/

VMS Functional Specification for Multipath Failover

DIGITAL CONFIDENTIAL

Revision/Update Information: X1.1.0

Glenn C. Everhart

26-June-1997

Abstract:

This functional specification describes the functional interfaces for Multipath Failover between multiple paths to storage devices.

Digital Equipment Corporation
Nashua, New Hampshire

OpenVMS Multipath Failover Working Functional Spec
Digital Confidential

Introduction

Due to the introduction of non-DSSI devices with multiple physical paths able to access them, the need has arisen for a way to allow access to them to failover from one physical path to another within a VMScluster, so that the cluster's ability to tolerate faults might not be degraded by choice of non-DSSI peripherals. These peripherals include normal SCSI disks connected by a shared SCSI bus ("SCSI clusters") and HSZ type controllers which have more than one controller to connect them to their hosts, among others.

Because the switching of DSSI disks depends on the design of MSCP in a very fundamental way, it has not proven feasible to use the identical way to switch non-DSSI paths that MSCP has used. Instead, a switching strategy based on using generic OVMS device properties has been devised, with a small amount of IO Exec infrastructure work added in to make this a low overhead choice. Because this infrastructure work has other value beyond this project, this document must describe both the operation and control needed for multipath switching and how that code works, and the I/O infrastructure which has been added. This will be done in separate chapters, with infrastructure discussed first, for the benefit of those who may want to use these facilities themselves at a later date.

I/O Infrastructure for Layering

In other operating systems, it has often proven convenient to add functions to existing drivers by means of pre and post processing the I/O with added functions. Such a facility exists within unix, called streams. In OpenVMS, this can be done with I/O interception, but the path has typically had several performance problem areas:

> * When code which has intercepted an I/O request gains control, it often needs to locate intercept specific data, and when there is no standard way to do this, an ad hoc solution must be devised. There have already been numerous problems introduced by systems products which used intercept techniques incompatibly, causing unwanted behavior or crashes.
>
> * When an IRP is arranged to be handled with an intercept or filter layer, postprocessing control has been regained by putting a postprocessing routine address into cell IRP$L_PID. Control is transferred here from the IPL 4 postprocessing code. The need to invoke an IPL 4 interrupt is a disadvantage, compounded by the need to get back to IPL 8 to actually complete the I/O and start new I/O with many drivers. This detour through IPL 4 is by itself a performance killer.
>
> * In addition, information needed to restore the original values in cells like IRP$L_PID must be stored somewhere. Given that the IRP itself has no cells for such storage, it has been necessary either to allow only one IRP at a time to function, or to keep information in auxiliary structures. Allocating and deallocating these in pool can be expensive.

To permit orderly implementation of multipath switching as an interception based software layer, a number of system changes have been made. These address the problems above, and define a way to perform interception which permits multiple intercepts to gain system access, to be added or removed in any order, and to function with low overhead when present.

*Code has been added to IOSUBNPAG which provides a means for an IRP to be postprocessed at fork IPL without a detour through IPL 4 processing, and which allows a postprocessing routine pointed at by IRP$L_PID to be called directly from fork IPL, without detours. This means that an intercept can add code at the start of an IRP's path through the system which will cause control to be regained at the end, and the postprocessing control transfer will be cheap.
* This code can be used to insert postprocessing into any desired I/O stream. It presumes and requires that the original I/O context be saved and restored when using it, so that postprocessing tasks intercepted may run normally after the intercepting code. (Any postprocessing code which tries to shortcut postprocessing may break layered I/O if it doesn't do this.)
* Code has been added to IOC$INITIATE which checks the kernel stack and forks (to clean it off) if the kernel stack is being pushed too low. This permits all drivers to use CALL_INITIATE safely so that all IRPs pass through the underlying driver start_io entry. DKdriver has been edited to do this.
* A standard format has been defined for an I/O intercept permitting multiple intercepts to coexist. The notion is that an I/O intercept is done by making an edited copy of the Driver Dispatch Table (DDT) and pointing UCB$L_DDT at it. The DDT copy is physically located within an "I/O intercept area" which for multipath failover happens to be located within the intercept driver's UCB. This intercept block contains some defined data storage, and if present as part of a larger structure, the larger structure can then be accessed. Multipath failover also defines a call to any unit of the intercept driver which can follow any links which may exist and determine whether a particular device has been intercepted by a given driver. So long as the format of the "I/O intercept area" remains the same, multiple intercepts can be added to a given I/O. This more or less exactly duplicates the kind of function that streams gives in unix, or that exists in NT.
* A compatible extension to the IRP definition has been made to hold context information. This consists of a context stack area large enough to hold several intercept contexts (which are 6-8 longwords in size typically), a context stack pointer, which must be an address within the IRP it is in, and a flag cell. The new format is recognized by its longer IRP length, by presence of and validity of its context stack pointer, and by presence of a "valid" flag word. The flag word also is used to keep track of cases where the context stack has overflowed and auxiliary storage exists. The possiblity that old style IRPs might be generated by privileged packages, and the possibility that a context stack might fill, means that intercept code must be able to store context in auxiliary structures if IRP space is not available. However, the expectation is that IRP based storage will be present in nearly all cases, and that this will make it unnecessary to deal with auxiliary storage. All new IRP fields are at the end, so no old offsets change at all.

The IRP extension has the format:

```
Old IRP (as in VMS V7.1)
IRP$L_CURCSP                ; Current context stack pointer, must point inside
                            ; this IRP
IRP$L_STKFLGS               ; Flags, flag IRP$V_ONSTACK implies the current
                            ; context stack has not overflowed
                            ; (Others zero for now)
IRP$A_CTXSTK                ; Block of storage for context stack, 64 longs
                            ; currently.
```

It is of course critical that an intercept maintain the context stack correctly. It should set the context stack pointer to a higher address in the IRP, then fill in its data behind it, so that another intercept will find the context stack pointer pointing at available space. When the intercept's postprocessing occurs, its data will be at the top of the stack so it may retrieve it, then lower the context stack pointer. Since each intercept's data will be at the top of stack when it is called, context data formats need be known only to the intercept which created them. This provides useful flexibility for other uses.

You will note also that the context stack pointer is an absolute address. This is done so that cloned IRPs will not be taken to have valid intercept data in them. Applications and drivers which create additional cloned IRPs in general set the IRP$L_PID field for themselves to have a chance to delete these IRPs. Thus the problem of an IRP being made to stray into an intercept's posting code cannot exist any more than it exists now with code such as MSCP server which sets this field.

The one case where the new fields can cause grief is where someone allocates a structure the size of an old (VMS 7.1 or less) IRP and copies the first part of a new IRP, including the header with irp$w_size in it, into this structure. Doing so will put an invalid size into IRP$W_SIZE and probably crash on deallocation. However, this is a highly suspect programming practice which needs to be fixed in any case, as it has many other ways to corrupt pool if used.

The Intercept Block

The standard intercept block has the following appearance:

The following area is the start of the intercept block needed for intercepts of driver entries. It has the following structure:

Unique ID - generally the DPT address of the intercept driver, which will
be unique for that driver on a given machine.
Intercept DDT - the address of our interceptor's DDT. That is, if a second
intercept occurs this cell should be filled in with a pointer
to the interceptor's copy of the DDT, so this becomes a
forward linked list outbound. If there is no intercept this
should be null.
Previous DDT - This cell holds a pointer to the DDT address as it was in
the ucb$l_ddt of the intercepted driver before we intercepted
the I/O and replaced the address with our own. This is an
inbound list, singly linked. We use these two lists to permit
intercepts to be added or removed.

(Two lists are used here since the intercepted UCB in general has nowhere to keep a list head, though we need to be able to go either way.)

ICsign - This cell holds a "magic number" which acts as a flag that
the intercept we find is using the same scheme we are. This
value is computed as hex F0070000 + DDT$K_LENGTH +
256*(length of the part of the intercept block that
precedes the DDT copy). Thus it is sensitive to the amount
of "stuff" preceding the DDT copy and to the DDT length.
Flags 1 - Longword of flags. Only ones defined are fi8ok and
nofipl8, to flag whether IPL8 postprocessing could be used
with the intercept code of this intercept and all
prior ones.
Flags 2 - Spare longword of flags
Userfiller[8] - 8 more longwords, reserved for any third parties who

might need or want to keep extra information in their
intercept blocks (SWdriver uses the first two longs to hold
the SW UCB address and the start of the array of path pointer structures address.)

DDT - Copy of the intercepted driver's DDT, edited to point to whatever code is inserted first. The theory is that code wanting to continue with the intercepted code original function can follow the back link and use that to call the original DDT functions.

Note too that the DDT contains an FDT pointer and several more, and that by inserting a copy of the original FDT with edits it is possible in like manner to chain FDT intercepts. At the end of an inserted FDT bit of code, one would locate and call the previous FDT table's routne.

The presumption is also that intercepts are added by grabbing the intercepted driver's UCB$L_DDT pointer and that intercepts generally do not know of other intercepts' existence, save that when locating their intercept UCBs (or other data structures; the intercept blocks don't have to be inside UCBs) they check that the one they pick has the correct driver DPT address. Thus their intercepts would be called first, so that "inner" pointers grabbed at any stage will be valid. (That is, pointers to "previous" routine addresses grabbed by an intercept driver.)

```
/* prepend start */
 long ucb$l_uniqid;          /* Unique ID to identify this UCB */
 long * ucb$l_intcddt;       /* Our interceptor's DDT addr */
 long * ucb$l_prevddt;       /* Previous DDT address  */
 long ucb$l_icsign;          /* Unique "magic number" that shows */
             /* that this area is a DDT intercept block in this */
             /* format. The magic number encodes the DDT length */
             /* to ensure compatibility in linking also.             */
 union xx5 {
             UINT lfg;     /* Block number */
             struct {
             UINT          ucb$v_fi8ok : 1;            /* OK to post at IPL8 */
             UINT          ucb$v_nofipl8 : 1; /* NOT OK to post at IPL8 */
             } pa;
            } ucb$l_icpflgs;
  union xx6{
             UINT lfg2;    /* Block number */
             struct {
             UINT          FILLER : 1; /* filler; this for future flgs */
             } pa;
            } ucb$l_icpflg2;
 /* Since this same intercept will be needed for all user intercepts also, or the traceback will fail, leave some space to be used by such intercepts should any third party applications need it. 8 longwords should be enough  for anything I can currently imagine, particularly since it can be used  to point to other structures should more space be needed. */
 long userfiller[8];         /* Space for user intercept dat$
/* prepend end */
  DDTA ucb$a_vicddt; /* Our intercept DDT copy      */
```

/* What follows is not always needed in the intercept block, but if FDT routines are to be intercepted these fields are recommended. */

```
long    * ucb$l_backlk;     /* Points at intercepted UCB */
FDT     ucb$l_myfdt;        /* Copy of user FDT */
long    * ucb$l_oldfdt;     /* Pointer to previous FDT */
long    ucb$l_mungd;        /* Flag that this UCB was altered */
```

Note: The DDT area starting at offset ucb$a_vicddt must be 8 byte aligned, which must be handled by seeing that the entire intercept structure is defined so this will be the case. The offsets within the structure must not change.

As can be seen, there are pointer cells pointing to what an intercept intercepted and to what intercepted the intercept. These are both filled in at intercept time or at cleanup time so that searching for one's own intercept is possible. The uniqid field is a value unique to each driver, the DPT address in swdriver. It must be unique for each driver in a processor (and the presumption here is that a driver does not intercept itself ever). The icsign field, on the other hand, is a pure magic number which lets us be sure we are dealing with the right intercept format. It is computed as

^XF0070000 + DDT$K_LENGTH + <256*<length of the storage from uniqid to the end of the DDTA field inclusive>>

which means that a change to the intercept block or to the DDT size automatically gives a different magic number.

The flags **fi8ok** and **nofipl8** are to be filled in based on a driver's indicating its ability to have I/O completion handled at IPL 8. In general any driver that can handle fast-io will be able to do this, but there can be exceptions. It would be desirable to allow there to be a UCB flag so that individual drivers could control this. At the moment no such provision yet exists.

Notice that the provisions made here allow intercept code to find its data quickly with some address arithmetic, and to add postprocessing code with minimal overhead.

At I/O posting time, the usual (V7.1 and earlier) code path presumed that if the cell IRP$L_PID had been filled in with a negative number, this would be taken as a system address and called with R5 pointing to the IRP, rather than normally post processing the IRP. This operation took place at IPL 4, and insertion of postprocessing code could then mean that a need to rise to fork IPL would exist.

The new code adds some processing to the IOSUBNPAG module. If the **fast-finish** and **finipl8** bits are both set in the IRP, it will be postprocessed at fork IPL and a routine whose address is stored in IRP$L_PID will be called from fork IPL, holding the fork lock, and without having gone through any detours via IPL 4 and IPL 8 interrupts. It is expected that the finipl8 flag will be cleared before a final postprocessing is done on the IRP in question, but the added code will simply be called with the Alpha JSR instruction, not needing IPL adjustment. This option need not be used, but is present to reduce the overhead of intercepts which need to perform some accounting after each I/O is done. The switching subsystem needs to do this, and as a result of this modification at least two PALcode calls per I/O can be avoided.

It must be noted that this call to the address at IRP$L_PID is not preceded by any calls to VMS cache routines such as CACHE$IOPOST as the IPL 4 calls are. These calls are intended for use when inserting processing ahead of normal post-processing and it is expected that the IRP will be restored to its initial state and re-posted from the intercept routine to allow other processing to occur Shortcuts to I/O completion violate the layering philosophy, and will bypass routines which VMS caching inserts in normal post processing (and may bypass other special purpose code in the VMS I/O posting code), possibly causing it to work incorrectly. Other I/O posting channels call these routines. This one is designed to be used where the IRP is passed on to the next of a possible chain of processing, and .is optimized for such use. The presence of the fast-finish bit is simply an artifact of the simplest and shortest code modification able to accomplish this.

Switching Driver Use of Infrastructure

Before moving to the switching system functions, a few words about the use of the above provisions by the switching system.

The switching subsystem intercepts a driver's Driver Dispatch Table and replaces entries for start_io, mount verify start/stop, auxiliary processing, I/O cancel, and pending I/O. These entries are also saved for each path known to the system, though only the first DDT is actually replaced.

The switching driver contains a set of structures describing each path used and holding a few statistics about the path (i/o count, error count, time of last switch). These are physically located in a block within the SWdriver UCB extension. To make these simpler to find within SDA or Debug, where it is not feasible to call SWdriver itself to give its intercept address, the userfiller[0] cell is filled in with the address of the SWdriver UCB and userfiller[1] is filled in with the beginning of the path array address. The array is an array of PTHPTR structures defined as follows:

```
typedef struct pthptr {
    DT_UCB * ucb$l_hstucb;          /* UCB of path              */
    void *   ucb$l_oldmv;           /* original mount ver entry */
    void *   ucb$l_oaltst;          /* original altstart entry */
    void *   ucb$l_qirp;            /* original pending_io entry */
    void *   ucb$l_cancl;           /* original cancel io entry */
    void *   ucb$l_aux;             /* original auxiliary io entry */
    long     ucb$l_enapth;          /* path enabled flag        */
    uint64   ucb$q_pthtim;          /* time of last I/O         */
    long     ucb$l_pthios;          /* Count of I/Os on this path */
    long     ucb$l_ptherr;          /* Count of errors on this path */
    } PTHPTR;
```

The array of these structures contains, currently, 8 entries. An unused entry can be found by the ucb$l_hstucb cell being non-negative. The UCBs pointed to by other than the first entry will be removed from system lists, and so can be found only by following the UCB$L_DDT pointers if UCB$L_DDT is not identical to DDB$L_DDT and finding these pointers within the intercept blocks.

Switching Subsystem Infrastructure
Switching Driver Calls

The switching driver is the central control site for switching control and provides a number of routines to allow external monitoring and control of switching and related utility functions. Each processor doing switching will have SWdriver units loaded, one for each "tuple" or set of paths to a given device. These paths are described internally by the PTHPTR structure shown above.

SWdriver can be controlled by issuing $QIO requests to the SW unit controlling a tuple or by calling a routine within SWdriver which is pointed to by the SWdriver DDT. The first method requires use of the IO$_PHYSICAL function (which is not generally used but which requires PHY_IO privilege to use) and is issued from process context. The second method is usable from elevated IPL and consists of calling a routine pointed to by the SWdriver DDT's ddt$ps_mntver_2 entry.

Where an IO$_PHYSICAL request is sent to SWdriver, SWdriver expects P1 and P2 to be the address and length of a 528 byte (or longer) buffer containing information about what to do, and which is used to return information in. The buffer is interpreted as having structure

```
Func:   .BLKL   1     ; Function to do
USL:    .BLKW   1     ; Unit, or whole long is used
USH:    .BLKW   1     ; Subfunction switch (for func.3)
ULEN:   .BLKL   1     ; other data
Rest:   .BLKL   128   ; more data
```

The functions are: (X = don't care, blank if full longword used by prev)

Func	USL	USH	ULEN	Rest
1	Path #	X	Len of dvc name	Dvc name, ASCII Connects the SW device to the named device, and if the device is not the first device on the path, hides the device. Uses given path number if the path number is in 0-7 range, or uses the next free path if path number is not in this range. Normally the path number will be 65535 on call so the next free path will be assigned.
2	X	X	Len of dvc name	Dvc name, ASCII Disconnects the named device from a switched tuple. (Present mainly for testing; not expected to be used save for diagnostics.)
3	Path #	0	X	X Switches the path to the path number in USL (0-7 with current implementation) if the device is idle. Returns an error if the device is not idle. (Idle means no outstanding I/Os.) Note that it is up to code calling this to issue a packack along the new path before use.
3	Path #	1	X	X Sends an I/O that makes the tuple UCB go and stay busy. It is not counted as an outstanding I/O (and does not go to the underlying devices) but is used to allow a device that is busy to be made forcibly idle.
3	Path #	2	X	X Completes the I/O which is holding the tuple's UCB idle, so that it can be unidled. Thus one can switch a busy path by using subfunction 1 to make it idle, subfunction 0 to switch it, and subfunction 2 to reenable it.

					If the $QIO parameter 4 is set to -1 when the call is made, the outstanding I/O count is zeroed also. This is considered an emergency override to allow switching where something has been wedged.
4	X	X	X		X
					Reports on the tuple. Returns data in the calling argument block as follows, using the "Rest" area:

Length of table in SWdriver in paths (currently 8) (Longword) Size of PTHPTR structure (Longword) Current path index (Longword) Time of last path switch (Quadword) For each assigned path: I/O count for this path (Longword) Error count for this path (Longword) Time of last switch for this path (Quadword)

5	UCB of device	X	0	Tells whether a given device indicated by its UCB address is intercepted by a SW device, and if so returns the SW UCB and unit number of the intercepting SW device. Any SW unit, including the template SWA0: unit, can be sent this request and it will determine the interceptor, even if the interceptor is a different SW unit. Thus normally this kind of request is sent to SWA0: which is loaded early in boot and is the template device for all SW devices. The output in the "Rest" area returned is as follows (All results longwords):

SW unit number intercepting device.
Address of PTHPTR arrays
Path index currently in use (0-7 currently)
Address of table of path selection order information (Table of longwords giving the path selection order for each path)
Length of tables in paths (currently 8)
Length of a PTHPTR structure
Intercepted UCB address (should match the input UCB)
Current outstanding I/O count on the device

6	Server IPID	Mbx name len		Mailbox name Store a server's mailbox and IPID information in a SWdriver unit to allow the SWdriver unit to send messages to the server via the mailbox. Normally called only by swserver. The mailbox name is just the ASCII name of the mailbox device.
7	X	X	X	X Clears out server information from SWdriver UCB, undoing function 6
8	X	X	X	X Returns IOSB of 6631 if this is a SW device with no paths assigned, or 6635 if it has one or more paths assigned. Used to sanity check that the device is a real SW device prior to sending additional queries. Also returns current path number in the second longword of the IOSB.
9	Path #	X	UCB of device	

					Same as function 1 (connects device path) but takes UCB address instead of device name.
10	Path #	X	UCB of device		Same as function 2 (disconnects path) but takes UCB instead of device name
11	1	X	X		Returns currently used path index information. Info in longwords is filled into Rest area as follows:

Index of current path (0-7)
Address of full array of PTHPTR structures
Address of currently used PTHPTR structure

12	X	X	X	X	Returns the same information as function 5, but for the SW device sent the function, not the SW device intercepting some particular other device.

Note that the format of the SW device UCB is known only to SWdriver, but certain structures are made available to other programs by these means so that various controls can be done. The SWdriver code generally assumes that all structures in its UCB are synchronized with the forklock when writing is to be done.

Since issuing $QIO from high IPL is impossible, these functions are also duplicated by a routine which is callable via the DDT$PS_MNTVER_2 cell in the SWdriver DDT. The call sequence is

int sw_fakeformat(char* buffer, int bufsiz, SW_UCB *ucb)

Where buffer and bufsiz describe a buffer as above, and ucb is the UCB of the SW device which controls the tuple of paths. Subfunctions 1 and 2 of function 3 do not operate in this environment, and function 8 returns only the 6631 or 6635 values as the function value. Generally the function value returns the status. Apart from these differences the functions are exactly as above.

At calls to SW_FAKEFORMAT it is assumed that the caller holds the **IO write mutex** and the **forklock**, to ensure that operations are properly synchronized.

Switching Subsystem Infrastructure Control Program Functions

The switching control program is a separate image which has a number of capabilities for setup and diagnosis of the switching subsystem. These are designed to permit manual setup of tuples of paths, inspection of statistics, manual switching, and path idling.

Paths are selected from enabled paths (any path connected to a SWdriver unit is enabled by default) in the order of the path ordering array within a SWdriver UCB. Initially this is a simple round robin order which matches the order in which paths were set up. The control blocks which contain the path enabled flags and the array of longwords containing the path order are externally available via calls to SWdriver as described above, so no further provision exists at this time to set the selection order or enabled state of paths. However, an application in kernel mode can acquire the forklock and rewrite the order array as it likes or clear any enabled bits to control this. While the control program initially does not have this function, it can be added at any time.

The control program supports the following commands:

SETUP
The command
  SWCTL/SETUP SWA0: devicename:

sets up a new SW unit and associates the device "devicename:" with that tuple as its primary path name for all subsequent tuples. It creates a new SWAn: device, and defines the DCL symbol **SW_DEVICE** as that device name for use by subsequent commands. The command will fail if SWA0: is not loaded to be its template, if the process lack CMKRNL privilege, if the device in devicename: cannot be found, or if (an option) a SW unit other than SWA0: is specified and it has already been set up.

Normally one sets up a SW device once, then uses the ADD function (see below) to add more paths to it.

ADD
The command
  SWCTL/ADD [/SECOND] SWAn: device:
will add **device:** to a previously setup **SWAn:** device. If the optional /**SECOND** qualifier is present, SWTCL will look for a UCB linked with the original UCB via the UCB$L_2P_UCB field as the UCB to be added. Thus, if one has a direct SCSI connection and an MSCP served path to the same device, the /SECOND qualifier permits one to automatically specify the hidden second path as the one to use in a failover situation.

REPORT
The command
  SWCTL/REPORT SWAn:
will print out a brief report showing the device name associated with the SW device, and some statistics about each path.

An example follows:

```
$ SWCTL/REPORT SWA2:
 Devices named _$33$DKA204: Switching Statistics
There are 2 paths, current is 0
 Path 0
 I/O count: 47094, Err count: 0
 Path 1
 I/O count: 0, Err count: 0
```

This is useful to inspect activity information for each path.

SWITCH
The command
  SWCTL/SWITCH SWAn: m
(where an example might be
  SWCTL/SWITCH SWA2: 1
to switch to path 1 from path 0 on the SWA2: device shown above)

will switch the SWAn: device given to use path **m** (where currently m ranges from 0 to 7, if there are 8 paths associated with a device), **provided** the underlying device was idle to begin with. If it was not idle, the path is not altered.

IDLE
The command
SWCTL/IDLE SWAn: m
will cause the device associated with SWAn: to be made idle by having it sent a dummy I/O request which is not processed. This causes the device UCB Busy bit to become set and remain set, which will allow it to drain all I/O requests. the device will remain idle until the UNIDLE command is given (see below). The idea is that this command can be used to idle a device so its path can be switched. The command does not by itself wait for the device to become idle; it merely stops new work to the device. (The value of m doesn't matter here.)

UNIDLE
The command
SWCTL/UNIDLE SWAn: m
will complete the dummy I/O which SWCTL/IDLE gave, allowing I/O to resume to a device idled by the SWCTL/IDLE command. The value of m doesn't matter.

PATHCHANGE
The command
SWCTL/PATHCHANGE SWAn: m
will idle the device controlled by the SWAn: pseudodevice, then issue a command to change its path (and retry up to 60 times with a 1 second wait between each try), will issue an IO$_PACKACK function to the new path if it succeeds, and then will unidle the device. By performing these operations in one invocation, the image permits use with less chance of causing hangs. This functionality should be used instead of the separate functionality if at all possible.

All of these functions can be used regardless of what the device path names are, and second and later paths connected will be hidden when SWCTL/ADD is used to connect them to a path tuple. Thus they are useful in testing with older hardware, or (**with *caution***) for use in providing failover in systems with such hardware. The chief caution needed is that SWCTL knows nothing about what paths really are connected to what, and thus is capable of connecting any sets of entirely different devices together as though they were paths to the same device. This will of course cause data corruption and loss if done, but so long as paths connected are **really** the paths to the exact same device, SWCTL can allow failover to be set up between them. This is slightly simpler where the /SECOND qualifier is used, since devices so related are already identically named and there is reason to believe they really are the same.

Switching Subsystem Infrastructure
IOGEN and Driver Naming Functions

The main hardware client for the switching subsystem initially is HSZ70 and above controllers, which will have firmware able to accept a device allocation class from the HSZ console and present it in byte 102 of INQUIRY data to the host processors.

By adding code into the device configuration logic, this INQUIRY data is read and used to set a device allocation class for devices connected to these HSZ controllers. The modules IOGEN-CONNECT.BLI, IOGEN-SCSI-CONFIG.BLI, and LOAD_DRIVER.BLI are altered to perform this logic in autoconfigure, and the routine SYSMANIOGEN.C is also altered to handle similar logic for the SYSMAN IO CONNECT command. In addition, small changes in DKDRIVER.MAR are needed to ensure the correct port driver is connected in all cases, and MSCP.MAR is slightly altered to ensure it does not permit a served connection hooked up via the switching driver to be served somewhere else. Finally, some new routines are defined (either added to IOSUBNPAG or added locally) to perform duplicate device searches and connect newly found duplicate devices to SW units.

The major autoconfig changes are in IOGEN-SCSI-CONFIG.BLI. This module does I/O to SCSI port drivers to decide what (if any) class drivers to load. Since the new device allocation class must be supplied to LOAD_DRIVER in order that it set up a DDB correctly, IOGEN-SCSI-CONFIG keeps a record of each device allocation class found in the INQUIRY message where the device is an HSZ, and when one of these allocation classes is found that is nonzero, the corresponding LOAD_DRIVER call has its allocation class item list filled in with the device allocation class (currently a byte) biased by adding 40000000 hex to it, which serves to tell LOAD_DRIVER that this is not a port allocation class. This device allocation class in fact will override any PACs in effect. Within LOAD_DRIVER, this bias is removed. Also, IOGEN-CONNECT folds the port letter to A in this case. Since this is happening in cases where there is NOT a Port Allocation Class (PAC), we initialize the ddb$l_port_id cell to zero always, and fill it in with the original port ID any time we fold the letter. A corresponding change in DKdriver then looks at this cell when doing its connect, and uses the ddb$l_port_id field to find its port driver if that field is nonzero. This breaks some of the old dependence on port and class letters being the same. MKdriver could, by the way, readily be modified the same way. Once the new device is loaded, a call is made to the IOC_STD$TEST_SWITCH routine to connect it to a switching tuple if it is a second or later path to some existing device.

The changes to support SYSMAN IO CONNECT are somewhat more extensive since in the old code there was no INQUIRY attempted. The code has been altered in sysman$io_connect to look at connects for drivers named %K* (second letter a K). If such a driver is being connected, a channel to the corresponding PK driver is assigned and an INQUIRY sent to the target device. If one is found and is an HSZ5x, 7x, or 8x, the device allocation class (if present) is set up (taking it from byte 102 of the INQUIRY data) to send to the load-driver call. Also an entry is added to the load_driver itemlist to retrieve the UCB of the loaded driver, so the switching code can connect it. Once the unit is connected, a call to the IOC_STD$TEST_SWITCH routine is made to connect in the switching logic if this is a second path to a device. The same underlying code is called for naming as is used in autoconfigure, so that naming will wind up the same. This can mean someone gives a command to connect DKB300: and finds a new 54DKA300: device, for example, and users will have to be alerted that this can happen. Since manual connection is somewhat uncommon, and since there isn't any simple alternative to allow identification of which port driver is to be used, this added complexity will need to be accepted. *(With fibre channel, it seems likely that a /PORT=abc qualifier would be needed to specify the (possibly multi letter) port name in manual connects.*

The IOC_STD$TEST_SWITCH function is the most important external routine used to connect a new device. It takes one argument, the UCB of the newly loaded device, and searches for any other devices of the same name on the system. If it finds any such, and if the first device is not already connected to a SWdriver unit, a new SWdriver unit is cloned and the first UCB hooked in as the master path. The supplied UCB is then added as a secondary path. If no other paths are found of the same name, however, no switching code is hooked in. Thus, SW units are created only as needed, and single path devices are not touched by switching code. The IOC_STD$TEST_SWITCH function calls the function IOC_STD$SETUP_SWITCH, which takes the two UCBs and actually arranges for them to be connected via the switching driver. This lower level routine may be called where one is in posession of the primary UCB for a path and a new UCB which also serves the same device, though it is expected that IOC_STD$TEST_SWITCH will more commonly be used.

Switching Subsystem Infrastructure
Switching Server Functions

The switching server is completely automatic in its operation. It has two functions: sending keep alive test messages to local SCSI disk paths, and switching idle paths to the "other" controller of a pair of HSZ controllers when notified of controller failure. The controller failure is signalled by an HSZ as a SCSI Unit Attention condition with Digital-specific error codes. This class of code is generated by an HSZ only when an otherwise successful transfer has occurred but the failure of the other controller has been noted. Thus, DKdriver has some special purpose code which will check for these codes and generate an alternate success I/O status with the 16384 bit set (which it otherwise does not use). When SWdriver notices an I/O completion with this status, it sends a message to the server, clears the extra bit(s), and treats the I/O otherwise as successful.

The server, when it starts, sends INQUIRY commands to SCSI devices it finds and records pairings it can find. When it receives a message from SWdriver indicating controller failure, it sends commands to SWdriver to switch all idle devices set to use the failed controller to the still-good side. Where these paths are not idle, the server takes no action (since mount verify will accomplish the switching in those cases). This avoids the extra time which might be needed for mount verification where a path was idle. In addition, the server generates a "keepalive" mailbix message to itself periodically which is used to trigger sending keepalive messages to its currently active paths.

These keepalive messages are implemented as SCSI INQUIRY commands, to avoid unwanted side effects which might be triggered by actual data motion commands. The INQUIRY command is sent via the IO$_DIAGNOSE interface and is sent every 400 seconds. This will provide some ability to detect failed hardware without overly burdening the I/O system with traffic.

Note that the switching system will run with the server not even present, and the presumption is that the server is started sometime after autoconfigure has run. The server provides a spot where one can add system or cluster global information and coordination, but for the initial version of switching, control can be handled by the other components of switching whether the server is present or not.

Some Notes for the Future:

It is worth noting here that a number of VMS components work in the I/O stream in ways which have not in the past taken I/O layering of this kind into account. Most notable is the caching subsystem, which contains a good deal of special purpose code. While examining the paths used for switching indicates that for disks and similar devices the cache software is not an issue, future work in cache areas will need to be aware of the possibilities of use of VMS I/O layering, and may be well advised to use it for cache operation as well.

The principles of switching as used here can be extended to modest numbers of I/O connections with little difficulty. Should connections to very large numbers of paths become possible, they will however reach some intrinsic limits. If this happens, I believe the simplest approach is to have each node keep track only of some of the paths possible.

Switching paths dynamically, where for example each IRP might be sent round robin to each enabled path, is also possible with the kind of switching code here provided. Very little of SWDRIVER would need to change to handle this as a routine I/O method. It might be sufficient to have the entire tuple of devices go into mount verify (as presently done) when any I/O fails, considering that currently DKdriver lets all I/O drain when mount verify starts. In principle one need only drain I/O on all paths, putting all the active ones into their MV state and taking them out again after switch. This scheme would have similar switching behavior to the current one, but might be able to achieve higher throughput for non error cases by spreading out load over many busses. There would have to be some book-keeping added to SWdriver to ensure all "active" devices had their begin-MV calls run and their end-MV calls run, and to distribute I/O across a set of paths, but the philosophy of the thing and much of its action would not be affected.

The behavior of the current switching depends strongly on that of the HSZ, which can take minutes to do a switch in its current incarnation. A topic for investigation is whether, when a path failure is detected, it makes any sense to send an IO$_AVAILABLE down the failed path prior to sending the IO$_PACKACK goes to the new one. Some alternative method to tell an HSZ that it is being commanded to fail over which is perhaps less ambiguous than the SCSI START command used with IO$_PACKACK could be useful here. SWdriver could readily send a function modifier, for example, with a MV packack and DKdriver could be made to understand to send such a function to an HSZ should the hardware be made able to use it. For the moment, the switching system does what it can, constrained by underlying hardware as well as drivers. It is the slowness of HSZ failover which drives the use of a several minutes long hysteresis time, though, in switching. It could be made faster if the hardware allowed.

Revision History
Revision X0.1, June 24, 1997 Everhart - Initial draft 24-Jun-1997