I/O Controller Data Integrity Extensions

Martin K. Petersen, Oracle Linux Engineering <martin.petersen@oracle.com>

Introduction

In order to provide customers with increased data integrity protection, Oracle has worked with industry partners to develop an integrity protection scheme that is standards-based and not specific to the Oracle database.

Work within the T10 committee that governs the SCSI protocols led to the ratification of the Protection Information Model extensions to the SCSI block protocol. The T10 Protection Information Model (also known as Data Integrity Field, or DIF) provides means to protect the communication between host adapter and storage device. However, the T10 specification only describes the protocol between the host adapter and the storage device. How the operating system interacts with the I/O controller is outside the scope of T10.

The Data Integrity Extensions described in this document specify how an I/O controller should interact with the host operating system to engage in exchange of protection information, thus enabling end-to-end data integrity.

T10 Protection Information Model

Enterprise drives using the SCSI protocol have long had the capability to be reformatted to bigger sector sizes such as 520 bytes. The extra 8 bytes of information per sector have traditionally been used by array firmware to store integrity information proprietary to the array. The T10 PIM was introduced as a way to use those extra bytes in an open and standardized fashion.

The 8 bytes of protection information are divided up as follows:

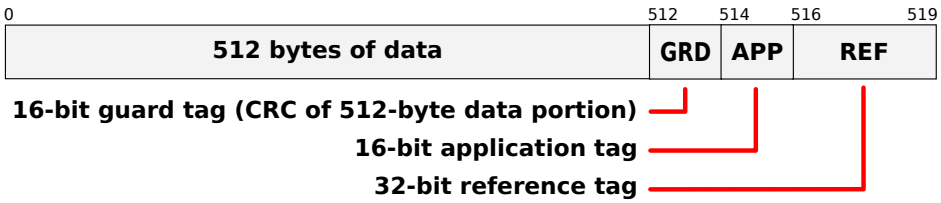

Figure 1 T10 Data Integrity Fields

The guard tag protects the data portion of the sector. The application tag is simply opaque storage. And finally, the reference tag is being used to protect against out-of-order and misdirected write scenarios.
Standardizing the contents of the protection information enables all nodes in the I/O path, including the disk itself, to verify the integrity of the data block.

Comparison of I/O Paths

A typical I/O submission scenario in an enterprise configuration is illustrated in figure 2. The only entity capable of using the 8 bytes of protection information is the array firmware.

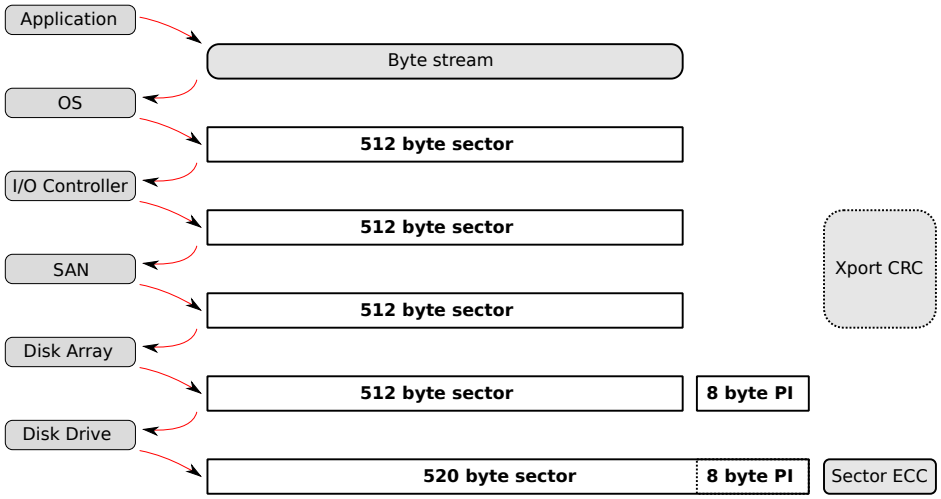

Figure 2 Normal I/O write: Application writes byte stream to OS. Filesystem writes in logical blocks that are multiples of 512-byte sectors. Depending on physical transport, a CRC may be applied on the wire. Array firmware generates 8 bytes of proprietary protection information. Disk stores 520-byte sectors and generates its own CRC.

A similar T10 PI-enabled configuration will look like figure 3. The I/O controller generates and appends the protection information and every subsequent node in the I/O path can verify the data integrity.

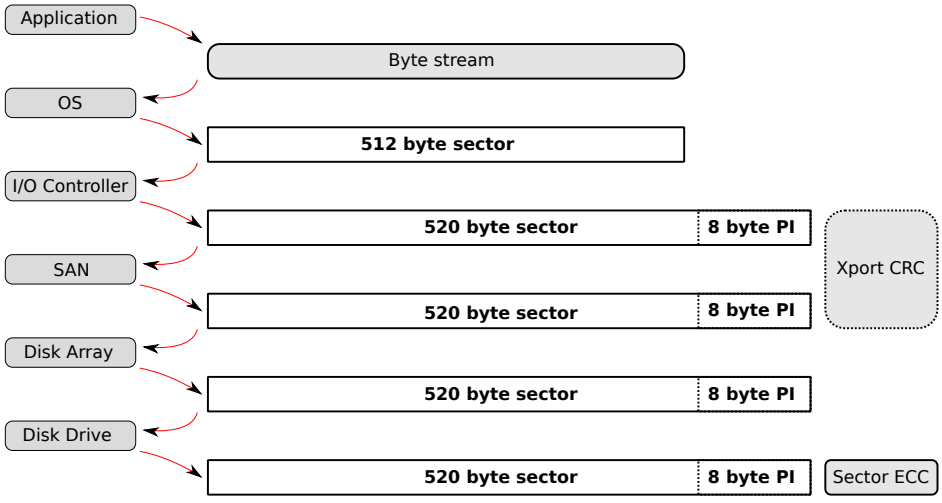

Figure 3 T10 PI I/O write: Application writes byte stream to OS. Filesystem writes in logical blocks that are multiples of 512-byte sectors. HBA generates protection information and sends out 520-byte sectors. SAN switch can optionally check protection information. Array firmware verifies protection information, optionally remaps reference tags and writes to disk. Disk verifies protection information before storing request.

Combining T10 PIM with the Data Integrity Extensions allows the protection information to be attached even higher up in the stack–either in the application or in the operating system. The entire I/O path is protected and true end-to-end data integrity protection is achieved.

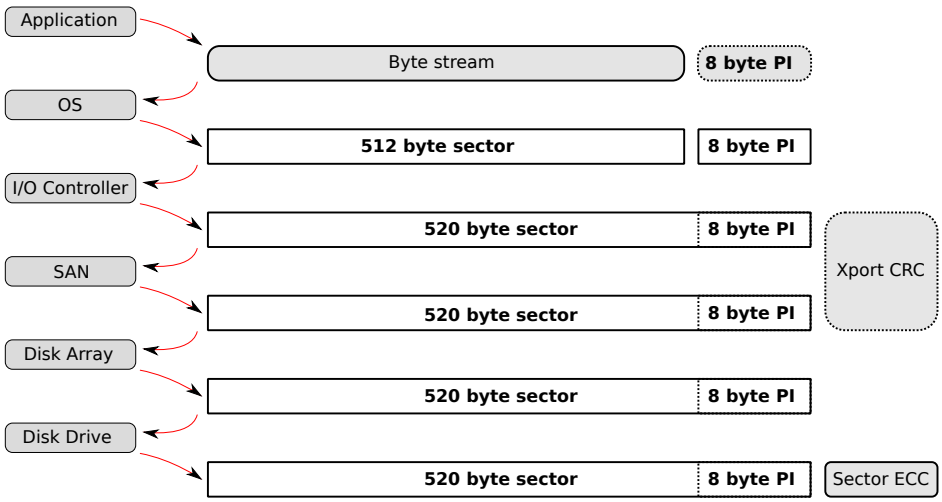

Figure 4 Combined DIX + T10 PI I/O write: Application writes byte stream to OS, optionally including protection information. Filesystem writes in logical blocks that are multiples of 512-byte sectors. If no protection information has been generated, OS automatically does so and attaches it to the I/O. HBA verifies data integrity, merges data and protection scatterlists and sends out 520-byte sectors. The remainder of the I/O path is similar to the T10 PI example above.

Figure 5 illustrates the protection coverage for the protection schemes mentioned above:

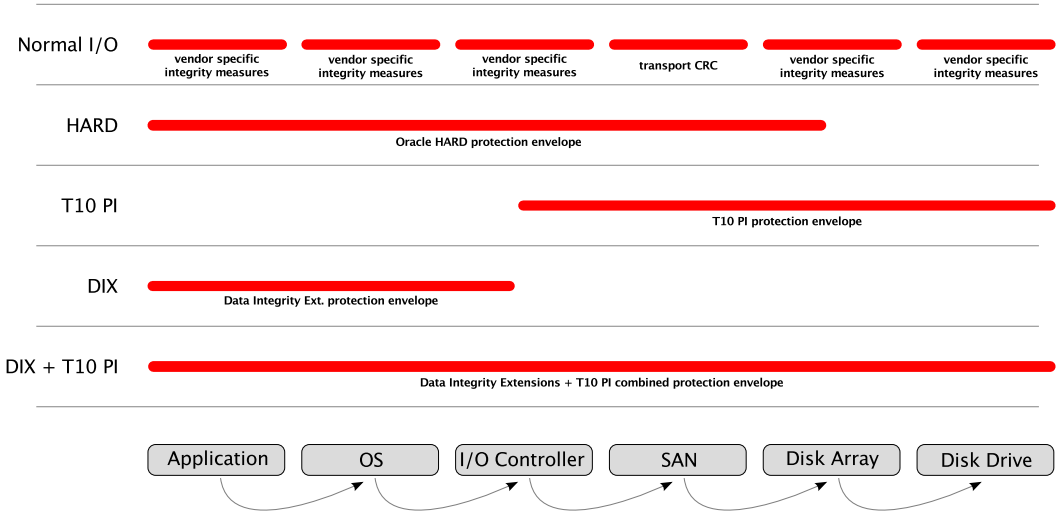

Figure 5 The Normal I/O line illustrates the disjoint integrity coverage offered using a current operating system and standard hardware. The HARD line shows the protection envelope offered by the Oracle Database accessing a disk array with HARD capability. T10 PI shows coverage using the integrity portions of the SCSI protocol. DIX shows the coverage offered by the Data Integrity Extensions. And finally, DIX + T10 PI illustrates the full coverage provided by the Data Integrity Extensions in combination with T10 PI.

1 Data Integrity Extensions

As shown above, in order to facilitate true end-to-end data integrity between application and storage device, integrity metadata must be made accessible to the operating system. We have developed a model in which the T10 protection information can be transferred to and from the operating system using standard DMA.

While the functionality of the operating system-level interfaces can be implemented on top of most currently available controllers, the Data Integrity Extensions have been designed to provide extra features that drastically improve performance. Most of these features are optional but if they are not present they put additional load on the system processor. We encourage vendors to implement the full feature set described below.

1.1 Scatter-Gather List Separation

On the wire between initiator and target the T10 SCSI Block Commands specification mandates that the protection information is interleaved with the data sectors. For instance 512 bytes of data followed by 8 bytes of protection information. This is very inconvenient for the operating system which usually stores application data in 4KB pages. Doing memory copies to interleave data and protection data wastes system resources. Our benchmarking shows that manual interleaving by way of scatterlists also impacts system performance.

Consequently, in our model interleaving of data and protection information has been delegated to the I/O controller. The controller driver will receive a request with two scatter-gather lists attached. One containing the data in/out buffer as usual and one containing the matching protection data in/out buffer.

When a WRITE request is received from the operating system the controller must verify that the data buffer and the protection buffer are in agreement and subsequently interleave the two buffers to get the block size mandated by the T10 Protection Information Model.

When a READ request is received from the storage device the controller must verify that the data portions match the protection information, split the blocks and DMA to the data and protection buffers respectively.

The scatter-gather list elements of the protection buffer will honor the same alignment constraints as those of the data buffer.

1.2 Protection Data Endianness

The protection information format is network-endian as per T10. This includes the protection data exchanged with the host.

1.3 Guard Tag Format

The format of the guard tag between initiator and target is specified in the T10 SBC standard as a cyclic redundancy check using a well-defined polynomium. A CRC is expensive to calculate in software and benchmarks show that it has a significant impact on the performance of a system.

Various algorithms were tested and the IP checksum was selected for the first implementation of this. The IP checksum is cheap to calculate and also has a few other properties that make it suitable for this purpose.

It should be noted that the T10 CRC checksum is mandatory and the alternate guard tag format is an optional feature. An I/O driver, utility or management tool can indicate to the operating system whether to use the T10 CRC, the IP checksum or (in the future) another format. The operating system is responsible for negotiating the best performing algorithm in the case of for instance multipathing using controllers with different checksum capabilities.

This negotiation capability is also used to pick a suitable checksum format when a software RAID device spans devices with different hardware sector size or guard tag capabilities.

The guard tag type is required to be a per-request property, not a global setting.

1.4 Reference Tag Remapping

For T10 SBC Type 1 devices the reference tag is defined to contain the lower 32-bits of the target LBA. For Type 2 devices the reference tag is an incrementing counter seeded in the 32-byte CDB. When using the Type 2 and Type 3 protection schemes the protection information may be provided with finer granularity than the device logical block size. The granularity (in bytes) is supplied in `dix_protection_interval`.

When an I/O request moves between layers the address space used for the reference tag may change. For instance an application may submit requests with the individual blocks numbered $0..n$. A filesystem may want the reference tag to indicate filesystem blocks, etc. Therefore transitions between layers in the I/O stack often involve remapping the reference tag from one linear address space to another. To facilitate this, a DIX-capable I/O controller can implement remapping functionality.

The OS will provide the I/O controller driver with the initial reference tag (`dix_ref_tag_in`). This value will be used to verify the protection received by the controller. An outbound initial reference tag will also be provided (`dix_ref_tag_out`). This value indicates the first reference tag that is to be sent to the host or the target. After verifying the received protection information the I/O controller must remap the reference tags starting with this value.

1.5 Checking and Error Handling

T10 SBC defines a set of flags that describe which of the three protection information tags need to be checked by the target storage device (`RDPROTECT` / `WRPROTECT`).

A similar but more comprehensive set of parameters are provided by DIX. These parameters indicate which of the tags in the protection information need to be verified by the controller firmware.

In case of a mismatch in the protection data the controller must return a suitable error to the operating system. The error notification must include in which request, at which reference tag offset the mismatch occurred, and whether it was a guard, reference or an application tag failure.

1.6 DIX Operations

In the case of both `READ` and `WRITE` requests the CDB passed to the I/O controller has a `RDPROTECT` / `WRPROTECT` field which indicates whether to transfer protection data between initiator and target. Given the OS-to-I/O controller interface is outside the scope of T10, there are no similar controls for this piece of the I/O path.

The OS will always prepare a CDB with appropriate `RDPROTECT` / `WRPROTECT` information depending on target format and capabilities. The request passed to the controller driver will also include information about which protection type the target has been formatted with, and which checksum is used for communication between OS and I/O controller.

Finally, the OS will also provide the I/O controller driver with an operation code which tells the controller which type of I/O to perform. The operation codes are:

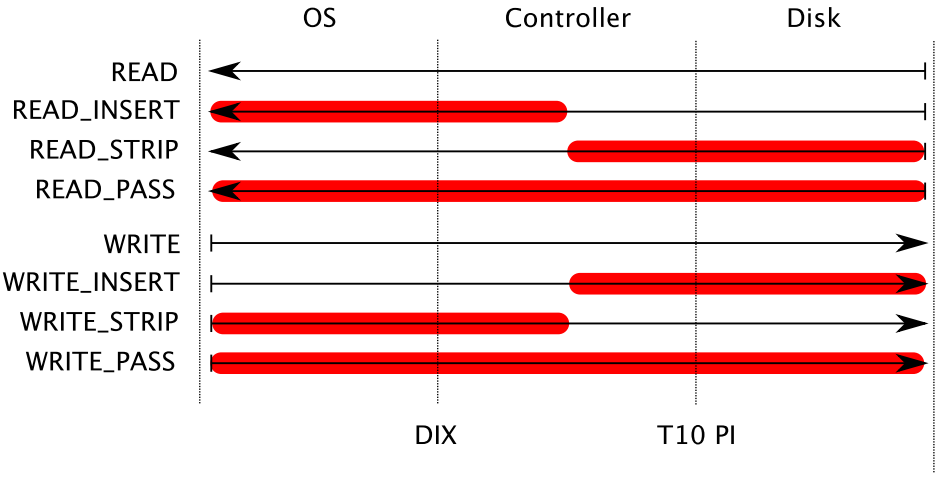

Figure 6 DIX operations: Red bars indicate protected path.

- READ and WRITE are for normal (unprotected) I/O
- READ_INSERT Read data from target, generate protection data and transfer both to OS.
- WRITE_STRIP Transfer data and protection data from OS, verify data integrity, discard protection data and write data to target
- READ_STRIP Read data and protection data from target, verify data integrity, discard protection data and transfer data to OS
- WRITE_INSERT Transfer data from OS, generate protection data and write both data and protection data to target
- READ_PASS Read data + protection data from target, optionally verify protection data, optionally convert checksum, optionally remap reference tag, transfer both data and protection data to OS
- WRITE_PASS Transfer data + protection data from OS, optionally verify protection data, optionally convert checksum, optionally remap reference tag, write both data and protection data to target

READ_INSERT and WRITE_STRIP are for DIX in combination with legacy (non-PI) devices. These two operations enable protection between OS and controller only. The contents of the guard tag and the reference tag are defined to be the same as in Type 1. Given that there is no place to persistently store the application tag the contents of the application tag must be ignored.

READ_STRIP and WRITE_INSERT are for regular T10 PI operation where no protection information is exchanged with the operating system.

READ_PASS and WRITE_PASS are for passing protection information through to the target. Checksum conversion may be taking place in the I/O controller if something other than the T10 CRC is used when communicating with the host.

1.7 Protection Modes

The controller driver must indicate its T10 PI and DIX capabilities to the operating system so that requests can be prepared accordingly. The following protection modes are defined:

Mode	Description
T10 PI Type 0	Normal, unprotected I/O.
T10 PI Type n	Protection supported between controller and a storage device formatted with T10 Type n protection information. The controller must implement the READ_STRIP and WRITE_INSERT operations

DIX Type 0	Protection DMA enabled between OS and controller. Storage device is not formatted with protection information. The controller must implement the READ_INSERT and WRITE_STRIP operations
DIX Type n	Protection DMA enabled between OS and controller. Storage device is formatted with T10 PI Type n. The controller must implement the READ_PASS and WRITE_PASS operations.

2 Parameters Affecting the I/O Execution

The operating system will provide the I/O controller driver with a set of flags and parameters that indicate how each I/O request should be handled. These are in addition to the information normally communicated by the OS like scatter-gather list descriptors, buffer length, command buffer, etc.

2.1 DIX_GUARD_CHECK

Type: Flag

If this flag is set then the I/O controller shall verify the received data buffer against the guard tags in the received protection information. If DIX_GUARD_CHECK is unset then the guard tag shall not be checked.

2.2 DIX_GUARD_IP_CHECKSUM

Type: Flag

If this flag is set then all guard tags exchanged with the operating system shall be in IP checksum format. This implies a conversion step for the READ_PASS/WRITE_PASS commands. If DIX_IP_CHECKSUM is unset then the guard tag exchanged with the operating system is in T10 PI CRC format.

2.3 DIX_REF_CHECK

Type: Flag

If this flag is set then the I/O controller shall verify the reference tag against the dix_ref_tag_in parameter. If DIX_REF_CHECK is unset then the reference tag shall not be checked.

2.4 DIX_REF_ESCAPE

Type: Flag

If this flag is set then the I/O controller shall not check blocks whose reference tag contains $0xFFFFFFFF$. This flag is only valid when DIX_APP_ESCAPE is also set.

2.5 DIX_REF_INCREMENT

Type: Flag

If this flag is set then the I/O controller shall increment the dix_ref_tag_in value by 1 for every block (dix_protection_interval bytes of data).

If this flag and DIX_REF_REMAP are set then the I/O controller shall increment the dix_ref_tag_out value by 1 for every block (dix_protection_interval bytes of data).

2.6 DIX_REF_REMAP

Type: Flag

If this flag is set then the I/O controller shall convert reference tags received to incrementing values starting with the one provided in dix_ref_tag_out.

If a received block has protection checking disabled (see the DIX_APP_ESCAPE flag) then that block shall still cause the reference counter to be incremented. The reference tag value received shall be preserved in the protection information transmitted. If DIX_REF_REMAP is unset then the reference tag received shall always be transmitted.

If DIF_REF_CHECK is unset and the received block has a reference tag that differs from the expected value then that block shall still cause the reference counter to be incremented. The reference tag value received shall be preserved in the protection information transmitted.

If DIF_REF_CHECK is set and the received reference tag does not match the expected value then the I/O shall be aborted with an I/O controller reference tag error.

2.7 dix_ref_tag_in

Type: uint32_t

Defines the initial reference tag that the controller shall compare against when verifying received protection information. I.e. when DIX_REF_CHECK is set. For T10 PI Type 1 devices this corresponds to the lower 32 bits of the Logical Block Address. For T10 PI Type 2 devices this corresponds to the Expected Initial Logical Reference Tag.

2.8 dix_ref_tag_out

Type: uint32_t

Defines the initial reference tag that the controller shall send out when transmitting protection information. This value is only valid when the DIX_REF_REMAP flag is set.

2.9 DIX_APP_ESCAPE

Type: Flag

If this flag is set then the I/O controller shall not check blocks whose application tag contains $0xFFFF$.

2.10 dix_app_tag

Type: uint16_t

Defines a constant application tag for an I/O request. This value is logically anded with the dix_app_tag_mask to determine the value to be set when transmitting protection information or the value to compare against when receiving it.

2.11 dix_app_tag_mask

Type: uint16_t

This mask defines which of the bits in dix_app_tag are valid.

If dix_app_tag_mask is $0xFFFF$ then all bits of the application tag should be set to/compared to dix_app_tag.

If dix_app_tag_mask is $0x0000$ then each application tag shall be considered data and be transferred verbatim.

2.12 dix_protection_interval

Type: uint32_t

This value defines the number of bytes of data that is covered by a unit of protection information. For Type 1 devices it is identical to the logical block size. For Type 2 and Type 3 devices the protection interval may be smaller than the logical block size.

3 Implementation Tables

Not all combinations of DIX parameters and flags make sense. The following tables are provided to guide implementors.

3.1 Guard Tag Handling

DIX operation	Target PI Type	GUARD CHECK	IP CHKSUM	APP ESCAPE	REF ESCAPE	Description
READ_INSERT WRITE_INSERT	any	-	-	-	-	Invalid, HBA generates guard tag

DIX operation	Target PI Type	GUARD CHECK	IP CHKSUM	APP ESCAPE	REF ESCAPE	Description
READ_STRIP	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from target, does not check guard tag, discards PI
	1,2,3	true	-	false	false	HBA receives PI from target, verifies CRC guard tag, discards PI
	1,2	true	-	true	false	HBA receives PI from target, verifies CRC guard tag unless app tag contains 0xFFFF, discards PI
	3	true	-	true	true	HBA receives PI from target, verifies CRC guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, discards PI
WRITE_STRIP	any	false	false	false	false	HBA receives PI from OS, does not check guard tag, discards PI
	any	true	false	false	false	HBA receives PI from OS, verifies CRC guard tag, discards PI
	any	true	true	false	false	HBA receives PI from OS, verifies IP guard tag, discards PI
	0,1,2	true	false	true	false	HBA receives PI from OS, verifies CRC guard tag unless app tag contains 0xFFFF, discards PI
	0,1,2	true	true	true	false	HBA receives PI from OS, verifies IP guard tag unless app tag contains 0xFFFF, discards PI
	3	true	false	true	true	HBA receives PI from OS, verifies CRC guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, discards PI
	3	true	true	true	true	HBA receives PI from OS, verifies IP guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, discards PI
READ_PASS	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from target, does not check guard tag, transmits PI to OS
	1,2,3	true	false	false	false	HBA receives PI from target, verifies CRC guard tag, transmits PI to OS

DIX operation	Target PI Type	GUARD CHECK	IP CHKSUM	APP ESCAPE	REF ESCAPE	Description
	1,2	true	false	true	false	HBA receives PI from target, verifies CRC guard tag unless app tag contains 0xFFFF, transmits PI to OS
	1,2	true	true	true	false	HBA receives PI from target, generates IP guard tag, verifies CRC guard tag unless app tag contains 0xFFFF, transmits PI to OS
	3	true	false	true	true	HBA receives PI from target, verifies CRC guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to OS
	3	true	true	true	true	HBA receives PI from target, generates IP guard tag, verifies CRC guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to OS
WRITE_PASS	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from OS, does not check guard tag, transmits PI to target
	1,2,3	true	false	false	false	HBA receives PI from OS, verifies CRC guard tag, transmits PI to target
	1,2	true	false	true	false	HBA receives PI from OS, verifies CRC guard tag unless app tag contains 0xFFFF, transmits PI to target
	1,2	true	true	true	false	HBA receives PI from OS, generates CRC guard tag, verifies IP guard tag unless app tag contains 0xFFFF, transmits PI to target
	3	true	false	true	true	HBA receives PI from OS, verifies CRC guard tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to target
	3	true	true	true	true	HBA receives PI from OS, generates CRC guard tag, verifies IP guard unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to target

3.2 Reference Tag Handling

DIX operation	Target PI Type	REF CHECK	REF REMAP	APP ESCAPE	REF ESCAPE	Description
READ_INSERT WRITE_INSERT	any	-	-	-	-	Invalid, HBA generates ref tag
READ_STRIP	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from target, does not check ref tag, discards PI
	1,2,3	true	-	false	false	HBA receives PI from target, checks ref tag, discards PI
	1,2	true	-	true	false	HBA receives PI from target, verifies ref tag unless app tag contains 0xFFFF, discards PI
	3	true	-	true	true	HBA receives PI from target, verifies ref tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, discards PI
WRITE_STRIP	any	false	-	false	false	HBA receives PI from OS, does not check ref tag, discards PI
	any	true	-	false	false	HBA receives PI from OS, verifies ref tag, discards PI
	1,2	true	-	true	false	HBA receives PI from OS, verifies ref tag unless app tag contains 0xFFFF, discards PI
	3	true	-	true	true	HBA receives PI from OS, verifies ref tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, discards PI
READ_PASS	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from target, does not check ref tag, transmits PI to OS
	1,2,3	true	false	false	false	HBA receives PI from target, verifies ref tag, transmits PI to OS
	1,2	true	false	true	false	HBA receives PI from target, verifies ref tag unless app tag contains 0xFFFF, transmits PI to OS
	1,2	true	true	true	false	HBA receives PI from target, verifies ref tag unless app tag contains 0xFFFF, remaps ref tag, transmits PI to OS
	3	true	false	true	true	HBA receives PI from target, verifies ref tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to OS

DIX operation	Target PI Type	REF CHECK	REF REMAP	APP ESCAPE	REF ESCAPE	Description
WRITE_PASS	0	-	-	-	-	Invalid, target not formatted with PI
	1,2,3	false	-	-	-	HBA receives PI from OS, does not check ref tag, transmits PI to target
	1,2,3	true	false	false	false	HBA receives PI from OS, verifies ref tag, transmits PI to target
	1,2	true	false	true	false	HBA receives PI from OS, verifies ref tag unless app tag contains 0xFFFF, transmits PI to target
	1,2	true	true	true	false	HBA receives PI from OS, verifies ref tag unless app tag contains 0xFFFF, remaps ref tag, transmits PI to target
	3	true	false	true	true	HBA receives PI from OS, verifies ref tag unless app tag contains 0xFFFF and ref tag contains 0xFFFFFFFF, transmits PI to target

3.3 Application Tag Handling

DIX operation	Target PI Type	app_tag_mask	Description
READ_INSERT	any	0x0000	Invalid, app tag must be set to a value
	any	0x0001 .. 0xFFFF	HBA generates PI, sets application tag for each block to dix_app_tag & dix_app_tag_mask, transmits PI to OS
WRITE_INSERT	0	-	Invalid, target not formatted with PI
	1,2,3	0x0000	Invalid, app tag must be set to a value
	1,2,3	0x0001 .. 0xFFFF	HBA generates PI, sets application tag for each block to dix_app_tag & dix_app_tag_mask, transmits PI to target
READ_STRIP	0	-	Invalid, target not formatted with PI
	1,2,3	0x0000	HBA receives PI from target, does not check app tag, discards PI
	1,2,3	0x0001 .. 0xFFFF	HBA receives PI from target, verifies each app tag against dix_app_tag & dix_app_tag_mask, discards PI
READ_PASS	0	-	Invalid, target not formatted with PI

DIX operation	Target PI Type	app_tag_mask	Description
	1,2,3	0x0000	HBA receives PI from target, does not check app tag, transmits PI to OS
	1,2,3	0x0001 .. 0xFFFF	HBA receives PI from target, verifies each app tag against dix_app_tag & dix_app_tag_mask, transmits PI to OS
WRITE_STRIP	any	0x0000	HBA receives PI from OS, does not check app tag, discards PI
	any	0x0001 .. 0xFFFF	HBA receives PI from OS, verifies app tag against dix_app_tag & dix_app_tag_mask, discards PI
WRITE_PASS	0	-	Invalid, target not formatted with PI
	1,2,3	0x0000	HBA receives PI from OS, does not check app tag, transmits PI to target
	1,2,3	0x0001 .. 0xFFFF	HBA receives PI from OS, verifies app tag against dix_app_tag & dix_app_tag_mask, transmits PI to target

4 Request Routing

It is important to emphasize that there are two distinct protection envelopes to consider:

- Transfer of protection information between operating system and I/O controller
- Transfer of protection information between I/O controller (initiator) and disk (target)

Whether to protect the path between OS and controller is up to the application, the operating system or system administrator preference. Thus there is no guarantee that an I/O request bound for a controller supporting the Data Integrity Extensions will provide a scatter-gather list for integrity metadata. It is a per-I/O property.

Similarly, whether the path between controller and disk should be protected with T10 PI is controlled by the RDPROTECT / WRPROTECT field in the CDB.

The two protection envelopes are completely *orthogonal*. And any combination can be expected on hardware that supports it.

Example: A WRITE request could include an integrity metadata scatter-gather list despite being bound for a storage device that is not formatted using T10 PI. In that case the controller must read and verify the protection information and then use standard 512-byte sectors when communicating with the target. The operation in this case is called WRITE_STRIP because it is a *write* request and the protection data must be *stripped* off of the I/O after verification. We refer to this mode of operation as DIX Type 0.

The following charts illustrate how to route READ and WRITE requests respectively:

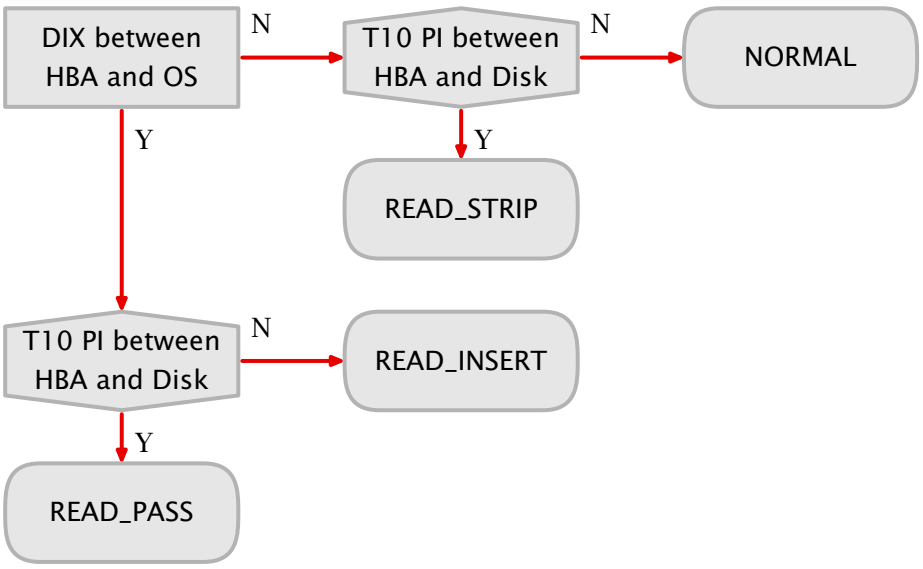

Figure 7 READ request

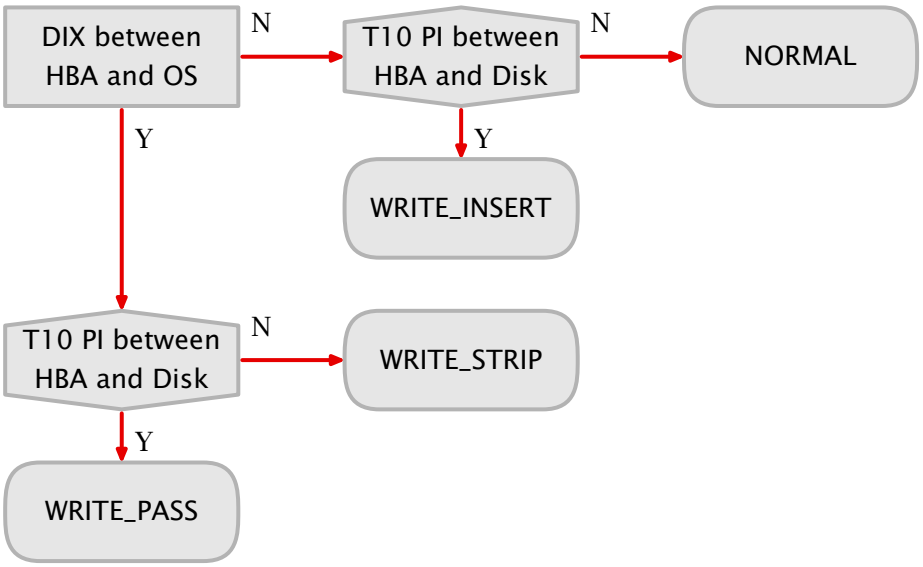

Figure 8 WRITE request

Revision History

Date	Change
2013-11-14	Add `DIX_REF_INCREMENT` flag.
2011-04-14	Reference tag remapping clarifications.
2011-03-16	Clarified PI movement in implementation tables.
2011-03-15	Fixed a few typos and corrected “Expected Initial Logical Block Address” in section 2.6 to “Expected Initial Logical Block Reference Tag”.
2010-09-01	Disallow `DIX_REF_ESCAPE` without `DIX_APP_ESCAPE`.
2010-08-24	Replace `DIX_ESCAPE_MASK` with two flags.
2010-07-04	Parameters affecting I/O execution. Reference tag remapping. Error handling section superceded by DIX check masks.
2009-11-19	Deprecated CONVERT operations and started referring to “T10 PI” instead of “DIF”.
2008-07-18	Added illustrations and incorporated request routing document.
2008-03-31	Clarify handling of uninitialized DIF data ($0xFFFF$ in the application tag).
2008-01-10	Require protection info to be network-endian.
2007-10-02	Initial version.