Mac OS 8:
SCSI Hard Disk Block Storage Plug-In

Design Document

I/O Device Drivers

Change History

Modified draft to reflect connection-based SCSI and other improvements since original was written.

1. Vision for Block Storage Family Plug-ins

As the prototype Block Storage Plug-In, the SCSI Hard Disk Plug-In sets the standard by which all other plug-ins for this family should be designed. The essential goals included modularity, simplicity and reduction of dependence on existing Apple internals. By maintaining compactness in design and implementation it was hoped that future plug-ins, many designed by outside vendors, would easily integrate into the Mac OS 8 environment.

As the SCSI Family has responsibility for hardware-dependent functionality, the plug-in was seen as a bridge between the generalities of the Block Storage Family and the specifics of SCSI. By keeping the number of data structures and procedures to the minimum required for functionality, and generally keeping things as simple as reasonable, this software achieves much of the design goals.

2. Functional & External Interface Description

This plug-in provides functional access to a given SCSI store. As such, it interprets requests from Block Storage, allocates and manages memory for data structures, creates SCSI CDBs, catalogs exceptions encountered both internal to the routines and those reported by SCSI, and reports specific status for each transaction with the device including, as needed, detailed success and/or failure data structures at the granularity of each memory parcel in a memory list or I/O list data structure. Additionally, SCSI-specific information not included in the Name Registry is obtained at initialization time for the plug-in's current, and future, use.
Interface with the plug-in is provided through the Block Storage Family. As a privileged, family-specific plug-in, no other access to the plug-in is possible.

Assuming initialization was successful, the SCSI plug-in assumes the device pointed to by the store existed on the SCSI bus at the time of the device's entry into the Name Registry and that said device is a valid SCSI device capable of random-access block storage consistent with non-removable media (fixed) disk drives. Since the plug-in initializes itself and manages the device for the operating system, no other assumptions are necessary for its performance.

Block Storage identifies the accessible functional elements of the plug-in via the data structure BSStoreMappingOps{} whose definition can be found in header file BlockStoragePPI.h. The elements in that structure reference routines Block Storage may use to access the store, including examination, initialization, command execution, mapping and removal algorithms. The specific interfaces are given below:

static OSStatus BSSCSIDiskPIExamine(BSStorePtr examineStore, UInt32 *certainty) determines the actual existence of a store this plug-in should concern itself with. If a valid device is identified, the value of certainty is set to kBSMPIDeviceTypeRecognized. This filter serves to eliminate non-fixed disk SCSI devices from the plug-in's scope of operation.

OSStatus BSSCSIDiskPIInit(BSStorePtr initStore) initializes the plug-in's private data storage for this store and identifies certain characteristics of the device and the bus it is on. This procedure performs a SCSI Bus Inquiry to obtain the size of the SCSI Parameter Block as well as other bus-specific information we may later wish to use. The device's properties are noted and stored in the store private data structure. SCSI Read Capacity is called to determine the functional limits for the device. After this routine completes, the SCSI device is fully accessible by the operating system.

As other disks are tested, especially older devices, it may be useful to identify "quirky" ones for specific behavior on the part of the plug-in. Currently, SCSI DMAs all data to and from the drive. Certain drives are known to misbehave when data is polled, pseudo-DMA style, and the initialization procedure can then be used to toggle the plug-in to inform SCSI of the need to behave in a non-standard manner when accessing these drives. Note: These drives shall be serviced by unique plug-ins derived from the standard SCSI plug-in.

OSStatus BSSCSIDiskPICleanup(BSStorePtr theStore) returns memory allocated in the initialization routine to the resident pool. The OS is then in the same condition it was in before the plug-in was initialized. This is only called when the device is no longer in use by the OS, i.e., unplugged or unneeded at system shutdown. The caller provides a pointer to the store.

BSIOStatus BSSCSIDiskPIIO(BSStorePtr theStore, BSBlockListDescriptorRef blocks, MemListDescriptorRef memory, BSIORequestBlockPtr ioReq, OptionBits options, BSErrorList **errors) is the run-time call to request the plug-in perform I/O to or from the store. The caller provides pointers to the store, the block list descriptor, the memory list (list of resident locations in memory to DMA or poll the data into), the request block pointer (unused by the plug-in) and the option bitmap. If an error occurs in processing (after successful memory allocation), the routine returns a pointer to a linked list of memory block identifiers specifying which portions of the request succeeded, if any, and which failed with the specific failure identifier. It is currently the responsibility of the caller to dispose of the memory pointed to by the ioErrorList pointer. (It is considered bad form for the caller to dispose of the memory allocated by the plug-in. It is assumed that a mechanism to dispose of the memory within the plug-in will someday be devised.) Errors identified in this routine are failures prior to execution by SCSI.

OSStatus BSSCSIIOCompletion(BSStorePtr theStore, void *finishedPrivateData, BSErrorListPtr returnedBSErrorList, OSStatus returnedStatus, BSErrorListPtr *errorListPtrPtr) handles each individual part of a transfer’s completion as toggled by SCSI. Non-memory fragmented transfers, for example, generate one CDB and result in one call to this routine. The caller provides pointers to the store, the private data, the returned error list (other errors detected by other plug-ins “downstream” of this one), the status returned by failed plug-ins “downstream” and a pointer to a pointer for returning this routine’s error list data structure, as described in BSSCSIDiskPIIO(), above.

OSStatus BSSCSIDiskPIGoToState(BSStorePtr theStore, BSAccessibilityState gotoState) currently sets the plugin into the online and offline states to allow or disallow access without removal of the plug-in.

OSStatus BSSCSIDiskPIGetInfo(BSStorePtr infoStore, BSStoreMPIInfo * info) creates the ascii device name and stores its functional properties in the mapping plug-in information data structure.

OSStatus BSSCSIDiskPIFormatMedia(BSStorePtr formatStore, BSFormatIndex formatType), the format command, and OSStatus BSSCSIDiskPIAddComponent(BSStorePtr destStore, BSStoreMPIComponent * newComponent, BSStoreInfo * storeNewInfo), are currently unused and are null routines.

3. Module Breakdown

Store examination is performed by BSSCSIDiskPIExamine(). Its sole purpose is to determine if the family expert has any devices pointed to in the Device Registry whose attributes are such that the SCSI hard disk plug-in should service said device. This is accomplished by validating component properties (such as, not more than one component attached to the store, is this an external device, etc. A comparison of the name entered in the device registry for the store (currently "DASD") completes the examination process. If all of the tests are successful, a certainty of kBSMPIDeviceTypeRecognized is stored in the address referred to as certainty. No attempt is made to communicate with, or otherwise validate the functionality of the device by this routine. The existence of a device the plug-in should potentially service results in a successful status value returned by this routine.

Initialization of the store using the plug-in is accomplished through BSSCSIDiskPIInit(). In order to avoid hanging in a call to the SIM, all communication through the SCSI Family in this routine is performed as synchronous calls. Failure to adhere to this convention could result in memory leaks as well as inability to access the device at all during the life of the current instantiation of the OS.

Private data structures used throughout the plug-in are created and initialized. The plug-in private data structure is created and associated with the store. Pools are created to allow the plug-in to manage memory without repeated calls to PoolAllocateResident(). The pool management routines included in the plug-in are able to return pointers to data structures with less overhead than the standard pool allocation routines, in part because not all elements of each data structure are necessary to clear. Each data structure used by the plug-in has a corresponding pool attached to the plug-in private data structure.

The basic device information necessary to make a simple call to SCSI is stored, and SCSIOpenConnection() is called, requesting read / write access to the store. If this call fails, the init routine aborts and the device is inaccessible for lack of a connection to it through the SCSI Family.

BSPIBusInquiry() is called to create a bus inquiry CDB and call SCSI synchronously to perform a bus inquiry command. The information received is stored in the private data structure for future use. Failure to successfully complete the bus inquiry causes the init routine to abort due to inability to analyze the nature of the bus the device is attached to.

Once the bus is validated, the pools are finally created via a call to BSPIAllocatePools(). Failure to create pools results in an abortion of the initialization routine, and the device is inaccessible for lack of resident memory.

BSSCSIReadCapacity() is called to create a read capacity CDB and call SCSI synchronously to perform a read capacity command. This is necessary as the partition creation software has no knowledge of the extent of the media on the device. All I/O requests are checked for overrun by the plug-in. Failure to read the capacity of the device aborts the init routine due to inability to determine its extent.

All aborts of this routine free memory allocated within it and yield the device inaccessible for the life of the current instantiation of the operating system.

The clean-up routine is extremely simple, de-allocating memory previously allocated from the resident pool in the initialization routine and returning status for that action.

Command execution code receives control and direction from Block Storage through a direct call. All memory blocks associated with the transfer are already locked into memory via a call to PrepareMemoryForIO() performed by Block Storage. BSSCSIDiskPIIO() accepts pointers to the memory and description of the blocks to be transferred, acquires pooled data structures required for the transfer and any errors which may or may not occur, and proceeds to determine the extent of the transfer, breaking it up into separate requests based on the pattern of contiguous memory blocks contained within the request block data structure. While not the most efficient means of managing transfers possible, it is necessary at this junction as the plug-in resides in the page-fault path (obligating Block Storage to prepare the memory versus the SCSI Family doing so).

The entire transfer is represented by the BSIOTotalXferStatus data structure. Each transfer is broken up according to the pattern of contiguous memory blocks into individual requests to SCSI, as represented by the BSHDOneXferInfo data structure. These are attached to the BSIOTotalXferStatus data structure, which in turn are attached to the private data structure. There may be several outstanding BSIOTotalXferStatus data structures (in a linked list) each containing at least one BSHDOneXferInfo data structure. This relationship is created in this routine.

After acquiring a pointer to the private data structure (pD) through a call to BSGetMappingPIPrivateData(), the BSIOTotalXferStatus data structure is acquired from its pool via a call to BSPIPoolGet(), a generic pool pointer allocation routine within the plug-in. If there are existing BSIOTotalXferStatus data structures outstanding, the linked list of the same is traversed, and the current pointer is attached to its end.

Next, the block list descriptor is examined, and a memory list data structure is created for each contiguous memory block represented within the transfer request. See BSBlockListDescriptorGetExtent() and MemListCreateSimpleDescriptor() in the Block Storage documentation for an understanding of how this is accomplished.

BSIOInitOneXferInfo() is called to initialize the data structure used to manage each transfer as it is represented by the fragmentation of memory. In this routine the specific memory list or I/O preparation table is attached to the BSHDOneXferInfo data structure, later to be sent to the SCSI Family. The preferred memory descriptor is the MemListDescriptorRef. It shall be used by this, and all future plug-ins. It greatly simplifies the management of memory for software "downstream" of the plug-in.

Providing that the BSHDOneXferInfo data structure has been successfully initialized, the transfer is next checked for an overrun condition. If none exists, BSSCSIMgr() is called to perform the SCSI-specific operations necessary to this I/O request. In this routine the BSSCSIXferData data structure is initialized with all information necessary to perform connection-based SCSI calls. The actual parameters sent in the calling sequence for a given command are contained in this data structure. BSSCSIReadWriteCDB() is called to create the SCSI CDB, and BSIOEnqueue() is then called to initiate the transfer, or queue it for later execution.

Assuming the transfer does not require queuing, BSSCSIDoIOAsync() first calls BSTrackOtherFamilyRequest() to enter this transfer into a kernel notification queue maintained by block storage. It is this mechanism which allows BlockStorage to call the plug-in back when I/O completes asynchronously. SCSIExecIOAsyncCmd()

asynchronously executes the requested command. Refer to the SCSI Family documentation for a description of this procedure.

When processing of the request is complete, Block Storage is awakened and calls BSSCSIDiskPIIOCompletion(). If no errors occurred, memory allocated for data structures is freed and good status is returned. If a failure did occur, each individual memory block request is tagged with its corresponding status, and an appropriate failure status is set. The error block request list (essentially a linked list of the memory blocks specified in the original request from Block Storage) is attached to the passed pointer to a pointer to an error list, and the bad status is returned. Block Storage then is able to specify which portions of memory contain successfully transferred data and which do not. Block Storage is additionally charged with the responsibility for de-allocating the memory allocated for the error list.

The key for being able to accomplish this is the (void *)finishedPrivateData pointer which returns, in this instance, the BSHDOneXferInfo data structure representing this I/O. It was identified to block storage in the call to BSTrackOtherFamilyRequest(). A pointer within this data structure (dubbed motherShip) holds the address of the corresponding BSIOTotalXferStatus data structure this separate I/O is part of. The completion routine determines whether or not all parts of a complete transfer request are finished, and logs their success or failure. If the entire transfer is complete, and no errors have been noted, kBSIOCompleted is returned to block storage denoting the entire transfer has finished. If no errors occurred, BSErrorListPtr pointer is NULL.

While the forgoing may sound complex, in fact the implementation has been simplified so as to allow vendors to create plug-ins with a minimum of effort. This should result in more rapid time from development to deliverables, a goal of the Mac OS 8 overall design.

Note: Flow illustration forthcoming in this space...

4. Internal Interfaces

As Block Storage is the only client, and as the plug-in only serves SCSI Family, this section is intended for this already well-informed audience and is therefore necessarily brief.

Examination

Plug-in device examination is accomplished through invocation of the following call:

OSStatus BSSCSIDiskPIExamine(BSStorePtr examineStore, UInt32 *certainty);

where examineStore is the identifier for the device store, and certainty is a returned value defining the level of certainty that this store should indeed be serviced by this plug-in. Non-zero OSStatus indicates a failure of one or more functions during examination.

Initialization

Plug-in initialization is accomplished through invocation of the following call as a task:

OSStatus BSSCSIDiskPIInit(BSStorePtr initStore);

where initStore points to a data structure describing the device store. Non-zero OSStatus indicates a failure of one or more functions during initialization. The plug-in aborts, and its memory is freed. The device cannot be accessed by this plug-in in case of any failure detected within this routine. Successful completion means a connection has been established to the device and it is ready for I/O.

Cleanup

Plug-in removal is accomplished through invocation of the following call:

OSStatus BSSCSIDiskPICleanup(BSStorePtr cleanupStore);

where cleanupStore is a pointer to a data structure representing the device store. Non-zero OSStatus indicates one or more data structures were not released, a fatal OS error.

I/O Execution

Invoking the Plug-In

Plug-in execution is initiated by the following call:

static OSStatus BSSCSIDiskPIIO(BSStorePtr theStore, BSBlockListDescriptorRef blocks, MemListDescriptorRef memory, BSIORequestBlockPtr ioReq, OptionBits options, BSErrorList **errors);

where theStore is a pointer to a data structure representing the device store, blocks is a pointer to a BSBlockListDescriptorRef used to determine the extent of the transfer into memory, memory is a pointer to a MemListDescriptorRef data structure describing the list of locations in memory where data is transferred to or from, ioReq is a pointer to a Block Storage I/O Request Block data structure containing a block-by-block map of the prepared memory blocks and the starting disk block and extent of the transfer, ioErrorList is a pointer to a pointer used by the plug-in to attach error results prior to invoking SCSI, and options is a bitmap currently differentiating between reads and writes.

Non-zero OSStatus indicates any of a multitude of potential failures. Refer to the OS error enumerations for these. If OSStatus is non-zero, errors should be checked for a non-NULL pointer. If so, it contains a linked list of memory blocks requested and the final status of each.

If no errors are detected in the request, or the allocation of memory for data structures necessary to process the request, a CDB is created for each divisible portion of the request. This division is based on discontiguous memory blocks as mapped by Block Storage in a call to PrepareMemoryForIO(). In sequential order, as defined by the list of memory blocks, each portion of the request is sent to SCSI asynchronously. First, a call is made to set up a wakeup call to Block Storage as shown:

OSStatus BSTrackOtherFamilyRequest(BSStorePtr theStore, BSIORequestBlockPtr ioReq, void * privateData, KernelNotificationPtr retNotify);

where theStore and ioReq are as passed into the plug-in, privateData is a pointer to a data structure used to identify this request (BSHDOneXferInfo), and retNotify is the address of a kernel notification structure to be filled in by BSTrackOtherFamilyRequest() and subsequently used by SCSIExecIOAsyncCmd() to establish the kernel queue for waking up Block Storage when the request has been completed by SCSI. OSStatus is non-zero only if the request cannot be tracked. Finally, SCSI is called asynchronously, as shown:

OSStatus SCSIExecIOAsyncCmd(ConnectionID connID, KernelNotification *kernelNot,SCSIDataObject dataObject, SCSICDBObject cdbObject, SCSIFlagsObject flagsObject, SCSIExecIOResult *resultBuffer, SCSIExecIOTag *ioTag);

is the connection-based SCSI asynchronous command for execution of a single I/O request to the device. All objects in this calling sequence are stored in the BSSCSIXferData data structure. The command returns the ioTag pointer to use as an identifier to track this request. Non-zero OSStatus indicates a failure in queuing this request by the SCSI Family or SIM.

SCSI Completion of Requests

When Block Storage is notified of an event initiated by the SCSI plug-in it respond by calling:

BSMappingIOCompletion BSSCSIDiskPIIOCompletion(BSStorePtr theStore, void *finishedPrivateData, BSErrorListPtr returnedBSErrorList, OSStatus returnedStatus, BSErrorListPtr *errorListPtrPtr);

where theStore is the same as was passed into the plug-in when execution began (see above), finishedPrivateData is the unique data structure identifying this request originally passed into SCSIExecIOAsyncCmd() (in this case, BSHDOneXferInfo), returnedBSErrorList, not currently used by the plug-in, references an error list spawned by a previous plug-in, returnedStatus is the final status of some other plug-in, and errorListPtrPtr is an address where we can attach an error list for the errant transfer.

```
	DriverDescription TheDriverDescription =
	{
		kDriverDescriptionSignature,
		kMac OS 8DriverDescriptor,
		{
			"\pTarget",
			{1,0,0,0},
		},
		{
			kDriverIsUnderExpertControl | kDriverIsLoadedAtBoot,
			"\pTarget",
			{0,0,0,0,0,0,0,0},
		},
		 {
			1,
			{
				kServiceCategoryBlockStorage,
				0,
				{0,0,0,0}
			}
		}
	};

void		BSPIDeallocatePools( BSSCSIPrivateDataPtr pD );
OSStatus	BSPIMakeMorePools( BSSCSIPrivateDataPtr pD );
void		*BSPIPoolGet( BSStorePtr theStore, int poolType );
OSStatus	BSPIPoolPut( BSStorePtr theStore, int poolType, void *aPtr );
OSStatus	BSPIAllocatePools( BSSCSIPrivateDataPtr pD );
OSStatus	BSSCSIDoIO( BSStorePtr theStore, BSHDOneXferInfoPtr oneXI);
BSHDOneXferInfoPtr	BSIOInitOneXferInfo( BSStorePtr theStore, UInt32 len, MemListDescriptorRef
			memory, UInt32 dataType, OptionBits options, int curXfer);
OSStatus	ScsiGetIOPBSize( UInt8 busID, UInt16 *byteCount, BSSCSIPrivateDataPtr
		privateData );
OSStatus	BSSCSIMgr( BSStorePtr theStore, BSHDOneXferInfoPtr oneXI, BSErrorListPtr
		errData, BSIORequestBlockPtr ioReq );
void		BSSCSITestUnitReadyCDB( BSSCSIXferDataPtr sXferData );
void		BSSCSIInquiryCDB( BSSCSIXferDataPtr sXferData );
void		BSSCSIReadWriteCDB( BSSCSIXferDataPtr sXferData );
```

```
void            BSSCSIReadCapacityCDB( BSSCSIXferDataPtr sXferData );
void            BSPICleanupDataStructures( BSStorePtr theStore, BSIOTotalXferStatusPtr tXStatus,
                BSHDOneXferInfoPtr oneXI );
OSStatus        BSSCSIReadCapacity( BSStorePtr theStore );
OSStatus        BSSCSIDiskPIExamine( BSStorePtr examineStore, BSMPIConfidenceLevel *
                confidence );
OSStatus        BSSCSIDiskPIInit( BSStorePtr initStore);
OSStatus        BSSCSIDiskPICleanup( BSStorePtr cleanupStore );
BSIOStatus      BSSCSIDiskPIIO(BSStorePtr ioStore, BSBlockListDescriptorRef blocks,
                MemListDescriptorRef memory, BSIORequestBlockPtr parentRequest, OptionBits
                options, BSErrorList ** errors);
OSStatus        BSSCSIDiskPIAddComponent(BSStorePtr destStore, BSStoreMPIComponent *
                newComponent, BSStoreInfo * storeNewInfo);
OSStatus        BSSCSIDiskPIGoToState(BSStorePtr theStore, BSAccessibilityState gotoState);
OSStatus        BSSCSIDiskPIFormatMedia(BSStorePtr formatStore, BSFormatIndex formatType);
OSStatus        BSSCSIDiskPIGetInfo(BSStorePtr infoStore, BSStoreMPIInfo * info);
OSStatus        BSSCSIDiskPIIOCompletion( BSStorePtr theStore, void *finishedPrivateData,
                BSErrorListPtr returnedBSErrorList, OSStatus returnedStatus, BSErrorListPtr
                *errorListPtrPtr );
```

6. Key Algorithms

In the humble opinion of the author, all algorithms not referenced in other sections of this document are eminently readable and require no further dissection here. Particular attention was paid to memory management and synchronization to ensure the code could survive with multiple instances of itself present in the OS. As the plug-in is in the page fault path, this is the most critical piece of code created for this module (refer to #7, below).

7. Critical Sections & Synchronization

As the actual memory requirements for the plug-in are difficult to assess at initialization time, and calls to PoolAllocateResident() are costly when made in the run-time execution path to the store, a unique memory management facility has been designed for plug-ins to address both issues. At initialization time a “chunk” of memory is allocated from the resident pool for the plug-in’s use on a per-data structure basis. This memory chunk is sufficient to store the default data structures for the store, with additional space for more data structures required for a TBD number of transfer requests from Block Storage. Management of the chunk is accomplished through calls to a plug-in function capable of determining whether sufficient space remains in the chunk, requesting additional memory from the resident pool as needed. The net result of this allocation method is a significant reduction in the number of requests of PoolAllocateResident(), thus speeding the throughput of the plug-in. De-allocation is likewise performed as a function call. Data integrity is assured through atomic synchronization mechanisms. Current design avoids calling PoolAllocateResident() for any further storage due to the risk of deadlock as the plug-in executes within the page fault path.

The memory manager implements the above by use of a linked list of data structure-sized memory blocks for the three key data structures and their associated memory. Allocation is always from the head of the linked list. De-allocation is always to the head of the linked list. Ergo, it matters not whether the list is accessed many times or only once, allocation and de-allocation times remain relatively static. As each instance of the code will act on pointers to the data structures from these pools, the only synchronization mechanism required for the plug-in is contained within the routines BSPIPoolGet() and BSPIPoolPut(). All other data is referenced off of the local stack for a given instance of the plug-in.

Synchronization is accomplished through a simple algorithm using CompareAndSwap(). In BSPIPoolGet(), the code obtains the address of the first data structure memory in the specified linked list. CompareAndSwap() is then called to replace the first address with the address pointed to by the poolNext pointer contained in the obtained data structure. If the first address has changed, the process is repeated (essentially, if the first address is in use, get the first address again). Likewise, in the BSPIPoolPut() routine, the pointer to the next element in the data structure to be returned to the pool is updated with the address of the first element in the pool. If that element has changed, the process is repeated. Clearly, in the vast majority of cases the code executes in a linear fashion. Occasionally (perhaps 5% of the time, in the estimation of the creators of the CompareAnd Swap() code) a second iteration is required. This code is efficient, executing roughly twice as fast as PoolAllocateResident(), and slightly faster than PoolDeallocate().

Multiple instances of the code should execute successfully with this design. In a multiprocessor environment, similar read-modify-write instructions should protect the code from parallel execution errors.

Note that all critical code sections are protected by an access lock.

8. Correctness Testing Methodology

The author tested the software using the debugger on a test platform of an 8100/80AV running TCL under Mac OS 8. BSClient() code created buffers of various sizes, filled with known data, which were written to the disk. The buffers were then zeroed out and the data read back from the disk. No errors were discovered. Additionally, the drive partitioning information was read correctly from the disk. Roughly a terabyte of data was read/written in this manner. Subsequently, the code ran successfully to the finder in Mac OS 8.

9. Correctness Testing Mechanisms

10. Fault Handling Methodology & Mechanisms

The ideal multitasking operating system responds in some manner to all errors, regardless of their severity. Fatal errors, ones which compromise the integrity of the kernel or its associated data structures, should precipitate an immediate and orderly shutdown and restart of the operating system. Errors of moderate severity, ones involving the failure of a device to correctly perform (especially where memory is, or is expected to be, modified, or where the device itself is incorrectly updated), should ideally be corrected by the operating system itself, with notification to the application that additional action was necessary to perform a given request, returning status indicating success or failure of the request. Minor errors, such as application-initiated mishandling of data structures, should be serviced by the application itself. This is the model Mac OS 8, and the SCSI plug-in, aspire to. Note that fault tolerance is a superset of this model, and requires additional hardware not envisioned in the Mac OS 8 world. However, if the operating system can perform adequately with respect to fatal errors, a fault tolerant layer can be created between the application and the kernel for future, unforeseen implementations of Mac OS 8 or its progeny on appropriate hardware platforms.

There are two primary types of errors possible in the execution of plug-in code and corresponding remedies for each. They are listed in order of precedence, although, in fact, any error detected by the plug-in is reported, and all plug-in errors are fatal to all, or a portion of, the SCSI request whose servicing yielded the error state.

Memory Allocation Errors

These are identified by the inability of memory pool allocation code to return a pointer to a memory block of the size requested by the plug-in. In a multitasking environment such errors are fatal, signifying corruption in the kernel. The plug-in may encounter such a situation when allocating memory for a data structure, or a data structure within a data structure. In either event the plug-in de-allocates any memory previously allocated in the current instance of the plug-in, returning appropriate failure status to Block Storage.

SCSI Errors

When SCSI errors occur, the error is flagged, the failed memory block is identified with an appropriate error code, and execution continues until all blocks requested have either erred or succeeded. The entire transfer is flagged as unsuccessful and the plug-in returns a linked list of request blocks each with its own individual status. This allows higher-level code to identify particular pieces of memory whose contents may not have been completely transferred to or from the device.

Returned Errors

As control is passed from the plug-in into other software components, additional errors may be detected. The scope of these may include, but is not limited to, memory allocation errors, software integrity errors, et cetera. Such errors are reported by the plug-in, provided they are not superseded by other errors listed above. As all errors identified as Returned Errors are passed to the plug-in by Block Storage, it is assumed that Block Storage has already noted the condition. This being the case, if a transfer succeeded, but there was a Returned Error, the final status is set to failure and the Returned Error is noted for the requested memory block, and the entire block request is returned to Block Storage. If the transfer failed and there was a Returned Error, the Returned Error is overwritten with the I/O failure status and the entire transfer is flagged as failed. Thus, if Block Storage detects a Returned Error subsequently passed to the I/O completion routine of the plug-in, but no requested memory block is flagged with the identical Returned Error, Block Storage could differentiate that a SCSI error also occurred.

Error testing should include generation of SCSI errors, and simulation of memory request failures. The latter is more difficult than the former, as fault injection here implies code modification to "lie" about the state of the memory and thus embodies the risk of injecting bugs into the code under test. The ideal test bed would include software external to the plug-in which would continuously request more and more resident memory until, in fact, no memory remained in the system. Said test software should first be tested for its robustness.

11. Performance Measurement Methodology

This module was designed to have a minimal RAM footprint (no larger than 4K) and minimal overhead to the path from Block Storage to SCSI. Its overhead should not increase as demand increases on the plug-in (with the notable exception of necessary overhead to obtain data from the memory pool if several instances of the plug-in simultaneously attempt to access the same data structures). Occasional, one-time increase in overhead for the purpose of requesting additional memory from the resident pool is acceptable.

12. Performance Measurement Mechanisms

13. Prototypes & Prior Experience (Experimental Justification)

This software design has been implemented for nearly a year with success.

14. Dependencies

This software was developed without requiring design modification of any existing code. During development, changes in the interface between the plug-in and the Block Storage Family were deemed prudent. Those changes are reflected in this document. As such, no further modifications are necessary or required.
It is necessary, however, that the SCSI Family Expert initialize the Name Registry prior to calling the plug-in initialization routine. The SCSI Family itself must be functional to the point of accepting a SCSI Bus Inquiry command synchronously at plug-in initialization time, and SCSI I/O commands thereafter. The Block Storage Family calls these routines, therefore its state must be sufficiently progressed to support these actions and process the results, including any errors which may result.

15. Compatibility

As the SCSI plug-in is unique to Mac OS 8, no compatibility comparison is possible or valid. Nevertheless, there is nothing in the plug-in design that should inhibit existing applications which access the SCSI Family directly from performing concurrently with plug-in requests. The only issue with respect to applications which access the plug-in through the Block Storage path would be how they expect error handling to occur, and this issue is (outside of retries) better dealt with by File Systems.

The design of this software highlighted modularity and common, simple C programming techniques. No hardware is directly referenced. There should be little difficulty (if any) porting this code to other platforms, or building this code with various C compilers.

16. Internationalization

17. Reviewers

18. Plan of Action

As the implementation of this design is functionally complete, additional work will primarily consist of enhancements to the initial design.

Appendices

- **Analysis of Results Document**
- **Code**
- **Instrumentation, Test & Verification Tools**

Analysis of Results Document

Presentation of the Results

Analysis of Correctness Testing

Analysis of Fault Handling Testing

Analysis of Performance Results

Code

Instrumentation, Test & Verification Tools