Date: June 28, 2000

To: T10 Committee (SCSI)

From: Jim Hafner (IBM) (hafner@almaden.ibm.com)

Subject: Suggested changes to OSD model in osd-r01

ABSTRACT:

The Object Based Storage Device Commands document (osd-r01.pdf, heretofore referred to as Rev01) defines a model for a new type of SCSI device where data is accessed by initiators via object addressing and the physical (or virtual-physical) addressing within the device is managed by the device itself. Additional, quality of service and other higher level concepts may be enabled with this object model. The model defined in rev01 appears (to this author) to be more complicated than it need be to meet the intended requirements. In this document, we outline an alternative model with respect to a number (but not all) features of the OSD proposal.

1.0 Outline of differences with Rev01

There are a number of differences between Rev01 and this proposal. These are summarized here:

a) An OSD is always an OSD. That is, it doesn't become one by the FORMAT command and can't change to a block based device within the context of the OSD command/service action set.

b) A block-based device cannot be created as an object within the OSD. (See Rev01, Appendix B, "Emulating a BBSD on an OSD".) Both of this function and the "creation" of an OSD belong, in our opinion, outside the scope of the OSD command specification itself (e.g., in the controller command set or in the vendor-specific space).

c) Sessions are states for I/O data transfers and are always established independent of objects. In Rev01, a Create of an object or an I/O operation on an object can create a session which persists beyond the I/O operation itself. This seemed unnecessarily too complex.

d) Simplified object model. There are only three types of objects in this model, each object has meta-data and data. The two "well-known" objects have pre structured data. (In other words, we combine the Storage Device Control Object and the Object Group List object into one; additionally, the Object Group Control Object and Object Group Object List are joined. In both cases, the "List" object becomes the "data" of the "Control Object".)

e) Objects get attributes (as part of their meta-data); Sessions get parameters. This is just vocabulary but it helps separate the concepts involved.

f) More things are optional. For example, sessions (other than the default session) are optional in this proposal.

g) The specific semantics of a number of service actions is different.

h) A READ is only valid against a UserObject (i.e., not one of the “well-known” objects”). A LIST action is used to get lists of GroupIDs and UserObject IDs. This has a rich and extensible syntax that can aid in navigating the objects in the OSD.

i) A number of major open design questions are raised (see 4.0).

Other differences will be seen by comparing this document with Rev01.

1.1 Design principles

In proposing these alternatives (and raising the design questions), we have tried to use the following design principles:

a) Simplify the model as much as possible

b) Minimize the resource requirements for mandatory functions while enabling optional functions requiring more resources

c) Separation of independent functions

d) Stay within existing SCSI model limitations, so as to minimize the extensions to customary SCSI protocols.

In particular, item (d) implies that we do not require any changes to customary usage of response data formats (status byte, sense data, etc.). See the discussion in 1.1.1.

1.1.1 Command parameters and Response data

The SCSI model customarily assumes that response data contains SCSI Status and if the status indicates CHECK CONDITION, then either included with the status or available on REQUEST SENSE, detailed Sense Data with ASC/ASCQs is provided. This OSD model (as defined in Rev01) seems to imply a different structure. See, for example, Table 20, “Response to CREATE object action” where SCSI status and additional data are embedded together. It is as if Rev01 presumes that status and additional data can be returned in response to any command or service action.

However, the OSD protocols would benefit from the ability to both send and receive complex data structures, along with user data, within a given command cycle. In general, this is not a requirement, though it would enable compound service actions (e.g., CREATE plus WRITE plus SET_ATTRIBUTES). Compound service actions may reduce latency.

In order to facilitate receipt of complex data within the response phase, it might be possible to define precise extensions to the format of response data (status, plus sense, plus new additional fields beyond the 18 or so bytes used customarily). This would enable

some simpler protocols in some instances and some compound protocols in others. Some things to note:

a) This currently doesn't fit the existing SCSI protocol, particularly in the context of parallel SCSI (where status doesn't come with additional data and sense data is provided for a CHECK CONDITION status, on request or as part of the response, *depending* on the underlying transport protocol.)

b)This might conflict with existing vendor-specific usages of longer sense data, though that might be mitigated by the fact that this is a completely new device model.

c) There are only two types of data that need to be returned in response data and they are mutually exclusive. Namely, either the command was successful (in which case the additional fields can be specified in an OSD-specific way) or the command was unsuccessful and the response contains status other than GOOD and sense data (in the usual way). Perhaps this provides a mechanism for implementing some of the semantics of Rev01 within the context of the current SCSI model.

For something like this to work, it might require changes to SAM-x.

In some cases, the OSD model might also benefit from additional parameters in the command phase, as well as having DataOut write-type data. This could be accomplished by appending this optional data in the CDB with the (potentially undesirable) consequences:

a) a given service action might have variable length CDB (that is, the additional length field of the CDB would not only be a function of the service action code); parsing such CDBs might be expensive.

b) the additional parameters, which requires a rich specification language, might force the size of the variable length CDB to be "unreasonable".

c) implementation changes required for host HBAs, changes to specification of transport protocols (like FCP), etc.

We raise these issues and alternatives in order to facilitate a detailed discussion of these questions. However, in this proposal, we have stayed within the customary usage of the command and response phase of the SCSI protocol. In our simplified approach, each service action is a simple operation; compound service actions are not defined.

2.0 Basic model

2.1 Object Types

An OSD is a logical unit within a SCSI device. As such, an OSD will return the OSD Device Type value in response to an INQUIRY command.[1] On OSD contains only objects. Objects have two types of information associated with them:

a) Meta-data

b) Data

Meta-data describe specific characteristics or attributes of the object (more details are given below and in 2.2). This will include the size of the meta-data itself, the total amount of bytes occupied by the object (including Meta-data), logical size of the object, as well as many other parameters. The Data is the actual data contained in the object (e.g., user data).

There are three different types of objects:

a) **Root**: this unique object is always present in the device. Its Meta-data contains device-global characteristics. This includes the total capacity of the logical unit, maximum number of objects (of each type, see the next two items) that it may contain, as well as certain quality of service characteristics (such as data integrity characteristics, e.g., if the device stores all its data in RAID5). Its Data contains the list of currently valid Group IDs (see next item). This is maintained by the OSD itself.

b) **Group**: this object is created by specific commands from an initiator. Its purpose is to contain a list of UserObjects (see next item), that share some common attributes. Its Meta-data contains its GroupID (a 32bit unsigned integer), the maximum number of UserObjects it can contain, the current number of UserObjects, the quota capacity of the group, the current capacity utilized by the group, as well as quality of service attributes common to all the objects in the group. The default attributes for a Group are inherited from the attributes of the device (i.e., Root object). The Data component of a Group is the list of currently valid UserObject IDs. This is maintained by the OSD itself.

c) **UserObject**[2]: these are the primary objects that contain user data. Consequently, the Data for this kind of object is user data; the OSD manages this data on behalf of the initiators. The Meta-data for a UserObject contains characteristics specific to the object. This includes the UserObject ID (a 64 bit unsigned integer), the logical size of the user data, and quality of service attributes. Default attributes for a UserObject are inherited from the attributes of the group in which it is contained.

1. This Device Type value (and the characteristic of being an OSD) should be immutable within the context of the OSD model. This is not exactly implied by the current draft (see FORMAT command of Rev01). However, a device (like a controller or RAID box) may have the ability to configure logical units as either block storage devices (like SBCs) or as OSDs. If this is the case, this creation or change of a logical unit from one type to another should be done in the context of controller commands (SCCs) or in vendor specific ways.
2. A different name for this might be better: e.g., “Basic” or “Generic” object.

From the factory[1], only the Root object exists. It attributes are the “factory defaults”. There are no Groups or UserObjects so the Data for the root object is empty. (A FORMAT command to the OSD will restore the device to this original state.)

There is only one Root object per OSD. There can be many Group objects (up to the capacity of the OSD). The Root and Group objects might be called “well-known” in that the structure of the meta-data and data associated with these objects is predefined. Additionally, for the Root object, the GroupID and ObjectID are predefined (zero in both cases). For the Group object, the GroupID is that of the group itself (it is assigned by the OSD when the group is created), and the ObjectID is predefined (zero). Only UserObjects can contain arbitrary data (the content of this data is owned by the initiators). UserObjects have GroupID of the group they belong to. Their ObjectID is that assigned by the OSD when the object is created.

Meta-data attributes for an object can be queried by the GET_ATTRIBUTE service action and may be changed by the SET_ATTRIBUTE service action.

To get a list of the valid GroupIDs, an initiator can do a LIST service action against the Root object. To get a list of the ObjectIDs in a group, the initiator can do a LIST service action against the Group object. READ and WRITE service actions to these objects are not allowed.

READ/WRITE/APPEND service actions are used to interface with the data of a UserObject.

2.2 Meta-data Attributes

The above description of the basic objects contains a couple of specifics about object meta-data. There are a number of open issues in this regard, many of which are covered in more or less detail in Rev01. Other issues are raised in 4.1.

NOTE: This proposal separates the notion of attributes for a session (referred to here as session parameters) and attributes of objects. This differs somewhat from Rev01.

Some Meta-data attributes of objects are immutable (forever); for example, the total byte capacity of the OSD is not changeable (that is, within the OSD framework directly). Mutable attributes come in three types. Those changed by the OSD autonomously, those changed by the OSD in response to other initiator actions that indirectly effect the attributes of an objects, and those that are set by an initiator action. An example of the first might be the total number of bytes used in a sparse representation of an object. An example of the second type is the size of the Data of a Group object; this changes with creation of new UserObjects, but this is not directly “set” by the initiator. Other examples of the second type are time of creation, time of last read, time of last write, position of “last written byte”, etc.

1. This includes the state of the OSD logical unit after it might be created under controller-type command.

Besides being able to query the existing attributes, some attributes should be immutable (that is, where the factory defaults are not-changeable). A method to ask what attributes are changeable is also defined. All attributes for an object are required to be persistent through power-cycles and resets. However, Data in a UserObject may be stored in volatile storage (cache). This is “flushed” to persistent store by the FLUSH service action.

2.3 Sessions

A session is a set of state information maintained at the OSD for the purposes of setting parameters for data transfer. They are used only for Read and Write (including Append) type actions of user data.

Every OSD has a default session, that determines the parameters of its underlying data transfer machinery. The SessionID of the default session is zero. Optionally, an OSD can support other session parameters. The SessionID of any session other than the default session must be non-zero. The SessionIDs are created by the OSD and provided to the initiator in returned data of an OPEN_SESSION service action. The parameters that govern a session can be queried (GET_SESSION_PARAMS) and changed (SET_SESSION_PARAMS).

In contrast to Rev01, Read and Write service actions will be handled only under an existing session. That is, such actions cannot create a session, they can only fall under the auspices of an existing one.

A CLOSE_SESSION is used to close one or all non-default sessions.

The number of sessions that an OSD supports is vendor-specific (but it must be at least one, for the default session).

Once a session is established, read/write actions can be requested within the context of that session.

One parameter of a session may be an expiration time. Consequently, some sessions may close automatically.

The close of a session or the change of parameters for a session shall not affect the data transfer for any read/write action already being serviced by the OSD within that session. Only new read/write actions specifying that SessionID shall be affected.[1]

1. This paragraph needs language analogous to the effect reservations have on existing commands. On the other hand, this is not the only possible design point for the interaction of existing operations and new settings.

3.0 Command Service Actions

3.1 Summary

Command service actions fall into a number of different categories, as described in the following subclauses. In short these are:

a) FORMAT

b) CREATE, DELETE

c) LIST

d) READ, WRITE, APPEND

e) FLUSH

f) GET_ATTRIBUTE, SET_ATTRIBUTE

g) Session Service Actions

The following additional service actions, if such are deemed to be of value, may be defined later:

a) OPEN, CLOSE (of a UserObject; see 4.9)

b) LOCK, UNLOCK (see 4.10)

c) Other (see 4.11)

Given the complex structure of attribute specifications, and the uni-directional data transfer limitations of SCSI, the use of attributes as optional parameters in some service actions is problematic. See 1.1.1 for a discussion of these and related issues. For simplicity, we've separated the setting of attributes to specific service actions.

3.2 FORMAT (Mandatory)

The FORMAT service action is used to restore the OSD to the factory default settings. The syntax is

FORMAT()

This action has the same effect as deleting all UserObjects, deleting all Group objects, and setting the attributes for the Root object to the factory default. There are no parameters or other identifiers required in this service action (however, notions such as the "immediate" behavior analogous to the FORMAT command for SBCs might be useful).

In contrast to Rev01, this command does not take an optional "length" field. This is (and should be) an immutable attribute of the OSD itself. The notion of changing this capacity value does not belong to the OSD itself, but to a higher level interface, e.g., a controller which owns the OSD as a configurable logical unit. The ability to grow or shrink the physical capacity allocated to an OSD (from the outside) is a useful thing, but is beyond the scope of this proposal.

The limits on number of Group objects and UserObjects supported by the OSD are attributes. Consequently, they can be set with SET_ATTRIBUTES after the FORMAT action. They may also be changed later, subject to defined consistency problems (e.g., a request to set the maximum number of groups to a value less than the existing count).

There is no data transfer for this service action.

3.3 CREATE/DELETE (Mandatory)

These two service actions use the following syntax:

SA(GroupID, ObjectID)

These are both DataIn service actions.

To create a Group object, the GroupID and ObjectID must be set to zero (to indicate that the Root object is being addressed). If the OSD can successfully complete this command, the OSD returns a GroupID in the parameter data. Once the group is created, the attributes for that group (e.g., quota and the number of UserObjects which can be allocated within the group) can be reset with the SET_ATTRIBUTES service action. As one consequence of successful processing of this service action, the newly generated GroupID is added to the Root object's Data.

To create a UserObject, the GroupID must be a valid GroupID for an existing Group object. The ObjectID must be zero. If successful, the returned parameter data shall contain the ObjectID for the new object. As one consequence of this action, the newly created ObjectID is added to the Data for the referenced Group object. It is an open question what the state should be of the Data bytes for this new UserObject; one option is uninitialized garbage, the other is initialized to all zero. Perhaps this should be an option in the CREATE service action?

The attributes of a UserObject (e.g., object logical size) may be reset with the SET_ATTRIBUTES service action.

In both CREATE for a Group or a UserObject, the returned parameter data might contain additional information. For example, the parameter data for creating a UserObject may also include remain Group quota. Details of this still need to be worked out.

NOTE: by putting the created object ID in the returned parameter data (as opposed to response data), we have disallowed the simultaneous creation/write of a new UserObject. See 1.1.1 for a discussion of the limitations of existing SCSI in this regard.

To delete a UserObject, the DELETE service action with GroupID and ObjectID is used. Successful completion of this service action frees any space and resources allocated for the object as well as removes the ObjectID from the Group object specified.

To delete a Group object, the DELETE service action with GroupID is used. The ObjectID must be zero. Successful completion of this action will result in deletion of all Use-

rObjects in the specified Group, freeing of any resources allocated to that Group and removal of the GroupID from the Root objects Data.

NOTE: Should DELETE group succeed if the group is not empty?

The parameter data for a DELETE service action contains the same (once this is formalized) data as a CREATE with the exception of the object ID.

The DELETE service action with GroupID set to zero is not allowed.

3.4 LIST (Mandatory)

The LIST service action is used to get data from the Root or a Group object. (The READ action is reserved for UserObjects only). This service action is a DataIn type service action and has the following syntax:

LIST(GroupID, Number, Index, [SortOrder], AllocationLength)

A zero GroupID refers to the Root object and non-zero GroupID refers to a valid Group object. Number specifies the number of IDs to be returned (GroupIDs if referencing the Root object and UserObject IDs otherwise). Index specifies the starting position of the IDs within the specified SortOrder, with initial position value of zero. SortOrder is optional. The default order is implementation dependent. Support for specific SortOrder rules is optional. Examples are numerically by object ID, by creation time, by last access time, by last write time, by size (e.g., quota for groups or object logical size of UserObjects), or any other predefined sort rule. The AllocationLength is the amount of space (in bytes) set aside in the DataIn buffer of the initiator. This might be more or less than is necessary for the Number*(Size of IDs) requested and is handled in the usual way (truncated if necessary). The returned data shall contain a header which specifies the sort order, the GroupID, and other additional data to allow for the returned parameter data to be self-parsable. (Details are TBD.)

There are no session parameters allowed for this data transfer.

3.5 READ/WRITE (Mandatory), APPEND (Optional)

These service actions take the following syntax:

SA(GroupID, ObjectID, [SessionID], TransferLength, Offset)

These service actions are used to read or write some or all of the Data for a given UserObject. Consequently, the GroupID and ObjectID shall each be non-zero and shall refer to an existing object in the OSD.

The READ action requests data from the specified UserObject, starting at the specified Offset, and for the specified TransferLength. The TransferLength indicates the number of bytes the initiator has set aside in the DataIn buffer.

The WRITE and APPEND actions write data to the specified UserObject of the size specified in TransferLength. For the WRITE this starts at the specified Offset. For the APPEND, it starts at the next byte offset beyond the last written by either a WRITE or APPEND. (The OSD must maintain this offset within its internal data structures, particularly, if it supports the APPEND service action.)

The SessionID for any of the service actions is optional (and optionally supported by the OSD). If supported and the SessionID is valid, this requests the data transfer for the service action be handled under the parameters (quality of service, etc.) of the specified session. If the SessionID is set to zero, then the parameters of the “default” session apply.

3.6 FLUSH (Mandatory)

This service action applies to UserObject Data. (The Root and Group Data are always persistent and flushed.)

FLUSH(GroupID, ObjectID, [Length, Offset])

The FLUSH, with non-zero GroupID and ObjectID, is used to commit any changes to a UserObject to persistent store (clear the volatile cache for the specified object within the optionally specified Length and Offset).

A FLUSH service action with non-zero GroupID and zero ObjectID requires the OSD to commit to persistent store all Data for all UserObjects in the specified Group. The Length and Offset fields are ignored.

A FLUSH service action with both the GroupID and ObjectID set to zero requires the OSD to commit to persistent store all Data for all UserObjects within the OSD. The Length and Offset fields are ignored.

3.7 GET_ATTRIBUTE/SET_ATTRIBUTE (Mandatory)

Attributes are the contents of the Meta-data for specific objects. Each object type has a specific set of attributes which cannot be changed directly, either because they are “factory defaults”, or because they are inherent to the object itself. Other attributes may be changed by SET_ATTRIBUTE service action.

These two service actions require the following syntax:

SA(GroupID, ObjectID, AttributeMask, Changeable, TransferLength)

The SET_ATTRIBUTE service action is a DataOut operation, the parameter data contains a description of the attributes that need to be set. In this case, the AttributeMask is ignored. The TransferLength indicates the amount of parameter data that will be transfered.

The GET_ATTRIBUTE service action is a DataIn operation. A Changeable bit set to one requests that the OSD send a summary of the attributes it supports (subject to the AttributeMask) and indicating which attributes are changeable and what range of a values they can attain. If the Changeable bit is set to zero, then the OSD returns the current value of the attributes specified in the AttributeMask. The TransferLength indicates the size of the initiators DataIn buffer.

The AttributeMask takes a role analogous to the PageCode (more precisely, a set of Page Codes) field in MODE SENSE/SELECT and the Changeable bit is analogous to the PC=changeable in MODE SENSE. See 4.1.

(More details on the structure of the AttributeMask and the data exchanged in these service actions need to be worked on, see 4.1, but some of the essential pieces are already part of Rev01.)

3.8 Session Service Actions

This suite of service actions is only suggested as a first pass on this subject.

NOTE: one major open question in the context of sessions is the scope of a session or set of sessions (limited to the requesting initiator, valid for all initiators holding the SessionID, etc.)

3.8.1 LIST_SESSIONS (Optional)

This simple service action requests that the OSD return a list of the SessionIDs for existing sessions. The syntax is

LIST_SESSIONS(AllocationLength)

The AllocationLength specifies the size of the DataIn buffer of the requesting initiator. The returned parameter data will contain a field containing a count of the number of open sessions (other than the default session), and a list of SessionIDs (subject to the AllocationLength limitations).

NOTE: if the scope of a session extends beyond the initiator which created the session, this service action will have to be mandatory.

3.8.2 OPEN_SESSION/CLOSE_SESSION (Optional)

Non-default sessions are created and destroyed with the OPEN/CLOSE_SESSION service actions. The syntax is:

OPEN_SESSION(AllocationLength)

CLOSE_SESSION(SessionID)

The OPEN_SESSION service action is used to request that the OSD set aside resources for a session. The returned parameter data contains a SessionID generated by the OSD (and possibly other data TBD). The AllocationLength indicates the size of the initiator's DataIn buffer and should be large enough for a SessionID (that is, at least 4bytes). If the OSD has the resources to create and maintain the state of a new session it returns the SessionID and responds with GOOD status. A session is open until closed either by an explicit CLOSE_SESSION action or by actions taken autonomously by the OSD itself (see 2.3).

The parameters of a new session are those of the default session. They can be changed, while the new session is open, by the SET_SESSION_PARAMS service action.

NOTE: creation and parameter setting for new sessions are separate service actions. This is a consequence of the limitations of the SCSI protocol.

The CLOSE_SESSION service action requires a 4byte SessionID as parameter. There is no data transfer for this service action. The SessionID can be any valid SessionID or the FFFFFFFFh value. An invalid SessionID is ignored and is not an error condition. The all FF value means that all sessions (except the default session) are to be closed. See 2.3 for the actions taken on the part of the OSD when a session is closed.

3.8.3 GET_SESSION_PARAMS/SET_SESSION_PARAMS (Mandatory)

These two service actions are used to query the session parameters of an existing session, or to change them. The syntax is:

SA(SessionID, TransferLength, Changeable)

The SessionID specifies the identifier of an existing session (this includes the value zero for the default session). The TransferLength specifies the (maximum) amount of data that shall be transferred through the DataIn (GET) or DataOut (SET) buffers.

The Changeable bit applies only to the GET_SESSION_PARAMS service action and only when the SessionID is zero. It requests a summary of the session parameters (of the default session) that can be changed for other sessions.

The SET_SESSION_PARAMS is only valid if the SessionID is not zero. In other words, the default session parameters are unchangeable.

The structure of the parameter data that describes the parameters of sessions is TBD.

4.0 Open questions

4.1 Meta-data attributes

The specification of the types of attributes objects (of each type) may have, how they are referenced, changed, packed into parameter data, etc., is still TBD. A partial specification is given in Rev01, clause 5.3, but more work is needed.

NOTE: As noted earlier, this proposal separates the notion of attributes for a session (referred to here as session parameters) and attributes of objects.

The Root object needs to have some way of describing (a) total physical capacity (b) QoS properties of that capacity, subdivided by QoS ranges (e.g., how much is on RAID5, how much is tape, how much has certain I/O characteristics), etc. Group objects need a way to describe the quota, current use, static QoS properties, etc. There is at present, no language for describing some of these properties.

A careful inventory of the minimal set and optional set of attributes is required. Additionally, a mechanism for vendor-specific and for extensibility of the attribute concept will have to be defined.

An AttributeMask is used in a service action in a manner analogous to Mode Page codes. Namely, a bit position points to a relevant Attribute Page. An Attribute Page will specify a set of attributes which have some commonality between them (e.g., RAID level and capacity). In this way, we can utilize a relatively small size AttributeMask to reference relatively large amounts of data and still have some room for extensibility. So, 64bit mask can reference 64 pages of data and each page of data can many more fields of quite arbitrary size. Is this sufficient to meet the long term requirements?

It may be desirable that the AttributeMask have different interpretations if the GroupID/ObjectID combination refers to the Root object, a Group object, or a UserObject.

There are open questions about the interaction of default attributes, setting/resetting of attributes, relationship of attributes of an object with its parent object (User to Group or Group to Root), and relationship of object attributes and session parameters.

This whole subject is a work-in-progress.

4.2 Size of UserObjects

There are a number of “size” notions applicable to a UserObject. The size of the Metadata is one such.

Additionally, there is the “object logical size”. This can have a number of different meanings. One interpretation is equivalent to End-of-Write. That is, this size is the offset (plus 1?) of the last byte written by a WRITE service action. End-of-Write is a useful (required) notion in the context of the APPEND operation.

Another interpretation is something like End-of-Object. This might be an offset that exceeds the last location written.

A third interpretation is that all byte addresses below that value return initialized values (zero unless expressly written with WRITE) and anything beyond is uninitialized, but any offset is addressable in any case.

There is/should be a distinction between this “size” and the physical size of the object. In particular, if the OSD can manage sparse objects, then logical size may be significantly larger than the physical size of the object.

The following are open questions in this context that, in our opinion, have not been adequately addressed (even here):

a) what “size” attributes are important

b) is there a notion of End-of-Write that is valid beyond the requirements of APPEND

c) is there a notion of End-of-Object that differs from End-of-Write

d) what is the initialization of bytes offsets below either “logical size” or EOW

e) is the physical size of an object information private to the OSD or should the initiator be able to get this information (at this granularity).

4.3 FORMAT

Should the FORMAT service action be a DataIn command, with returned data summarizing some/all of the default attributes of the Root object of the OSD?

Alternatively, should the FORMAT service action be a DataOut command, used to set the default attributes, such as any “capacity” parameters of the Root object? This shouldn’t include the total size of the OSD, but might include maximum number of Groups, maximum number of UserObjects, maximum number of UserObjects in any given Group, maximum quota for Group, maximum size of UserObject’s Data, etc. Should the FORMAT command enable changes to something other than the “factory default”? See the MODE SENSE/SELECT options for persistence for a model of how this might be done.

4.4 CREATE/DELETE

Specification of the data returned in the CREATE action (beyond the ID of the object created) and also of the data returned in a DELETE action is TBD.

It’s an open question whether the DELETE action on a group object should succeed if the Group object is not humpy of UserObjects or whether it should fail in this case or whether there should be a bit in the service action which effectively says delete the group and all its objects unconditionally.

4.5 LIST

It is possible to unify the LIST service action for Root/Group objects into the READ service action (essentially LIST just “reads” the Data for these objects). However, to get the full semantic richness of LIST (with sort, etc.) would require adding features to READ which apply only to a subset of the GroupID/ObjectIDs that can be specified. Additionally, it is not clear that allowing SessionIDs for the LIST function are necessary. Finally, READ

is more naturally a counterpoint to WRITE; but WRITE can not apply to Root or Group objects, so it seemed more natural to add the LIST action. Is this correct?

A specification of the predefined SortOrders for LIST is TBD, as is, the subset of these which are required of all OSDs and which are optional. This specification should be extensible and should allow for vendor-specific options as well.

4.6 READ/WRITE/APPEND

Given some interpretation to “allocation” or “object logical size”, it is open whether this size can change automatically (i.e., independent of SET_ATTRIBUTES), whether it can grow dynamically or whether this “dynamic” is a specific attribute of a given object. If size is static, then READ/WRITE/APPEND beyond the size limit should clearly fail. If size can grow dynamically, then WRITE/APPEND have, in effect, unlimited addressing. READs beyond the size limit may or may not fail. If they don’t fail, it is open what the result is. There are three choices (a) only return data within the size (End-of-Write or End-of-Object), (b) return zero outside the End-of-XXX (c) return uninitialized garbage.

If the SessionID in a READ/WRITE/APPEND service action is not valid (for whatever reason, see 4.8), then one of three design choices are possible: (a) fail the request with error message; (b) service the request with the default session; (c) service the request with the default session, but report this as RECOVERED ERROR status.

4.7 FLUSH

It is open whether the FLUSH syntax should support offsets/lengths or (always) mean the entire object. We’ve given the more general syntax, but have no strong feeling about this.

4.8 Sessions

It is not yet decided how volatile or non-volatile the state or parameters of a session should be. One argument is that they should be volatile so that resets and power-cycles should invalidate all non-default SessionIDs. An argument can be made for either persistence or for optional persistence.

It is unclear what happens if the Meta-data attributes of the object being accessed and the parameters of the session conflict (e.g., the object may be on tertiary storage and so not be streamable, but the parameters of the session may request/require streaming).

It is open whether the expiration time of a session should be allowed to be infinite and if so, what that means for persistence through resets/power cycles.

Should parameter changes on a session be allowed to affect existing commands?

Are sessions “owned and private” to a given initiator or can any initiator use a valid SessionID in its I/O operations? If the latter, then some mechanism like LIST_SESSIONS

would be required. If the former, then what is the response to GET_SESSION_PARAMS from an initiator that didn't open the session in the first place?

This document proposes a LIST_SESSIONS service action. As noted, this is only required if sessions can be shared by different initiators. If sessions are private to an initiator, then this may not need to be defined. If it is defined, then the returned list of SessionIDs should include only those which are valid for the requesting initiator.

4.9 Open/Close

It is an open question whether there is a need for OPEN/CLOSE semantics on a UserObject. The above OPEN/CLOSE_SESSION can be used for setting data transfer requirements.

One possible value in OPEN on an object is for locking purposes. But is this better suited to a separate LOCK semantics (see 4.10), if such semantics are required from the OSD? Another possible purpose is a hint to the OSD that I/O operations will be forthcoming on this object, allowing the OSD to do some prefetching, cursor placement (SEEK?), etc. An alternative to an OPEN of this sort may be achieved with a READ of zero bytes (and perhaps with an offset, to "set the cursor"). A counterpoint is that READ is typically stateless, whereas an OPEN migh imply a maintained state until CLOSE.

Other unanswered questions with respect to OPEN/CLOSE semantics are

a) who owns the OPEN (an initiator?)

b) who can issue the CLOSE (anyone, only the owning initiator, does it matter?)

c) how long does an OPEN last? Do they automatically CLOSE if the initiator "goes away", e.g., FC PRLOs? Do they close after a time-out?

d) is only one OPEN allowed on an object?

4.10 Locking

It is an open question whether locking semantics are a required feature of an OSD, whether, the OSD should support such semantics at all, or whether the ability to put a lock on an object is a changeable attribute. But, perhaps this can be accomplished within the context of Persistent Reservations if those can be extended to objects (analogous to elements within a medium changer). If it is judged that an independent locking semantics is required, a number of questions need to be answered:

a) who owns the lock (an initiator or the holder of a lock handle)?

b) what data is required to release the lock?

c) can the lock be released by a non-owner and if so, how hard should that be?

Once these questions are answered, the particulars of LOCK/UNLOCK service actions can be specified.

4.11 Other possible OSD functions

One suggested function of an OSD is something akin to version control on objects. This might be an optional feature (as it might require many resources). The version control would allow an initiator to do READ-type operations against older versions of an object. How this might be specified is still open for debate. One choice is a service action which takes an existing object, creates a new object which is a (possibly logical) copy of the original. The new object would have a back reference to its source object. The source object would become read-only (and perhaps indelible). This would enable a chain of objects, which indicate the history. Another choice is an attribute of a single object that requires version control. Then recovery or reads against the older versions would be possible. The event that drives a new version state may be either any write operation or by explicit service action.

4.12 Access Controls

The current OSD leaves open the question of access controls, aiming to define that within the context of the full security model (encryption, etc.).

However, the SPC model of Access Controls (see ftp://ftp.t10.org/t10/document.99/99-245r8.pdf, and 99-245r9.pdf when available) allows for (a cryptographically insecure but manageable) set of access controls at the group level. It is worth investigating the value of this type of access control as a preliminary step to full cryptographically secure mechanisms, defined at the UserObject level.

4.13 Security issues

In this proposal, we have avoided the issue of security (beyond what is mentioned above for Access Controls). This is an area where much work is required, but we feel that the first step in defining an OSD standard is to get the basic model defined precisely, before issues of security obscure the fundamental issues. In other words, we need minimal function before we can secure those functions.

On the other hand, some of the issues in SCSI-security may extend beyond the OSD model, e.g., Key Management may have broader implications and applications. We believe that the entire subject of SCSI-security should be addressed in the larger context (SPC-x?), before any specific issues are settled within the context of OSDs.

4.14 Miscellaneous

This section contains a summary of some of the smaller open questions raised above, and perhaps a few others:

a) rules for interaction of attributes for objects, their default attributes, session attributes, etc.

b) limits on expiration time for open sessions (infinite) and on persistence of sessions through resets and power cycles.

c) state of allocated Data for new UserObject on creation (initialized or not; optionally specified in CDB or not).

d) Should a DELETE of group succeed always even if the Group object is non-empty (i.e., contains UserObjects)? Should this be an optional parameter in CDB?

e) Should FLUSH support length/offset semantics? Should this be optional?

f) What are the requirements for Lock/Unlock? Can they be met with Persistent Reservations at the UserObject level? Are locks required for Root and Group objects?

g) What types of Sort Order are useful in the context of the LIST (IDs) action?

5.0 PRESENTATION STARTS ON NEXT PAGE

Basic OSD Device Model

An OSD is not a Controller

- physical size is fixed (e.g., from factory)
- Can’t create LUs internally
- Controller can create OSD (or BSD) across physical storage

OSD manages user data with associative addressing model (ID + offset + length)

Objects can have arbitrary length, be sparse, stored in different types of media, etc.

Mapping OSD Function to SCSI

SCSI limitiatons:

- One-way data transfer
- Status cannot come with additional data!
- not all protocols bundle Sense Data with CHECK CONDITION status

CDBs should be easily parsed

- widely varying length is not necessarily good

CONCLUSION: Compound service actions (e.g., CREATE+WRITE+SET_ATTRIBUTE) difficult to map to SCSI protocol

Three basic objects

Each object has Meta-data and Data.

Root object:

- Meta-data: global data for OSD, default attributes for groups and user objects
- Data: List of GroupIDs

Group objects:

- Meta-data: attributes for group, default attributes for user objects
- Data: List of UserObjectIDs

UserObjects:

- Meta-data: attributes for object
- Data: user data

Examples of Meta-data

Root:

OSD capacity (by type, RAID5, tape, etc.), max number of Groups, UserObjects, default attributes of Groups and UserObjects, etc.

Group:

quota, size (many meanings), max number of UserObjects, default attributes of UserObjects, etc.

UserObject:

size (many meanings), type of storage, time-stamps, EOW, EOO, etc.

Sessions

Sessions are opened independent of object I/O

- establish a set of characteristics/parameters for data-phase (via GET/SET_SESSION_PARAMS)
- valid only for R/W operations on user data (Data of UserObject)
- non-default sessions are optional
- all R/W operations are under existing session (default or pre-established)

Some open questions:

- who owns the session (one initiator?)
- can it be shared by other initiators?
- who can close a session (anyone?)
- interaction of changes to session (e.g., close) and existing R/W operations

Required Service Actions

FORMAT (DataNone): restores OSD to “factory default”

CREATE/DELETE (DataIn): creates or deletes Group object or UserObject

LIST (DataIn): list objectIDs in specified sort order

READ/WRITE/APPEND (DataIn/DataOut/DataOut): UserObject data transfer

FLUSH (DataNone): commit UserObject data to persistent store

GET/SET_ATTRIBUTE (DataIn/DataOut): query/change attributes for Root/Group/UserObject; also get the set of “changeable” attributes

Session Service Actions

LIST_SESSIONS (DataIn): get a list of current SessionIDs

OPEN/CLOSE_SESSION (DataIn/DataNone): create/destroy session

GET/SET_SESSION_PARAMS (DataIn/DataOut): query/change parameters for existing session; also get the set of “changeable” parameters

Other TBD Service Actions

OPEN/CLOSE (of UserObject)

LOCK/UNLOCK (of Group or UserObject)

VERSION (on UserObject)

Major Open Questions

Can/should we extend SAM-x to enable compound service actions?

Do we really need/want compound service actions?

Is T10/SCSI the right protocol to define an OSD?

Issues:

- limitations on compound service actions
- security (encryption, authentications, capabilities, etc.)