To: T10 Technical Committee
From: Rob Elliott, HP (elliott@hp.com)
Date: 8 June 2007
Subject: 07-242r0 SAT-2 On-disk data format

Revision history
Revision 0 (8 June 2007) First revision, as informally discussed in the May 2007 SAT WG

Related documents
sat2r00 - SCSI-to-ATA Translation - 2 (SAT-2) revision 0
sas2r09a - Serial Attached SCSI - 2 (SAS-2) revision 9a
ata8-acs-r3g - ATA-8 ATA/ATAPI Command Set (ATA8-ACS) revision 3g

Overview
In a SAS environment, SATA devices can be exposed to more than one STP initiator port - either over time (one STP initiator port is replaced by another), or concurrently (since SAS allows multiple initiators to coexist).

An expander containing an STP/SATA bridge coordinates access to each SATA device attached to it using affiliations. Affiliations prevents multiple STP initiator ports from confusing each other's transport layers by accessing the SATA device at the same time. Between commands, however, the affiliations can be released, and the STP initiator ports can obtain access to the same media.

To implement some more complex SCSI features, a SAT-2 SATL needs to store information on the ATA device so it can:

a) recover that information after power loss, hard reset, logical unit reset, or I_T nexus loss;
b) transfer that information to a replacement SATL; and
c) agree on the contents of that information with another SATL that is concurrently accessing the SATA device.

Complex SCSI features include:

a) Mode page values.
   A) If the mode page is shared, the same value needs to be returned for all I_T nexuses.
   B) If the mode page is per-I_T nexus, each I_T nexus needs its own value.
   C) If the mode page is saveable, the value(s) need to be preserved through power loss.
b) Persistent reservations commands.
   A) Registrations and persistent reservation status need to be shared by concurrent initiators and preserved through power loss.
c) Read-modify-write commands like ORWRITE and XPWRITE. These commands ask the SCSI device server (here, the SATL plus the ATA device) to implement an atomic operation. If two initiator ports request the same command concurrently, they must be performed atomically to avoid interleaving like this:
   1) Initiator A reads data
   2) Initiator B reads data
   3) Initiator A modifies and writes new data
   4) Initiator B modifies and writes new data. Initiator A's modifications are lost.
4) Any command that maintains state through power loss
5) Any command that maintains state through hard reset (if the SATL is unlikely to do so itself)
6) Any command that maintains state through logical unit reset (if the SATL is unlikely to do so itself)

The ATA command set doesn't define any commands that perform atomic operations (e.g. Test And Set, Compare and Exchange, etc.); such a command would simplify coordinating access across concurrent STP initiators. It does ensure that READ and WRITE commands are atomic on a logical block basis. So, a mutual exclusion algorithm such as Lamport's Bakery Algorithm is needed to ensure that multiple initiators can access shared data structures atomically. Even that may have intractable issues, meaning all that can be reliably implemented is persistence of values across power loss for one STP initiator port at a time.

Lamport's Bakery Algorithm

The wikipedia rendition of Lamport's Bakery Algorithm is:

```
Enter, Number: array [1..N] of integer = {0}; // declaration and initial values of global variables
Thread(i) {
        while (true) {
                Enter[i] = 1;
                Number[i] = 1 + max(Number[1], ..., Number[N]);
                Enter[i] = 0;
                for (j = 1; j <= N; j++) {
                        while (Enter[j] != 0) {
                        // wait until thread j receives its number
                        }
                        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
                                // wait until threads with smaller numbers or with the same
                                // number, but with higher priority, finish their work
                        }
                } // for
        // critical section...
        Number[i] = 0; // non-critical section...
        } // while (true)
} // Thread
```

Each SATL can be considered to be running a Thread. The critical section is where the SATL modifies a shared data structure stored on the SATA device.

All the global variables (Enter, Number) could be stored in a SAT metadata area of the ATA disk. It is probably simplest to dedicate one full logical block per variable so they can be read or written atomically (e.g., Enter[0] is on LBA x, Enter[1] is LBA x+1, etc.). Since only a few bits are needed for the variables, the extra storage in the logical block could be used to store debug information about the SAT that wrote the logical block (e.g., SAS address, timestamp, etc.).

Unfortunately, there is no global way to assign the index numbers for the Enter and Number arrays; unlike threads in an OS, where each thread has a unique number, the only unique value a SATL knows about is its STP initiator's 64-bit SAS address. An array of 2^{64} possible SAS addresses is too large to maintain on disk, and any hashing algorithm applied to those values is bound to have collisions.

A new SMP function could be defined in the SAS expander to return an incrementing number that is different for each STP initiator port concurrently accessing the SATA device (since the expander was powered on). A SATLs could use this value as its thread ID.

STP initiators could be required to enumerate all STP initiators in the SAS domain and assign themselves relative numbers. With zoning, however, they may not be able to detect other STP initiators. Expander assistance seems crucial.

On-disk data format

The on-disk data format could be compatible with the SNIA DDF format. This would increase the chance that if a SATL ever sees a DDF structure or a DDF software ever sees a SATL structure on a disk, it can realize something might be there worth preserving and not treat the disk as free for overwriting.

Figure 1 shows the DDF on-disk structure and how DDF would be nested inside of SAT.

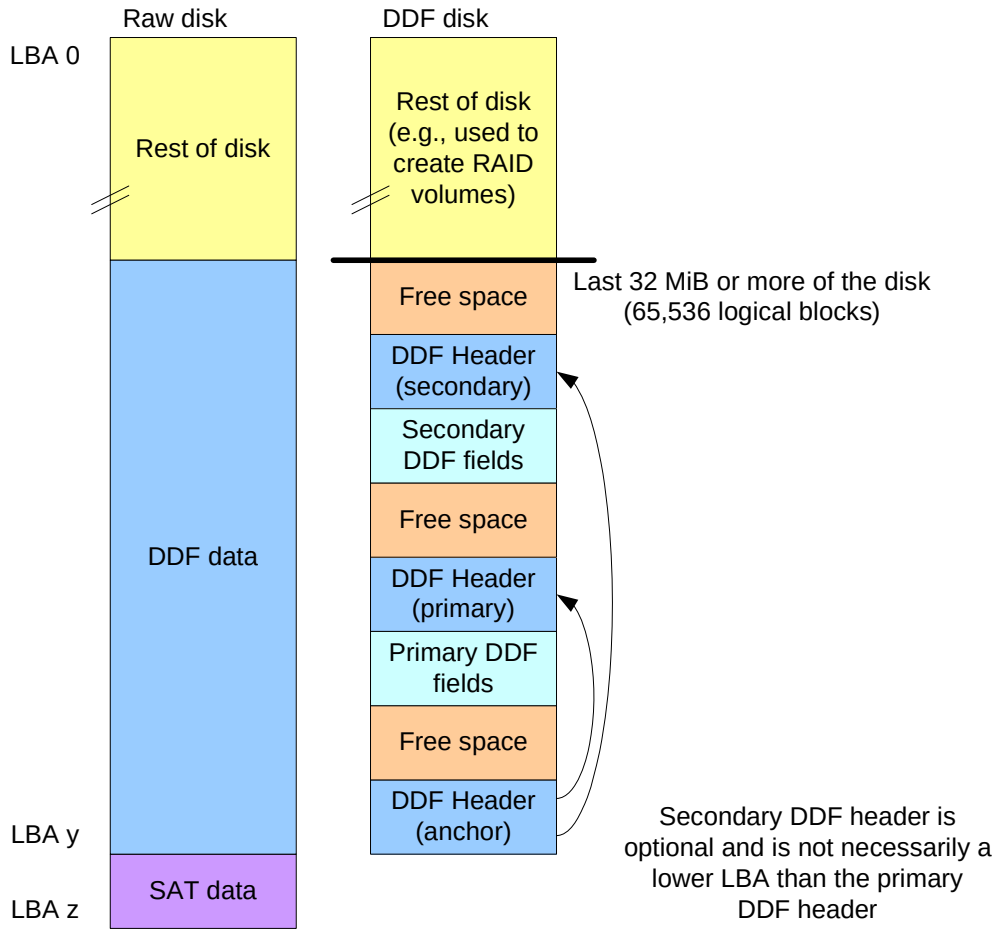
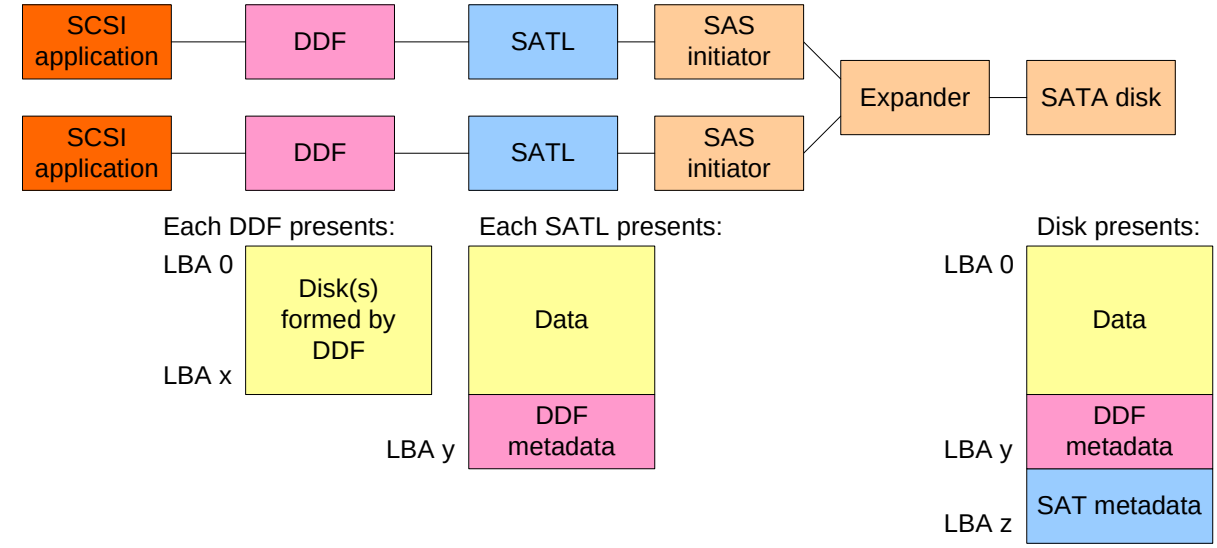


Figure 1 — DDF on-disk structure

Figure 2 shows how SAT and DDF headers can coexist.



If the disk is ever exposed to DDF directly, it would be safer if DDF recognized it as not a raw disk. Ideally, DDF should just ignore the SAT metadata and find the nested DDF metadata.

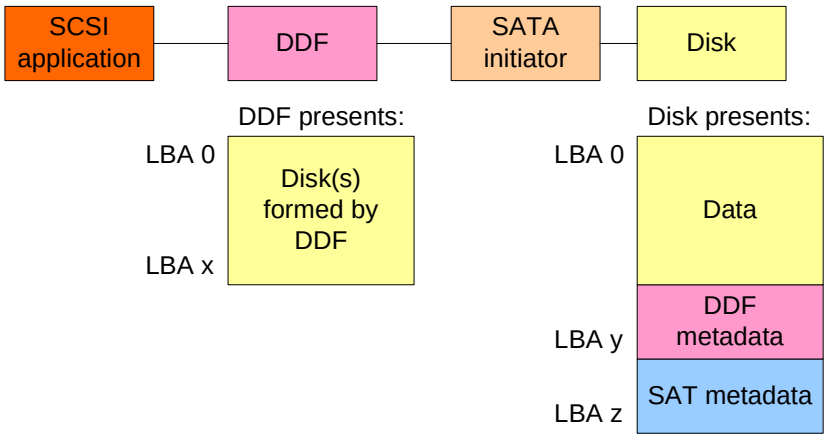


Figure 2 — SAT and DDF header nesting

The DDF_rev field indicates if the DDF header follows a SNIA DDF specification or a vendor-unique usage. It does not define exactly which fields are expected to be honored in the vendor-unique usage. Table 1 lists what seem to be the key fields in the DDF header.

Table 1 — DDF Header key fields

Field	Usage
Signature	DE11DE11h (i.e., “Dell Dell”)
CRC	Covers the entire logical block
DDF_Header_GUID	Information about the controller that created this header: a) 8-byte T10 vendor ID string b) 8-byte PCI ID (Vendor ID, Device ID, Subsystem Vendor, Subsystem ID) or FFFF<anything>h if unknown c) 4-byte timestamp (seconds since 1 Jan 1980 GMT); and d) 4-byte random number
DDF_rev	“xx.yy.zz” format where x, y, and z are numbers “CCxxxxxx” for vendor unique usage, where C is non-numeric
Sequence_Number	Unused in anchor
Timestamp	Seconds since 1 Jan 1980 GMT
Open_Flag	00h = DDF not being written 01h - 0Fh = DDF being written FFh = anchor
Foreign_Flag	00h = controller recognizes the configuration 01h = controller does not recognize the configuration
Primary_Header_LBA	Points to primary DDF header
Secondary_Header_LBA	Points to secondary DDF header
Header_Type	00h = anchor, 01h = primary, 02h = secondary
Workspace_Length	Must be >= 32,768 (16 MiB)
Workspace LBA	Points to area for controller vendor-specific usage

Table 2 lists how SAT would set the key fields in the DDF header to keep DDF software from stomping on it.

Table 2 — DDF Header key fields for SAT

Field	Size in bytes	Usage
Signature	4	Set to DE11DE11h (i.e., “Dell Dell”)
CRC	4	Set to cover the entire logical block
DDF_Header_GUID	24	Information about the controller that created this header: a) 8-byte T10 vendor ID string b) 8-byte PCI ID (Vendor ID, Device ID, Subsystem Vendor, Subsystem ID) or FFFF<anything>h if unknown c) 4-byte timestamp (seconds since 1 Jan 1980 GMT); and d) 4-byte random number
DDF_rev	8	Set to “SAT-2.0 ” (i.e., vendor unique usage)
Sequence_Number	4	Set to 00000000h (unused in anchor)
Timestamp	4	Set to the number of seconds since 1 Jan 1980 GMT
Open_Flag	1	Set to FFh (i.e., anchor)
Foreign_Flag	1	Set to 00h (i.e., last controller that wrote the header recognized the configuration)
Primary_Header_LBA	8	Set to 00000000_00000000h
Secondary_Header_LBA	8	Set to 00000000_00000000h
Header_Type	1	Set to 00h (i.e., anchor)
Workspace_Length	4	Set to 00000000h
Workspace LBA	8	Set to 00000000_00000000h

Data structures

This information is shared by all I_T nexuses:

a) Mode parameter header and block descriptors
b) Mode page values
c) List of mode page policies (shared, per-target port, and per-I_T nexus per page)
d) Reservation information: PRgeneration, type, scope, identity of the reservation holder
e) Identifying information for REPORT/SET IDENTIFYING INFORMATION commands
f) Access controls information: enabled/disabled, other information

This information is per-target port:

a) Mode page values

This information is per-I_T nexus:

a) Initiator port identifier
b) Relative target port identifer
c) Mode pages
d) Reservation information: registration key, initiator port identifier, relative target port identifier
e)