Multi Path Switching

Design Spec V2 Draft, 1-Jul-1996

Glenn C. Everhart

Problem Statement: The HSZ40 series with the next release of HSOF will offer a dual bus failover capability. This is characterized by some new INQUIRY information so that a host can be informed that the failover is possible, and new logic to provide a "preferred" initial path. (A single SCSI bus failover is also offered, but that requires no software changes.) In addition, a number of other situations exist where more than one path to a device exists, and where currently no means to control which path is used exists. This situation exists in the case where a device on a shared SCSI bus, for example, can be accessed directly or via a server such as the MSCP server. Lack of an ability to notice the failure of one path and switch to another is a notable weakness in parts of the OVMS I/O system currently and needs to be fixed. In the future it may also be desirable to be able to share the I/O load to devices across several access paths. Some controllers (EMC's, for example) support this now, but OVMS cannot use the features. In addition, some device names appear twice (or conceivably more) in OVMS displays and other devices may appear with multiple, and different, names. This can cause confusion at best, or file system corruption in cases where multiple names are used for the same storage.

For the HSZ, when the devices come up under current autoconfiguration, it is to be expected that each device will appear twice, once via its path over the first SCSI bus from the HSZ40 to the host, once via the other. Notwithstanding this, the devices are not duplicated, and because they have two aliases, the file system can readily corrupt file structures located on these devices.

Some means to control access so that a single path is used at any given moment, and so that normal VMS operations will not notice the dual path, is needed, which will allow access to the devices via the second SCSI bus in the event the first fails. Allowing accesses to be shared over the busses initially is highly desirable as well, and is supported to a degree by the HSZ firmware. (This is done by allowing a preference to be stated for each device, so that some devices can be set to be "preferred" over each bus.)

This failover must be available for disks. It should be available for other devices also.

Background:

Some HSZ devices have multiple SCSI bus connections, and the issue of failover between them has arisen. These connections can be connected either to the same SCSI bus (providing dual paths to that bus so that the failure of either controller does not prevent access to devices connected to the HSZ) or to different SCSI busses.

If both SCSI controllers on the HSZ are connected to the same SCSI bus, the HSZ will be able to handle failover within itself so that a host on the bus will not notice any change. However, when each controller is connected to a different SCSI bus, the host must be involved.

In this case, an HSZ might be on two ports on a system, with two SCSI controllers, and all LUNs attached to the HSZ will therefore show up twice; a disk might show up as DKB300: and as DKD300:, for example, if the HSZ were connected to the second and fourth SCSI adapters on the machine. At the HSZ itself, it is possible to set a preferred path to the device, and it will appear unready on the other path, but both could be configured and would refer to the same device.

Having dual names for the same storage violates the VMS cluster naming scheme and can result in disk corruption, so this situation by itself is not satisfactory.

Fortunately the HSZ itself provides certain bits of information which an operating system can use to figure which devices are which.

First, when in this dual-bus configuration, an HSZ will return some extra data in INQUIRY responses. This data includes:

* The serial number of this controller
* The serial number of the alternate controller
* A bitmask of LUNs which are preferred for this controller
* State of the "other" controller

Therefore one can determine, from the INQUIRY data, if the device is an HSZ, what this and the "other" controller is, and whether this particular device is preferred on "this" controller. (The bitmask changes to reflect the actual situation, so that if one controller fails, all LUNs are marked as preferred on the other). This extra information is present only in the dual bus case (the serial numbers being nulled otherwise). This permits a driver to determine, when configuring a device, that this particular path to the device is the preferred one or is an alternate non-preferred one. Moreover, the controller serial numbers are unique and visible to all nodes on a cluster, so that if a device name is chosen based on them, it will automatically be the same for all cluster nodes./

In addition, the HSZ firmware is being given the ability to notify drivers when a controller fails. This presumes that some devices are active on each controller, and works by having the HSZ detect the controller failure. If this happens, the next I/O to the good controller will receive a CHECK CONDITION status (unit attention). The sense data then uses some vendor unique sense codes for failover (and eventually failback) events and returns the good controller serial number, the failed controller serial number, failed controller target number, and a bitmask of LUNs moved. In addition, when this happens, the surviving controller kills (resets) the other controller to keep it from trying to continue operation.

This information can permit the processor to be notified of a path failure without necessarily having to incur timeout and mount verify delays. On VMS, however, a SCSI adapter on a failed path may have I/O in various states within its control, and if this is the case, some method of extracting it is needed. The usual path for this function is for timeouts to occur and force I/O requeue and mount verify. Where I/O is in progress to a device, there is no convenient external handle available to extract it (and the notion that as a side effect of a successful I/O on, say, mkb200:, we might stop and redirect all I/O active on DKD400:, seems likely to be far more complex and error prone than can be tolerated, if it can be done at all on all adapters). Therefore this information is likely to be most useful where the failed path devices are in fact idle. Where I/O is in progress at some stage within a SCSI adapter, it will have to be timed out or otherwise cleared from the adapter before a path switchover can take place. (This also means that in the event a transient failure occurs, nothing will be left "in the pipeline" to a device at switch time.)

Actual HSZ switchover is done by a SCSI START command (which is done as part of the IO$_PACKACK operation in VMS) so that host software has some control.

There is a proposal to the SCSI-3 committee which details a more general configuration, in which some number of devices are controlled by a set of controllers, where a device may be accessible from one or more of the controllers at a time. It is anticipated that LUN ownership might have to be established in this case via reserve/ release to set initial path preference (if only one path at a time may be used).

This proposal defines some SCSI commands which may be sent to a storage control device to report which controllers and devices are associated and to set up access. Since these devices will have their own LUNs and device types (apart from disks, tapes, etc. behind them) it is apparent that an io$_packack to a disk would have to have been preceded by some FC initialization commands. The unit init code of a new class driver may be the most logical place for such commands. Failover or failback is to be reported by ASC/ASCq event codes, same as for the HSZ.

While this suggestion is not yet definite, this specification does attempt to be generally compatible with it. (A server, for a specific case, can communicate with a control device if need be when a failover is signalled.)

In addition to issues with controllers like HSZ, issues already exist with the MSCP server in SCSI clusters. At times, disks are visible with two paths, one via MSCP and one via direct shared SCSI connections. Until the advent of SCSI clusters, a SCSI disk could be reached either locally or via MSCP, never both. Now, however, both paths can exist and be useful, but no failover is possible. Failover is supported with DUdriver, but not generally with other drivers such as DKdriver. Wherever there is at least one redundant path to a device, however, it is highly desirable that the system be able to switch paths invisibly to the good path when one of the paths fails.

As can be seen in Figure 1, there are several ways that an HSZ with failover can be connected to a system in addition to two SCSI ports on the same machine or one each on two clustered machines. In the top drawing, the disk on the HSZ might appear as 2DKA100: and also as 2DKB100 on the two machines directly on the shared SCSI busses, and would be visible as both names via the MSCP server in the third machine, if no code changes were made.

With our code changes, DKdriver would invalidate either the DKA100 or the DKB100 path at startup, prior to the MSCP server being involved, so that only one path would appear to VMS. NOTE HOWEVER that once the switching code was loaded, BOTH paths would be known to it, and the HSZ-preferred path would in fact be the one used.

If the disk is the boot device, of course, and loading of the switch code is later than very early in the boot path, in the top configuration it is not a problem so long as the systems in fact boot the same device and the CPU C does also. Only one device would be left visible by DKdriver on CPU A and B in the top picture, and CPU C would see only one MSCP served device.

The bottom picture is one which requires some configuration restrictions for boot time use, so that CPU A and B, and also CPU C, use the same port letter and allocation class. This would require that CPU C "see" a different allocation class database from CPU A and B if port allocation class were used, so that the same device name is common to the cluster. For use in data devices, however, the switching system can read a configuration file to assign either the name DKA100 or DKB100 to the disk, while using the local path, whichever that may be.

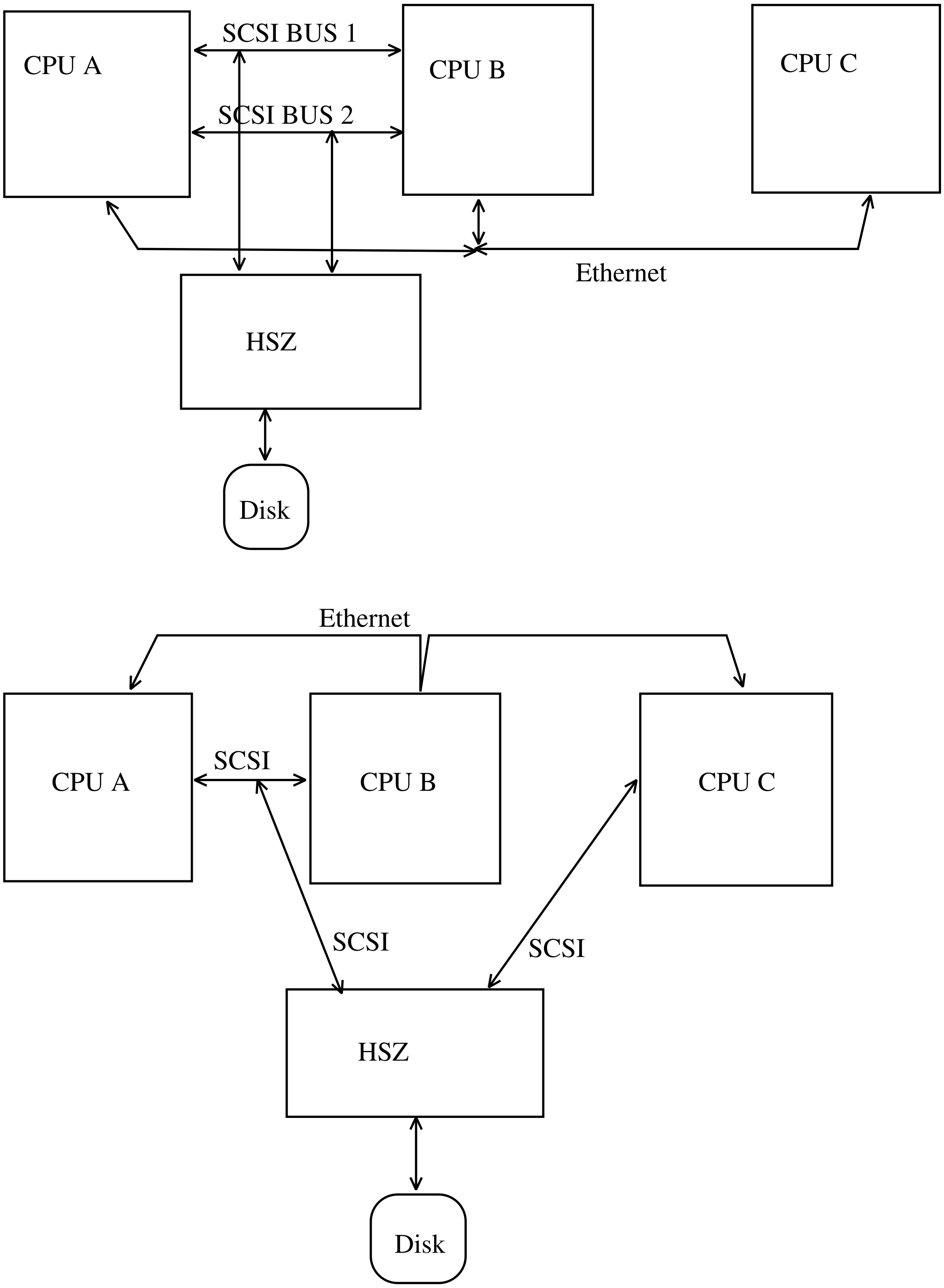

Fig. 1 - Some HSZ Configurations

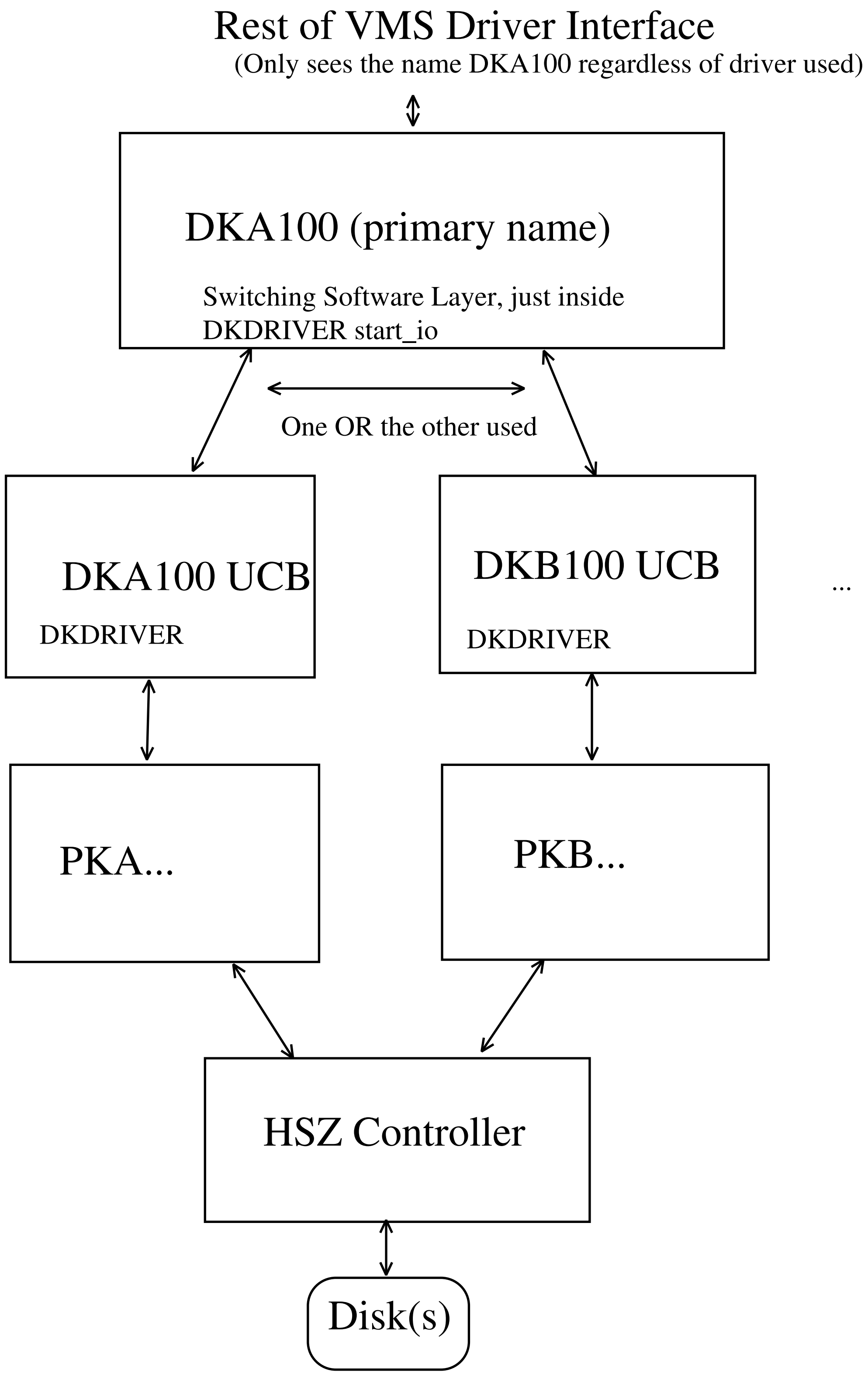

Figure 2. The Software Layering Used

As can be seen in Figure 2, switching as proposed herein presents a common **device name** to VMS, but the switching code can apply that device name **regardless** of which device path is actually used (and will select the online one at startup). This concept is vital to understanding what is being discussed. VMS sees ***only the external name the switching software provides, and not the underlying names.*** The switch code's purpose is to hide the fact that there are extra drivers, UCBs, port drivers, or the like, cleanly from the rest of OVMS. While this report covers directly only the HSZ failover case, the approach can be applied where one of the paths is using another disk driver and port driver transport to switch between that and another path, but there will be no notice from the controller to assist the failover.

Goals:
* Support HSZ failover for HSZ7x type controllers where two SCSI busses are connected to a single machine.
* Support other multipath failover cases such as SCSI to/from MSCP paths.
* Be compatible with planed HSG failover mechanism (which is generally similar to the HSZ one, with some differences due to the changes between SCSI-2 and SCSI-3)

Non-Goals:
* Support more than 2 busses (as yet)
* Support the case where both HSZ controllers are on a single bus (this is supported within the HSZ)
* Solve the device naming problem generally
* Dynamic routing or load balancing between paths to a device in full detail. (That is, it is expected that the solution must function to switch a failed path, but it is not initially necessary for a solution to load share between multiple paths dynamically in the first design pass.)
* Describe details of compatibility with the HSG proposed failover scheme.

Discussion of goals:

Much more complex situations may arise in the future, where devices are reachable via any of several paths. Controllers are under discussion which have 16 bus interconnects available to different computers, and which will need to do load balancing, and will need to have devices handled in such a way that confusion does not result due to multiple names. The approach discussed herein does not attempt to deal with this complexity yet, but to find a way to deal with the part of the failover problem defined by the HSZ firmware (HSOF 3.0 and later) which requires host CPU cooperation. It does attempt not to constrain its implementation too much, so that extension of the switching to more than two busses, routing of I/O dynamically between several paths, and failover between paths regardless of their method of connection can be contemplated as extensions to it rather than total reworks. All these are possible, but all will require additional design effort, which is not covered directly here. The techniques here appear to be usefully extensible in the directions mentioned, but the full set of issues around any of these other but related problems has not yet been addressed.

The configurations being addressed are therefore limited at this time to the dual-bus HSZ cases. The more general case of many paths via many controller types with possible load balancing is not addressed here, save in part and with important issues over how to generalize synchronization boundary conditions not dealt with in their full generality. That general discussion is beyond the scope of this design. The design proposed here is also a VMS variant of the kind of driver interface called "streams" in the Unix world. This is an interesting sidelight which may be suggestive but going beyond this sidebar comment is also beyond the scope of this design.

This document should be considered as the design spec for HSZ failover primarily, though critiques of the design where it may be over specialized in ways which will make it harder to solve follow-on problems might be appropriate.

Approach:

For HSZ, it was initially considered that some form of altering SCSI connections' purely SCSI structure based "routing" to devices might be feasible for the switching needed here. However, in principle, two SCSI busses can be controlled by entirely different SCSI port drivers, so that an attempt to alter connections on the fly at port level could involve considerable complexity in ensuring that port driver specific structures were initialized as the port drivers expect. (These initializations are not all alike.) Also, a "port level" approach does not deal with the appearance of multiple class driver units after autoconfiguration. Idle drivers might be revectored, but any links between SCSI structures and class drivers would need to be traced and reset, and any future asynchronous events would need to be blocked from access to the structures during this time, and any port driver specificities in the SCDT in particular would be a problem. Finally, this approach would not help with failover between SCSI and MSCP or other servers.

Since the failover scheme used in DUdriver is basically near the top of the I/O chain in the class driver, this seemed a more promising direction to go, and had the extra advantage that it can facilitate failover between DK and DU.

Therefore a simpler approach has been investigated. In this approach, a dedicated device switching subsystem inserts itself at the class driver external entry points and serves to abstract the device name used a level from the actual device driver paths. Thus, above this level, the OVMS system sees one device with one name which corresponds to the one piece of storage underneath. Below this level, two or more paths may exist, all leading to the same physical storage. This level's job, then, is to switch all I/O seen by the rest of OVMS (at a documented and fairly stable interface) between these underlying paths. It is not necessary that I/O order be preserved (since the SCSI drivers do not in general preserve it), except that in certain well defined circumstances such as path or cluster node failure, mount verify, and a few more, the guarantees offered by SCSI of order must be maintained. (See Figure 3)

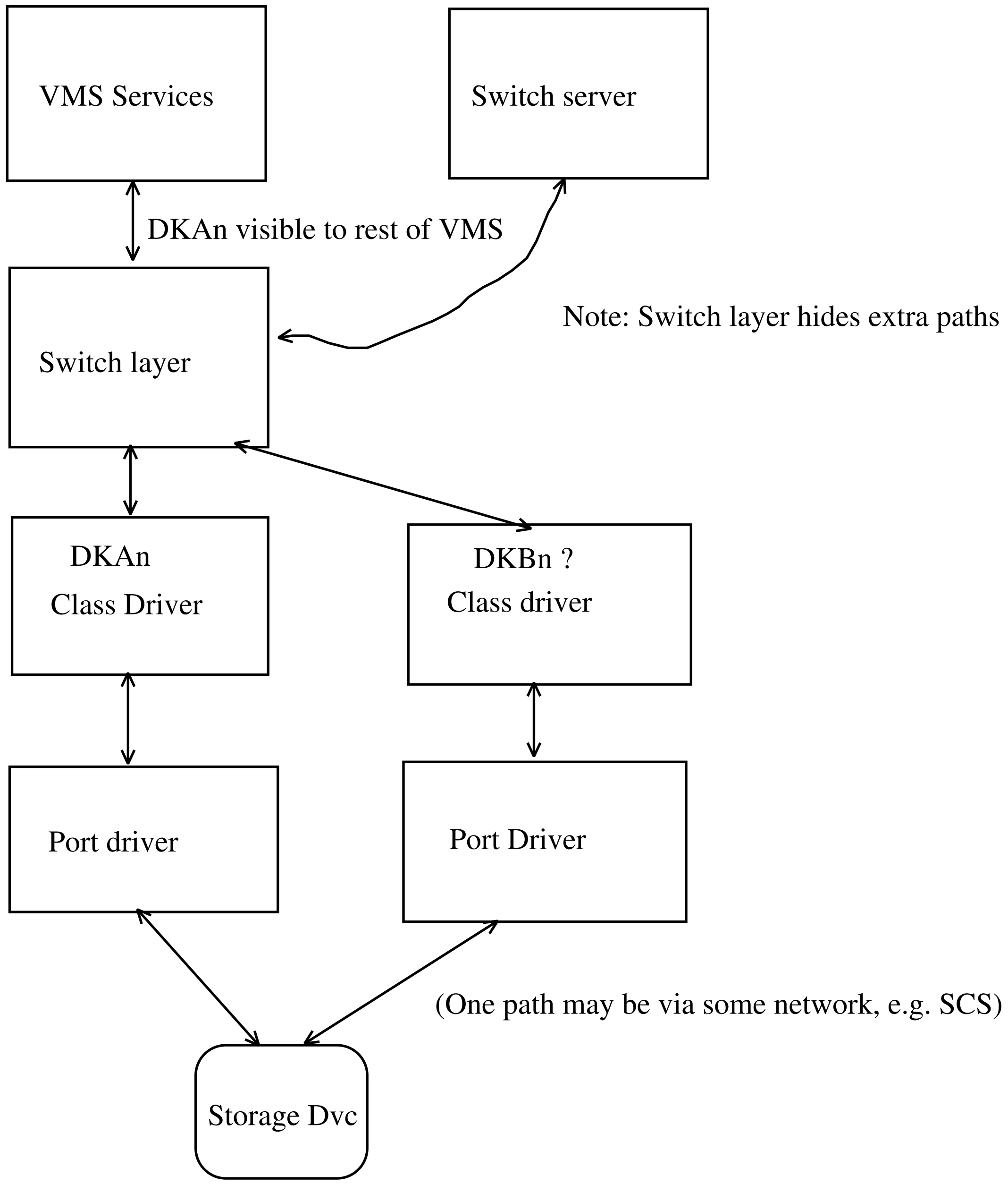

Figure 3. Layering with Switching Subsystem Added above Class Drivers

This approach involves small modifications to DKdriver (and possibly, but not necessarily, other drivers) to recognize HSZ units which are non-preferred path aliases of other devices and to mark them so that the MSCP server and normal VMS mounting services do not attempt to access them. This will ensure that for each device, one and only one mountable, serveable class level device appears. The alternate path will however still be autoconfigured, so that the SCSI connections will be created and initialized as at present by the class drivers. The alternate path will however have its data structures set so that they will be effectively invisible to normal VMS users. This will mean that the device will exist, but will not be found by VMS search routines as a device available for channel assignment.

Then at some point moderately early in system startup, but after autoconfigure, a switching driver will be inserted. This driver will implement the failover policy by gaining control at the class driver start_io entry point for the preferred path device, and doing monitoring or switching. Sufficient units of this virtual driver will be connected to handle all pairs of disks present, and a server will be started which will scan the device configuration (from the SCSI data base which by then will have been set up) and connect the pairs of disks appropriately, and also remain active awaiting notification of failures so that it can direct the failover of idle devices to a remaining good path. Only one server is required for any number of such devices.

(Insertion of the switching component earlier in the boot sequence is possible, and may be desired at some point, but it must occur at least after all local disks are configured. This may not be difficult with the new file oriented configurator, but remains to be investigated. The basic feasibility of the approach appears adequate even if startup is deferred to one of the startup scripts, though earlier connection may make it unnecessary to use a sysgen parameter, at the expense of some early boot code to effectively rename a device. For purposes of this report, the concept of early load of the switching functionality is deferred, to be replaced temporarily with some configuration restrictions on using the failover. When the boot path is being worked over in conjunction with supporting more device names, earlier load and configure of switching can more conveniently be addressed.)

(The switching driver will also intercept all other relevant entries pointed to by the DDTAB tables of drivers, to ensure that where the device is being accessed, the accesses are properly routed to the "live" device. Entries relevant are altstart, mountverify, pending io, auxiliary routines, and cancel io from current examinations; register dump appears not to need to be switched due to its calling usage. The pending I/O entry will be used primarily to ensure that I/O is seen even if a driver directly pulls requests off its queue.)

The intercept driver’s monitoring function will monitor I/O requests coming to the device so that when an IO$_PACKACK coming from mount verify is seen after the Nth time (initially, the third), this will be taken to mean that the I/O via the currently active path is infeasible, and that it is time to try switching. When this happens, I/O packets will have their path switches. The driver will either be set to requeue IRPs received to the alternate path driver (and gain control at I/O posting time to complete the I/O in the original device’s context), or to stop doingthis and allow IRPs to continue to the original start_io entry of the initially preferred path's driver. Also, some "Special" I/O status returns will be monitored (implemented as alternate success statuses in the current thinking) so that a server can be notified if an I/O returns from one controller and indicates the HSZ has found that its other controller has failed.

The switching driver can switch paths on command as well, provided that there is no I/O active on a device being switched. I/O is defined as active if an IRP has been seen at the driver's start-io entry point and has not been seen at I/O postprocessing.

IMPORTANT:

What VMS needs for valid file structures is that the device **name as seen by the rest of the system** be uniform. Once the switching component is present, this name can be that of **either** path, regardless of which device is actually preferred. The intent is not to force a preferred allocation of HSZ slots, but to set names uniformly, permitting the HSZ console choice of actual path preference to be honored. The switching takes place "under" the chosen device name, with the initial state of the switch being set so that the preferred device is used initially. If the switching software is to be loaded early in boot path, some cooperation with DKdriver to honor the HSZ preference (or later, generic SCSI3 preferences) will be needed. This is not expected to be a large amount of code.

DESIGN:

There are two new component, SWDRIVER and SWCTL, and some modifications to DKDRIVER used to produce the failover. (Similar changes can be made to other class drivers in a second pass; the switching software is largely independent of device class and can readily have those limitations removed for devices which cannot support mount verify.)

DKDRIVER CHANGES:

DKdriver is to be modified so that in unit init, when it looks at INQUIRY data from the HSZ, it determines whether this device is on a "non preferred" path (this being returned by the HSZ INQUIRY data). If so, it sets the DEV$V_NOCLU bit in its DEVCHAR2 field so that the MSCP server will not initially serve the non-preferred path. The preferred path for naming will be chosen as that with the lower controller serial number where possible (or the higher, depending on a parameter which must be set the same clusterwide). Thus, all nodes will see the same path, and it will be possible to boot the cluster even if one path's controller is down. (The problem of a shared SCSI bus being sampled by one node with path A down and soon after by another node with path A back up again is otherwise rather intractable.) In this way, boot time consistency can be assured in naming and access. It is of course vital that DKdriver do this, so that another system not directly connected may not start using the non-preferred device path via MSCP before we can mark it invalid. The system must use and know about only the one preferred name, uniformly across the cluster. (Data devices are a more tractable problem, since all one needs there is to ensure that the same names are used for mounts everywhere, and by the time mounts can occur, the switching code should be running.)

In the HSZ failover case, the device will come up with two aliases, and will return to each DKdriver unit the conroller serial numbers of the "current" and "other" path controllers. Thus a given device might be visible as, say, DKA300 and DKB300, but while (where the "A" controller happens to be the preferred path) DKA300 will be identified as a disk and come up normally, this code will cause DKB300 to be reset to not be visible to other nodes or to users. (Once the switching code is in place, the MSCP server will see only one name to serve, but that name will in fact represent two device paths either of which may be active, and which fail over locally.)

The HSZ will experience timeouts when a bus fails, which will produce mount verify conditions. In addition, should the HSZ detect a controller failure, it will allow failover to take place and will signal this by generating CHECK CONDITION on the next I/O to the "good" side controller. (In additjon, the server which is described later will periodically check the controllers to ensure prompt operator notice as well as failover.)

The CHECK CONDITION operations within DKdriver to handle UNIT ATTENTION will in fact return success with the current DKdriver. To preserve the status that the devices are operating correctly, yet allow the switching server to obtain the signal, DKdriver will, in this situation, return alternate success reports which will set the 16384 bit of the I/O status word (unused by DKdriver in any other context) and also the 8192 bit if this is a failback. These returns will be sent to the DKdriver caller. However, it is expected that the switching driver SWdriver will act upon them. The I/O status will "really" always be SS$_NORMAL in this case, and DKdriver will check the sense data flags to ensure that the (Digital vendor unique) codes are present before setting these flag bits in the return code. DKdriver will NOT however perform any switching operations on its own. This means that minimal DKdriver modification is made here, but the vital information needed is present and passed on by DKdriver to layers of the failover system above it. Where these alternate success statuses are seen by the switching driver, it will remove them prior to really completing the I/O, thus hiding any unusual behavior from applications or other VMS layers.

SWDRIVER

SWdriver stands for "SWitching Driver" and is a two way (currently) toggle switch sending I/O either to one disk or another, assuming the disks used are in fact the same but accessed over different paths. (Extending the driver to be an N-way switch should be straightforward, treating paths 3-N the same as path 2, but is not needed for any currently known problem. Future systems may however require this.)

If Bus B fails and some operation is completed on Bus A (these being the two busses on the HSZ40), the HSZ will generate CHECK CONDITION responses which DKdriver and other drivers need to be able to turn into statuses the switch can recogzize. The CHECK CONDITION data will indicate that Bus B has failed, not that anything is wrong with the current device on Bus A. To perform failover promptly when this happens, it will be necessary to have some server aware of the whole HSZ configuration and able to command switchover promptly. Accordingly, the switch driver is programmed to send a signal to a server when it recognizes such a condition, so that the server can command switchover to the remaining path.

This server can have the necessary global configuration information so that all devices can be switched to the good path. (The server will also send an IO$_PACKACK to get the device to come online at that time, before anything else is queued there.) Also, some code will be added to DKdriver to ensure the controller serial numbers are made available to the server, so that it can find the pairs of controllers automatically, rather than needing to have it generated by a customer.

Periodic polling of devices will also be added to the server component here, so that an operator can be notified of device failover. (There is a special I/O path in the switching driver allowing the server to contact all actually-known channels in spite of the otherwise opaque overloading of the chosen device name.) The server will initially determine device pairs by issuing INQUIRY packets using io$_diagnose, so that DKdriver need not store information about controller IDs. It will ensure that the UCB$V_NOASSIGN flag is set in UCB$L_STS of nonpreferred paths to help set these invisible, and will make such other modifications as shall be needed to ensure that the scan_device routines in VMS exec cannot see the extra paths either. These must scale so that multiple extra paths can be managed. The extra path will also be marked as DC$_MISC so that device scans in programs that look for disks will not find these extra devices. An additional check in the SHODEVUTL.BLI source will need to be added to ensure that these devices are normally not shown in SHOW DEVICE. Note that nothing in VMS will be affected by this since we are ensuring that the device VMS sees is normal.

Operationally, then, autoconfig does not change. Since DKDRIVER will be altered to ensure that no disks are served via multiple paths, the switch logic can be loaded during normal startup commands and need not run very early in the boot path. Tapes and generic devices for the most part are not made visible as early, and it is possible that resetting the alternate units' characteristics for those device types can be done by the switching software itself, after autoconfiguration shall have run. If this causes problems, the tape driver will need to be edited also to prevent too-early detection of tape alternate paths. Loading the switching code after full VMS is up simplifies it greatly, at the cost of failover not functioning until this code is loaded. Normal disk operation would be unchanged by the switch (the actual intercept is synchronized at fork level, which is necessary for any access to the intercepted path), but an HSZ controller failure would not be recovered if it occurred within the first few seconds (up to a few minutes) of system operation. However, once the software loaded, a switchover could be accomplished, presuming the failed devices were in mount verify state and had not timed out during the interval. Thus even in the case of a very early controller failure, a remedy could be applied partially "ex post facto". (The swdriver code would simply have to count MV Packacks starting after they had been going a while.) Only a system disk failure early on would not be covered in this way, since the recovery code would not load, and this can be considered much the same as a failure during early booting; a reboot would use the other controller and succeed.

In only one case does something unusual need to be done: when the boot disk is on the higher numbered controller. In this case, setting a boolean sysgen parameter will allow boot off a higher serial number controller by making it preferred. While this effectively changes the device physical names, a configuration file option will allow them to be effectively reset for all but the system disk. It is hoped that this will be a rare circumstance.

The system will then, when running, see one device name per device, and the path switching will take place below the start_io level in a way invisible to anything in VMS above driver level. By simply requeueing the IRP, high performance can be achieved and only minimal changes to driver operation (mainly to handle the new information in the INQUIRY data and the extra CHECK CONDITION flags) are needed, none of them of major import. The functionality here is completely orthogonal to the device naming scheme in use, and in practice it doesn't matter what the device name scheme is so long as IOC$SEARCHDEV can still find both devices. It is further expected that the qio server will eventually perform operations somewhat akin to this.

By functioning in this way, the system will avoid adding greatly to the complexity of DKdriver (et.alia) and can be extended to handle other failover situations rather simply, though the custom signals from the HSZ will not be used only in limited ways.

It should be added that for SCSI drivers, the mere startup of mount verify does not in itself mean that bus failover is appropriate, since SCSI RESET can be a normal part of system function. This is why the switch is not set to switch paths at the first pack-ack (or indeed at the start of the mount verify condition). This is also the reason why the switch does not simply intercept the start-mount-verify driver entry. In fact, the IO$_PACKACK will generate a SCSI START command on the new path, which the HSZ40 needs in order to switch its internal indicators. This situation is different from that obtaining for DUdriver, where mount verify generally does mean a path failure may have occurred.

SWDRIVER INTERNALS

SWdriver is an intercept driver which intercepts disk start-io entries. This is done by code which creates a copy of the DDT table, located in the intercept driver's UCB, and points the intercepted driver's UCB$L_DDT vector at it. This permits a per-drive intercept and is done in such a way that the vector can be intercepted by other similar intercepts totally reversibly, and in any order, just so they follow the connection logic (which has been published). (Because the intercepted DDT is located within the intercept driver UCB, the intercept code can locate the intercept driver UCB using this DDT. Some additional code exists to allow the code to be sure it has this data for its own intercept, not another on a possible chain of them.)

When the intercept is present, start-io for the "primary" path disk now points at the intercept address within a unit of SWdriver, which also knows the UCB addresses of the "primary" and "secondary" path devices. An IRP entering here is first examined to see if it is a mount verify pack-ack IRP (and counted; if 3 of these are seen in a row, SWdriver switches to the "secondary" path.) By using mount verification in this way, SWdriver assures that I/O through the failed path has been idled. (The mount verify driver entries are NOT used because for SCSI a mount verify condition does not necesarily mean a bad path.)

SWdriver also counts up outstanding I/O and arranges to gain control at I/O post time (so it can count down the I/O and post it). This is done by saving IRP$L_PID and replacing it with an address within SWDRIVER which will count the I/O down and, after replacing modified fields, perform a real I/O completion on the IRP.

Now if the I/O request is being routed to the primary path, SWdriver just calls the primary path start-io entry and returns. Since it is entered as part of the primary driver, it has all needed locks.

If on the other hand the path routed to is the secondary, SWdriver calls IOC$INSIOQC instead, redirecting the IRP to the secondary device. The primary device is unbusied in this case also, since SWdriver is acting in lieu of the primary device, which will not in fact get any I/O when it is routed this way. IRP$L_UCB is pointed at the secondary device during this operation, to be replaced with its original value when I/O is posted.

In all cases, when the I/O completes (and without a detour through IPL 4 if assembled that way), SWdriver regains control. At this point it decrements the outstanding I/O count, replaces a few IRP fields it needed to regain control, and completes the I/O (via a call to COM$POST, since it has no right to alter the underlying driver's busy or unbusy state). If on the secondary path, SWdriver checks the I/O to ensure that mount verification is begun on it also, as this would not otherwise be done. The I/O checking, mount verify processing, and postprocessing is all done in the context of the primary path, so that the primary path remains mounted and apparently active, though the secondary path may in fact be the one in use.

To save volatile parameters from an IRP during the switching, SWdriver currently overwrites the IRP argument areas (which are used prior to start_io but are not used after that point) to hold a number of IRP fields which are being reused to route the packet.

The usage is as follows:

Field:	Saves contents of:
IRP$Q_QIO_P1+4	IRP$L_STS (if fast finish shortcut only)
IRP$Q_QIO_P2	IRP$L_MEDIA (block number)
IRP$Q_QIO_P2+4	IRP$L_PID (PID, used to capture post processing)
IRP$Q_QIO_P2+8	IRP$L_UCB

While it is of course possible to allocate *another* structure to hold this information, these IRP fields are used by no other driver code since they are present only to make the $QIO arguments available to FDT code, completed before start-io code can be run. It may be desirable to consider extending the IRP to supply dedicated fields for this functionality, or perhaps to consider reusing some of the structures shadowing uses where the device is not shadowed, and otherwise use some separate structure. This approach does however provide very fast operation. The fields mentioned are saved and restored so that the IRP can be passed to another driver, yet have its I/O posted in the context of the correct driver. Saving IRP$L_MEDIA is necessary to ensure that IRPs which are re-inserted in device I/O queues at the start of mount verify have the correct block information. The UCB and PID fields must be altered to redirect the IRP to another driver and regain control when the I/O is posted by that driver. The IRP$L_STS field must also be treated this way if a "shortcut" to avoid IPL 4 processing is used, which is also present to minimize extra code caused by this approach, using the fast path I/O processing to eliminate most of the completion overhead which would otherwise be seen due to the need for two request completion calls.   SWdriver also has an interface for program controlled path switching. This is built using the IO$_RETCENTER function code (chosen rather arbitrarily as one that is not used generally and thus can be defined without much confusion) sent to SWdriver itself. (It is meant as a private interface.) This code passes a single parameter, 1 or 2, to indicate whether to take the primary or the secondary path. When this function is sent to SWdriver, it will switch to the selected path, provided that its count of active I/O (I/O seen at start-io and not yet seen at I/O post) is ZERO. When the HSZ sends notice that "the other controller has failed", the switch server sends a **packack** to the currently inactive path to flush out all I/O before switching in this way. The secondary device exists independently and is just addressed directly. The primary device, recall, has its start-io entry stolen, so there is code in SWdriver which will notice an I/O with all I/O function modifiers set, and which will strip all these and send the I/O to the primary path, whether it is connected or not for other purposes. The reason for this packack is to ensure that any "left over" activity on the path will be flushed, and also to issue the necessary SCSI functions to activate the path. This will be required for HSZ40 and up, and is likely to be important for others.

To interact with the failover server, SWdriver sends messages to a mailbox allocated by the failover server and whose UCB address has been stored in part of the SWdriver UCB extension. Thus SWdriver can use CALL_WRTMAILBOX, a documented interface, to send messages to the controller indicating that a mount-verify-initiated switchover has occurred, or that an I/O status with the 16384 bit set has been seen. These messages are simply sent, provided the server is present. The server is sent enough information to tell which devices are involved, and one server can handle any number of pairs of switched devices. It has the convention that SW units must be allocated and enabled starting with unit zero. (There is a UCB table in SWdriver which limits the number of units permitted, but its size is an assembly parameter and can be made as large as needed. Currently it is set for 500 units or less.)

Mount Verify and /Foreign Mounts

The mount verify service functions only with a normally mounted device. It is desirable for similar service to be optionally available for foreign device pairs, where a database vendor may be handling the disk itself. This cannot be the default, but is sensible as a general matter.

Fortunately, there is a server available which is able to handle much of the complexity here. If this function is implemented, it is feasible for swdriver to notice error codes that currently result in mount verify being used, communicate these to the server, and have the server/switch driver call mount verify entry points (if any) in the appropriate drivers (to flush I/O) and within the intercept driver to requeue any I/O that may have been outstanding, handle device busy, and for the server to issue the periodic packack functions via its private "wormhole" I/O functions permitting access to separate paths as needed. (The "wormhole" functions use patterns of some of the function modifier bits as flags as currently planned, so that the design scales easily to a modest number of paths, one or two dozen perhaps being a practical maximum. This should exceed what will be needed.)

By the use of such functions, this system should be able to provide what amounts to mount verify functions on foreign devices, and thus to handle failover.

Defect Containment

The investigation has resulted in a driver and control suite already which will serve as a source of a code count. The software written for this purpose (not counting some library functions used to allow the optional configuration file to be free form) totals some 3216 lines of code. It is estimated that another ~250 lines of code will be needed for the automatic controller-pair recognition, and the DKdriver lines already added (to side copies) to support these functions total 180. Thus there are so far about 3400 lines of code and the total for HSZ failover functionality may be expected to total when all is said and done 3650 to 4000 (to pick a round number) lines of code.

The bogey number of defects expected in 4000 lines of code at one per 40 lines of code would be 100. However, for code which is unit tested already (the driver and control daemon code) this estimate is reported high, and an estimate of 10 defects per KLOC is suggested for that segment of the code. This would mean about 34 defects in the code so far, plus another ~10 in code to be generated.

Not all of this code is new (in that some older virtual disk driver examples were built on which have been functioning for several years) and the switching driver code has been tested in one system, which is why it is expected that a lower defect count will cover the code so far.

Methods for defect removal include (in addition to unit tests):

* Overall design - minimal modifications will be introduced into the (already complex) SCSI drivers to support the failover functions. This can be expected to be the chief contributor to defect containment, since the effects of changes to existing SCSI drivers form a small fraction of the overall effort and their function is limited to reporting information to the failover system on the whole.
* Reviews. It will be important to have the code in the driver reviewed so that its design, and particularly its detailed control flow, can be reviewed. The same goes for the server components particularly where privileged.
* Stress testing. The code must be tested in SMP and large cluster environments to catch any timing subtleties.