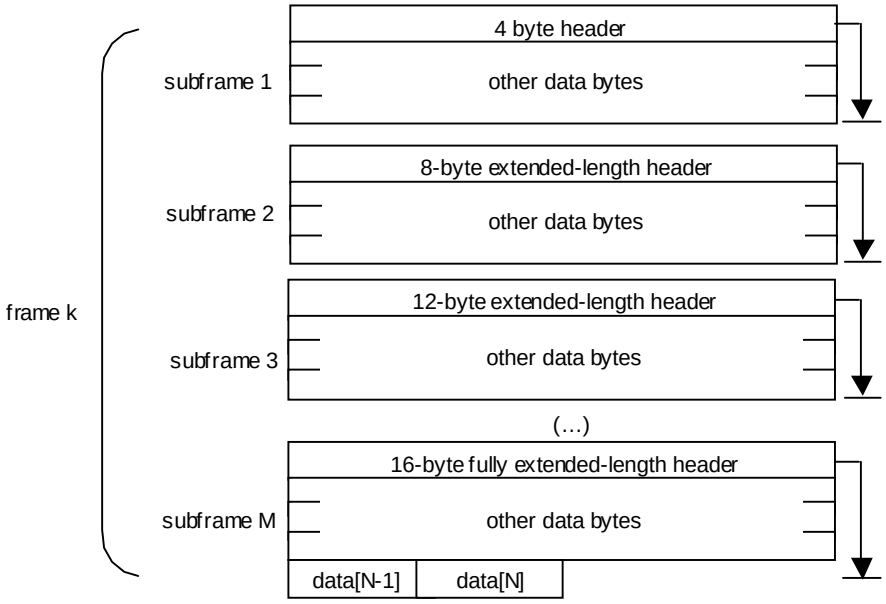

Figure C.1 – Subframe components

The common smaller subframes have a compact 4-byte header and the header can be extended, as necessary to specify larger subframe lengths and to identify context-free *tCode* values. The actual size of the final subframe in each frame may be less than the subframe's header-specified size.

C.2 Subframe headers

C.2.1 Common subframe formats

All subframes start with a few identifiers and are padded to the nearest quadlet, to facilitate their processing at the consumer, as illustrated by Figure C.2.

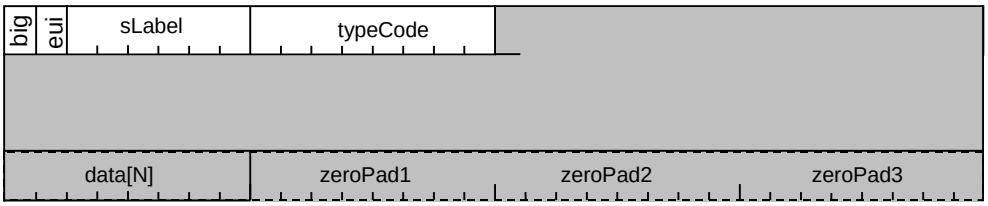

Figure C.2 – Common subframe format

The ***big*** bit values of 0 and 1 are used to identify packets with 16-bit and 48-bit length values respectively.

The ***eui*** bit values of 0 and 1 are used when the *tCode* meaning is context-dependent or context-free respectively.

The 6-bit ***sLabel*** (subframe label) is used to identify this request, for error logging purposes and for properly affiliating response subframes with their request. One of these *sLabel* values is used to label request packets that have no affiliated response, as specified in Table C.1.

Table C.1 – sLabel values

Value	Name	Description
$0\text{-}3E_{16}$	—	Distinctive labels for request and response subframes
$3F_{16}$	NULL	Common label for responseless request subframes

Several of the *typeCode* values are used to label response packets, as specified in Table C.2. The meaning of the other *typeCode* values depends on their *typeSpecifier* context, as discussed below.

Table C.2 – typeCode values

Value	Name	Description
$0\text{-}FD_{16}$	—	Distinctive subframe identifiers for request subframes
FE_{16}	REJECTED	Common label for rejected response subframes
FF_{16}	RESPONSE	Common label for accepted response subframes

For non-RESPONSE *typeCode* values, the 48-bit *typeSpecifier* context is concatenated with the 8-bit ***typeCode*** value, to generate the 56-bit *typeSpecifier* value that identifies these subframe type.

C.2.2 Basic subframe headers

The basic subframe has a one-quadlet header that specifies the format and size of the following data, as illustrated in Figure C.3.

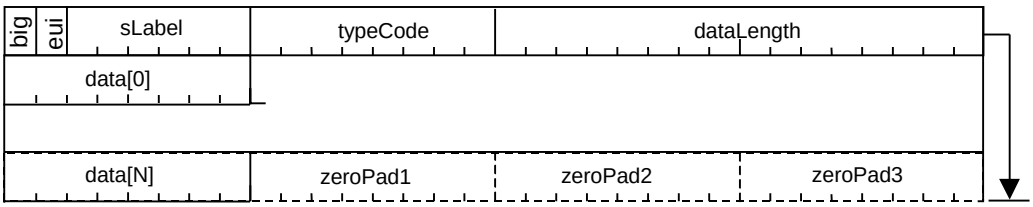

Figure C.3 – Basic subframe format

For these subframe formats, the ***big*** and ***eui*** bit values shall be zero. The 6-bit ***sLabel*** (subframe label) and ***typeCode*** values are defined in C.2.1.

The 16-bit ***dataLength*** specifies the number of data bytes in the remainder of the subframe. Note that the data bytes are padded, when necessary, so that the next subframe addresses remains quadlet aligned.

C.2.3 Extended length header

Distinctive *big=1* and *eui*=0 bits identify the subframe headers that contain an extended data-length values, as illustrated in Figure C.4.

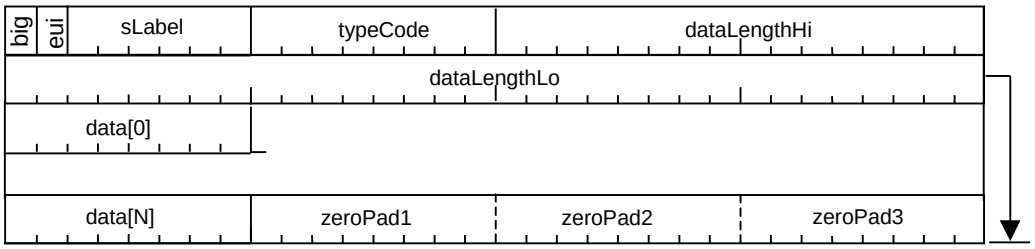

Figure C.4 – Extended packet formats

The 6-bit ***sLabel*** (subframe label) and ***typeCode*** values are defined in C.2.1.

The 16-bit ***dataLengthHi*** and 32-bit ***dataLengthLo*** fields are concatenated to generate a dataLength value.

The 48-bit *dataLength* value specifies the number of data bytes in the remainder of the subframe. Note that the data bytes are padded, when necessary, so that the next subframe addresses remains quadlet aligned.

C.3 Extended identifier header

Distinctive *big=0* and *eui*=1 values identify the subframe headers that contain a context-free *typeCode* value, as illustrated in Figure C.5. The meaning of these *typeCode* values depends on the header's 64-bit specifier value.

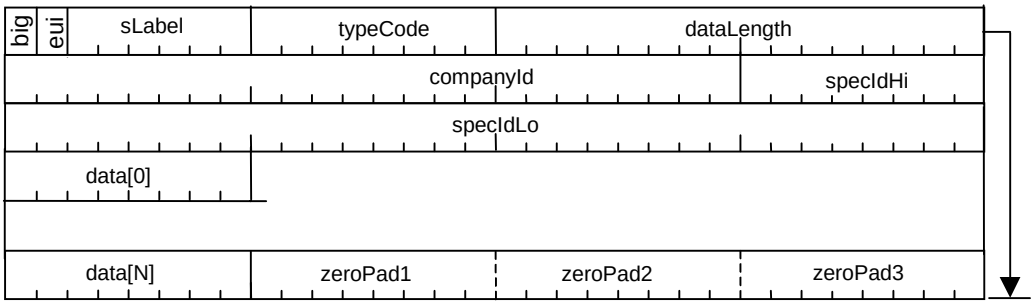

Figure C.5 – Extended identifier format

The 6-bit ***sLabel*** (subframe label) and ***typeCode*** values are defined in C.2.1.

The 16-bit ***dataLength*** specifies the number of data bytes in the remainder of the subframe. Note that the data bytes are padded, when necessary, so that the next subframe addresses remains quadlet aligned.

A 24-bit ***companyId***, 8-bit ***specIdHi***, and 32-bit ***specIdLo*** values are concatenated to form a specifierId. The 64-bit *specifierId* value uniquely identifies the standard that defines the meaning of the subframe's ***typeCode*** value.

C.3.1 Fully extended header

Distinctive *big=1* and *eui*=1 values identify the subframe headers that contain a large length specifier and a context-free *typeCode* value, as illustrated in Figure C.5.

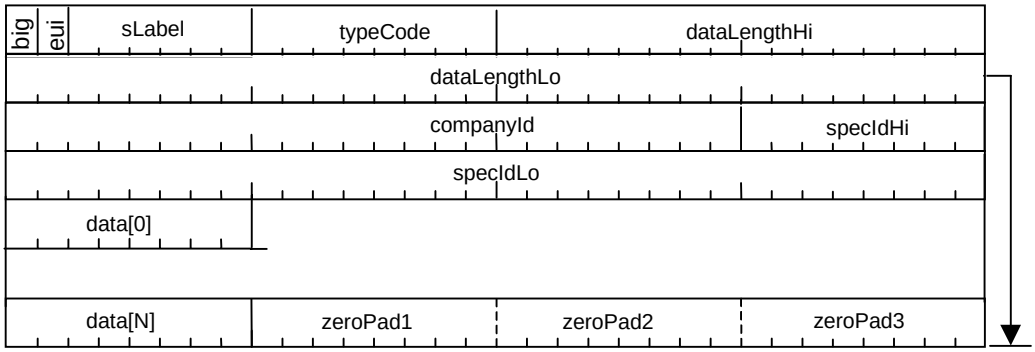

Figure C.6 – Fully extended packet formats

The 6-bit ***sLabel*** (subframe label) and ***typeCode*** values are defined in C.2.1.

The 16-bit ***dataLengthHi*** and 32-bit ***dataLengthLo*** fields are concatenated to generate a dataLength value.

The 48-bit *dataLength* value specifies the number of data bytes in the remainder of the subframe. Note that the data bytes are padded, when necessary, so that the next subframe addresses remains quadlet aligned.

A 24-bit ***companyId***, 8-bit ***specIdHi***, and 32-bit ***specIdLo*** values are concatenated to form a specifierId. The 64-bit *specifierId* value uniquely identifies the standard that defines the meaning of the subframe's ***typeCode*** value.

Annex D: State machine communications for automatic disconnection

This section describes the communications between asynchronous serial bus connections state machine and AV/C command state machine defined in References [R8] and [R9].

D.1 ALLOCATE_ATTACH_FRAME disconnection sequence

The following figure indicates the disconnection sequence of ALLOCATE_ATTACH_FRAME subfunction.

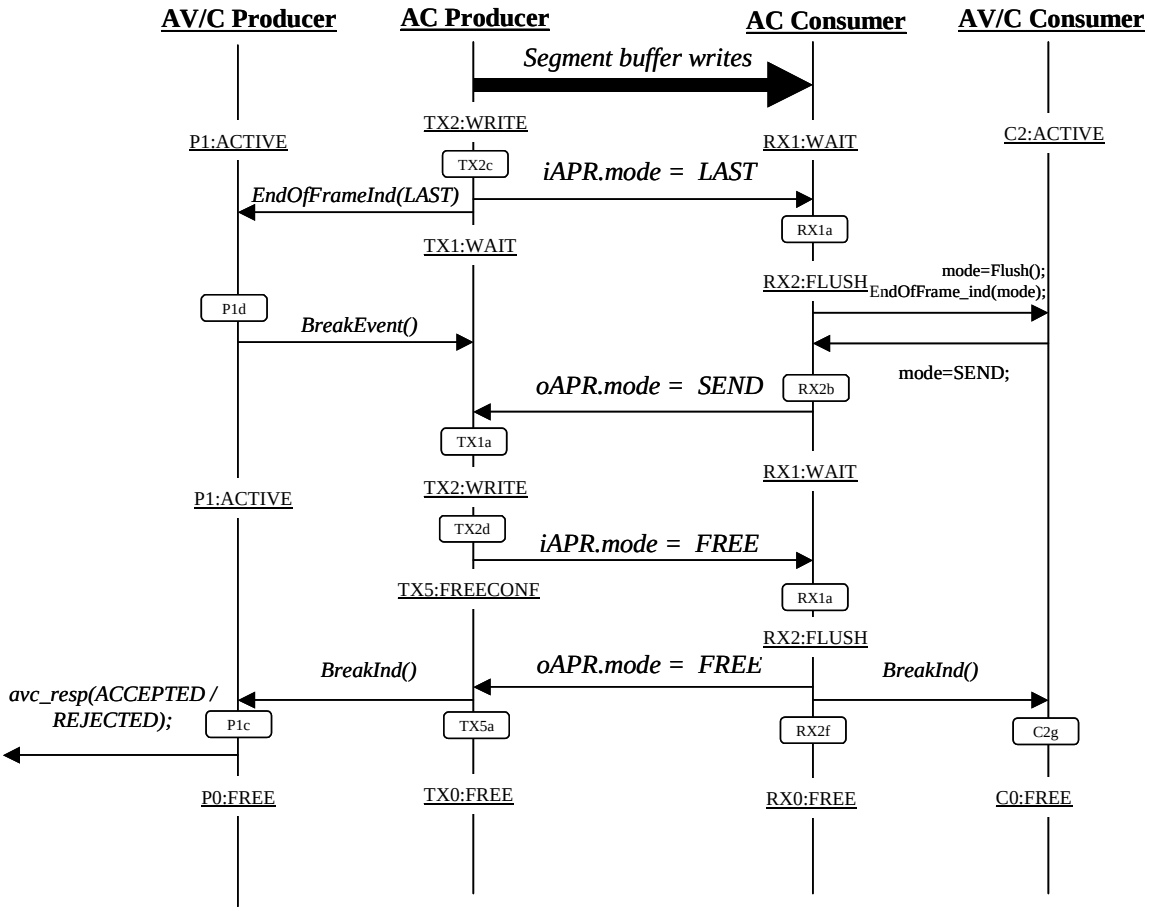

Figure D.1 – ALLOCATE_ATTACH_FRAME disconnection sequence

D.2 ATTACH_FRAME disconnection sequence

The following figure indicates the disconnection sequence of ATTACH_FRAME subfunction.

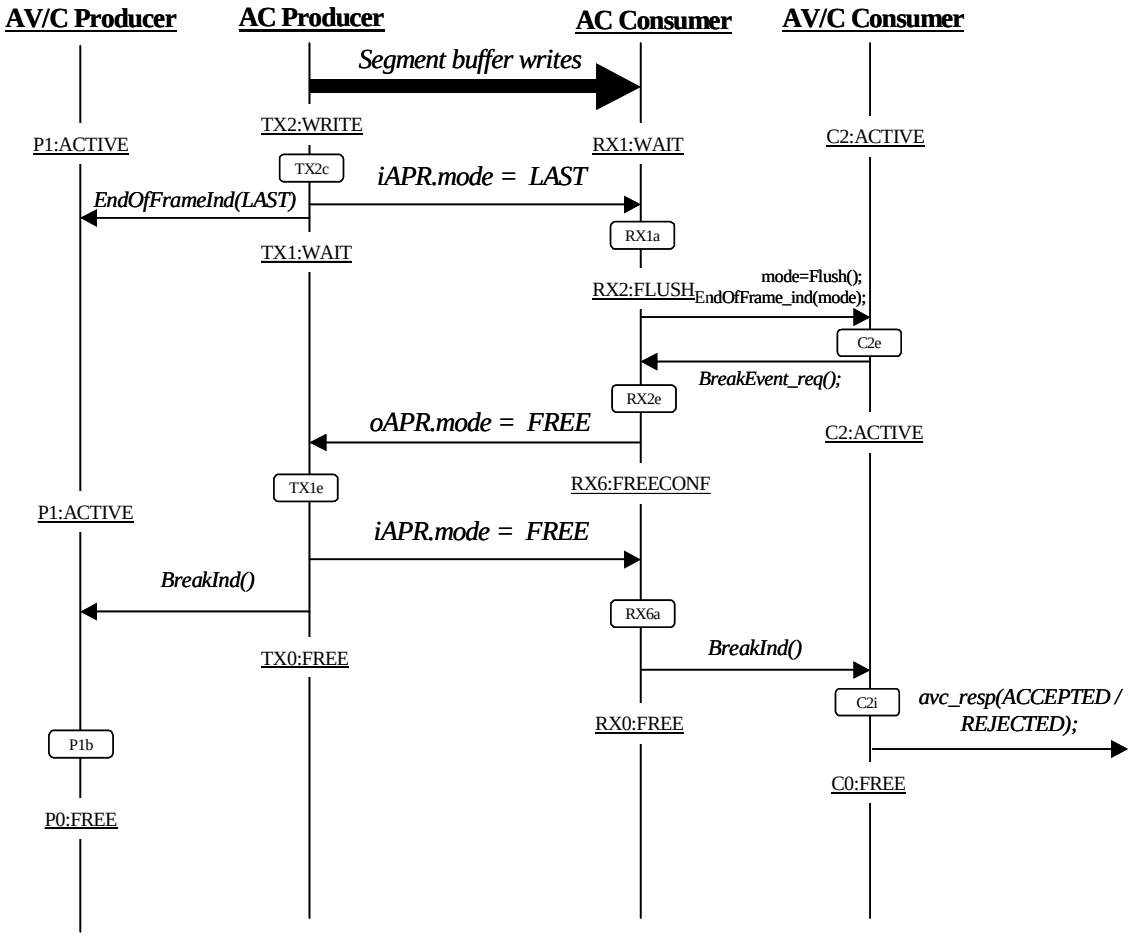

Figure D.2 – ATTACH_FRAME disconnection sequence