FROM:	Andy Green Peter Johansson
TO:	T10 SBP-3 working group
DATE:	August 2, 2001
RE:	Multi-processor enhancements to FAST_START

This document is a revision of the FAST_START enhancement proposal first suggest by David Wooten, next discussed at the Colorado Springs T10 meeting and subsequently described by Andy Green in 01-248r0. Most of the changes reflect editorial considerations in preparation for addition of the material to the SBP-3 working draft, but there are also minor technical changes.

In Colorado Springs the working group tentatively agreed that the target could cache “fast start” packets received out of order, but conversations at the 1394 Trade Association meeting in Vancouver, BC between Messrs. Anderson, Fuller, Johansson and Wooten exposed a flaw in the reasoning. In short, we concluded that a target fetch agent could make safe use of “fast start” packets only if the initiator guarantees their in-order delivery or if the previous ORB pointer equals the current value of the ORB_POINTER register. Because of difficulties with unordered delivery and processing of both “fast start” ORBs and fetched ORBs, it is not possible for a target to accumulate ORBs (with their associated previous and next ORB pointers) and later link them into a coherent list.

6.4.7 FAST_START register

The FAST_START register permits an initiator to signal a new task to an idle fetch agent by means of a single block write request addressed to the register. This write-only register shall support block write requests whose *destination_offset* is equal to the address of the FAST_START register and whose *data_length* is a multiple of four and less than or equal to the vendor-dependent size of the register (see 7.6.10) but shall reject all other requests. The format of this register is illustrated below.

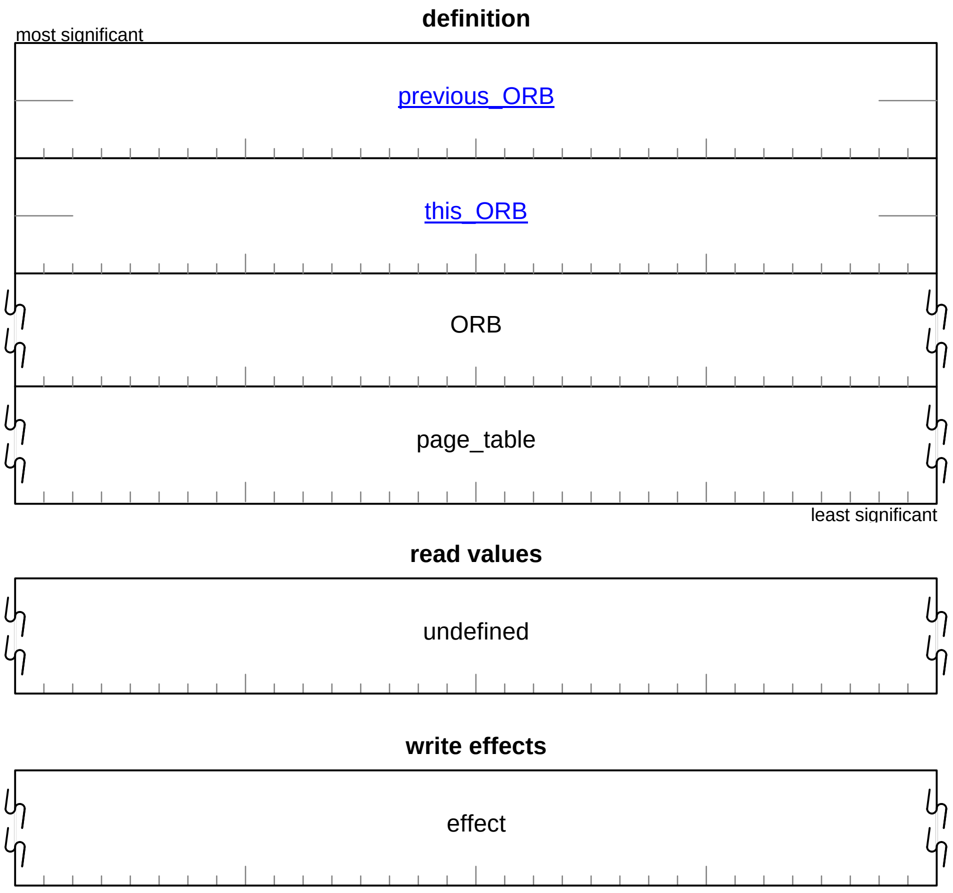

Figure 46 – FAST_START format

The *previous_ORB* field shall conform to the address pointer format illustrated by Figure 12 and shall either be a null pointer or reference an ORB in initiator memory whose *next*_ORB field is equal to the *this*_ORB field in the block write request addressed to the FAST_START register. When *previous_ORB* is not a null pointer, the ORB's Serial Bus address shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as a result of login) and the *previous_ORB* field.

The ~~*ORB_offset_hi* and *ORB_offset_lo* fields together form an *ORB_offset* field which shall reference~~ *this_ORB* field shall conform to the address pointer format illustrated by Figure 12 and shall contain the address of an ORB in initiator memory whose contents are identical to the *ORB* field in the block write request addressed to the FAST_START register. The ~~corresponding~~ ORB's Serial Bus address shall be formed from the concatenation of the 16-bit node ID of the initiator (available to the target as a result of login) and the ~~*ORB_offset*~~ *this_ORB* field ~~and two least significant bits of zero~~.

The *ORB* field shall contain an ORB whose format conforms to those specified by 5.1. The length of the ORB is variable and shall be determined by the combination of *rq_fmt* and the type of fetch agent (normal or stream) associated with the FAST_START register. An initiator shall not address a block write request to the FAST_START register whose *data_length*, in bytes, is less than eight plus the size of the *ORB* field. The target shall reject a block write request addressed to the FAST_START register if its *data_length*, in bytes, is less than ~~eight~~ sixteen plus the size of the *ORB* field.

The *page_table* field, if present, shall immediately follow the *ORB* field. If the format of the *ORB* field includes a nonzero *page_table_present* bit, the *page_table* field shall contain zero or more page table entries whose order and content shall be identical to those contained within the page table referenced by the *data_descriptor* field in the ORB referenced by ~~*ORB_offset*~~ *this_ORB*. The *page_table* field may be a subset of the page table referenced by the ORB, but no partial page table entries shall be present (see 5.2). The target shall derive the number of immediately available page table entries from the *data_length* of the block write request addressed to the FAST_START. The number of page table entries is limited by the maximum size of the FAST_START register.

~~The *data_length* of block write requests addressed to the FAST_START register shall be a multiple of four.~~

The effects of a write transaction to the FAST_START register are dependent upon the value of its *previous_ORB* field and the value of *st* in the associated AGENT_STATE register. If the ~~target~~ fetch agent is in the DEAD state, writes to the FAST_START register shall be ignored. If the ~~target~~ fetch agent is in the ACTIVE state, a write to the FAST_START register shall be interpreted as if it were a quadlet write request addressed to the fetch agent's DOORBELL register (the data payload shall be ignored) ~~register may cause unpredictable target behavior~~. Otherwise, when the ~~target~~ fetch agent is in the RESET or SUSPENDED state, the value of the *previous_ORB* field determines the effect of a write to this register ~~shall cause *ORB_offset*~~. If *previous_ORB* contains a null pointer, *this_ORB* shall be stored in the associated ORB_POINTER register, the *ORB* and *page_table* fields shall be stored in the target's working set and the agent shall transition to the ACTIVE state. When *previous_ORB* is not null, the target may perform these actions if and only if *previous_ORB* is equal to the fetch agent's ORB_POINTER register. See 9.1.5 for a precise definition of fetch agent state transitions that involve the FAST_START register.

9.1.4 Use of the FAST_START register (informative)

An initiator ~~aware that a fetch agent is in either the RESET or SUSPENDED state~~ may signal new task(s) to the fetch agent by a block write request addressed to the fetch agent's FAST_START register (see 6.4.6). The block write request contains ~~the address of~~ pointers both to the ORB to be commenced and to the previous ORB (*i.e.*, the ORB whose *next_ORB* field references the ORB to be commenced), a copy of the ORB itself and, optionally, page table data associated with the ORB.

Significant overhead reductions may result from the use of the FAST_START register, since the target need not fetch either the address of the ORB or the ORB itself. In cases where the block write request contains the entire page table, the target need not fetch the page table; even if the entire page table is not written to the FAST_START register (it may be too large), the target may significantly reduce startup latency by fetching the remaining page table entries concurrently with task execution.

Although an initiator may achieve optimal performance improvement by writing to the FAST_START register when the fetch agent is in either the RESET or SUSPENDED state, the register may also be used when the fetch agent is active. In this case, the target ignores the data payload of the block write request and behaves as if a quadlet write had been addressed to the fetch agent's DOORBELL register. There are several ways by which an initiator may securely know that a fetch agent is in the RESET or SUSPENDED state. If the initiator has not written to either of the fetch agent's ORB_POINTER or FAST_START registers since the most recently completed login or reconnect operation or the most recent write to the fetch agent's AGENT_RESET register, the fetch agent is in the RESET state. Similarly, if the initiator has not written to either of the fetch agent's ORB_POINTER or FAST_START registers since the target last stored a status block with a *src* field equal to one (see 5.3.1), the fetch agent is in the SUSPENDED state.

NOTE – An initiator may use the above methods to deduce fetch agent state whether or not the target implements the FAST_START register. If the register is not supported, startup latency for an idle fetch agent may be reduced by writing the address of an ORB directly to the ORB_POINTER register instead of a write to the DOORBELL register.

There are two variants to the use of the FAST_START register, one suitable for single-threaded initiators and the other suitable for multi-threaded (possibly multiprocessor) initiators. If the initiator implementation guarantees that no more than one block write request to the FAST_START register is attempted while the fetch agent is idle, it may set the *previous_ORB* field to a null pointer; this causes the idle fetch agent to unconditionally update the ORB_POINTER register with the value of *this_ORB*. Multi-threaded initiators may not be able to satisfy this constraint for ordered writes to the FAST_START register, in which case the method outlined below may be used:

a) Construct the ORB (with a null *next_ORB* field) and associated data structures in system memory. The address of the ORB is designated *this_ORB*;

b) In an effectively atomic operation (*i.e.*, one protected within a critical section), obtain the current tail pointer to the linked list of active ORBs, save it as *previous_ORB* and replace the tail pointer with *this_ORB*;

c) Store *this_ORB* in the *next_ORB* field of the ORB referenced by *previous_ORB*;

d) Initiate a block write request to the fetch agent's FAST_START register; its data payload should include *previous_ORB*, *this_ORB*, a copy of the ORB and, optionally, page table information.

The presence of a non-null *previous_ORB* field permits the fetch agent to ignore FAST_START write requests that arrive out of order.

Either a single ORB or a linked list of ORBs may be signaled in a single block write request to the FAST_START register, dependent upon the value of the *next_ORB* field in the ORB contained within the block write.[1] Once a successful completion response is received for the block write request, the initiator may append to the linked list of ORBs by the methods described in 9.1.2.

9.1.5 Fetch agent state machine

The operations of a target fetch agent are specified by the figure below. The state of a fetch agent is visible in the context displayed by the AGENT_STATE and ORB_POINTER registers described in 6.4. The state machine diagram and accompanying text explicitly specify the conditions for transition from one state to another and the actions taken within states.

The target shall qualify all writes to fetch agent CSRs by the *source_ID* of the currently logged-in initiator. A write to a fetch agent CSR by any other Serial Bus node shall be rejected by the target by one of the following methods:

– an acknowledgment of *ack_type_error*;

– an acknowledgment of *ack_complete* (although the write is ignored); or

– an acknowledgment of *ack_pending*. When the target subsequently responds, the response code shall be *resp_type_error*.

The recommended target action is to indicate a type error, either by an acknowledgment of *ack_type_error* or an acknowledgment of *ack_pending* followed by *resp_type_error*.

[1] When more than one ORB is signaled by a write to the FAST_START register, the algorithm described for multi-threaded operations is modified to update the linked list tail pointer with the address of the final ORB in the list to be appended rather than with the value of *this_ORB*.

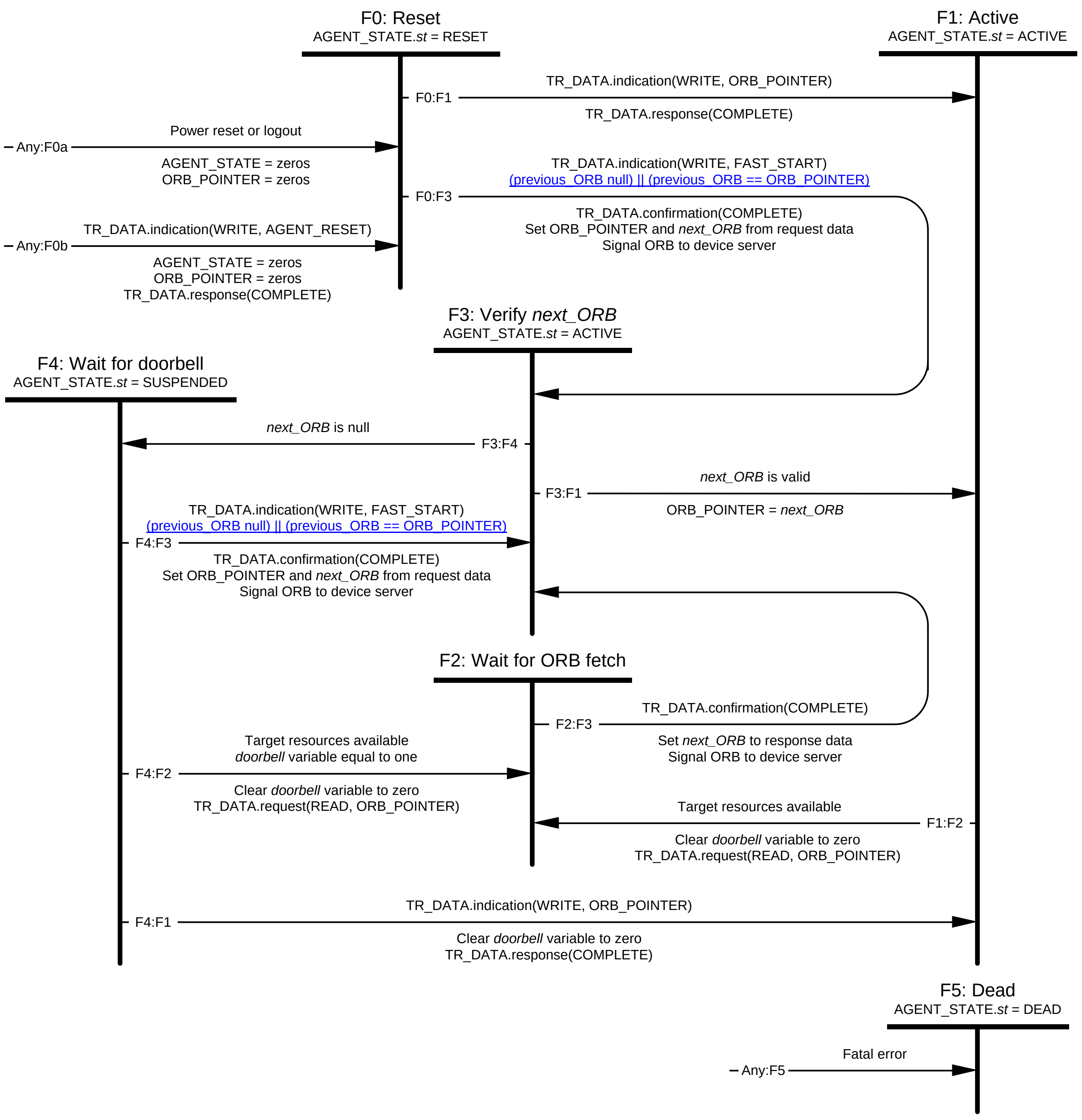
F0: Reset
AGENT_STATE.*st* = RESET
F1: Active
AGENT_STATE.*st* = ACTIVE
F0:F1
TR_DATA.indication(WRITE, ORB_POINTER)
TR_DATA.response(COMPLETE)
Any:F0a
Power reset or logout
AGENT_STATE = zeros
ORB_POINTER = zeros
F0:F3
TR_DATA.indication(WRITE, FAST_START)
(previous_ORB null) || (previous_ORB == ORB_POINTER)
TR_DATA.confirmation(COMPLETE)
Set ORB_POINTER and *next_ORB* from request data
Signal ORB to device server
Any:F0b
TR_DATA.indication(WRITE, AGENT_RESET)
AGENT_STATE = zeros
ORB_POINTER = zeros
TR_DATA.response(COMPLETE)
F3: Verify *next_ORB*
AGENT_STATE.*st* = ACTIVE
F4: Wait for doorbell
AGENT_STATE.*st* = SUSPENDED
next_ORB is null
F3:F4
F3:F1
next_ORB is valid
ORB_POINTER = *next_ORB*
TR_DATA.indication(WRITE, FAST_START)
(previous_ORB null) || (previous_ORB == ORB_POINTER)
F4:F3
TR_DATA.confirmation(COMPLETE)
Set ORB_POINTER and *next_ORB* from request data
Signal ORB to device server
F2: Wait for ORB fetch
F2:F3
TR_DATA.confirmation(COMPLETE)
Set *next_ORB* to response data
Signal ORB to device server
Target resources available
doorbell variable equal to one
F4:F2
Clear *doorbell* variable to zero
TR_DATA.request(READ, ORB_POINTER)
Target resources available
F1:F2
Clear *doorbell* variable to zero
TR_DATA.request(READ, ORB_POINTER)
TR_DATA.indication(WRITE, ORB_POINTER)
F4:F1
Clear *doorbell* variable to zero
TR_DATA.response(COMPLETE)
F5: Dead
AGENT_STATE.*st* = DEAD
Fatal error
Any:F5

Figure 68 – Fetch agent state machine

Transition Any:F0a. A power reset shall cause the fetch agent to transition to the RESET state from any other state. The AGENT_STATE and ORB_POINTER registers (that control and make visible the operations of the fetch agent) shall be reset to zeros.

Transition Any:F0b. A quadlet write request by the initiator to the AGENT_RESET register shall cause the fetch agent to transition to state F0 from any other state. The fetch agent shall zero the AGENT_STATE and ORB_POINTER registers before the transition to state F0. Transaction label(s) for outstanding request subaction(s) shall not be reused until either the corresponding response subaction completes or a split time-out expires; in the former case, the response data shall be discarded.

State F0: Reset. Upon entry to this state, the *st* field in the AGENT_STATE register shall be set to RESET. The fetch agent is inactive and available to be initialized by an initiator.

Transition F0:F1. An 8-byte block write of a valid *ORB_offset* to the ORB_POINTER register shall update the register and cause the fetch agent to transition to state F1. The target shall confirm the block write request with a response subaction of COMPLETE.

Transition F0:F3. A block write ~~of a valid *ORB_offset*, ORB and optional page table data~~ to the FAST_START register may affect the state of the fetch agent. If the *previous_ORB* field does not contain a null pointer and is not equal to the ORB_POINTER register, the fetch agent state shall not change and the target shall confirm the block write request with a response subaction of COMPLETE. Otherwise the fetch agent shall update the ORB_POINTER register with the value of ~~*ORB_offset*~~*this_ORB*, shall update the *next_ORB* variable with the contents of the *next_ORB* field from the ORB contained within the block write request and shall cause the fetch agent to transition to state F3. The fetch agent shall also make the ORB and any page table data available to the device server for execution. Once these actions are complete, the target shall confirm the block write request with a response subaction of COMPLETE.

NOTE – If the *previous_ORB* field is either null or equal to the ORB_POINTER register, a target may interpret a block write addressed to its FAST_START register as if the ~~first eight bytes~~ *this_ORB* field had been written to its ORB_POINTER register. Although it is less efficient to elect transition F0:F1, it is functionally equivalent to transition F0:F3.

NOTE – When the fetch agent is in the RESET state, it is not necessary to write to the DOORBELL register upon either transition F0:F1 or F0:F3.

State F1: Active. Upon entry to state F1, the *st* field in the AGENT_STATE register shall be set to ACTIVE. In this state, the fetch agent may use the address information in the ORB_POINTER register to fetch ORBs from the initiator as resources permit.

Transition F1:F2. The availability of target resources is an implementation-dependent decision. Typically, the resources might be space in device memory to hold an image of the ORB while the command is scheduled for execution and subsequently completed. In any case, the fetch agent clears the *doorbell* variable to zero and then issues a block read request to obtain the ORB from system memory.

State F2: Wait for ORB fetch. The fetch agent is suspended and awaiting a read response for a block read directed to the address contained in the ORB_POINTER register.

Transition F2:F3. Subsequent to a block read request, issued as described above, the fetch agent may accept a block read response that contains either the *next_ORB* data or an entire ORB intended for execution by the device server. If a read response is received whose *source_ID*, *destination_ID* and *tl* fields match the *destination_ID*, *source_ID* and *tl* fields, respectively, of the read request, the fetch agent shall copy the *next_ORB* field from the response data to the *next_ORB* variable before making the transition to state F3. When the response data contains an entire ORB not yet in the device server's working set, the fetch agent shall make the ORB available to the device server for execution.

State F3: Verify *next_ORB*. Upon entry to state F3, the *st* field in the AGENT_STATE register shall be set to ACTIVE.[2] The *next_ORB* variable contains information about a subsequent ORB that may be linked in order after the one just fetched. As described in 5.1, the *next_ORB* pointer encodes the address of the next ORB. The actions of this state determine whether or not the *next_ORB* pointer is null.

Transition F3:F1. If the *next_ORB* variable does not indicate a null pointer the fetch agent shall update the ORB_POINTER register with the value of *next_ORB*.

[2] Although this action is redundant in the case of transition F2:F3, it is necessary for transitions F0:F3 and F4:F3.

Transition F3:F4. The fetch agent shall transition to a suspended state, F4, if *next_ORB* contains a null pointer. A null pointer is defined in 5.1 and exists if the most significant bit of the variable is one.

State F4: Wait for doorbell. Upon entry to state F4, the *st* field in the AGENT_STATE register shall be set to SUSPENDED. The fetch agent is suspended; the ORB_POINTER register contains the address of the ORB whose *next_ORB* field was null at the time state F4 was entered.

Transition F4:F1. If an indication of a write to the ORB_POINTER register is received, the fetch agent shall clear the *doorbell* variable to zero and confirm the write transaction with a response subaction of COMPLETE. After the confirmation, the fetch agent shall transition to state F1.

Transition F4:F2. Whenever the *doorbell* variable is equal to one, the fetch agent shall clear the *doorbell* variable to zero, issue a read request to obtain a fresh copy of the *next_ORB* field from the ORB whose address is contained in the ORB_POINTER register and then transition to state F2. The *doorbell* variable is set to one as the result of a quadlet write request of any value to the DOORBELL register, whether the write request is received in this or any other state.

The fetch agent may issue either an 8-byte block read request (to fetch just the *next_ORB* field) or it may reread the entire ORB. The initiator shall insure that system memory occupied by the ORB remains accessible, as described in 9.3.

Transition F4:F3. A block write ~~of a valid *ORB_offset*, ORB and optional page table data~~ to the FAST_START register may affect the state of the fetch agent. If the *previous_ORB_* field does not contain a null pointer and is not equal to the ORB_POINTER register, the fetch agent state shall not change and the target shall confirm the block write request with a response subaction of COMPLETE. Otherwise the fetch agent shall update the ORB_POINTER register with the value of ~~*ORB_offset*~~ *this_ORB*, shall update the *next_ORB* variable with the contents of the *next_ORB* field from the ORB contained within the block write request and shall cause the fetch agent to transition to state F3. The fetch agent shall also make the ORB and any page table data available to the device server for execution. Once these actions are complete, the target shall confirm the block write request with a response subaction of COMPLETE.

NOTE – If the *previous_ORB_* field is either null or equal to the ORB_POINTER register, a target may interpret a block write addressed to its FAST_START register as if the ~~first eight bytes~~ *this_ORB* field had been written to its ORB_POINTER register. Although it is less efficient to elect transition F4:F1, it is functionally equivalent to transition F4:F3.

Transition Any:F5. Upon the detection of any fatal error, the fetch agent shall transition to state F5. Examples of fatal errors include, but are not limited to:

- the failure of the addressed node to acknowledge a read request;
- the failure of the addressed node to respond to a read request (split time-out);
- a busy condition at the addressed node that exceeds the target's busy retry limit;
- a data CRC error in a response subaction.

Some of these errors may be recoverable if retried by the target.

The fetch agent may also be instructed to transition to the dead state as a result of an error in command execution detected by the device server.

State F5: Dead. The dead state is a unique state that preserves fetch agent information in the AGENT_STATE and ORB_POINTER registers. Writes to any fetch agent register except AGENT_RESET shall have no effect while in state F5.

E.2 ORB_POINTER or FAST_START write request

A consequence of a write to either the ORB_POINTER or FAST_START register ~~is valid only~~ when the target fetch agent is in the RESET or SUSPENDED state~~. A consequence of the write~~ is that, if successful, the target fetch agent transitions to the ACTIVE state. If no acknowledgement is received by the initiator after a write to either the ORB_POINTER or FAST_START register when the fetch agent is in either of these states, the initiator should not retry the write. The recommended method for error recovery is a write to the AGENT_RESET register. An exception is a write to the FAST_START register with a non-null *previous_ORB* field; because the target compares the *previous_ORB* field to the ORB_POINTER register, the write may be retried.