DATE: August 5, 1982

TO: R & D Personnel

FROM: Garry Kessler

SUBJECT: Requirements Spec for the Prime I/O System (PIOS)

REFERENCE: PE-TI-989 (Prime I/O System)

KEYWORDS: PIOS, DSAG, PDA, streams, input/output, device independence

ABSTRACT

The new Prime I/O System (PIOS) will rewrite the current I/O system to include the following features:

- I/O redirectability and device independence
- abstract objects
- generalized name space (links, search lists)
- record I/O
- simple IPC primitives

This paper presents the goals and requirements of PIOS.

Table of Contents

Executive Summary....4

Glossary....5

1 Motivation....9

2 Goals....11

3 Competitive Summary....12
- 3.1 Summary of Features Discussed....12
- 3.2 Hierarchical Name Space....13
- 3.3 Generic Files....13
- 3.4 Name Space Conventions....14
- 3.5 Access Control....14
- 3.6 Connections....15
- 3.7 I/O Objects....15
- 3.8 Types of I/O....15
- 3.9 More on Record I/O....16
- 3.10 Primos Deficiencies....18

4 Overview....19
- 4.1 Characteristics of PIOS....19
- 4.2 The User's Viewpoint....20
  - 4.2.1 Abstract Objects....20
  - 4.2.2 Name Space Operations....21
  - 4.2.3 Stream Operations....22
  - 4.2.4 Examples....22

5 Requirements....25
- 5.1 I/O....25
  - 5.1.1 Name Space....25
  - 5.1.2 Stream Operations....27
  - 5.1.3 Types of I/O....27
    - 5.1.3.1 Common I/O Operations....27
    - 5.1.3.2 Record I/O....28
    - 5.1.3.3 Block I/O....28
    - 5.1.3.4 Unpended I/O....29
- 5.2 Ease of Use....29
- 5.3 Compatibility....29
- 5.4 Access Control....30

5.5 Error Reporting.....30
5.6 System Startup and Configurability.....30
5.7 Implementation.....30
5.8 Extensibility and Portability.....32
5.9 Database Management.....32

6 Comparison of PIOS with Competition.....34

7 Dependencies.....35

8 Non-requirements.....38

9 Issues.....39

Executive Summary

The Prime I/O System (PIOS) Project proposes to rewrite and enhance the following areas of Primos:

- o file system tables and access methods
- o input/output at all levels
- o dependency on particular CPU architecture
- o access control
- o name space management

This project will also support the following architectural changes:

- o DSAG service oriented environment (PDA - Prime Distributed Architecture)
- o distributed file system intelligence (e.g. smart disk controllers)
- o portability of Primos to other CPU families (i.e. isolation of machine dependent code)

In addition, PIOS will add to Primos features offered by Prime's competitors, and thereby improve Prime's position in the marketplace. These features include:

- o I/O redirectability
- o device independence
- o record I/O
- o uniform I/O interface
- o support for language I/O

PIOS is a major rewrite of the Ring 0 part of the operating system. Rewriting is more cost effective than patching because of the extent of the changes required, and the intractability of the existing code. Since the I/O system accounts for a very large percentage of operating system bugs (70% for Rev. 19 - cf. PE-TI-989), restructuring the system should result in a significant improvement in reliability and maintainability.

Compatibility will be assured; no software other than Primos and its utilities is required to change. Since many of the changes are being made for the benefit of other software (e.g. device-independent I/O for languages) commitments from other groups will be needed to put the new features into use.

Glossary

Here we define terminology used in this specification. Underlined words indicate terms that appear elsewhere in the glossary.

Abstract object - A named set of data and operations. The operations defined on the data can include conventional ones like read and write, as well as ones of arbitrary complexity defined by the user; the operations also include access control information. Examples of objects are files, devices, and services; all I/O is done to/from objects. There are three kinds of objects: simple objects, composite objects, and generic objects.

Aliasing - The ability to temporarily redirect the target of an I/O operation from a "hard-wired" global name to some substitute target, as for debugging purposes. In PIOS, aliasing is done using the associate stream operation.

Association - To associate two names is to relate or connect them in preparation for doing I/O; associating names is the first step to defining a stream. The names associated can be two local names, a local name and a global name, or two global names (aliasing). All associations are local, i.e., process-specific.

This notion is more general than the usual one of connection, since any two names can be associated.

Asynchronous I/O - See unpended I/O.

Block I/O - An I/O interface that typically transfers data by reference rather than by value (no copy mode I/O). No formatting of data into records is done.

Composite object - An object which contains more than one object within itself; the internal objects are called subobjects. A composite object could be used when a number of objects are logically equivalent (e.g., servers in a service), when the internal structure of objects must be hidden from users (e.g., certain Data Management objects), or to conglomerate many small objects into one large one.

Connection - An association between a generic file known to a process and a resource known to the system. A less general notion than association, since one end must be a local name.

Device independence - The ability to use a single interface to perform I/O to all types of objects. This allows programs to be written which do not depend on the peculiarities of any particular device. See I/O redirectability.

Distributed system - A distributed system has many units capable of varying degrees of computation and data storage. Each unit is independent of all others in that units do not share main memory. All units are interconnected in such a way that users need not be aware of discrete resources or system topology. Rather, they perceive the system as a set of services for information storage, retrieval, and modification.

File elements - A file system feature which allows files to be stored on contiguous areas of disks for efficient I/O.

Filter - A kind of stream which performs a transformation on the data which passes through it. A filter is associated with a particular target in a stream and specifies how data entering the target is to be transformed; in this sense it is an association from a target to itself (a loop).

Fork - A special association which allows one end of a stream to be split into two parts, thus sending the data to two different destinations.

Generic file - A local (process-specific) name used to refer to global names to provide I/O redirectability.

Generic object - An object whose name is in the local name space of a process. A generic object is used as an I/O port: the physical or logical resource it is related to can be redefined by a process in order to redirect I/O.

A generic object is a more general notion than a generic file, since access control and non-standard operations can be defined on an object.

Global name - A name in the hierarchical name space which is known across all processes.

Hierarchical name space - A name space in which names are organized into a vertical (hierarchical) structure as well as a horizontal (flat) structure.

The name space seen by a process is divided into global names and local names. When a global name is the same as a local name, the local name is given preference.

I/O redirectability - The ability to change the source or sink of an I/O operation without modifying an existing program. In a fully general I/O system, this ability is available even for programs which do not use a device independent interface (i.e., which refer directly to physical resources). This last ability is known as aliasing. General redirectability is provided by the association operation.

I/O system - That portion of the operating system that enables users to locate and use non-main memory storage and communications resources; it includes facilities to identify, share, use and manage a system's I/O resources.

Links - File names that point to other file names instead of data.

Local name - A name which is known only to a particular process.

Object - See abstract object.

Naming service - A service which manages the hierarchical name space.

Open - Operation which defines the direction of flow and the logical access of data in a stream; completes the definition of a stream.

PDA - Prime Distributed Architecture. A distributed operating system currently under development.

Record I/O - An I/O interface that performs data blocking, unblocking and formatting functions for the user.

Search lists - Lists of directories to look for names. A search list is a list of "hints" to a naming service.

Server - One of a set of processes which implements a service.

Service - A set of processes (servers) which insulate users from the physical characteristics and location of a resource. A service is implemented as an abstract object, so that data can be transferred between it and a user process using the same I/O interface used for all objects. A service can be used to implement a set of operations on other abstract objects.

Simple object - An object which does not contain subobjects.

Sink - Object to which data flows in a stream; data is written to the sink.

Source - Object from which data flows in a stream; data is read from the source.

Stream - An I/O stream is a logical conduit through which data flows. To define a stream, a user must specify the I/O objects in the path of the stream, the direction of the data flow, and the way the data is to be read/written. The path is defined by associating the objects; the direction of flow and the logical access is defined by opening the stream.

Subobject - An object which is part of a composite object. A subobject may also be composite.

Target - Either end of an association; a source or a sink.

Unpended I/O - An I/O interface that permits overlapping of I/O and CPU times (aka asynchronous I/O).

1 Motivation

The I/O system is that portion of the operating system which enables users to locate and use non-main memory storage and communications resources; it includes facilities to identify, share, use and manage a system's I/O resources. The current I/O system has several inadequacies, all of which can be attributed to its haphazard design and implementation:

- Lack of features offered by our competitors: I/O device independence, redirectability, file elements, and record management support for fixed and varying length records are common operating system features among our competitors. (File elements are a file system feature which allows files to be kept on contiguous disk records for efficient I/O.) Prime lacks these and is at a competitive disadvantage.

  In addition, the lack of support for language I/O causes extra layers of software to be produced and obscure code to be written, increasing an already high maintainance load (see below).

- The system is difficult to use and understand. This shows up in misleading terminology, the existence of obsolete features, and the definition of assymetric operations. For example, although segment directories are called directories, operations that can be performed on them are very different from those that can be performed on a UFD or sub-UFD. In addition, segment directories were introduced to fill a need for objects that function somewhat like directories but are large. In a file system which allows large directories and files, such a special case would not be needed. Another place where assymetry shows up is in obtaining the length of a file; getting the length of a SAM file is a much more expensive operation than for a DAM file because the length of a SAM file is not stored in the file header. Such a disparity is unnecessary, inconsistent, and inefficient.

  A final example is the use of highly encoded physical device numbers. To add a disk to the system, the user must know the number of platters in the partition, the controller the drive is on, and the device number of the drive; he must then correctly encode this information into the physical device number. The difficulty of use is bad enough, but this procedure also results in resource limitations: Prime machines cannot handle more than two disk controllers for the simple reason that only a single bit is reserved in the physical device number to specify the controller. Prime machines are limited in the size of partitions because of the number of bits allowed for the number of platters in a partition. As a result, a full 600Mb disk cannot be used as one partition. These restrictions are unacceptable.

- o Lack of full transparency. Users must be aware of network topology when performing some operations. For example, accessing a remote file can fail even though the machine on which it resides is on the user's network; this can happen if the partition has not been explicitly added to the user's machine. The requirement that the operator explicitly add remote disks to network nodes was intended as a security feature; however this feature often gets in the way more than it is useful, and attempts to handle a security problem outside of Prime's official security mechanism ACLs. For consistency all security features should be implemented under one mechanism.

- o The system is not easily debugged or modified. On the order of 2/3 of all bugs reported are in the I/O system (cf. PE-TI-989).

- o Inflexibility - Many soft resources are statically allocated at system startup; reconfiguration requires a system reboot. This makes it difficult to customize a system to user needs. It also places an added burden on developers who must do frequent reloads and reboots.

2 Goals

The overall goal of PIOS is to remove the inadequacies of the current I/O system and improve Prime's competitive position. Specifically, PIOS will:

- o lay a foundation for PDA (Prime Distributed Architecture).
- o make the system easier to use.
- o make the system more extensible and portable.
- o add new features (e.g. device independence, I/O redirection).
- o provide support for intelligent devices (intelligent disk controller, I/O processor).
- o maintain compatibility with existing commands.
- o improve maintainability and reliability.
- o provide support for database management products.

3 Competitive Summary

Three systems were chosen as representative of the marketplace in which Primos competes. Data General's AOS and Digital's VAX/VMS are Prime's most consistent competitors. Bell System's UNIX is included as a widely accepted operating system in academic environments. All three offer I/O systems superior to Prime's.

An I/O system allows users to identify, share, use, and manage a system's I/O resources. This section discusses competitive offerings in the first three areas and only touches on resource management since such issues are implementation oriented.

3.1 Summary of Features Discussed

- hierarchical name space
  A conveniently structured way for referring to I/O objects (files, devices, and services).

- generic files
  A set of local (process-specific) names used to refer to I/O objects.

- shorthand methods for referencing long names
  Methodologies for hiding the real names of objects and for reducing typing.

- security and access control
  Limiting access to objects.

- sophisticated connection capabilities
  Associating names for the purposes of doing I/O.

- one I/O interface for all objects
  Using connections to perform I/O on a variety of objects without concern for peculiarities of specific objects.

- record I/O
  An I/O interface that performs data blocking, unblocking and formatting functions for the user.

- o block I/O
  An I/O interface that typically transfers data by reference rather than by value (no copy mode I/O).

- o unpended I/O
  An I/O interface that permits overlapping of I/O and CPU times (aka asynchronous I/O).

3.2 Hierarchical Name Space

Systems associate names with resources so that resources may be easily identified and used. A hierarchical name space is one in which names are organized into a vertical (hierarchical) structure as well as a horizontal (flat) structure. Naming hierarchies are desirable because they are flexible, uniform and easy for users to learn and use.

Names are arranged in a hierarchy in all three systems. Devices are members of a distinguished directory near the top of the hierarchy. Generally some special character (or character sequence) is used to identify the distinguished directory, such as the '@' in @CON0 for the AOS system console. Both AOS and VAX naming hierarchies include network node names at their top levels; UNIX does not.

The AOS naming hierarchy has an additional unique feature. It allows whole subtrees (i.e., other disks) to be inserted and removed anywhere below the top level of the hierarchy. VAX/VMS permits this only at a single point near the top of the hierarchy.

3.3 Generic Files

Names are ultimately resolved to resources by the operating system. Different systems use a variety of mechanisms for resolving names. However, the concept of generic files is common among them.

Generic files are a set of names that are private to a process. The process may associate devices, files or services with generic files. When the process performs I/O on a generic file, the I/O is performed on the associated device, file or service. This insulates the process from specific knowledge of I/O objects, and allows I/O bindings to be made at run time rather than at coding time. Deferring bindings makes programs more flexible.

The various systems support different generic files as standard. UNIX supports generic files for input and output. AOS supports generic files for input, output, data and list. VAX/VMS supports generic files for input, output, error, command input, command source and library. The tradeoff here is simplicity (few files) versus control (many files).

Generic files may be connected to other generic files for ease of use. For example, interactive users have process input and output assigned to a console file. The console file is assigned to the terminal. This extra level of indirection through the console file permits users to conveniently redirect both input and output by redirecting the console file.

In addition to the standard generic files, VAX/VMS allows users to make up their own, and provides an extra layer of insulation with an alias facility (logical names).

3.4 Name Space Conventions

UNIX and AOS provide some name space conveniences in the form of links and search lists. Links are file names that point to other file names as opposed to data. Search Lists are lists of directories to look for names. VAX/VMS does not have search lists and this is a weakness. In general, however, all systems provide some mechanisms for abbreviating and hiding resource names. These mechanisms can be used to build complex name space structures (more complex than trees).

3.5 Access Control

All systems include mechanisms to control access to resources by controlling access to their names. AOS uses access control lists. VAX/VMS uses a variant of access control lists and Bell UNIX uses passwords. Access control lists (ACLs) are preferred over passwords because they are easier to use and provide more flexibility and security. With ACLs complex access controls can be defined. Specific users or groups of users can have different subsets of operations they are allowed to do. With passwords, knowing the password gives you all access.

3.6 Connections

A connection is an association between a generic file known to a process and a resource known to the system. The association gives the process access to the resource through the generic file. The concept of establishing connections is fundamental to I/O systems that use generic files and is found in all the systems discussed here.

AOS and VAX/VMS support only simple connections while UNIX supports some more sophisticated constructs known as forks and filters. Forks are specialized connections that permit a single generic file to be associated with multiple system resources. This is useful, for example, in making multiple copies of an output file. Filters are also a specialized type of connection. They perform transformations on data that pass through them. An example might be an ASCII to EBCDIC conversion filter. Sophisticated connections are desirable because they provide users with increased flexibility in the I/O system which makes more complex problems easier to solve.

3.7 I/O Objects

I/O objects are system resources that are available to users to perform input or output operations. In AOS and VAX/VMS, devices, files and server processes are all potential I/O objects. In UNIX devices and files are I/O objects; processes are available in some extended UNIX systems. Having one connection mechanism work with a wide variety of I/O objects produces systems that are consistent, flexible and easy to use; users need to learn only one interface.

In UNIX and AOS objects are divided into two classes, character class and block class. Typically this distinction reflects the inherent capabilities of the objects. Character objects perform I/O a character at a time. Block objects may perform I/O many characters at a time.

3.8 Types of I/O

Three broad types of I/O operations are offered in the systems discussed here. All transfer data between user processes and system resources.

In record I/O data is transferred in units of logical records, one record at a time. This mode of I/O is directly accessible via language I/O. Of the I/O modes discussed here, it is the most sophisticated; it provides users with a logical record interface which performs required data blocking, unblocking and reformatting. Record I/O corresponds to the I/O model found in many high level languages.

In block I/O data is transferred in arbitrary sized chunks or blocks, one block at a time. In this case no restructuring of the data into logical records is performed. This mode sacrifices some of the functionality of record I/O in order to improve performance. Typically block I/O is a no copy I/O mode where data is passed by reference rather than by value. Obviously, block I/O is advantageous when the record structure of the data is not of interest and record formatting overhead would be wasted. Bulk copying is such an application.

In unpended I/O more than one request for data transfer may be outstanding at the same time (aka asynchronous I/O). In other words a process need not wait for one request to complete before issuing another. This mode is used to overlap I/O times with CPU times. Bulk store backup applications are users of unpended I/O. Unpended I/O can be used on both records and blocks.

VAX/VMS offers all three types of I/O. AOS does not offer unpended I/O directly, but the effect may be obtained via multitasking. UNIX offers only block mode I/O. The claim is that users do not need the generality of multiple record types. The tradeoff among all of the systems is flexibility and performance versus complexity.

3.9 More on Record I/O

Record I/O provides users with the greatest insulation from the peculiarities of individual I/O objects. The system performs formatting operations on the data automatically; users see data in units of standard logical records. Both VAX/VMS and AOS use records. UNIX permits only one record type (dynamic; see next paragraph).

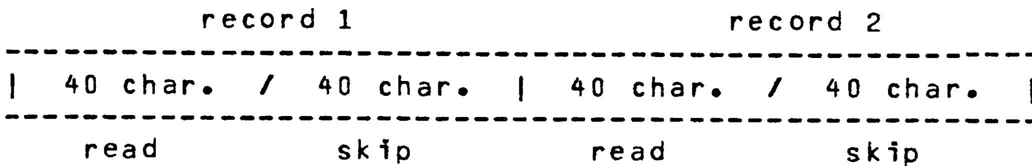
Records have a variety of types distinguished by how their lengths are established. Dynamic records have lengths that are specified with every read or write. Both UNIX and AOS support them. AOS supports three other record types as well. Fixed length records have a predefined, but settable, common length. Variable length records have their lengths encoded at their beginnings (32 bits). Delimited records are terminated by a special character (settable). VAX/VMS supports fixed and variable records and a hybrid type called variable with fixed control. Table 1 summarizes the record types supported by each system.

Support for a variety of record types is a convenience for languages and applications. Language specifications dictate the need for various record types. Furthermore, it should be possible to read data with a record type that is different from the type with which it was written. This is possible under both AOS and VAX/VMS, but with different results.

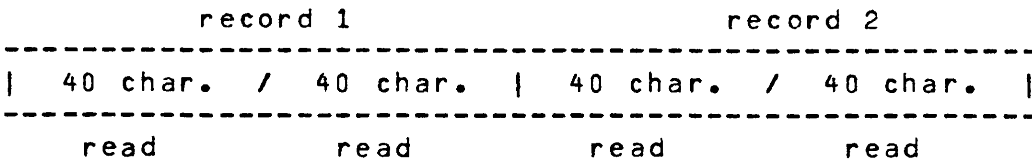
Record Type	System: AOS	VMS	UNIX
Dynamic	yes	no	yes
Fixed	yes	yes	no
Varying	yes	yes	no
Delimited	yes	no	no
Hybrid	no	yes	no

Table 1: Record Types Supported

VAX/VMS strictly adheres to the record type of objects. An attempt to read an 80 character, fixed length file as a 40 character, fixed length file will result in a truncation of the second 40 characters in each record. This allows short, fixed size buffers to be used when the desired data is towards the beginning of the records.



AOS adheres to record types of objects more loosely. An attempt to read the same 80 character, fixed length file 40 characters at a time will result in successive reads of 40 characters sequentially reading <u>all</u> of the data in the file. This allows for the reading of objects with fixed sized buffers without regard to the true type of the object.



3.10 Primos Deficiencies

- Hierarchical name space
  The Primos file system is a hierarchical name space. However, it lacks the generality and conveniences of VAX/VMS, AOS, and UNIX. The hierarchy does not include network nodes or system devices. Shorthand conveniences such as links, synonyms and search lists are either not provided or have only recently been introduced.

- Generic files
  Primos has no support for generic files.

- Security and access control
  Primos is improving its access control mechanism for files. The ACL mechanism at Prime is superior to those on other systems in that group names are supported. However, there is no access control on devices; services are not supported.

- Connection capabilities
  Primos has no connection capabilities.

- One I/O interface for all objects
  Primos does not support interchangeable access to files and devices. Language libraries do provide some support in this area, but they require active use on the part of the application as opposed to being a feature of the system provided for free. Therefore, applications typically do not support interchangeable accesss to objects.

  Much I/O is done by direct calls to I/O drivers. This results in a different and often obscure interface for each device, and causes maintainance and ease of use problems.

  There is no service process support in Primos.

- Record I/O
  There is no record I/O support in Primos. Languages do implement record I/O; however, there is no guarantee that a file written with one language can be read with another.

- Block I/O and Unpended I/O
  Block I/O is supported by Primos and is not a major deficiency. Unpended I/O is supported only for tapes.

4 Overview

In this section, we present an overview of PIOS from a user viewpoint. The term "user" is purposely left ambiguous; it may range from a non-programmer all the way to a sophisticated systems programmer.

4.1 Characteristics of PIOS

1. Device-independent I/O

   PIOS provides one interface to perform all input/output functions to all types of I/O objects. This interface can be used by compiled code (to do language I/O), or by application programs directly. The source/target of the I/O can be changed for any run of the program, without modifying the code.

   The source/sink of the I/O can be a PDA <u>service</u>, allowing a service oriented environment to be integrated into programs and programming languages in a natural way. A service is a set of processes, or servers, which performs information retrieval, storage, or transformation while insulating the user from the physical characteristics and location of the actual resources.

2. I/O Redirectability

   Besides providing a device-independent I/O interface, PIOS allows a user to redefine the source and target of any I/O operation. This is done by "connecting" a conduit or stream to the source or target desired. PIOS provides a uniform facility for locating local or remote files, devices, and services. The program will not need to concern itself with how this is done.

3. Hierarchical Name Space

   <u>N. B.:</u> This part of PIOS has very strong dependencies on the design efforts of DSAG; cf. the section on "Dependencies".

   PIOS provides a tree structured name space that will be similar to today's system in most ways, though its internal structure will be redesigned.

   The naming and configuration of media will be improved over today's octal partition numbers. In particular, any source/sink for information will be named.

The PIOS name space will include devices and services. All extensions will be consistent with any name space design produced by DSAG.

Users will use name space operations to create, delete and modify the attributes of I/O objects.

4. Access Control

PIOS will provide a high level of access control over all objects. This control will be compatible with the current ACL mechanism. The goal here is to ensure that data or services cannot be accessed by unauthorized persons.

5. Portability

Device dependent code will be isolated and a device independent interface developed, so that alternative media such as intelligent disks can be simply "plugged in" to the rest of the system. Isolating hardware dependent code, and defining clean interfaces to this code will facilitate migrating Primos to future processors.

4.2 The User's Viewpoint

This section describes at a high level how PIOS might be used. It is intended to give a feel for the way PIOS "fits together" at the user level. Some assumptions have been made about how PIOS will be implemented; these assumptions may not hold in the final design of PIOS.

All I/O is done from and to _abstract objects_. Objects are defined using _name space operations_, and the datapaths used to do I/O are defined using _stream operations_.

4.2.1 Abstract Objects

The source and sink for any I/O operation can be any abstract object; this is the central concept of PIOS. An _abstract object_ is a named set of data and operations. Files, devices, and services are all examples of abstract objects. Objects having arbitrary operations and attributes can be defined using PIOS operations. The operations defined on the object can include conventional ones like read and write, as well as ones of arbitrary complexity defined by the user; the operations also include access control information. This kind of very general control over data manipulation is similar to the language concept of abstract data types. Using one interface for object definition and access control results in a very general and flexible,

yet simple mechanism.

The name of any object can be either global (system-wide) or local (process-specific). Global objects have their names in a global name space; local objects are known only to the process that created them.

There are three kinds of objects: simple, generic, and composite. A simple object is one which does not contain other objects as its data portion; it is a single named set of data and operations. A file is an example of a simple object; the contents of the file are the data part of the object and the operations defined are usually just the standard ones of read, write, and execute. A generic object is used as an I/O port; there is no "real thing" to which the name always applies. Generic object names are always local and are used to associate a local name (the generic object's name) with a global object. Generic objects are used to write programs and processes with runtime redirectable I/O. A generic object is more general than a generic file because arbitrary operations can be defined on them, and users can make their own generic objects. An example of a generic object might be STDIN, a standard input port for a process; STDIN would probably be associated with a user's terminal by default. A composite object is a complex object containing more than one object (subobjects) as its data portion. A composite object could be used when a number of objects are logically equivalent (e.g., servers in a service), when the internal structure of objects must be hidden from users (e.g., certain Data Management objects), or to conglomerate many small objects into a large one. Composite objects are usually managed by a separate process called an object manager.

4.2.2 Name Space Operations

Name space operations are used to create, alter, and destroy abstract objects, which are the sources and sinks (targets) of all data transfer. Users may create any kind of object they like. An object is created by specifying its name, the type of object, where the name resides (local or global name space), the data, and the operations including access rights. Removing an object's name from the name space causes its destruction.

Each process can reference objects in the global name space, which has the names of all objects that can be referenced system-wide. This name space is managed by a name space service, which is responsible for maintaining the integrity of the name space across all nodes in the distributed system, and for resolving names into references to specific objects.

Processes will also have a local name space which contains the names of objects that are known only to that process. Any kind of object can be in the local name space, but the most likely inhabitants are generic object names. These are local because the specific binding for each name (the object it is associated with) is different for each process. Another example of a locally known object is a subobject known to the

object manager of a composite object; the composite object's name would (most likely) be global, but the names of the subobjects would be known only to the object manager.

4.2.3 Stream Operations

Stream operations are used to create, alter, and destroy data conduits (streams). The operations define the source and sink of an I/O stream, and the logical and physical characteristics of the data transfer. Typically, physical characteristics are hidden from the user, while the logical characteristics are not.

The most basic stream operations are associate and open. Associate is used to "connect" names in the (local or global) name space, in order to define the path that the data is to take in the upcoming I/O. The command can be used to associate two local names, two global names, or a local name and a global name. The "connection" operation found in other system is less general since it allows only the last kind of association.

The open operation is used to define the direction of data flow and the kind of access. For instance, if the name HERE is associated with the name THERE, the stream could be opened for reading from HERE to THERE in records 80 characters long. In most cases, the open operation should be implicit. That is, associating HERE and THERE and then doing a read operation on object HERE should implicitly define the direction of data flow and some "reasonable" kind of access.

To summarize, the associate operation is used to define the objects in the path of a stream, and the open operation is used to activate the stream.

4.2.4 Examples

1. As an example of the use of generic objects, suppose a typical PIOS system has an electronic mail service, associated with the generic object name MAILMAN. To send a letter, the user merely writes to the object MAILMAN; to receive his mail, he reads from MAILMAN. More generally, rather than using the standard read and write operations to receive and send mail, special operations "send" and "receive" could be defined on the generic object MAILMAN. The send operation could do such things as forwarding, returning mail to sent unknown addresses, etc. The receive operation could sort incoming mail into separate files according to who sent them, delete "junk" items, etc. Since MAILMAN is a local object, a user could customize his mail service by defining any operations he wishes on MAILMAN. In addition, by associating the name MAILMAN with some local temporary file, he could redirect the I/O (while debugging a program that uses MAILMAN, say).

2. PIOS I/O operations can be used at command level or from programs. Typically, a user would associate two object names at command level, thereby defining the targets of the I/O to follow. The system at this point would also determine certain physical characteristics of the anticipated data flow, from its knowledge of the objects involved. The direction of flow and logical access is not yet defined. Let us say a user wishes to write into a file named "report_file" using a program which writes into the generic object OUTPUT. He would first create an association between the generic object OUTPUT and the file "report_file". The command might look like:

```
associate OUTPUT -and Report_file
```

That OUTPUT is associated with report_file is not significant to the user's program since it references only the generic object, OUTPUT; if he substitutes Printer5 for Report_file, the program will continue to work.

Programs may also issue association requests. For instance,

```
call as$asnam (OUTPUT, Report_file)
```

The association merely specifies the endpoints of a datapath; it does not specify the direction or logical characteristics of flow. To do this, the user must <u>open</u> the association for some specified logical access. This can be done at command level:

```
open OUTPUT -for logical_access_definition
```

or, from a program:

```
call as$open (OUTPUT, logical_access_definition)
```

PIOS verifies that Report_file, which is associated with OUTPUT, may be accessed as requested by the logical_access_definition and initializes OUTPUT for such access. An example of a logical_access_definition might be to open the association for write access using fixed length records with a record length of 80 characters. Actually, it should be possible for most opens to be implicit. The system should be able to determine from the physical characteristics of the objects and the operations performed what sort of default opening would be reasonable. This would allow the user to do I/O without being aware of logical file formats, file units, etc.

The open command completes the definition of a stream. The user is now ready to invoke his program and write into Report_file. His program can say:

```
call as$write (OUTPUT, buffer)
```

5 Requirements

In this section we give the high level requirements of PIOS. The intent here is to define the overall design criteria to be used in later functional and design specifications. The requirements were chosen using the goals and the competitive analysis.

5.1 I/O

All I/O is done to and from objects. These objects are defined using name space operations, and the datapaths used to do I/O are defined using stream operations. Before we discuss proposed I/O operations, then, we will discuss name space characteristics and stream operations.

5.1.1 Name Space

The full definition of name space characteristics and operations is dependent on interaction with DSAG. Here we describe the general features PIOS would expect to have in this design.

PIOS will extend today's hierarchical name space to include devices and services. Links and search lists will be supported. Aliasing will be provided through the association facility.

It will be possible to add subtrees anywhere in the name space.

Users can define their own objects using name space operations. These operations will include:

- o Create_generic_name - Enter a new local name in the local name space and create a generic object with specified operations and access rights.

- o Destroy_generic_name - Delete a local name from the local name space and destroy the corresponding generic object.

- o Create_global_name - Enter a global name in the global name space and create a (simple or composite) object with specified operations and access rights.

- o Destroy_global_name - Delete a name from the global name space and destroy the corresponding global object.

o Change_name - Change a local or global name.

The deletion operations will have two forms: a soft delete and a hard delete. A soft delete will allow recovery of the deleted object for some (probably small) system defined time after the delete operation. A hard delete destroys the object immediately.

PIOS will support simple, generic, and composite objects.

Objects will be able to span partitions, although subobjects may be reauired to be on one parition.

Files will be allowed to cross partitions.

File elements (contiguous allocation of files) will be supported.

Objects will have attributes which can be examined:

o Get_all_attributes - Retrieve values of all object attributes.

o Get_attribute - Retrieve value of a particular attribute.

o Set_attribute - Set value of a particular attribute.

These operations would be an extension of current file attributes to abstract objects.

Generic objects will be provided for physical devices (e.g., TTY, PRINTER, MAGTAPE, DISK, etc.). These generic objects will reference the service for the particular device. The names of specific generic objects may be different in the final design.

Access to all devices will be through services. Users will communicate with services using Interprocess Communication (IPC). The definition of an adequate IPC mechanism is the responsibility of DSAG (cf. "Dependencies").

Other services will also have generic objects associated with them (e.g., MAILMAN). NULL will be the generic name for the bit bucket.

In addition, PIOS will provide a large number of predefined generic objects for such things as command input, command output, error messages, etc.

5.1.2 Stream Operations

User level I/O will be device independent and run-time redirectable; user programs need not know the characteristics of I/O sources and targets. This will be accomplished using streams. The stream operations supported by PIOS will be fully specified in the functional specification. The types of operations which should be considered are:

- Associate - Associate two targets as endpoints of a stream; the targets can be any kind of object.

  If a name is in both the local and global name spaces of a process, the local name is preferred.

- Disassociate - Remove the association between two targets.

- List_associations - List all or part of a process's current associations.

- Shadow - Associate one output object with another output object. This causes a split stream in which the output goes to two places (one of them a "shadow" of the first).

- Transform - Associate data transformation software with a generic object name.

- Remove_transformation - Remove data transformation from a generic object.

- Open - Activate stream with a particular logical access method.

- Close - Deactivate stream.

5.1.3 Types of I/O

PIOS will support record and block I/O; unpended I/O will be provided only through IPC primitives. First we will discuss I/O operations common to both record and block I/O.

5.1.3.1 Common I/O Operations

All terminal I/O will be done using an abstract terminal type (Standard Terminal Interface - STI; see "Dependencies"); users should not need to know the characteristics of particular terminals.

I/O interfaces will be well integrated with language I/O; this will eliminate the need for extra layers of software. Files written using one language will be readable by other languages.

The I/O system will be structured such that future use of an I/O processor (IOP) can be easily assimilated.

I/O operations supported will include:

- o Read - Read item (a block or a record).

- o Write - Write item.

- o Set_position - Position I/O target.

- o Get_position - Retrieve position of I/O target.

- o Set_state - Change state (TBD in functional specification) of I/O target.

- o Get_state - Retrieve state (TBD) of I/O target.

- o Get_characteristics - Retrieve the physical chracteristics of an object (such as record type).

5.1.3.2 Record I/O

PIOS will support fixed length, varying length, and delimited records. When the two ends of a stream are connected to records of different type, short records will be padded and long records will be truncated. Users will be able to read the physical characteristics of I/O objects (such as block size) so that they can recognize record type mismatchs.

5.1.3.3 Block I/O

PIOS will support block I/O. In this form of I/O information is transferred in chunks equal to the physical bock size. No formatting of data into records is done.

5.1.3.4 Unpended I/O

Unpended I/O will only be provided in the form of IPC primitives:

o Send - Send a message.

o Receive - Receive a message.

5.2 Ease of Use

PIOS will be "friendly". Interfaces will be simple, each command having one well-defined function. All objects will be manipulated using one interface. The user environment will be as transparent as possible; e. g., it should not be necessary for a user to do anything special to access a remote file rather than a local file. However, PIOS will provide facilities for users who need to operate on a "lower" level than most; system transparency does not imply inflexibility. Where restrictions must be imposed (e. g., the size of files or directories), they will be such that they are never encountered by most users. The extent of these changes is TBD.

5.3 Compatibility

PIOS must be able to run on all 50-series CPUs and memory configurations, and using all 50-series I/O controllers unchanged (unless a hardware design error is found which cannot be circumvented, in which case a field ECO may be required).

All user programs must continue to run, without modification, recompilation, or reloading.

All current Primos commands must continue to work as today.

No compatibility requirements are imposed on operator or system administrator interfaces; there should be a valid technical reason for any change made.

It is acceptable that Prime-supplied software which is not part of Primos be modified, recompiled or reloaded as part of the PIOS release. However, such requirements must be technically justified, clearly documented, and called to the attention of the affected groups.

5.4 Access Control

PIOS must in no way reduce the security and integrity of the system. The current ACL system will be extended to include devices and services. In general, all objects will be protected by ACLs.

It must not be possible for a user to compromise any Primos database. All permitted accesses to Primos databases will be done by procedure call or message transmission, never by direct memory references.

5.5 Error Reporting

All error messages will be informative and in plain English; terse messages, and unnecessary use of jargon will be avoided.

Timely reporting of asynchronous errors is dependent on the definition of a central error reporting facility (see "Dependencies"). Asynchronous errors may be reported by signaling a condition or by setting a status variable that a process can interrogate; determining which is better must be decided jointly with DSAG. Synchronous errors will be reported by setting a status variable.

5.6 System Startup and Configurability

It must be easy to load, boot, and configure the system; this requirement should aid the implementation of PIOS, as well as reduce administrative headaches in the field. Specifically, configuration must be simpler than today; it must be possible to bring resources on and off line dynamically. The number of coldstart parameters should be minimized in favor of dynamic allocation just after coldstart. It is desirable that many parameters (e.g. the number of page maps available) be variable during system operation.

The current use of physical device numbers will be eliminated in favor of a more friendly and flexible naming scheme.

5.7 Implementation

PIOS will be written in a suitable systems programming language, such as MODULA II. Even if such a language is not available when the implementation begins, the selected language will be used in the design specification of PIOS as a pseudo-code. PMA will be permitted inside the machine dependent kernel where absolutely required; this prohibition is especially important because of the anticipated arrival of the NSP machine. The use of other languages is prohibited.

PIOS will provide a completely new set of subroutine interfaces in the areas it covers. The detailed requirements on these interfaces are left to the functional specification of PIOS, but a number of general ones are given here.

1. Each interface should have a single purpose (e.g. "create file" but not "create, delete, or verify existence of file").

2. The datatype of each argument should not depend on the value of another argument. In cases where this function is needed (e.g. an argument of arbitrary type), a pointer to the actual data should be used.

3. The number of arguments for each interface should be minimized, but not at the cost of loss of function.

4. Argument datatypes requiring a descriptor (e.g. char(*) var) should be used only when required. Each such interface must have a companion writearound that can be called from a language that does not support descriptors, such as F77.

5. "Side effects" will be minimized or eliminated. Wherever possible programs will be isolated from the internal representation of system data structures; this means that access to system common areas will be rigidly controlled through special modules.

6. All gates (ring 0 interfaces) must work securely even if the value of one or more arguments changes while executing in the gate.

7. Machine dependent code will be minimized and isolated. This involves specifying a set of machine independent interfaces that the remainder of the code uses (machine dependent kernel).

8. To ensure the maintainability and extensibility of PIOS, modularity and structure will be emphasized in design and implementation. Agreed upon interfaces and protocols will be strictly adhered to. "Shortcuts" for efficiency will not be taken.

9. The code for PIOS will be "self-documenting" as much as possible; well-commented code will <u>not</u> be a substitute for coherent external documentation (specifically, functional and design specifications).

5.8 Extensibility and Portability

It must be possible to extend PIOS to include new functions or support new technologies (e. g., intelligent disk controllers, I/O processors) without rewriting or replacing large sections of code; ideally, we should be able to just "plug in" new code. The emphasis on structure and documentation in PIOS design should permit us to get close to this ideal.

To ensure that PIOS is not made obsolete by architectural advances, it must be possible to move PIOS to new CPU families with a minimum of effort. The identification and isolation of machine dependent code should allow this (see #7 under "Implementation").

The definition of a machine dependent kernel is especially important because of the proposed NSP machine.

5.9 Database Management

Determining the full requirements of the database management products is ongoing. Some conclusions have been reached; some issues involve other projects or have been deferred pending the results of the I/O performance study. Some requirements are addressed by items discussed elsewhere in this specification. Additional requirements are:

- o It must be possible for very large objects to span partitions. Subobjects are constrained to be on one partition.
- o It must be possible to restrict objects to single nodes.
- o It must be possible to tell when an object crosses a network boundary.
- o It must be possible to replace subobjects with other subobjects.
- o ROAM must be able to securely manage its objects using name space conventions and servers.
- o PIOS must provide a large number (TBD) of dynamically allocated file units.

The following items are not the responsibility of PIOS:

- o Management of ROAM objects is up to DM.

- o DM must provide the ROAM object portion of generic copy command.
- o ROAM will continue to do its own recovery.
- o Data management supports a kind of object (keyed access) not supported in language I/O. It must be possible for languages to read keyed access objects using their random and sequential interfaces. DM must provide support for this.

6 Comparison of PIOS with Competition

PIOS corrects the deficiencies enumerated in Section 3.10.

o Hierarchical name space

PIOS will expand the name space to include network nodes, devices, and services; links and search lists will be supported. Aliasing will be provided via the association mechanism for doing connections.

o Generic files

PIOS will provide generic objects, a more general feature than generic files. Generic objects allow the specification of access control and non-standard operations on the names defined.

o Security and access control

All objects will be protected using ACLs. Objects include file, devices, and services.

o Connection capabilities

PIOS will provide very general and flexible connection capabilities with its stream operations. Full device independence and I/O redirectability will be supported.

o One I/O interface for all objects

All types of objects will be referenced using one interface, namely the PIOS name space and stream operations.

o Record I/O

PIOS will support record I/O.

o Block I/O and unpended I/O

Block I/O is supported today. Unpended I/O will be done using IPC primitives.

7 Dependencies

PIOS has a number of strong dependencies on other groups which affect both its schedule and proposed functionality. In this section we list the dependencies and the groups affected. This list is in approximate priority order, starting with the most critical dependencies. However, be cautioned that the last items on the list are still quite important; the first items are critical.

1. Distributed System Architecture Group (DSAG)

   Importance: supercritical

   Risk: Cannot complete PIOS functional specification, prolonged delay of project design and implementation.

   PIOS's heaviest dependencies are here. DSAG is involved in for the design of three critical areas: the hierarchical name space, interprocess communication, and error reporting. PIOS cannot be fully specified without DSAG's proposals in these three areas. PIOS requires that DSAG produce a specification in these areas consistent with the goals and requirements in this document. Close consultation between DSAG and the PIOS team is required.

2. Data management support (Data management/Performance STD)

   Importance: critical

   Risk: Incomplete functional specification, possible inadequate design, performance problems.

   Data management supports a kind of object (keyed access) not supported in language I/O. It must be possible for languages to read keyed access objects using their random and sequential interfaces.

   In addition, it is a goal of PIOS to provide data management products with favorable functionality and performance perks. Defining these depends heavily on the data management group's aid in reviewing our proposals, and on the performance task force's success in determining where performance enhancements should be concentrated.

3. Language library support (Translators)

   Importance: critical

   Risk: PIOS can be implemented without this, but performance will not improve, and using the new features will be harder for users.

PIOS will strive to integrate its I/O operations with language I/O. This will eliminate the need for extra layers of software used to translate language constructs into Prime's I/O mishmash. However, eliminating the need and taking advantage of it are different. The language libraries will have to be modified to use the new PIOS interface.

4. Copy primitives (DM/OS)

   Importance: critical

   Risk: PIOS can be implemented without this but not fully utilized. Certain utilities, such as MAGSAV/MAGRST, will be affected.

   A generic copy command must be provided for objects. This requires the cooperation of the DM group, since they will be expected to provide a copy primitive for ROAM objects.

5. Standard terminal interface STI (Networks)

   Importance: Very important

   Risk: PIOS can be implemented and used without this, but both would be more difficult. Needed at least for second release of PIOS.

   Achieving the goal of full device independence and of representing any user keyboard as a single generic object implies the existence of an abstract terminal type. This kind of feature is being pursued by the STI project.

6. Ring 0 debugger (Tools group/STD)

   Importance: Very important

   Risk: Implementation will take much longer and the resulting product will be less reliable.

   There has long been a need for a high level debugger (comparable to DBG) which can be used to debug ring 0 code. This is a difficult problem because Prime's hardware protection prevents the direct use of standard debugging techniques (most especially breakpointing) except under special circumstances. There is also the problem that debugging an operating system on line implies the existence of a surrogate operating system which can run things when the experimental software breaks. Despite these difficulties, two solutions have been proposed, one involving a two machine system (possibly a P850), the other an extension of the Microprocessor Debugging System (MDS). Given the unusual complexity of the PIOS project (and later the full PDA effort), the availability of a high level ring 0 debugger is essential to its timely completion.

7. Systems programming language (Translators)

   Importance: Very important

   Risk: Same as for #6.

   Like the ring 0 debugger, the need for a powerful, flexible systems implementation language has been long felt. Possible choices include MODULA II and SPL.

8. Secure data paths (Networks)

   Importance: Very important

   Risk: A distributed system ==> more network traffic ==> more security problems ==> more unhappy customers. PIOS can be implemented and used without this.

   PIOS will do nothing to reduce the security of Primos and will attempt to improve it. However, a distributed computing environment implies heavy network traffic. Currently there is no way to ensure that inter-machine transmissions are secure. If PDA is to be a secure system, this problem must be solved.

9. System boot(OS)

   Importance: Important

   Risk: Same as #6.

   The abilities to boot the system in a single step, and to do partial system builds and loads are very desirable and affect the schedule.

8 Non-requirements

The following are not the responsibility of PIOS:

- The design of a full IPC mechanism to provide complete I/O asynchrony. This is the responsibility of DSAG.
- User devices. PIOS does not propose to simplify the addition of new devices.
- Keyed access to files - DM must do it.
- IEEE floating point - The problem here is files having both IEEE floating point and Prime floating point. Recognizing this fact requires multi-typed files. PIOS will not support this.
- Conversion from ASCII 7 to ASCII 8.
- Logging attempted security violations.

9 Issues

At this stage of the PIOS project, some proposed design features are still controversial or require more thought. In this section we list the current issues and their status. It is expected that eventually this section will be deleted, and all issues will become either requirements or non-requirements.

1. Extent of DBMS performance perks - This is still being discussed. Some areas are deferred until the recommendations of the performance task force are available.

2. Compressed records - The question here is how to handle blank compression. This is deferred until the recommendations of the performance task force are available.

3. Magtape labels - The question here is what kind of label should be used in device independent magtape I/O. A Marketing recommendation is needed here.

4. General lock management and support for atomic operations. Level of support for transactions. Discussions with DM are ongoing.

5. Version control to support hard/soft crash recovery.

6. Phasing out password directories.

7. Phasing out old commands.

8. Features and design of the physical file system. This is part of the PIOS project, but full specification must wait for the recommendations of the performance task force.

9. Reliability, Availability, and Servicability - The specification of specific RAS requirements is deferred pending the completion of the I/O performance study.

10. Performance perks - The recommendations of the I/O performance task force are needed to specify these.