Generic Interface for Configurable Disk I/O Systems

[†]Rex di Bona, [*]Alan Dearle, [†]James Farrow,
[†]Frans Henskens, [†]Anders Lindström,
[†]John Rosenberg, [*]Francis Vaughan

[*]Department of Computer Science
University of Adelaide, S.A. 5005
Australia

{francis, al}@cs.adelaide.edu.au

[†]Department of Computer Science
University of Sydney, N.S.W. 2006
Australia

{rex, matty, frans, anders, johnr}@cs.su.oz.au

Abstract

Disk drivers are rapidly increasing in complexity and decreasing in understandability. We present a new design for the organisation of disk drivers based on a module paradigm. Disk drivers are constructed in modules that may be placed in the order that best fulfils an application's requirements. We show that a simple interface between the modules leads to a powerful mechanism for building disk drivers.

1. Introduction

The complexity of disk drivers in operating systems is rapidly increasing. Drivers are called on to provide more and more functionality. Drivers have to perform additional functions such as striping, spanning, and replication. This paper describes a departure from traditional disk drivers which permit this additional functionality to be provided in easily constructed building blocks. Each block individually performs a simple task. The power of this system is in the flexible designs that can be constructed from the simple blocks.

The traditional approach to disk device drivers is to have the driver linked into the kernel. If a change in functionality is required the driver has to be modified and the kernel relinked. User code is restricted to stub routines that call the kernel drivers. This has two problems: that of the code being overly large, and that the time spent in the kernel increases, reducing performance.

Recent systems have attempted to address these issues. The approach taken has been to implement a core driver in the kernel, and have the remaining functionality in user space. Even when most of the functionality is removed from the kernel and placed into user space we only solve one problem, that of time spent in the kernel. There is still a monolithic piece of code in user space that is difficult to construct and potentially slow to execute.

In our design we remove the need for a monolithic piece of code in user space. The design presented breaks this piece of code into multiple modules; each module performs one simple function. We connect these modules together, on a per application basis, to gain the functionality desired. An application developer can use modules from a predefined set provided, or may produce a custom module if the required functionality is not already available. The developer selects the modules required, and arranges them in the order that is desired.

Since an application only uses those modules required, the code that must be executed for each disk access is reduced. In a monolithic driver there is a requirement for a series of tests and loops to perform the operations requested, eventuating in only a small fraction of the kernel driver being executed. In the modular system these tests and loops are not necessary; only the code required is placed into the modules that are called. In keeping with this paradigm there are multiple kernel drivers, each driver for one type of hardware controller unit. The kernel drivers only support those operations that are required for the controllers.

The design of the structure arose out of the desire to produce a simple, generic, driver for the Grasshopper project. The Grasshopper project is designing a persistent operating system for experimental purposes [5]. As such, one of the project goals is to design a system that allows experimentation with a variety of different high level systems. The persistent nature of Grasshopper is not crucial in the design of the disk interface, and we feel the interface is equally suited to non-persistent operating systems.

In summary, we propose a system where the disk driver is spread between multiple user modules in user space and a set of core drivers in kernel space. Each kernel driver provides access to one hardware device. Each user module performs one simple function A user may set up a graph of modules in user space between the application and the kernel to provide the functionality required. This approach reduces both kernel and user level overheads and complexities when compared to other systems.

1.1 Background

There are two possible approaches to constructing a disk driver. Either all the functionality can be provided by a single module, or the code can be spread amongst many modules. Traditional systems are built using the conglomerated approach, while most recent systems use a layered approach. We extend the layered approach into a modular approach, where the modules may be connected in any order.

Traditionally, device drivers are integral parts of the kernel. Device functionality was dictated by the included drivers. A kernel rebuild was necessary whenever changes were made to any portion of, for example, a disk driver.

The layered approach, as utilised in [3, 6, 7, 10, 11], provides a fixed hierarchy of modules that are used to perform the task of controlling disk drives. For example, in Mach [6] there are chip dependent and chip independent layers. This approach improves the kernel response to interrupts. In SunOS the SCSI disk drivers have been replaced with the Sun Common SCSI Architecture (SCSA). The SCSA has three layers, the controller layer, the drive type layer, and the file system layer. Choices [7] uses the internal type system to build a hierarchy of disk types. The functionality of a disk is defined totally by the position of its code in the type system.

Each of these systems is characterised by having multiple layers with different interfaces between the layers. While it is possible to mix and match to a certain extent, the only choice is between objects in the one layer. What we shall describe is a system where the objects have a common interface, and may be placed in the best order for each application.

The idea of composing processing structures out of simple building blocks is not new. Using UNIX pipes, for example, it is possible to construct arbitrarily complex programs from a collection of tools by joining them in various ways. All programs have the same interface (that of a character stream). A program does not know which programs, if any, are upstream or downstream of itself. Data flows from one program into another anonymously. We feel that this anonymity is a desirable feature as it enforces clear, generic, interfaces.

1.2 Adding Functionality

This placement of complexity in user space is advantageous in that it is easier to write and debug a small kernel driver, and then construct the user level modules, than it is to write all portions of a single kernel driver at once. As an example we consider the advent of disk jukeboxes. These are devices that store multiple disk platters in a rack, and place one platter in a disk drive to be read or written. These devices are becoming more common as the need for large accessible storage systems becomes more prevalent. Jukebox systems are quite different from traditional disk drives. They require additional,

special commands, namely position disk and load/unload disk, to be sent before the normal read/write commands.

A jukebox cannot be attached to a system with an older style disk interface as the requirements of the jukebox do not align with the functionality of the traditional kernel based disk drivers. In the module system all that would be required is to construct a jukebox control module, and place this module as the first module above the kernel interface, and a jukebox could be accessed as though it was a single, extremely large, disk. The upper level modules do not require any changes. The jukebox module would also have to access a second kernel driver. The additional kernel driver would be a driver that allowed commands to be sent to the Medium-changer in the jukebox. This new kernel driver is almost identical to a standard SCSI disk driver [9].

1.3 Advantages of a Modular Approach

The modular approach has three major advantages. The first is that less time is spent in the kernel, which has been shown to improve the kernel response time [6, 4]. The second advantage is that it is possible to customise the interface between an executing application and the underlying disks to improve performance. Only those modules required need be attached. For example, in experiments on databases, where exact placement of data on disk is important the database engine can be placed directly above the kernel driver, allowing the highest possible bandwidth and exact placement of data, while a file server that does not need exact placement, but does require vast storage space could use a spanning module on top of many drives to access multiple drives in a independent fashion.

The third, and perhaps the most important advantage for an experimental system, is that having a more modular approach allows for easier experimentation. An experimental module can be constructed and tested without disturbing or modifying the existing modules. This allows additional modules to be developed and tested while the system is running, and does not require a reboot to make these modules available for general use.

The remainder of this paper is structured as follows. Section 2 gives an example of the sort of structures we can compose using the model. In Section 3 we describe the assumptions of the model we use. In Section 4 we outline the model itself.

2. Examples of Modular Designs

The model we shall describe is very powerful and flexible. The power of the model is in the ability to compose complex processing structures out of simple building blocks. In this section we present three examples that demonstrate this power and flexibility.

2.1 Partitioning a Multi-disk System

Consider a system that has multiple disks which may be of different types. These disks are to be presented to the application as a single partition. For conventional systems, the kernel drivers must know how to achieve a spanning partition. In the typical case, in which each partition is wholly contained on a single physical disk, this ability is a hindrance. The resultant kernel is larger than necessary because of the superfluous code, and driver execution is less efficient due to unnecessary tests.

Suppose we wish to allocate three disks of differing sizes evenly between four applications. Figure 1 shows a solution using the scheme detailed in this paper. A two module system would be used. The three disks each have a kernel interface, K1, K2, and K3 We use a spanning module (Sm) to provide the abstraction of a single disk. The capacity of this disk is the combined size of the three underlying disks. Above this module we would place the partitioning module (Pm). The module would be configured to provide four partitions, each partition being accessed by an application.

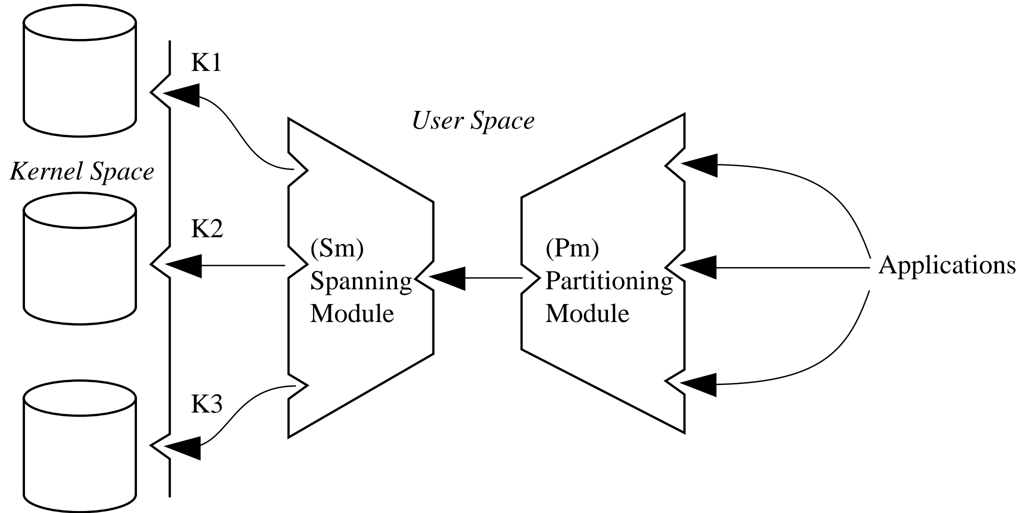

Figure 1: Partitioning Spanned Drives

The spanning of the physical devices is transparent to the applications. Significantly, the spanning is also transparent to the kernel device drivers. The functionality of the kernel drivers is identical irrespective of whether the driver is used for spanning or single disk partitions.

2.2 Security and Allocation on a Single Disk

One of the advantages of the modular system is the ability to adopt different policies on differing parts of a collection of disks. For example, suppose we need to store a sensitive data object on a portion of a disk, and have the remainder of the disk available for general allocation. If we were attempting to do this with a monolithic kernel disk driver the kernel driver would have to have separate policies for the different sections of the disk. In one section the data would be encrypted for security reasons, and in the other section the disk space has to be shared amongst several applications. This requirement of having different policies would complicate the kernel driver.

The kernel driver would be required to differentiate the requests for each portion of the disk. It must implement the different policies desired for each portion. This would require the kernel driver to be designed with this functionality as a goal. In general a kernel based driver would not be able to anticipate all the different policies requested by users.

To solve this situation using modular blocks we use three modules. As shown in Figure 2 attached to the kernel driver is the partitioning module (Pm). Connected to the partitioning module is an encryption module (Em) for the secure data, and an allocating module (Am) to perform general disk allocation. This ensures that the data for the sensitive application is encrypted when stored on the physical disk, and that the non-sensitive data is stored in raw form. Applications store and retrieve data without knowledge of any transformation on the stored data.

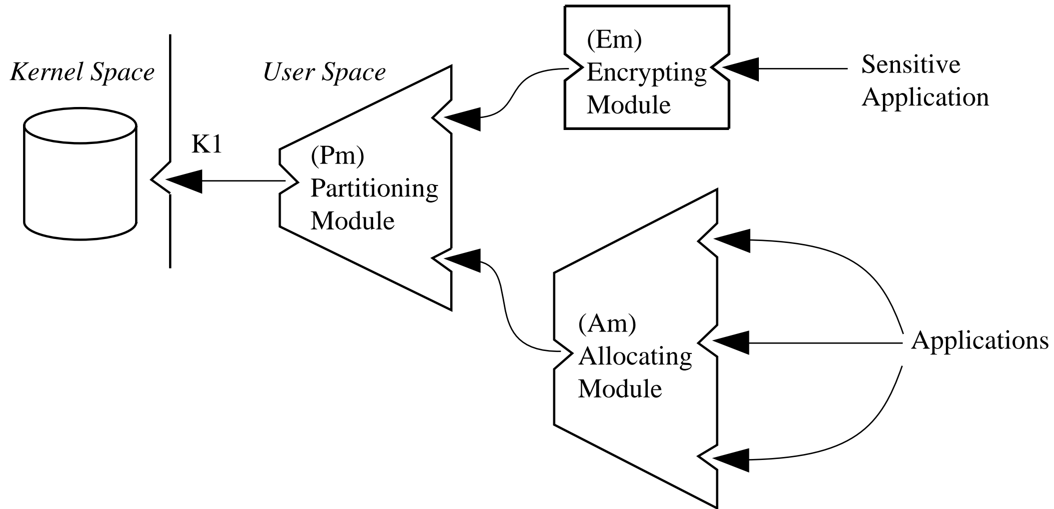

Figure 2: Implementing Different Policies

2.3 Remote Disks

On some occasions it is desirable to have the data for one node stored on another node, for example dataless nodes. A dataless node has only the operating system stored on local disks and all user data is stored on remote nodes. In this instance we would se a remote disk. A remote disk appears to be locally attached, when in actuality the physical device is attached to a remote node. It is not necessary for the remote node to be executing the same operating system provided the two nodes conform to a storage protocol.

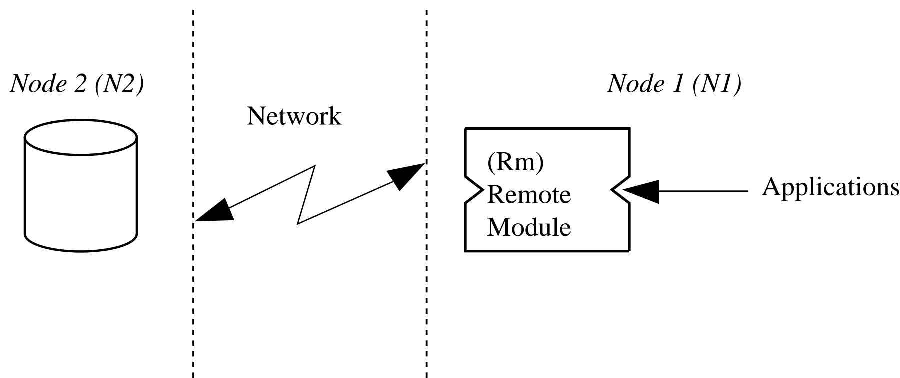

Figure 3: Remote Disks

This type of remote storage is very useful during experimentation as it allows a reliable system to be used as the disk host for the system running the experiment. This allows disks to be prefabricated, and examined irrespective of the state of the experiment.

Figure 3 shows a remote disk configuration. The disk attached to node N2 is available on node N1. The remote disk module on node N1 takes disk operations and converts them into network requests to node N2. Node N2 then performs the operations on the local disk.

In all the above examples the interface that is finally presented to the applications executing on the local node is identical. Any combination of modules can be placed between the physical disk and the application. The number of combinations that are

possible is limitless, although some combinations are less than useful: a spanning module combining all portions of a partitioning module is basically a two module delay line.

3. Model Assumptions

There are several assumptions that the model uses to improve the efficacy of operations. The first is that it is possible to efficiently move data between modules. The second is that a module is granted no special privileges; it is normal user code.

In a system where the structure is constructed from a collection of independent modules it is important to allow data to move between these modules as rapidly as possible. Disk drive systems, by their nature, are used for high speed traffic. Copying data between modules would incur an unacceptable performance overhead. For this reason, the modular approach will only work on systems where data can be referenced by easily passed parameters. Since users can write their own modules it is important to be able to pass data between the kernel and a user module, and between user modules.

Disk drivers, by their nature, and for efficiency reasons, access raw data. Since we are advocating that users be able to write disk handling modules there must be some mechanism that restricts a malicious piece of user code from accessing unauthorised data. Furthermore, modules are not run in kernel mode. They may only access the resources that are available to any other user program. A module is treated no differently from any other piece of user code.

4. The Disk Module Model

In this section we provide an abstract view of the modules which comprise a disk subsystem. Operations supported by the proposed modules are described. Some examples of modules provided by a typical operating system are presented.

4.1 The Disk Modules

Each disk module is a self contained entity providing a core set of interface routines. This set comprises entry points, each of which performs a single operation. Disk modules communicate with each other using this core interface. Apart from the core interface, some disk modules provide additional operations that allow access to functionality particular to the module.

Each disk module accepts calls on its interface, provides some functionality, and then calls the interface routine of some other module in the disk module stack. The lowest module in the stack calls the disk interface provided by the kernel.

Kernel disk functions are accessed using the same core interface as that provided by disk modules. The kernel implements the connection between the disk module stack and the physical disk controller. To achieve this the kernel must provide a driver for each kind of disk controller attached to the machine.

4.2 Disk Module Operations

The core interface operations supported by each module may be grouped into three categories according to their function. These categories are:

- read/write - to move data to and from the disk.
- control - to manipulate the disk configuration.
- status - to obtain information about the disk configuration.

Additionally a module may implement extra operations which are accessed through its extensions to the core interface. For example an encrypting module would provide an interface routine to set the encryption key.

The interface specification does not detail whether the operations are synchronous, or asynchronous in nature. For ease of implementation we have deemed that operations on a module will be synchronous. It should be noted that a synchronous interface does not preclude equivalent asynchronous operations. Asynchronous operations can still be performed by using a threads based system where a thread blocks, waiting on the operation to complete, while the main thread continues to execute [1, 2]. The full list of mandatory operations is:

- read_block
- write_block
- flush
- initialise
- reset
- format
- eject
- retrieve_layout
- release_block
- pre_allocate_block

The *read* and *write* operations are used to move data to and from a disk drive. The operations take two arguments: the block of data, and the disk address that is being accessed. The *read* operation will only return when the data has been retrieved. The *write* operation completes when the block of data has been queued for writing by the kernel. The user level modules operate synchronously for writes. The operations may fail for several reasons, e.g.; if the disk is unavailable, if the disk has been removed (for removable media), or if the data address is invalid.

Although logically all operations are synchronous, the *write* operation can return before the write has occurred, but may only return when it is guaranteed that the write can be scheduled and, barring hardware failure, will succeed. If synchronous writes are required there is a *flush* operation provided. The *flush* operation will not return until all outstanding writes have been completed.

The *flush*, *initialise*, *reset*, *format*, and *eject* operations are used to control the disk. As mentioned, *flush* commits writes to the physical disk. The *format* operation causes the specified disk to be formatted. After a format all previous information is lost. The *eject* operation causes a removable media to be ejected from the drive. The *reset* operation is provided for failure handling. If an operation fails the module may attempt to reset the drive thereby bringing it back (hopefully) to a known state, and the failing operation may be retried before declaring the operation a failure. *Initialise* is used to inform the module that a change has occurred in the configuration of the hardware, and to ask the module to re-check its knowledge of the hardware configuration. This is useful if a drive is replaced, brought on-line or taken off-line.

The *pre_allocate_block* and *release_block* operations are used to allow trapping of invalid disk block references. Trapping these references is useful for some disks, such as magneto-optical disks that allow sector based formatting. We can use this information to perform sector based formatting. Other drivers, such as the allocating module use the information from *pre_allocate_block* and *release_block* to perform sector allocation and release.

The result of a *read* or *write* operation on a block that has not been preallocated, or that has been released, is defined on a module-by-module basis. Higher level modules return an error, but some modules implicitly perform a *pre_allocate_block* on all blocks, and ignore the *release_block* operation. For read accesses to blocks which have not been

previously written these modules return a data block with undefined contents. Write operations are always successful.

The *retrieve_layout* operation is used to allow optimisation or configuration based on the physical disk layout. This operation returns the physical layout of the disk, including the size of the disk (in blocks), the number of heads, sectors per track, and the number of cylinders. If any of the disk parameters cannot be supplied then a zero is returned for that parameter. This is required as some modern disks do not have any externally useful information apart from the size of the disk, due to hidden internal layout optimisations. Composite modules, such as the spanning module cannot return any useful information apart from the size of the composite disk, as the geometries for the individual disks may vary greatly.

The operations outlined above comprise the core functionality of each module. Every module must perform these operations. Some modules may perform the operation by passing the operation to a lower level module. Other modules may modify the parameters before passing the operations to a lower level module.

Modules may have other operations in addition to the core functionality. These operations are usually used during the construction of a module stack, and are specific to each class of module. Examples of additional operations include *get_new_partition* for the partitioning module, *add_drive* for the spanning module, and *set_encryption_key* for the encrypting module.

4.3 Standard Modules

As an example of the types of user level modules that can be constructed the Grasshopper system comes with the following modules as standard:

- spanning
- striping
- replicating
- partitioning
- allocating
- remote
- encrypting

Each of the modules uses one or more parent disks to store information, and presents one or more logical disks to applications. The parent disks may be the logical disks of other modules, or physical disks controlled by the kernel. The logical disks presented are either directly used by applications, or are the parent disks for other disk modules.

The spanning and striping modules are used to make multiple parent disks appear as a single logical disk. The difference between the modules is in the algorithm used to determine the layout of data on the parent disks. The spanning module presents a logical disk that is the concatenation of the parent disks. It allows parent disks of different geometries to be combined into a single logical disk. The striping module requires all the parent disks be identical in configuration and places consecutive blocks o the logical disk on separate parent disks.

The replicating module is similar to the striping module, but implements a strategy of having multiple parent disks that hold identical data. If a disk fails the remaining disks can be used to continue service requests and to rebuild the data on the failed disk.

The partitioning module uses a single parent disk and presents multiple logical disks to its clients. The number of logical disks and their size is constrained only by the parent disk. Each logical disk has a geometry fixed at create time. Once set, a logical disk cannot be changed in size, but may be deallocated.

The allocating module interfaces to a single parent disk and presents multiple logical disks, each of variable size. Each of the logical disks has no defined geometry. All geometry parameters are returned as zero on *retrieve_layout* operations.

The allocating module operates by allocating blocks for each logical disk from a general pool of blocks. The general pool of blocks are stored on the parent disk. The allocating module allows sparse logical disks, so the higher level modules can use disk lock indices up to the maximum implementation value for a disk index. To facilitate the bookkeeping of disk blocks the allocating module responds to the *pre_allocate_block* and *release_block* operations by allocating the block from the general pool or returning the block to the general pool. Allocation of a block may fail if the parent disk is full, but once a block is allocated, writes to that block are guaranteed to succeed.

The allocating module may allocate the blocks for operations using the algorithm most appropriate. The interface specification does not define anything about the internal organisation of the data, only that the data is retrievable. The module may use pointers, or bitmaps [8] depending on the constructor of the module.

The remote module presents to applications a logical disk that is actually stored on a remote host. This allows a disk stored on a reliable host (e.g. UNIX), and be available for testing drivers at all levels. The disk contents may be examined on the reliable host to determine if the drivers are working correctly.

The encrypting module is used to perform simple encryption using a one bit rotor style encrypter. It encrypts writes from its logical disk to its parent disk, and decrypts reads from the parent disk to the logical disk. It is used for encryption of data being placed onto removable media where the physical security of the media is questionable.

These drivers are all small pieces of easily written code. Each builds on a standard module interface library. For example, the spanning module is implemented by an additional 62 lines of C code.

5. Conclusion

This paper describes the disk driver structure in the Grasshopper operating system. The module based system provides a simple and clean interface between driver modules. The modules may be combined to provide flexible and powerful disk storage systems.

The module abstraction allows applications to access disk storage in a uniform manner, irrespective of the data layout desired. Furthermore the kernel driver is a simple and clean driver. The kernel need not implement any management policy.

The advantages of the system include the ease of constructing specialised disk stacks, the ability to write and commission modules with new functionality on a running system, and the flexibility with which experiments on the disk interface can be performed. The system reduces the amount of code necessary in the kernel, and improves the kernel understandability and maintainability.

Currently, the design is being implemented on the SUN Microsystems 3/60 prototype of Grasshopper. Implementation will shortly move to DEC Alpha based machines, where the full operating system is being developed.

Although the Grasshopper operating system is a persistent system the design presented is not dependent upon persistence for operation, and would perform in a more conventional environment.

Acknowledgements

The work described in this paper is supported by Australian Research Council grant A49130439 and by an equipment grant under the Alpha Innovators program from Digital Equipment Corporation.

Thanks must go to both Stephen Norris and David Hulse who have recently joined the Grasshopper project for their comments on drafts of this paper.

References

[1] Cheriton, D. “Multi-Processing Structuring and the Thoth Operating System”, ICS-79-19, University of Waterloo, May 1979.

[2] Cheriton, D. “The V Distributed System”, Communications of the ACM, 31(3):314-333, March 1988.

[3] Chorus-Systemes “Overview of the CHORUS Distributed Operating System”, CS/TR-90-25, Chorus Systemes, 1990.

[4] Clark, R., Jensen, E., Reynolds, F. “An Architectural Overview Of The Alpha Real-Time Distributed Kernel”, Usenix Workshop on Microkernels and Other Kernel Architectures, April 1992.

[5] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindstroaom,A., Rosenberg, J. and Vaughan, F. “Grasshopper: An Orthogonally Persistent Operating System”, Technical Report,Departments of Computer Science, Universities of Adelaide and Sydney, 1993.

[6] Forin, A., Golub, D., Bershad, B. “An I/O System for Mach 3.0”, Anonymous FTP from mach.cs.cmu.edu.

[7] Grasshopper, “Self Stabilising Entities in Grasshopper”, Technical Report, Departments of Computer Science, Universities of Adelaide and Sydney, In Preparation.

[7] Kougiouris, P. “A Device Management Framework for an Object-Oriented Operating System”, Masters Thesis, University of Illinois, 1991.

[8] Long, D. “A Note on Bit-Mapped Free Sector Management”, Operating System Review, 27(2):7-9, April 1993.

[9] SCSI-2 “Small Computer System Interface - 2 Standard”, Document X3T9.2/86-109, Rev 10B, Global Engineering Documents.

[10] Sun Microsystems “SUN Common SCSI Architecture”, Document Number 800-4701-10, Revision B, Sun Microsystems Inc., March1990.

[11] Sun Microsystems `“SUN Common SCSI Architecture”, Document Number 800-4700-10, Revision A, Sun Microsystems Inc.,November 1989.