Proposal for a Common Parallel File System Programming Interface Version 1.0

Peter F. Corbett, Jean-Pierre Prost
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Chris Demetriou, Garth Gibson, Erik Riedel, Jim Zelenka
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Yuqun Chen, Ed Felten, Kai Li
Department of Computer Science
Princeton University
Princeton, NJ 08544

John Hartman, Larry Peterson
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Brian Bershad, Alec Wolman
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

Ruth Aydt
Pablo Research Group
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

This document proposes an interface to parallel file systems intended for use with a variety of parallel computers. This proposal is based on the separation of programmer convenience functions from high-performance enabling functions. We propose that the former be supported above this interface, possibly in client libraries. The latter, functions that enable high performance, are defined by this proposed API under the assumption that these functions are more likely to need system and vendor-specific support.

Specifically, this proposal includes functions which support reading and writing with scatter-gather addressing for memory and file ranges, and asynchronous operations. It also includes mechanisms that permit client control over client caching, and file access and layout hints. Finally, it includes a mechanism by which this API can be extended and extensions for fast file copy and batching collective I/O operations.

Contents

Context

This proposal is being developed by the Scalable I/O Initiative (SIO), a consortium of universities, national laboratories, and industries studying parallel and scalable I/O systems for large parallel computers. This proposal is not a commitment on the part of any member of SIO to support these interfaces. However, it is intended that within the SIO effort several implementations of parallel file systems compliant with this interface be produced on several different platforms. We do not expect these interfaces to be finalized until implementation and user experience are obtained. SIO will foster such implementation and application development experience. The ultimate goal of this effort to produce a common parallel file system interface is two-fold: to support research in the area of parallel I/O, and to eventually recommend additions of parallel I/O interfaces to the χ/Open and POSIX standards.

This document contains a basic API plus several extensions. Sections 3 through 14 in this document contain the basic API, which all conforming implementations must implement. Sections 15 and 16 contain extensions to the API which may optionally be provided by implementations.

Within the SIO research community, proposals (and counterproposals) for future modifications to this API are journalled in a separate document called "Proposal for a Common Parallel File System Programming Interface; Part II: What's in Progress."

Perhaps unavoidably, this document is more about the description of interfaces than it is about their rationalizations. We apologize in advance for your many unanswered questions.

1 Introduction

The intent of the interfaces presented here is to add to the standard χ/Open XPG 4.2 interfaces, which were earlier defined in IEEE Standard 1003.1 (POSIX). It is widely recognized by vendors of distributed memory parallel computers and workstation clusters, such as IBM and Intel, that extensions to the χ/Open XPG 4.2 and POSIX interfaces to support high performance file I/O for parallel applications are desirable. However, there is little agreement about what these extensions should be. This results in part from vendor extensions that exclusively emphasize the capabilities of a specific machine or application class. As a result, it is not currently possible for programmers to write application programs using extended file system interfaces that are portable from one parallel computer to another.

Clearly, there is a need for a new set of standard interfaces, preferably a set of extensions to the χ/Open XPG 4.2 interfaces, if we wish users and third party software vendors to use the extended features of parallel file systems. The SIO community has chosen to divide the file system interface into two levels: a *low-level* interface which hides machine-dependent details and contains only those features needed to provide good performance, and a *high-level* interface which provides features for programmer convenience and to support particular application classes.[1] This document describes only the low-level interface.

There are portions of this API which provide functionality that is redundant with the function provided in the χ/Open interfaces. This is to enable some SIO members to develop complete experimental file systems with just this API, without the added burden of implementing a complete χ/Open compliant file systems interface. In the cases of redundant interfaces, the SIO functions can simply be implemented as wrappers over the standard functions. However, these functions should be implemented in such a way as to ensure that all libraries written to this API can run properly.

Our two-level approach arises from the conflicting goals of some aspects of different extended interfaces. For example, in a discussion of the commonalities between IBM's PIOFS and Intel's PFS in February 1995, we identified little more than the basic UNIX functions in common. Largely this is because IBM had chosen to support the concept of dynamic partitioning and subfiles, while Intel supported a set of file modes to define the semantics of parallel access. Our two-level approach moves the implementation of the special character of these parallel file systems (Intel I/O modes or IBM subfiles) to high-level libraries and proposes a low-level interface capable of efficiently supporting both of these and other specialized parallel file system function sets. The approach follows CMU's December 1994 suggestion, in that the new interfaces are low level, but are powerful for implementing high-level parallel I/O libraries.

[1] MPI-IO is an example of such a high-level interface.

The usage scenario is that I/O libraries can be easily and efficiently built on top of the interfaces provided by this API. Each vendor is free to implement whatever libraries they wish on top of these interfaces. Likely libraries include MPI-IO, a PIOFS subfile library, and a library which supports Intel's I/O modes.[2] It is simpler to implement or share a library at this level than to implement the function in the vendor-specific file system itself. Also, third party vendors (or groups such as SIO) can produce libraries that could compile and run on another vendor's machine. In addition, these interfaces could be a compiler target.

Code written to this low-level API is intended to be portable. By this we mean source compatibility. In particular, each implementation of this API is free to assign different bit lengths to most types and different bit values to all constants, except as noted. Because the size of fields is implementation dependent, the range of some variables may also vary. In some cases this may limit source compatibility, so we have tried to require comfortably large limits wherever possible.

1.1 Independent Messaging and Minimal Synchronization

One view of a parallel application is of a set of tasks, typically executing on different nodes, communicating among themselves, possibly via shared memory. There are a variety of abstractions, toolkits, and mechanisms for communicating from which a particular parallel application may choose. One principle of this low-level API is to avoid dependence on the application's

[2] We do not intend to prescribe the software structure of an implementation of PIOFS or PFS built with this API. Our expectation is that implementations will be efficient enough to allow libraries built entirely on the interfaces in this API to obtain high performance. For example, an application coded for an SIO-based Intel I/O-mode library should run efficiently on an IBM SP2 offering these interfaces. Of course, when this application runs on a Paragon, it is not required to use the I/O-mode library in favor of the native PFS interfaces.

chosen method for communication. This means that a low-level parallel file system client implementation may not be aware of application-level messages and certainly cannot expect to use the same methods for communicating with its peer client agents. Of course, each client agent of the low-level parallel file system must be able to communicate with the parallel file system servers (if any). The method of this communication is implementation specific and will most likely be unavailable to the application programmer.

Another guiding principle in the design of this API is to discourage unnecessary synchronization of the client applications or of the client agents of the parallel file system. To this end, this API is designed to admit efficient low-level parallel file system implementations which restrict internal communication to a single client and the parallel file system server(s) responsible for a particular file. That is, this API does not require that client agents of the parallel file system directly communicate. This means that a compliant parallel file system implementation need not provide coherent distributed shared memory, shared file pointer synchronization, or collective I/O barrier synchronization. As described below, distributed shared memory may be avoided with application-managed weakly consistent caches and collective I/O barrier synchronization can be made implicit by requiring the application to distribute an opaque collective I/O handle defined by the parallel file system.

1.2 No Shared File Pointers

One of the original points of disagreement in the development of the API was support for shared file pointers. Some parallel file systems exploit shared file pointers extensively while others avoid this implicit synchronization as much as possible. The position of this API is similar to the latter: that shared file pointers can require extensive synchronization of the client agents of the parallel file system; that they implicitly synchronize the application's tasks; and that they can easily lead to excessive synchronization, slowing the application. Further, we contend that if this level of application synchronization is valuable, it should be provided by the higher level parallel file system libraries which may have access to peer-to-peer messaging systems and can be customized to specific applications' needs. For these reasons this API does not support shared file pointers; in fact, it does not support file pointers at all, requiring the offsets for all I/O operations to be explicitly provided.

1.3 Scatter-Gather Transfers

Batching transfers is a powerful strategy for improving performance. A parallel file system implementation can be expected to try to batch accesses to the disk, transfers between machine nodes, and buffer manipulations. Traditional UNIX read-write interfaces transfer contiguous file regions and contiguous memory regions, dramatically reducing batching opportunities for applications that manipulate large, non-contiguous data regions. Correspondingly, a principle extension for high-performance file systems is the compact representation of transfers of non-contiguous regions, commonly known as scatter-gather. In the core of this API proposal, the expressive power of scatter-gather is limited to a list of strided (vector) regions. [3]

1.4 Asynchronous I/O

The API provides interfaces for asynchronous reads and writes. Outstanding accesses can be polled or waited upon (either singly or as a list of accesses).

1.5 I/O Controls

This API allows applications to get and set file status data (such as file sizes), get and set performance-related information (such as file caching and layout), and perform various operations (such as cache consistency) via a general I/O control mechanism. Vendors can define their own control operations, allowing the API to be extended easily.

Some controls, notably data layout and capacity preallocation controls, may be performed much more efficiently as a group and/or at the time a file is created or opened. For this reason, multiple controls may be specified in the

[3] Beyond this proposal, some SIO researchers have shown an interest in nested lists of strided regions.

same operation, and the extended open interface in this API allows a set of controls to be executed when a file is opened. Because of the large amount of work that might be done by a set of controls, the API allows failure of I/O controls to fail the overall open or control operation immediately, and allows implementations to declare that certain controls may not be issued as a part of the same operation.

1.6 Client Caching

Because parallel files will experience concurrent read-write sharing, maintaining client cache consistency could become quite expensive. An implementation of this API may provide no client caching (for example, in some parallel systems the latency for fetching a file block from a server's cache may be low enough to not warrant client file caches). It may also provide strong consistency using shared memory mechanisms. However, many parallel applications will synchronize concurrent sharing at a higher level and can explicitly determine when to propagate written data from their local caches and when to refresh stale data from their local caches. This API enables these applications to improve their client cache performance by requesting weak consistency on a particular open file and to issue the appropriate propagate and refresh controls. In the case of weak consistency, an implementation may divide the file address space into fixed sized consistency units (cache lines or blocks) which are entirely present in a client cache if at all. Concurrent write sharing of a weakly consistent file within one consistency unit is not guaranteed to have reasonable semantics.

Note that this API makes no requirement that a low-level parallel file system implementation control or even detect unintentional read-write sharing, that is, read-write sharing by tasks that are parts of multiple uncoordinated parallel applications. In situations like this, which are common to many file systems, the atomicity of file creation can be used by higher level tools to provide simple advisory locks by using the existence of a file to signify a held lock.

1.7 File Access Pattern Hints

Allowing an application to provide hints about file accesses can substantially improve performance, particularly when a large amount of data is read non-sequentially (but predictably), or when a large number of small files are read one at a time. There are at least two distinct approaches to giving hints: explicitly listing an ordered sequence of future accesses (such as “read block 5, then block 7), and describing an access pattern with a single identifier (such as “random access,”“sequential access,” or “will not access”). Because it is not clear how to interpret a set of hints that intermingle these approaches, this API provides separate hint classes for each, does not specify how to interpret combinations containing both, and allows vendors to add new classes of hints as needed. To allow applications to provide information to the file system as early as possible, hints can be applied to open file descriptors or to files that have not yet been opened. In either case, hints apply only to the task that issued them, and not other tasks.

1.8 Extensions to this API

In discussing earlier low-level API proposals, we found that there are some features that are almost universally agreed upon, and a few features that have significant constituencies but were not supported by all members of the group. We thus chose to define the low-level API as a *basic API* plus a set of optional *extensions*. An extension is a feature that:

- has significant research value;
- impacts performance, at least on some architectures; and
- is not trivial to implement correctly;

As a part of the basic API, implementations must provide mechanisms for allowing applications to determine which extensions are supported. Those mechanisms are detailed in Section 14.

1.9 Collective I/O

As mentioned in Section 1.3, batching is a powerful mechanism for improving performance. When multiple client nodes access one file at the same time, batching can again be useful, particularly when each client's access is a complex pattern but the sum of all client accesses is a large contiguous access (e.g. the whole file). Accesses of this type are known as "collective I/O," and this API includes an extension which provides collective I/O facilities.

Current collective I/O mechanisms commonly exploit the implementation system's task identifiers or task groups to name the members of a collective I/O. In this API we avoid dependence on the systems' task naming mechanisms by dynamically defining an opaque identifier for a collective I/O that is distributed via the application's communication system and presented to the parallel file system by each participant (client involved in the collective I/O). With this mechanism we enable at least three types of batching. First, the parallel file system implementation may choose to wait for all participants to join the collective I/O before doing any of the work. Second, the application can provide a hint describing the total work to be done by the collective I/O at the time the collective I/O is defined. Third, a collective I/O may be defined to have multiple iterations, avoiding multiple defining operations and enabling earlier collective hints.

1.10 Checkpoints and File Versioning

Many parallel applications want the ability to create checkpoints of their files. Others want the ability to efficiently create a series of versions of a file over time. Rather than directly supporting checkpoints or file versions, this API includes an extension which offers a generic "fast copy" operation. A fast copy might be implemented as duplication of a file's metadata, with shared pointers to all data pages, each of which is marked copy-on-write. The tracking of copies is left up to the applications (or higher level parallel file system libraries).

1.11 File Names and Access Protection

When these interfaces are merged with POSIX it is expected that POSIX conventions will be adopted for directories and access control. However, during SIO research, compliant implementations need not deal with these (important) issues.

This API does not define directories or directory operations. Files may be named in a flat name space, though implementations may choose to offer additional name space management. A directory structure is not viewed as essential to parallel file system performance and can be provided by vendor-defined extensions as needed.

Similarly, access control checking, permission specifications, and user and group identifiers are not specified by this API. Implementations which provide access control management are expected to do so via vendor-defined extensions.

1.12 File Labels

An important issue for higher level library systems and application systems is interoperability. To support interoperability without inserting header data into the file's actual data, the low-level API was offers a small amount of application controlled data called a *label* for each file. A file's label is stored in its metadata.

2 Document Conventions

This document describes both the "basic" (or "core") API and extensions to the basic API. The basic API is described in Sections 3 through 14, and the extensions are described in Sections 15 and 16. Some sections of this document refer to "this document," which is meant to indicate the entirety of the basic API and the extensions described herein.

Implementations wishing to conform to this API must provide all of the types, definitions, and functions specified in the basic API, including those necessary to determine whether or not extensions are present.

2.1 Typesetting Conventions

Type definitions, functions definitions, and constants (including control operation identifiers) are typeset in the **bold** font.

Function names are typeset in the **bold** font and are followed by parentheses, e.g. **sio_open()**.

Variables, structure members, and function arguments are typeset in the *italic* font.

2.2 Definition of Terms

Throughout this document (except where explicitly noted) the phrase "file system" is used to indicate a file system which provides this API, and "implementation" is used to refer to the implementation of such a file system. Except where noted, the terms "application" and "higher-level library" are used interchangeably, and are meant to indicate the programs or libraries which are using this API to access parallel files.

Throughout this document, several words or phrases are used to indicate how given functionality must be used or implemented. For clarity, they are defined here:

"will," "shall," or "must"

When describing functionality provided by file system implementations, these terms indicate that conforming implementations have to implement the functionality as described.

When describing behavior of applications, these terms indicate the behavior of properly-written applications (i.e. applications behaving in other ways are considered buggy).

"should"

When describing functionality provided by file system implementations, this term suggests that an implementation provide the functionality in the manner described, but that doing so is not necessary for conformance.

When describing behavior of applications, this term indicates that the described behavior is the preferred behavior, but that other behavior may be correct.

"may"

When describing functionality provided by file system implementations, this term indicates that conforming implementations can implement functionality in the manner described, but doing so may not be suggested.

When describing behavior of applications, this term indicates that the described behavior is allowed, but not necessarily encouraged.

"undefined"

Undefined behavior is not specified by this standard, and is usually a result of a programming error or similar problem. Applications must avoid invoking undefined behavior. File system implementations may produce completely arbitrary results when undefined behavior is invoked, including producing random data, on disk or in memory buffers provided, or generating an exception.

"unspecified"

Unspecified behavior is not specified by this standard, but is usually the result of a correct programming practice. Behavior is left unspecified to give file system implementations freedom to implement functionality in different ways. Unspecified behavior must not have harmful permanent effects on the application or its data, and should be documented in individual implementations' documentation. Portable applications must not rely on unspecified behavior causing the same results on multiple file system implementations.

2.3 How to Read this Document

It is recommended that you read sections 6,8,9,10,11,12, and 13 before sections 3,4, and 5. The reason for this is that sections 3,4, and 5 provide definitions which refer to functions explained in later sections.

3 The sio_fs.h Include File

File system implementations must provide a C include file named **sio_fs.h** which contains the data type definitions, constants, and function declarations and/or prototypes for all functions defined in this document. Implementations which provide extensions not defined in this document may require additional files be included to use those extensions. Implementations which do so must still define the extension support constants and extension identifiers (see Section 14.1) for the extensions in **sio_fs.h**.

Applications or higher-level libraries must include **sio_fs.h** in their source files before referencing any of the types, constants, or functions described in this API.

4 Data Types

This section defines the data types which are referenced in the basic API, and gives brief explanations of the rationale behind them. Types used exclusively by extensions are not defined here–they are defined with the extensions.

All of the types defined in this section must be provided by conforming implementations. Vendors may provide additional types with names of the form **sio_vend_***vendordefinedname***_t**, where *vendordefinedname* can be a name of the vendor's choosing. All other type names beginning with **sio_** and ending with **_t** are reserved for future use by this API.

Except where otherwise noted, the sizes of all non-structure data types are fixed on a per-implementation basis and those data types must be fully copyable (i.e. they must not contain any pointers to other objects).

4.1 File Descriptor

All file descriptors are described as being of type **int**, primarily for compatibility with other systems (including UNIX) which use **int**s as file descriptors. A task may have up to **SIO_MAX_OPEN** parallel files open at any given time.

4.2 File Name

All file names are character strings terminated by a byte with the value zero, and are described being of type **const char ***. (They must never be modified by the system, and thus are **const**.) File names must not be longer than **SIO_MAX_NAME_LEN** characters, including the terminating zero byte.

4.3 Memory Address

Memory addresses are described as being of type **void ***. Each task must only access its own or a shared address space. Attempting to access memory for which the task does not have access permission produces undefined results.

4.4 sio_async_flags_t

This is an unsigned integral type used as a set of bits. Currently it can contain one of **SIO_ASYNC_BLOCKING** or **SIO_ASYNC_NONBLOCKING**. These flags indicate whether or not **sio_async_status_any()** will block waiting for an asynchronous I/O to complete. The use of these flags is described in Section 10.2.

4.5 sio_async_handle_t

This is an opaque type used to identify asynchronous I/Os.

4.6 sio_async_status_t

```
typedef struct {
      sio_transfer_len_t count;
      sio_return_t status;
} sio_async_status_t;
```

This structure is used to return the status of an asynchronous I/O. For a successful operation, *count* is set to the number of bytes transferred, and *status* is set to **SIO_SUCCESS**. For an unsuccessful operation, *status* is set to a value which indicates the nature of the error, and count is set to the number of bytes guaranteed to have been transferred correctly (see Section 10.2).

4.7 sio_caching_mode_t

This is an unsigned integral type used by the client cache control interfaces, and is defined in Section 12.

4.8 sio_control_t

```
typedef struct {
    sio_control_flags_t flags;
    sio_control_op_t op_code;
    void *op_data;
    sio_return_t result;
} sio_control_t;
```

This type is used to store the information associated with a control operation (see Section 13). Control operations are specified by providing the appropriate operation code in *op_code*, an indication in *flags* of what to do if the control cannot be performed, and a pointer to a data buffer (if necessary) in *op_data*.

The *result* field is set by the function performing the control operation to indicate success or failure.

4.9 sio_control_flags_t

This is an unsigned integral type used as a set of bits. Currently it can contain one of **SIO_CONTROL_MANDATORY** or **SIO_CONTROL_OPTIONAL**. These flags indicate whether failure of this control operation will cause the entire set of control operations to fail, with semantics as described in Section 8.1.

4.10 sio_control_op_t

This is an unsigned integral type used to indicate a control operation code. Control operations codes which are part of the basic API are defined in Section 13.

4.11 sio_count_t

This is an unsigned integral type with the range [0... **SIO_MAX_COUNT**]. It is used to represent a quantity of objects.

4.12 sio_extension_id_t

This is an unsigned integral type used to contain extension identifiers. See Section 14.1.2 for more details about its use.

4.13 sio_file_io_list_t

```
typedef struct {
    sio_offset_t offset;
    sio_size_t size;
    sio_size_t stride;
    sio_count_t element_cnt;
} sio_file_io_list_t;
```

This structure is used to describe a collection of regions within a file that is involved in a parallel file system operation. Its purpose is to encapsulate the description of many simple transfers into one larger and more complex transfer to enable the file system to be more efficient in the execution of the total transfer. Each **sio_file_io_list_t** structure describes a sequence of equally-sized and evenly-spaced contiguous byte regions within a file; this is sometimes called a "strided" access pattern. Common matrix decompositions can be described with such data structures.

The structure describes a set of *element_cnt* contiguous regions, each of size *size*, with the first region beginning at offset *offset* from the beginning of the file, and the beginning of each subsequent region starting *stride* bytes after the start of its predecessor. These contiguous byte regions may overlap; see Section 9 for details.

4.14 sio_hint_t

```
typedef struct {
    sio_hint_flags_t flag;
    sio_file_io_list_t *io_list;
    sio_count_t list_len;
    void *arg;
    sio_size_t arg_len;
} sio_hint_t;
```

This structure is used to store hint information (see Section 11). The *flag* field describes the access patterns being hinted, and the *io_list* and *list_len* fields describe the regions of the file to which the hint applies. The *arg* and *arg_len* fields contain a pointer to a hint-specific argument and the (non-negative) length of the argument, respectively. These fields allow different types of hints to require different types of arguments, while using the same hint interfaces.

4.15 sio_hint_class_t

This is an unsigned integral type which contains the class identifier of hints passed with the **sio_hint()** and **sio_hint_by_name()** functions. Each class of hints contains one or more hint types whose interaction is specified. Interactions between hint types of different classes are unspecified. This document defines the **SIO_HINT_CLASS_ORDERED**

and **SIO_HINT_CLASS_UNORDERED** constants to describe mandatory hint classes, and reserves constants whose names begin with with **SIO_HINT_CLASS_VEND_** for use by vendors. See Section 11 for more details about hints and hint classes.

4.16 sio_hint_flags_t

This is an unsigned integral type used as a set of bits. It is used to describe the hint information stored in a **sio_hint_t**. See Section 11 for a list of possible values for this type and explanations of their use.

4.17 sio_label_t

```
typedef struct {
    sio_size_t size;
    void *data;
} sio_label_t;
```

This type is used to store a file label, which can contain application-managed descriptive information about its associated file. The *data* field points to a memory buffer *size* bytes long. The **SIO_CTL_GetLabel** and **SIO_CTL_SetLabel** control operations use this structure in different manners; see Section 13.9 for more information about this structure's use.

4.18 sio_layout_t

```
typedef struct {
    sio_layout_flags_t flags;
    sio_count_t stripe_width;
    sio_size_t stripe_depth;
    sio_layout_algorithm_t algorithm;
    void * algorithm_data;
```

```
} sio_layout_t;
```

The number of parallel storage devices over which the file's data are striped is contained in the *stripe_width* field, while the (non-negative) number of contiguous bytes stored on each device (the unit of striping) is contained in *stripe_depth*. The *stripe_width* does not include any devices containing redundancy information, such as ECC codes or duplicate copies of the data. The *algorithm* field indicates the style of layout used for the file to provide guidance in the interpretation of the *stripe_width* and *stripe_depth* fields. The *algorithm_data* field is used to store algorithm-specific information about the layout.

The *flags* field indicates which portions of the **sio_layout_t** structure are being provided to the system or should be filled in by the system as described in Section 13.8.

4.19 sio_layout_algorithm_t

This is an unsigned integral type whose value indicates the style of layout used for an SIO file. The layout algorithm describing a simple round-robin striping across all storage devices used for a file is **SIO_LAYOUT_ALGORITHM_SIMPLE_STRIPING**. This must be defined, though not necessarily supported, by all implementations. Implementations may choose to support additional layout algorithms that describe layouts in more detail or provide for more complex storage system architectures. The *algorithm_data* field in the **sio_layout_t** structure can be used to store additional information about the layout algorithm.

Layout algorithm names beginning with **SIO_LAYOUT_ALGORITHM_VEND_** are reserved for use by vendors. All other names beginning with **SIO_LAYOUT_ALGORITHM_** are reserved for future use by this API.

4.20 sio_layout_flags_t

This is an unsigned integral type used as a set of bits. It may contain zero or more of **SIO_LAYOUT_WIDTH**, **SIO_LAYOUT_DEPTH**, or **SIO_LAYOUT_ALGORITHM**, bitwise ORed to specify the fields of an **sio_layout_t** structure are to be returned or set.

4.21 sio_mem_io_list_t

```
typedef struct {
    void *addr;
    sio_size_t size;
    sio_size_t stride;
    sio_count_t element_cnt;
} sio_mem_io_list_t;
```

This type is similar to **sio_file_io_list_t** except that it describes a collection of regions within one memory space that is involved in a parallel file system operation, rather than a collection of file regions. Its purpose is to encapsulate the description of many simple transfers into one larger and more complex transfer in order to enable the file system to be more efficient in the execution of the total transfer. Each **sio_mem_io_list_t** structure describes a sequence of equally-sized and evenly-spaced contiguous byte regions within the memory space.

The structure describes a set of *element_cnt* contiguous regions, each of size *size*, with the first region beginning at address *addr*, and the beginning of each subsequent region starting *stride* bytes after the start of its predecessor. These contiguous byte regions may overlap; see Section 9 for details.

4.22 sio_mode_t

This is an unsigned integral type used as a set of bits to specify the mode of a file operation. For example, the mode flags **SIO_MODE_READ** and

SIO_MODE_WRITE can be specified together or separately to open the file for reading and/or writing, or to indicate what operation is being hinted. Other flags are documented in Section 8.2.

4.23 sio_offset_t

This is a signed integral type whose absolute value is in the range [0... **SIO_MAX_OFFSET**].[4] This type is signed to allow an offset variable to be decremented in a loop, and have the loop terminate when the variable becomes negative.

4.24 sio_return_t

This is an unsigned integral type used by functions in this API to return a result code.[5] The constant **SIO_SUCCESS**, whose value must be 0, denotes success.

Other values indicate specific errors which have been encountered in processing this request (the enumeration of standard error codes is included in Appendix A). Error code names beginning with **SIO_ERR_VEND_** may be used by vendors for vendor-specific error codes. All other error code names, beginning with **SIO_ERR** are reserved for future use by this API. At least 16384 error codes (including 0, for **SIO_SUCCESS**) must be available for use by this API.

4.25 sio_size_t

This type is used to describe sizes of file and memory regions. It is a signed integral type whose absolute value is in the range [0... **SIO_MAX_SIZE**]. It is signed to allow expression of reverse strides for operations such as **sio_sg_read**().

[4] We do not take advantage of the defined behavior of C, which allows the effect of negative signed numbers to be achieved by using large unsigned numbers that are congruent modulo 2^n. $2^{63}-1$ is a sufficiently large offset that the extra factor of 2 possible by using unsigned offsets is not expected to be important before machines with 128 bit word sizes become widely used for high performance computing.

[5] An earlier version of this document used UNIX-style returns, where 0 indicated success, and -1 indicated failure, with specific UNIX error codes being set in the global error register. This was deemed inappropriate for two reasons. One is that the values of UNIX error numbers vary from platform to platform, as does the specific list of errors available. Another more serious problem is that it is difficult for multi-threaded applications to express different errors to different callers using a single global error register. Some systems, such as **pthreads**, provide a thread-specific error register for this reason. This was also deemed unacceptable, because it would require the parallel file system to be aware of the threading mechanism.

4.26 sio_transfer_len_t

This is an unsigned integral type in the range [0... **SIO_MAX_TRANSFER_LEN**]. It is used to count the total number of bytes transferred in I/O operations. This type differs from **sio_size_t** in that a single I/O operation may transfer many buffers whose length is represented by **sio_size_t**, hence **sio_transfer_len_t** is needed.

5 Range Constants

This section describes the constants used in this basic API to specify the ranges of data types. These constants are implementation-specific. However, for each of them, both a minimum value and a recommended value are given.

5.1 SIO_MAX_ASYNC_OUTSTANDING

This constant specifies the maximum number of outstanding asynchronous I/O requests that one task can have at one time. The minimum value is 1, and the recommended value is 512.

5.2 SIO_MAX_COUNT

This constant specifies the maximum number of items that can be defined by an **sio_count_t**. The minimum value is $2^{16}-1$, and the recommended value is $2^{32}-1$.

5.3 SIO_MAX_LABEL_LEN

This constant specifies the maximum length of a file label. The minimum value is **SIO_MAX_NAME_LEN** (whose minimum value is 256 bytes). The recommended value is the maximum of 1024 and the implementation's value of **SIO_MAX_NAME_LEN**.

5.4 SIO_MAX_NAME_LEN

This constant specifies the maximum length of a file name. The minimum value is 256, and the recommended value is 1024.

5.5 SIO_MAX_OFFSET

This constant specifies the maximum value for a file offset. The minimum value is $2^{63}-1$, and the recommended value is $2^{63}-1$.

5.6 SIO_MAX_OPEN

This constant specifies the maximum number of open files that a task can have at one time. The minimum value is 256, and the recommended value is 512. Note that a task may still fail to open a file before reaching this number because of system resource exhaustion.

5.7 SIO_MAX_SIZE

This constant specifies the maximum size in bytes of a variety of objects in the API. The minimum value is $2^{31}-1$, and the recommended value is $2^{63}-1$.

5.8 SIO_MAX_TRANSFER_LEN

This constant specifies the maximum number of bytes that can be transferred by a single I/O operation. The minimum value is **SIO_MAX_SIZE**, and the recommended value is $2^{63}-1$. Since several components of a scatter-gather I/O list can be transferred at once, **SIO_MAX_TRANSFER_LEN** must be greater than or equal to **SIO_MAX_SIZE**.

6 File Attributes

This section describes the attributes associated with an SIO file. The file attributes are unique to each SIO file and visible to all tasks opening the file. These attributes include the logical, physical, and preallocation sizes of the file, file label, and file layout information. Extended controls may define additional file attributes.

6.1 File Sizes

The **logical size** of an SIO file is the number of bytes from the beginning of the file (offset zero) to the end of the file (the largest offset from which data can be read successfully). The file may contain regions which have not yet been written (referred to as "holes"), which are read as zeros. The logical size can be increased or decreased with the control operation **SIO_CTL_SetSize** (see Section 13). Decreasing the logical size via **SIO_CTL_SetSize** corresponds to truncating the file, and increasing it creates a hole extending from the previous end of file to the new end of file. A file's logical size can also be increased by writing data past the current end of file.

The **physical size** of an SIO file is the amount of physical storage in bytes allocated to store the file data (excluding metadata). It may be different from the logical size of the file because of fixed size allocation blocks and because each implementation has the freedom to store data in any appropriate manner, including not storing the content of holes and the use of data compression techniques. The user has no direct control over the file's physical size.

The **preallocation size** of an open SIO file is the minimum logical size to which the file system guarantees the file may grow without running out of space. When a file is opened (created), its preallocation size defaults to its physical size (zero) unless a **SIO_CTL_SetPreallocation** control operation (see Section 13) is specified in the **sio_open()** call. Preallocation size is not affected by any operation defined by this API other than

SIO_CTL_SetPreallocation control operation and **sio_close()**.

6.2 File Label

The **file label** of an SIO file is a part of the file's metadata that is accessible to the user for storing descriptive information about the file without keeping a header in the file itself. Labels are intended to support interoperability by associating information about a file's representation (including file type, version, writing application, etc) with the file itself. Labels are not necessarily the same length in all implementations, but must always be long enough to record a maximum length file name for that implementation. This allows representation information too large to fit in a file label to be stored in a separate file named in the file label. The size of a label is given in the **sio_label_t** containing the label. This size is at least as large as an implementation's longest name which must be at least 256 bytes. The maximum size of a label in any specific implementation is given by the constant **SIO_MAX_LABEL_LENGTH** and is recommended to be at least 1024 bytes.

6.3 File Layout

The file layout of an SIO file expresses the placement of the file bytes on the parallel storage devices. Some implementations may allow the user to specify the file layout when the file is created with the **SIO_CTL_SetLayout** control operation. Other implementations may allow the user to query the file layout parameters with the **SIO_CTL_GetLayout** control operation, but not to set the layout. Still others may choose not to reveal anything about the underlying file layout and will support neither of the layout control operations.

A given file layout consists of the number of parallel storage devices over which the file data are striped, the number of contiguous bytes constituting each striping unit, and the algorithm which specifies the striping pattern of the striping units. For example, a simple striping pattern on four storage devices using a striping unit of 1024 bytes would look like the following (the starting byte number of each striping unit is shown):

Storage Unit 0	Storage Unit 1	Storage Unit 2	Storage Unit 3
0	1024	2048	3072
4096	5120	6144	7168
8192	9216	10240	11264
12288	13312	14336	15360
16384	17408	18432	19456
20480	21504	22528	23552
24576	25600	26624	27648
28672	29696	30720	31744
⋮	⋮	⋮	⋮

Note to implementor: The underlying implementation may employ advanced redundancy encodings or dynamic data representation (compressed and uncompressed or mirrored and parity protected). In cases like these, these layout parameters may be insufficient. In these cases the width of a stripe should be interpreted as the parallelism of accesses of at most an aligned striping unit.

7 Error Reporting

To make it easier for applications to deal with SIO error codes, the function **sio_error_string()** is provided. This function takes a **sio_return_t** value and returns a **const char ***. The sio_error_string function maps error codes to meaningful error strings. When passed an error code that is not defined by the implementation, **sio_error_string()** must return a string indicating the error number and noting that the error code is unrecognized.

7.1 sio_error_string

Purpose

Translate a **sio_return_t** into a string.

Syntax

#include <sio_fs.h>

const char *sio_error_string(sio_return_t *Result***);**

Parameters

Result The return code to translate.

Description

This function translates a return code to a string. The string pointed to must not be modified by the program, and may be overwritten by subsequent calls to **sio_error_string()**. If the implementation supports NLS (the suite of internationalization functions mandated by χ/Open XPG 4.2), the contents of the returned error message string should be determined by the setting of the **LC_MESSAGES** category in the locale.

8 Basic Operations

This section defines the basic operations that can be performed on parallel files. Interfaces are provided to open and close parallel files, to remove files from a parallel file system, and to perform control operations on parallel files.

This section defines some operations that appear to be similar to functions already supported in the POSIX standard. These operations exist so that implementations of this interface can be written without having to implement the entire POSIX interface. Implementations that do support complete POSIX interfaces must still support the functions in this section, although their implementation may use the POSIX functions.

Three of the functions defined in this section, **sio_open()**, **sio_control()**, and **sio_test()**, allow the application to specify a set of controls to be applied to a file. Because **sio_control()** provides the simplest introduction to the use of controls, it is described first.

8.1 sio_control

Purpose

Perform a set of control operations on a given file.

Syntax

#include <sio_fs.h>

sio_return_t sio_control(int *FileDescriptor*, **sio_control_t** **Ops*,
sio_count_t *OpCount***);**

Parameters

FileDescriptor The file descriptor of the open parallel file on which to perform the control operations.

Ops An array of control operations to be performed.

OpCount The number of control operations in the array referenced by *Ops*.

Description

This function performs the set of control operations specified by the *Ops* argument on the open file specified by the *FileDescriptor* argument. Each control operation is either mandatory or optional, depending on the bits set in its *flags* field. If any of the mandatory operations would fail, the **sio_control()** operation fails and returns **SIO_ERR_CONTROL_FAILED**. In contrast, the failure of an optional control does *not* cause **sio_control()** to fail. The status of the individual controls can be checked after **sio_control()** returns, via the *result* field in the **sio_control_t** structures.

The application must not assume any ordering on the execution of the controls in *Ops*; the implementation is free to examine and/or execute the *Ops* in any order. Those control operations that succeed may take effect in any order.

If the **sio_control()** operation succeeds, then all of the mandatory controls take effect and have their result codes set to **SIO_SUCCESS**. With regard to the optional controls, one of two situations can occur:

- all of the optional controls take effect and have their result codes set to **SIO_SUCCESS**; or
- at least one of the optional controls fails and has its result code set to a control-specific error value. The remainder of the optional controls may individually 1) fail and have their result code set to a control-specific error value, 2) take effect and have their result code set to **SIO_SUCCESS**, 3) not be attempted and have their result code set to **SIO_ERR_CONTROL_NOT_ATTEMPTED**.

If the **sio_control()** operation fails for any reason, then all of the control operations in *Ops* are annulled, that is, they have no permanent effect on the file system. If **sio_control()** fails, none of the controls will have their *result* field set to **SIO_SUCCESS**. In this case, the implementation may set the result field of a particular control to a control-specific error code if that control would have failed or if the control caused the **sio_control()** to fail, or to **SIO_ERR_CONTROL_WOULD_HAVE_SUCCEEDED** if that control would have succeeded had the **sio_control()** operation not failed, or to **SIO_ERR_CONTROL_NOT_ATTEMPTED** if the **sio_control()** failed before the implementation checked whether or not that control would have succeeded.

Section 13 defines the control operations included in the basic API.

Return Values

SIO_SUCCESS
All mandatory control operations succeeded.

SIO_ERR_CONTROL_FAILED
At least one of the mandatory control operations failed.

SIO_ERR_CONTROLS_CLASH
Some of the mandatory control operations are incompatible with each other and cannot be performed together by this implementation. If a control operation fails with this error, then at least two of the individual control operations must also have their result fields set to **SIO_ERR_CONTROLS_CLASH**.

SIO_ERR_INVALID_DESCRIPTOR
The *FileDescriptor* parameter is not a valid file descriptor.

8.2 sio_open

Purpose

Open a file for reading and/or writing.

Syntax

#include <sio_fs.h>

sio_return_t sio_open(int **FileDescriptorPtr*, **const char** **Name*,
sio_mode_t *Mode*,
sio_control_t **ControlOps*,
sio_count_t *ControlOpCount***);**

Parameters

FileDescriptorPtr On success, this will contain the file descriptor of the newly opened file.

Name The name of the file to open. The name must be at most **SIO_MAX_NAME_LEN** characters in length.

Mode The mode used to open the file. Must include at least one of **SIO_MODE_READ** and **SIO_MODE_WRITE**, or both ORed together. May also include **SIO_MODE_CREATE**.

ControlOps An array of control operations to be performed on the file during the open.

ControlOpCount The number of operations in the array specified by *ControlOps*.

Description

This function takes a logical file name, and produces a file descriptor which supports reading and/or writing, depending on the value of *Mode*. If the named file does not exist and *Mode* has the **SIO_MODE_CREATE** bit set, then the file will be created; if the bit is not set then **SIO_ERR_FILE_NOT_FOUND** will be returned. If **SIO_MODE_CREATE** is set and the file already exists, **SIO_ERR_ALREADY_EXISTS** will be returned.

As part of the operation of opening the file, **sio_open()** performs the control operations described by *ControlOps* and *ControlOpCount*. The control operations have the same meaning and are treated in the same way as in the **sio_control()** function.

If the **sio_open()** operation fails for any reason, then all of the control operations are annulled and have their result codes set in the same way **sio_control()** sets the result codes when it fails.

Note that the semantics of **sio_open()** do not require any permission or security checks. Implementations not embedded in a POSIX file system that wish to provide file permissions can check those permissions on open and can allow those permissions to be set via implementation-specific control operations.

Return Codes

SIO_SUCCESS
The open succeeded.

SIO_ERR_ALREADY_EXISTS
SIO_MODE_CREATE was specified and the file already exists.

SIO_ERR_CONTROL_FAILED
At least one of the mandatory control operations would have failed.

SIO_ERR_CONTROLS_CLASH
Some of the mandatory control operations specified are incompatible with each other and cannot be performed together by this implementation.

SIO_ERR_FILE_NOT_FOUND
The file did not exist and **SIO_MODE_CREATE** was not specified.

SIO_ERR_INVALID_FILENAME
The *Name* parameter is not a legal file name.

SIO_ERR_IO_FAILED
A physical I/O error caused the open to fail.

SIO_ERR_MAX_OPEN_EXCEEDED
Opening the file would result in the task having more than **SIO_MAX_OPEN** open file descriptors.

8.3 sio_close

Purpose

Close a previously opened file.

Syntax

```
#include <sio_fs.h>
sio_return_t sio_close(int FileDescriptor);
```

Parameters

FileDescriptor The file descriptor of the open parallel file to close.

Description

This function closes an open file. All resources associated with having the file open will be deallocated. Cached pending writes are made visible to other nodes before **sio_close()** returns (see Section 12 for details). The results of any asynchronous I/Os in progress at the time **sio_close()** is called are unspecified, and the handles for those I/Os may be invalidated by the system. Applications may ensure that all asynchronous I/Os are complete by calling **sio_async_status_any()** prior to calling **sio_close()** (see Section 10.2). Pre-allocated space, unnecessary for the physical file associated with the open file, may be released.

Note to implementors: Implementations should close all of a task's open parallel file descriptors when the task terminates.

Return Codes

SIO_SUCCESS
The close succeeded.

SIO_ERR_INVALID_DESCRIPTOR
The *FileDescriptor* parameter does not refer to a valid file descriptor previously returned by **sio_open()**.

SIO_ERR_IO_FAILED
A physical I/O error caused the close to fail.

8.4 sio_unlink

Purpose

Remove a file from the parallel file system.

Syntax

```
#include <sio_fs.h>
sio_return_t sio_unlink(const char *Name);
```

Parameters

Name The name of the file to remove.

Description

This function removes a file from the parallel file system, deallocating any space that was allocated for the file. The semantics of unlinking an open file are implementation-specific; possibilities include (but are not limited to) allowing tasks which have this file open to continue to use their open file descriptors, allowing subsequent I/O operations on the file to fail, and allowing **sio_unlink()** itself to fail if the file is open.

Return Codes

SIO_SUCCESS
The unlink succeeded.

SIO_ERR_FILE_NOT_FOUND
The file did not exist.

SIO_ERR_FILE_OPEN
The file *Name* is open and the implementation does not allow open files to be unlinked.

SIO_ERR_INVALID_FILENAME
The *Name* parameter is not a legal file name.

SIO_ERR_IO_FAILED
A physical I/O error caused the unlink to fail.

8.5 sio_test

Purpose

Use mode and control operations to determine attributes of a file by name, without opening the file.

Syntax

#include <sio_fs.h>

sio_return_t sio_test(const char **Name*, **sio_mode_t** *Mode*,
sio_control_t **ControlOps*,
sio_count_t *ControlOpCount***);**

Parameters

Name The name of the target file.

Mode The access mode to be tested. May include one or more of **SIO_MODE_READ**, **SIO_MODE_WRITE**, and **SIO_MODE_CREATE**, ORed together.

ControlOps An array of control operations to be performed on the file.

ControlOpCount The number of operations in the array specified by *ControlOps*.

Description

This function allows an application to test for the existence of a file or test whether a file can be created, and get the attributes of the file, without opening or creating the file.

This function is similar to **sio_open()**, except for two differences:

- It does not actually open or create the specified file.
- It is not allowed to perform any control operations that change the permanent state of the file system.

This function may only use controls that do not change the permanent state of the file system. Of the controls defined in this document, only the following may be performed by **sio_test()**: **SIO_CTL_GetSize** **SIO_CTL_GetAllocation**

SIO_CTL_GetPreallocation **SIO_CTL_GetLayout** **SIO_CTL_GetLabel** **SIO_CTL_GetConsistencyUnit**.

Controls that change file state will return **SIO_ERR_CONTROL_NOT_ON_TEST**. If implementation-specific controls are defined, the implementation must specify whether or not each additional control modifies file state.

Provided a disallowed control is not specified, this function succeeds if a call to **sio_open()** with the same parameters would have succeeded.

If this function fails for any reason, then the result codes of the individual *Ops* are set in the same manner that **sio_open()** sets the result codes of its *Ops*.

Return Codes

SIO_SUCCESS
The test succeeded.

SIO_ERR_ALREADY_EXISTS
SIO_MODE_CREATE was specified and the file already exists.

SIO_ERR_CONTROL_FAILED
At least one of the mandatory control operations would have failed.

SIO_ERR_CONTROL_NOT_ON_TEST
At least one of the control operations changes the file state and may not be used with **sio_test()**.

SIO_ERR_CONTROLS_CLASH
Some of the mandatory control operations specified are incompatible with each other and cannot be performed together by this implementation.

SIO_ERR_FILE_NOT_FOUND
The file did not exist and **SIO_MODE_CREATE** was not specified.

SIO_ERR_INVALID_FILENAME
The *Name* parameter is not a legal file name.

SIO_ERR_IO_FAILED
A physical I/O error caused the function to fail.

SIO_ERR_MAX_OPEN_EXCEEDED
Opening the file would result in the task having more than **SIO_MAX_OPEN** open file descriptors.

8.6 sio_rename

Purpose

Rename a file.

Syntax

#include <sio_fs.h>

sio_return_t sio_rename(const char **OldName*,
const char **NewName***);**

Parameters

OldName The current name of the file.

NewName The new name of the file.

Description

This function changes the name of the file *OldName* to *NewName*. The semantics of renaming an open file are implementation-specific; possibilities include (but are not limited to) allowing tasks which have this file open to continue to use their open file descriptors, allowing subsequent I/O operations on the file to fail, and allowing the rename itself to fail if the file is open.

Return Codes

SIO_SUCCESS
The rename succeeded.

SIO_ERR_ALREADY_EXISTS
NewName already exists.

SIO_ERR_FILE_NOT_FOUND
OldName did not exist.

SIO_ERR_FILE_OPEN
The file *OldName* is open and the implementation does not allow open files to be renamed.

SIO_ERR_INVALID_FILENAME
One of the file names is not a valid name for a file.

SIO_ERR_IO_FAILED

A physical I/O error caused the function to fail.

9 Synchronous File I/O

This section introduces new functions for file read and write operations. These provide file system functions previously unavailable in UNIX systems, as they allow strided scatter and gather of data in memory and also in a file.

One of the primary performance-limiting problems for file systems and parallel programs arises when the data-moving interfaces are restricted to moving single contiguous regions of bytes. This restriction causes applications to ask too frequently for small amounts of work and it denies the system the ability to obtain performance benefits from grouping (batching, scheduling, coalescing). Our first step toward removing this limitation is to offer interfaces that allow the transfer of multiple ranges in a file to or from multiple ranges in memory. We call this capability *scatter-gather*.

The read and write operations introduced in this section are not like traditional read/write operations. Rather than describing file and memory addresses as linear buffers, these calls describe them as lists of strided accesses. Each element of the list specifies a single strided access, consisting of a starting address (offset), size of each contiguous region, stride between the contiguous regions, and the total number of regions in the strided access (see Section 4 for the formats of these elements). Data are copied from the source buffer to the destination in *canonical order*. The canonical order of an individual strided access is the sequence of contiguous byte regions specified by the access. The canonical order for a list of strided accesses is simply the concatenation of the canonical orders for the strided accesses. Intuitively, all byte regions specified by the canonical ordering in a file are concatenated into a contiguous zero-address based virtual window. The byte regions specified in memory are also concatenated in canonical order into this virtual window. Each byte of the virtual window corresponds to one byte of the file and also to one byte of memory. The number of bytes specified in the two lists must be equal.

We place no restrictions on the values of addresses occurring in the canonical ordering of the data structure from the file or memory. This mapping may be increasing, decreasing or non-monotonic in the file or memory, and may cover a given byte more than once.

Note that the file system need not access the file or memory in canonical order. Data can be accessed in the file or memory in any sequence as preferred by the file system to optimize performance. The canonical sequence of file regions is used only to compute the association of the file data with memory regions.

If the source list (i.e. the memory buffer during a write or the file buffer during a read) contains the same region more than once then its data will be copied into the destination buffer multiple times. If the destination list contains the same region more than once then the resulting contents of the duplicated region are undefined.[6]

Applications must not access an I/O operation's memory buffer while the operation is in progress. For example, a thread in a multi-threaded application must not read or write a buffer while another thread has an I/O in progress using the same buffer. Failure to avoid such accesses may corrupt the task and/or file in undefined ways, including leaving the contents of the file corrupted or causing the task to fault. Applications that wish to share I/O buffers between threads must explicitly synchronize the threads' accesses to those buffers.

It is expected that many users of this API will desire simpler interfaces to this functionality. In addition to the basic POSIX interfaces, the interfaces in Appendix B are easily built on the interfaces provided in this API. These, or similar simplified interfaces, could easily be provided by a high-level library, and are not defined by this API.

[6] No function to check for duplicate regions in the destination list is provided. However, such a function could be implemented as part of a higher-level library built on top of this API.

9.1 sio_sg_read, sio_sg_write

Purpose

Transfer data between a file and memory.

Syntax

```
#include <sio_fs.h>

sio_return_t sio_sg_read(int FileDescriptor,
                         const sio_file_io_list_t *FileList,
                         sio_count_t FileListLength,
                         const sio_mem_io_list_t *MemoryList,
                         sio_count_t MemoryListLength,
                         sio_transfer_len_t *TotalTransferred);

sio_return_t sio_sg_write(int FileDescriptor,
                          const sio_file_io_list_t *FileList,
                          sio_count_t FileListLength,
                          const sio_mem_io_list_t *MemoryList,
                          sio_count_t MemoryListLength,
                          sio_transfer_len_t *TotalTransferred);
```

Parameters

FileDescriptor The file descriptor of an open file.

FileList Specification of file data to be read or written.

FileListLength Number of elements in *FileList*.

MemoryList Specification of the memory buffer containing data to be read or written.

MemoryListLength Number of elements in *MemoryList*.

TotalTransferred Used to return the total number of bytes read or written.

Description

These functions move data between a list of file locations and a list of memory locations. All I/O must be done to a single file, in the *FileDescriptor* argument.

The mapping between the collection of file regions specified by *FileList* and the collection of memory byte regions specified by *MemoryList* is in matching indices in the canonical ordering of the corresponding **sio_file_io_list_t** and **sio_mem_io_list_t**.

If the total transfer cannot be completed because a file address is not valid (i.e. reading beyond the end of the file), these interfaces will complete successfully, and return in *TotalTransferred* the index of the first byte in the canonical ordering that could not be transferred (following the UNIX example); bytes preceding this index in the canonical ordering have been transferred successfully and bytes following (and including) it may or may not have been transferred successfully.

Implementations may return a value less than the actual amount transferred if the operation was not successful; in particular, an implementation may indicate that zero bytes were transferred successfully on all failures.

Return Codes

SIO_SUCCESS
The function succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the I/O.

SIO_ERR_INVALID_DESCRIPTOR
FileDescriptor does not refer to a valid file descriptor.

SIO_ERR_INVALID_FILE_LIST
The file regions described by *FileList* are invalid, e.g. they contain illegal addresses.

SIO_ERR_INVALID_MEMORY_LIST
The memory regions described by *MemoryList* are invalid, e.g. they contain illegal addresses.

SIO_ERR_IO_FAILED
A physical I/O error caused the function to fail.

SIO_ERR_NO_SPACE
The file system ran out of space while trying to extend the file.

SIO_ERR_UNEQUAL_LISTS

The number of bytes in *MemoryList* and *FileList* are not equal.

10 Asynchronous File I/O

Asynchronous I/O allows a single-threaded task to issue concurrent I/O requests. The parallel file system supports up to **SIO_MAX_ASYNC_OUTSTANDING** (see Section 5.1) asynchronous I/Os at a time for each task. Asynchronous I/O functions merely initiate an I/O, returning to the task a handle that may be used by the task to wait for the I/O to complete, to check its status of the I/O, or to cancel the I/O.

These handles are of type **sio_async_handle_t**, which is an opaque type defined by the system. Only the task that issued the asynchronous I/O is able to use the **sio_async_handle_t** associated with the I/O to retrieve the status of or cancel the I/O. Other tasks that wish to retrieve the status of or cancel an I/O must contact the task that initiated the I/O.

10.1 sio_async_sg_read, sio_async_sg_write

Purpose

Asynchronously transfer data between a file and memory.

Syntax

```
#include <sio_fs.h>

sio_return_t sio_async_sg_read(int FileDescriptor,
                               const sio_file_io_list_t *FileList,
                               sio_count_t FileListLength,
                               const sio_mem_io_list_t *MemoryList,
                               sio_count_t MemoryListLength,
                               sio_async_handle_t *Handle);

sio_return_t sio_async_sg_write(int FileDescriptor,
                                const sio_file_io_list_t *FileList,
                                sio_count_t FileListLength,
                                const sio_mem_io_list_t *MemoryList,
                                sio_count_t MemoryListLength,
                                sio_async_handle_t *Handle);
```

Parameters

FileDescriptor The file descriptor of an open file.

FileList Specification of file data to be read or written.

FileListLength Number of elements in *FileList*.

MemoryList Specification of the memory buffer containing data to be read or written.

MemoryListLength Number of elements in *MemoryList*.

Handle Handle returned by the operation, which can be used later to determine the status of the I/O, to wait for its completion, or to cancel it.

Description

These functions behave similarly to **sio_sg_read()** and **sio_sg_write()**. A successful return, however, indicates only that the I/O has been queued for processing by the parallel file system.

Handle is a task-specific value which may be used to poll for completion, block until the I/O completes, or cancel the I/O. The handle remains valid until either the task completes, or **sio_async_status_any()** indicates that the I/O transfer associated with *Handle* is no longer in progress. While a handle is valid it counts towards the **SIO_MAX_ASYNC_OUTSTANDING** asynchronous I/Os that a task may have.

As in synchronous I/O, applications must neither access nor modify the contents of a memory buffer while an asynchronous I/O is in progress on that buffer. Doing so may leave the buffer and/or the file in an undefined state, and may cause the task to fault. See Section 9 for details.

Return Codes

SIO_SUCCESS
The function succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not allow the I/O.

SIO_ERR_INVALID_DESCRIPTOR
FileDescriptor does not refer to a valid file descriptor.

SIO_ERR_INVALID_FILE_LIST
The file regions described by *FileList* are invalid, e.g. they contain illegal addresses. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

SIO_ERR_INVALID_MEMORY_LIST
The memory regions described by *MemoryList* are invalid, e.g. they contain illegal addresses. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

SIO_ERR_IO_FAILED
A physical I/O error caused the function to fail.

SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED
The I/O request could not be initiated because doing so would cause the calling task's number of outstanding asynchronous I/Os to exceed the limit.

SIO_ERR_NO_SPACE
The file system ran out of space while trying to extend the file. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

SIO_ERR_UNEQUAL_LISTS
The number of bytes in *MemoryList* and *FileList* are not equal. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

10.2 sio_async_status_any

Purpose

Get the status of asynchronous I/Os.

Syntax

#include <sio_fs.h>

sio_return_t sio_async_status_any(
  sio_async_handle_t **HandleList*,
  sio_count_t *HandleListLength*,
  sio_count_t **Index*,
  sio_async_status_t **Status*,
  sio_async_flags_t *Flags*);

Parameters

HandleList An array of **sio_async_handle_t**s identifying the asynchronous I/Os for which status is desired.

HandleListLength The number of elements in *HandleList*.

Index Used to return the index of handle within *HandleList* for which status is returned.

Status Pointer to an **sio_async_status_t** to be filled in.

Flags Determines whether or not the operation blocks or returns immediately.

Description

This function retrieves the status of one of the asynchronous I/Os specified by *HandleList*. The index of the handle within *HandleList* for which the status is returned is stored in *Index*. The system may return the status for any of the handles, provided that if any of the I/Os are complete or canceled, then the status for one of these I/Os is returned and not the status of an I/O that is still in progress.

It is important to note that once the status for an I/O indicates that the I/O is no longer in progress (i.e. it completed or was canceled) the handle for the I/O is no longer valid. If

it is subsequently passed to **sio_async_status_any()** the value **SIO_ERR_INVALID_HANDLE** will be returned if the handle is still invalid, otherwise the status of the new asynchronous I/O will be returned if the handle has been reused.

The task may place a dummy handle in the **HandleList** by setting the entry to **SIO_ASYNC_DUMMY_HANDLE**. The system ignores a handle with this value, allowing the task to retrieve the status for a set of handles using the same *HandleList* array, by replacing the handle for the I/O just finished with the dummy value.

If the *Flags* parameter includes **SIO_ASYNC_BLOCKING**, this function will not return until at least one of the I/Os has completed. If it includes **SIO_ASYNC_NONBLOCKING**, this function returns immediately, regardless of whether or not one of the I/Os has completed.

Note to implementors: When an I/O is canceled the *count* field in *Status* will contain the number of bytes guaranteed to have been transferred prior to the cancellation. Implementations may always set this value to zero, indicating that none of the bytes are guaranteed to have been transferred.

Status Results

The following values are returned in the *result* field of the *Status* structure, indicating the status of the I/O:

SIO_SUCCESS
: The I/O has completed or been canceled. The *count* field contains the number of bytes transferred.

SIO_ERR_INVALID_FILE_LIST
: The file regions described by the *FileList* parameter passed to the function that initiated the I/O are invalid, e.g. they contain illegal addresses.

SIO_ERR_INVALID_MEMORY_LIST
: The memory regions described by the *MemoryList* parameter passed to the function that initiated the I/O are invalid, e.g. they contain illegal addresses.

SIO_ERR_IO_CANCELED
The I/O was canceled without completing. The *count* field contains the number of bytes *guaranteed* to have been transferred successfully prior to the cancellation. Implementations may set *count* to zero.

SIO_ERR_IO_FAILED
A physical I/O error caused the function to fail.

SIO_ERR_IO_IN_PROGRESS
The I/O is still in progress.

SIO_ERR_MIXED_COLL_AND_ASYNC
The implementation does not support mixing of asynchronous and collective I/O handles, and a mix of handle types was supplied.

SIO_ERR_NO_SPACE
The file system ran out of space while trying to extend the file.

SIO_ERR_UNEQUAL_LISTS
The size of the memory buffer doesn't match size of the file regions to be accessed.

Return Values

SIO_SUCCESS
An I/O has completed or been canceled, the index and result of which are stored in *Index* and *Status*, respectively.

SIO_ERR_INVALID_HANDLE
At least one of the elements of *HandleList* is neither a valid handle for an asynchronous I/O nor a dummy handle. *Index* will contain the index of one of the invalid handles.

SIO_ERR_IO_IN_PROGRESS
All I/Os are still in progress.

10.3 sio_async_cancel_all

Purpose

Request that a collection of asynchronous I/Os be canceled.

Syntax

```
#include &lt;sio_fs.h&gt;

sio_return_t sio_async_cancel_all(
                    sio_async_handle_t *HandleList,
                    sio_count_t HandleListLength);
```

Parameters

HandleList An array of **sio_async_handle_t**s identifying the asynchronous I/Os to be canceled.

HandleListLength The number of elements in *HandleList*.

Description

This function is used to request that asynchronous I/Os be canceled. It is not guaranteed that the I/O will not complete in full or in part; an implementation may ignore cancel requests. A canceled read leaves the contents of the I/O's memory buffer undefined. Likewise, if a write is canceled, the contents of the regions of the file regions being written are undefined.

The status of a canceled request remains available until an **sio_async_status_any()** reports its completion. An application should test for this status or its maximum outstanding asynchronous I/Os will appear to diminish.

Note to implementors:

An implementation may ignore cancellation requests altogether. In this case a call to **sio_async_status_any()** on an I/O that whose cancellation was requested should return the normal, uncanceled completion status of the I/O.

Note to implementors: Implementations are encouraged to avoid reusing the same handles for different asynchronous I/Os within the same task. A handle becomes invalid once the I/O is no longer in progress and its status has been retrieved, but bugs may cause a task to use such an invalid handle. If the system has reassigned the handle to a new I/O the task will end up affecting the new I/O, instead of getting an invalid handle error. Although this behavior is caused by a bug in the application, avoiding reuse of handles will help track the problem.

Return Values

SIO_SUCCESS
: The request for cancellation was accepted. This does not mean that the I/Os were actually canceled.

SIO_ERR_INVALID_HANDLE
: One of the elements in *HandleList* is not a valid handle for an asynchronous I/O.

11 File Access Pattern Hints

File access pattern hints provide a useful mechanism for users and libraries to disclose the intended use of file regions to the file system. The hints, if properly given, allow file systems to implement significant performance optimizations. Many parallel scientific programs, for example, have access patterns that are anathemic to some file system architectures. These applications could benefit if the file system accepted access hints that protected the application from the performance consequences of the default file system behavior. For example, access hints can be used by the file system to choose caching and pre-fetching policies.

Hints are issued with the **sio_hint()** and **sio_hint_by_name()** interfaces described in Section 11.3. These interfaces indicate a file, a hint class, and a list of hints. Hints apply only to the future accesses of the task passing in the hints, they are not associated with the accesses of other tasks. There are two hint classes specified in this API: ordered and unordered. Vendors are encouraged to extend this API with vendor-defined hint classes, which must have names beginning with **SIO_HINT_CLASS_VEND_**. Within any class of hints, the interaction of all hint types must be specified, but the interaction of hint types from different classes need not be specified. In particular, two calls issuing hints with different hint classes for the same open file may not be meaningful to an implementation. However, since the information in these hints are not commands, the file system implementation has broad freedom not to act where hint combinations are not meaningful.

The intent of hints is to allow the application to precisely specify what its future access patterns will be. The hint interface does not provide specific guarantees of how implementations will interpret these hints. Different implementations are free to choose different strategies for responding to hints (including ignoring them completely), but the application's description of its future accesses must conform to this interface.

System performance may be degraded due to inaccurate hints. Implementations should attempt to protect against such performance degradation, but are not required to. Similarly, applications should not assume that the file system can always limit the performance impacts of inaccurate hints (accesses that have been hinted, but will not actually be performed) and should make use of the cancel options to minimize these effects.

11.1 Ordered Hints

In a set of ordered hints, each hint indicates a particular future access to be issued by the calling task, and the sequence of issued hints indicates the order of these future accesses. The total order of future accesses expressed by multiple invocations of the hint interfaces is determined by logically concatenating the hint array in each invocation onto the end of the hint array built by previously issued hints. This allows access to different files to be ordered. The accesses to different files predicted by one hint are expected to occur after the accesses predicted by all hints preceding it in the total order, and before the accesses predicted by all hints following it in the total order.

The *flag* field of each **sio_hint_t** in the class of ordered hints can contain the following flags that can be ORed with each other:

SIO_HINT_READ or SIO_HINT_WRITE

SIO_HINT_READ indicates the hint describes a read access. **SIO_HINT_WRITE** indicates the hint describes a write access.

Exactly one of these flags must be specified for each hint. When used to cancel a hint the flags in the cancel request must match the hint's flags.

SIO_HINT_CANCEL_ALL or SIO_HINT_CANCEL_NEXT

Regardless of the file specified by the hint interface call and the regions specified by the *io_list* fields in the **sio_hint_t** structures, **SIO_HINT_CANCEL_ALL** indicates that all previously issued hints should be ignored.

SIO_HINT_CANCEL_NEXT indicates that the previously issued hint matching the file and region specified with this **SIO_HINT_CANCEL_NEXT** whose predicted access is next to occur should be ignored. A hint is considered "outstanding" if the data transfer request predicted by the hint has not yet occurred. It is expected the data transfer requests will take place in the sequence given by the total ordered list of hints for the task, with the possibility that not all transfer requests will have corresponding hints. The "next outstanding hint" will be the first matching hint in the set of ordered hints

previously issued by this task for which no corresponding for transfer request has occurred.

A previously issued hint's profile "matches" the current hint's profile if the hints pertain to the same file, and the regions specified by the *io_list* entry in the **sio_hint_t** structures are the same and the **SIO_HINT_READ** or **SIO_HINT_WRITE** flag matches.

No more than one of these flags may be specified for each hint.

Note to implementors: Implementations are not required to keep track of "outstanding" hints. The concept of "outstanding" only describes the application's intent in issuing the hint, and does not describe the implementation's behavior. In implementations that do not keep track of "outstanding" hints the **SIO_HINT_CANCEL_NEXT** hint may not be useful.

11.2 Unordered Hints

In an unordered set of hints, each hint independently specifies information about some set of future accesses. There is no explicit ordering among the accesses predicted by unordered hints. These predictions remain in effect until explicitly canceled.

The flag field of each **sio_hint_t** in the class of unordered hints can contain the following flags:

SIO_HINT_READ and/or SIO_HINT_WRITE
: **SIO_HINT_READ** indicates that the hint describes read accesses. **SIO_HINT_WRITE** indicates that the hint describes write accesses. If **SIO_HINT_READ** and **SIO_HINT_WRITE** are given together, they indicate that the hint describes a read-write access.

  At least one of these flags must be specified for each hint.

SIO_HINT_CANCEL_ALL or SIO_HINT_CANCEL_MATCHING
: **SIO_HINT_CANCEL_ALL** suggests that the file system ignore all previously issued unordered hints from this task, regardless of the file and file regions given in any of these hints. **SIO_HINT_CANCEL_MATCHING** suggests that the file system ignore all previously issued unordered hints from this task which match the given **sio_hint_t**.

  No more than one of these flags may be specified for each hint.

SIO_HINT_SEQUENTIAL, SIO_HINT_REVERSE, SIO_HINT_RANDOM_PARTIAL, SIO_HINT_RANDOM_COMPLETE, SIO_HINT_NO_FURTHER_USE, or SIO_HINT_WILL_USE
: Each hint expresses an access pattern predicted for the file region given by the hint. When changing a predicted access pattern on a region, a **SIO_HINT_CANCEL_MATCHING** hint should be issued to cancel the old hint before the new access hint is issued. The interpretation of multiple predicted access patterns on the same region or partial (overlapping) region is unspecified. These patterns are:

  SIO_HINT_SEQUENTIAL
  : The entire region will be accessed in non-overlapping blocks whose starting offsets increase monotonically.

  SIO_HINT_REVERSE
  : The entire region will be accessed in non-overlapping blocks whose starting offsets decrease monotonically.

  SIO_HINT_RANDOM_COMPLETE
  : Accesses in the region will have starting addresses and sizes that vary without pattern but the entire region will be accessed.

  SIO_HINT_RANDOM_PARTIAL
  : Accesses in the region will have starting addresses and sizes that vary without pattern and the entire region may not be accessed.

  SIO_HINT_NO_FURTHER_USE
  : No further accesses are expected in the region.

  SIO_HINT_WILL_USE
  : All data in the region will be accessed although no explicit pattern can be predicted or excluded.[7]

  Exactly one of these flags must be specified for each hint.

[7] This pattern should be used in cases where **SIO_HINT_RANDOM_COMPLETE** cannot because the access pattern might not be random.

11.3 sio_hint, sio_hint_by_name

Purpose

Issue a set of predictions about the future accesses of this task.

Syntax

#include <sio_fs.h>

```
sio_return_t sio_hint(int FileDescriptor,
                      sio_hint_class_t HintClass,
                      const sio_hint_t *Hints,
                      sio_count_t HintCount);
```

```
sio_return_t sio_hint_by_name(const char *FileName,
                              sio_hint_class_t HintClass,
                              const sio_hint_t *Hints,
                              sio_count_t HintCount);
```

Parameters

FileDescriptor The file descriptor of an open file to which these hints apply.

FileName The name of a file, not necessarily an open file, to which these hints apply.

HintClass The class of the hints being issued.

Hints An array of file access pattern hints.

HintCount The number of entries in the *Hints* array.

Description

This function reports the application's knowledge of future access patterns to the file system. The purpose of issuing this information is to enable optimizations in the dynamic behavior of the parallel file system. This knowledge is expressed as a set of hints, all from the same hint class. The interpretation of mixtures of hint types from different hint classes is unspecified. Hints can be applied to an open file using **sio_hint()**, or to a named file (which may not be open) using

sio_hint_by_name(). Each **sio_hint_t** structure in the *Hints* array describes a hint type applied to a list of file regions and optionally hint-specific arguments.

If the *size*, *stride*, and *element_cnt* fields for a particular **sio_file_io_list_t** in a hint are all zero, then the region being specified begins at the offset given by the *offset* field of that **sio_file_io_list_t** and continues until the end of the file. The entire contents of a file are specified as the region whenever an **sio_file_io_list_t** contains zero in the four fields: *offset*, *size*, *stride* and *element_cnt*.

The implementation may not act on any specific hint or on any hints at all.

Return Codes

SIO_SUCCESS
The function succeeded.

SIO_ERR_FILE_NOT_FOUND
The specified file did not exist.

SIO_ERR_HINT_TYPES_CLASH
The class of this hint differs from the class of another hint previously issued for the same file region.[8]

SIO_ERR_INVALID_CLASS
The hint class given in *HintClass* is not a valid hint class.

SIO_ERR_INVALID_DESCRIPTOR
FileDescriptor does not refer to a valid file descriptor created by **sio_open()**.

SIO_ERR_INVALID_FILENAME
The name given by *FileName* is invalid.

[8] As mentioned above, the effects of mixing hints of different classes for the same file region are undefined. This error code is provided for implementations that attempt to resolve hints from different classes.

12 Client Cache Control

The basic API includes facilities to control caching of data in client memory. The caching interfaces are specified such that it is a valid implementation strategy to simply ignore all cache control calls. The only requirement of an implementation that ignores these calls is that it must provide strongly consistent semantics.

The client caching mode of an SIO file may be specified by including the **SIO_CTL_SetCachingMode** control operation when making **sio_open()** or **sio_control()** calls.

This API specifies client caching modes with the type **sio_caching_mode_t**, which can have the following values:

SIO_CACHING_NONE
: Completely disable client caching.

SIO_CACHING_STRONG
: Allow strongly-consistent client caching. The file system may choose to provide caching with strong sequential consistency, or provide no caching at all.

SIO_CACHING_WEAK
: Allow weakly-consistent client caching. The file system may provide no client caching, strongly-consistent client caching, or weakly-consistent client caching.

Caching mode names beginning with **SIO_CACHING_** are reserved for future use by this API. Vendors may define their own caching modes by naming them with the prefix **SIO_CACHING_VEND_**.

An SIO parallel file system implementation's default client caching mode must provide sequential consistency. That is, it must be either **SIO_CACHING_NONE**, **SIO_CACHING_STRONG**, or a vendor-defined mode that provides strong sequential consistency.

If client caching is not disabled by using a caching mode of **SIO_CACHING_NONE**, the file system on a client node is free to maintain local copies of file data for both read and write operations.

In a system with strongly-consistent caching, every write forces the client node to immediately make the file system aware that the file has changed. This also requires that client nodes either check the validity of cached data before providing them to applications to satisfy a read, or be notified whenever cached or potentially cached data have changed.

On the other hand, weakly-consistent client caching allows the file system to avoid the messaging and bookkeeping which a sequentially consistent caching mode mandates, while providing the application with the benefits of caching. With this form of caching, client nodes may defer exposing all or part of a set of changes to a file until instructed otherwise by the application. Likewise, a client node need not confirm the validity of cached data with the server unless explicitly instructed to do so by the application.

An application informs the file system that data written on a file descriptor should become visible to other readers via the **SIO_CTL_Propagate** control operation. If the changed data have not already been exposed to the rest of the file system, this is done so immediately. Note that all, none, or part of this changed data may already have been exposed to the rest of the file system.

Likewise, an application informs the file system that locally cached data may be stale using the **SIO_CTL_Refresh** control operation. Reads of refreshed regions of a file are guaranteed to yield either the most current available data, or data that were not stale at the time of the most recent refresh operation. That is to say, if the data returned by the read are stale, it was made so *after* the refresh.

It is assumed that applications using weakly-consistent client caching either do not share data between nodes, or provide their own internal synchronization to coordinate when nodes must propagate and refresh data.

Thus, the way in which a node **A** would write data which are then read by a node **B** is:

A writes data to region **R**

A propagates data in region **R**

(Implicit:) **A** and **B** synchronize; **B** becomes aware that new data in region **R** are available

B refreshes data in region **R**

B reads data in region **R**

The granularity of caching is known as the *consistency unit*. This defines both the size and the alignment of the blocks of data within the file for which the file system insures that all non-conflicting writes are merged into the file. Tasks on different nodes cannot use weak consistency and achieve consistent parallel updates within a single consistency unit. Any conflicting writes within a single consistency unit will be resolved by an arbitrary selection of a winning writer when the data arrive at a server. The size of the consistency unit is implementation specific, and is represented by the constant **SIO_CACHE_CONSISTENCY_UNIT**. Additionally, the control operation **SIO_CTL_GetConsistencyUnit** can be used to retrieve the consistency unit for a file descriptor.[9] Applications should not make any assumptions about the size of the consistency unit; it may vary between individual bytes, cache lines, pages, and file blocks depending upon the implementation of the file system.

The motivation for providing weakly-consistent client caching as an option within the parallel file system is to allow parallel applications that could benefit from a decrease in the total amount of data being transferred between clients and servers to exercise relatively fine-grained control over the consistency of their local caches. **SIO_CTL_Propagate** and **SIO_CTL_Refresh** operations can be piggy-backed onto synchronization steps that already exist in parallel applications. These primitives allow application programmers and toolkit developers the mechanisms necessary to ensure consistency of the local parallel file system cache, without requiring the parallel file system to enforce any consistency model itself.

This implementation of weakly-consistent caching is only intended to cope with sharing among the tasks of a parallel application. To avoid unintended sharing among independent applications, traditional methods based on detecting conflicts at open time and disabling caching or resorting to strongly-consistent caching may be used.

[9] Currently, this should always yield **SIO_CACHE_CONSISTENCY_UNIT**. This is intended to allow for future extensions, which may provide different consistency units within the same implementation.

Some implementations may choose not to provide weak client cache consistency by ignoring a **SIO_CTL_SetCachingMode** operation that specifies the **SIO_CACHING_WEAK** mode, as well as the **SIO_CTL_Propagate** and **SIO_CTL_Refresh** operations. In this case, the **SIO_CTL_GetCachingMode** should return a value of **SIO_CACHING_NONE**, **SIO_CACHING_STRONG**, or a sequentially-consistent vendor-defined caching mode as appropriate, and **SIO_CTL_Propagate** and **SIO_CTL_Refresh** should always return success. (This way, an application which can tolerate weakly-consistent caching will not see extraneous errors in its absence.[10]

Note that client caching is controlled on a per-file descriptor basis, so it is possible to have a file opened with one client caching mode on one file descriptor and with a different mode on another file descriptor.

Descriptions of the **SIO_CTL_GetCachingMode**, **SIO_CTL_SetCachingMode**, **SIO_CTL_Propagate**, **SIO_CTL_Refresh**, and **SIO_CTL_GetConsistencyUnit** control operations are given in Section 13.

Note to implementors: The routine **sio_close()** implicitly performs a **SIO_CTL_Propagate** on the file descriptor. This causes all cached writes to be exposed to the file system at the time the file is closed, if they have not been already.

[10] Since weak caching mode can be implemented using strong caching, it is possible that an application running on one node may see data modifications that have not yet been propagated on a remote node. This is normal, since a weakly-consistent caching policy may expose the results of writes soon after or immediately as they occur.

13 Control Operations

This section describes the file control operations that can be performed using the functions **sio_control()**, **sio_open()**, **sio_test()**.

These control operations allow properties of files, file descriptors, and the file system to be set and retrieved.

Control operations are performed by invoking **sio_open()**, **sio_control()**, or **sio_test()** with the list of operations to be performed. Each operation description, an **sio_control_t**, includes the code of the operation to be performed, a pointer to the data to be manipulated by that operation, and space for a result code. In the following sections, information is provided about the various operation codes that must be implemented by file systems that conform to this API.

Operation names beginning with **SIO_CTL_** are reserved for use by this API. Operation names beginning with **SIO_CTL_VEND_** may be used by vendors to define vendor-specific operations.

13.1 SIO_CTL_GetSize, SIO_CTL_SetSize

Purpose

Get or set the file's logical size.

Affects

Open file

Parameter Type

Pointer to a **sio_offset_t**.

Description

Applications may query and adjust the logical size (see Section 6.1) of a file using these control operations. The **SIO_CTL_SetSize** operation causes the logical size of the file to be set to the value in the **sio_offset_t** pointed to by the *op_data* field of the **sio_control_t**. Setting a file's logical size may change the amount of storage that the file uses, but is not guaranteed to do so. An application wishing to preallocate storage for a file should use the **SIO_CTL_SetPreallocation** control operation.

The **SIO_CTL_GetSize** operation causes the logical size of the file being operated on to be placed in the **sio_offset_t** pointed to by the *op_data* member of the **sio_control_t**.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the operation.

SIO_ERR_IO_FAILED
A physical I/O error caused the operation to fail.

SIO_ERR_NO_SPACE
The system needs to increase the amount of storage used by the file but cannot.

13.2 SIO_CTL_GetAllocation

Purpose

Get the file's physical size.

Affects

Underlying file.

Parameter Type

Pointer to a **sio_offset_t**.

Description

The **SIO_CTL_GetAllocation** operation causes file's physical size (see Section 6.1) to be placed in the **sio_offset_t** pointed to by the *op_data* field of the **sio_control_t**.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the operation.

SIO_ERR_IO_FAILED
A physical I/O error caused the operation to fail.

13.3 SIO_CTL_GetPreallocation, SIO_CTL_SetPreallocation

Purpose

Get or set amount of space preallocated for the file.

Affects

Underlying file.

Parameter Type

Pointer to a **sio_offset_t**.

Description

The **SIO_CTL_GetPreallocation** operation causes the amount of space preallocated (see Section 6.1) for the file being operated on to be placed in the **sio_offset_t** pointed to by the *op_data* field of the **sio_control_t**.

The **SIO_CTL_SetPreallocation** operation causes the amount of space preallocated for the file being operated on to be set to the value in the **sio_offset_t** pointed to by the *op_data* field of the **sio_control_t**. A preallocation applies to an open file and will be reset to zero when the file is closed. While open, writes by other tasks that extend the physical size of the file may reduce the unconsumed preallocation.

If either the **SIO_CTL_GetPreallocation** operation or the **SIO_CTL_SetPreallocation** operation is supported, both must be supported.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the operation.

SIO_ERR_IO_FAILED
A physical I/O error caused the operation to fail.

SIO_ERR_NO_SPACE
: There isn't enough free space in the system to satisfy the request.

SIO_ERR_OP_UNSUPPORTED
: The operation is not supported by the system.

13.4 SIO_CTL_GetCachingMode, SIO_CTL_SetCachingMode

Purpose

Get or set the file's caching mode.

Affects

File descriptor.

Parameter Type

Pointer to a **sio_caching_mode_t**.

Description

The **SIO_CTL_GetCachingMode** operation causes the caching mode of the file descriptor to be placed in the **sio_caching_mode_t** pointed to by the *op_data* field of the **sio_control_t**.

The **SIO_CTL_SetCachingMode** operation causes the caching mode of the file descriptor to be set to the value of the **sio_caching_mode_t** pointed to by the *op_data* field ¡of the **sio_control_t**. SIO implementations which provide support for multiple caching modes may elect not to provide support for changing the caching mode of an open file.

Result Values

SIO_SUCCESS
: The operation succeeded.

SIO_ERR_INCORRECT_MODE
: The mode of the file descriptor does not permit the operation.

SIO_ERR_ONLY_AT_OPEN
: The system does not allow the caching mode of an open file to be changed. Caching modes can only be changed as part of **sio_open()**.

SIO_ERR_OP_UNSUPPORTED
: The system does not support **SIO_CTL_SetCachingMode**.

13.5 SIO_CTL_Propagate

Purpose

Force locally cached writes to be made visible to other nodes.

Affects

Cached writes associated with file descriptor.

Parameter Type

Pointer to a **sio_file_io_list_t**.

Description

This operation allows a task to force the parallel file system to make any data associated with a particular set of byte ranges visible to other nodes in the system (see Section 12 for information about why this might be necessary), as specified by the **sio_file_io_list_t** pointed to by the *op_data* field of the control request. If *op_data* is **NULL**, the propagation will apply to all bytes in the file. If the *size*, *stride* , and *element_cnt* fields of the **sio_file_io_list_t** pointed to by the op_data field are all zero, then the set of bytes to be propagated begins at the offset specified in the *offset* field of the **sio_file_io_list_t** and continues until the end of the file.

This operation only affects those bytes written via the given file descriptor; if an application writes to a file using more than one file descriptor, it must perform a propagate operation on each of them to guarantee the dirty data become visible to other nodes. While it is guaranteed after a propagate operation completes that all locally cached writes for the specified file regions have been exposed to the rest of the file system, it is *not* guaranteed that some or all the changed data was not visible to the rest of the file system *prior to* the propagate. That is, weakly-consistent client caching implies only that cached writes will be exposed to the rest of the file system *no later than* the completion of the propagate operation.

Result Values

SIO_SUCCESS

The results of all writes on this file descriptor in the specified region(s) have been exposed to the rest of the file system.

SIO_ERR_INVALID_FILE_LIST

op_data is not **NULL** nor a pointer to a valid **sio_file_io_list_t**.

13.6 SIO_CTL_Refresh

Purpose

Inform the file system that locally cached data may be invalid.

Affects

Blocks in client's cache containing data for this file.

Parameter Type

Pointer to a **sio_file_io_list_t**.

Description

This operation informs the parallel file system that data cached for a file may be stale, that is, superseded by more recent writes (see Section 12 for information about why this might be necessary). Future reads to the specified client region(s) are guaranteed to not yield data that were stale at the time the refresh operation began.[11] File region(s) are specified by the **sio_file_io_list_t** pointed to by the *op_data* field of the control request. If *op_data* is NULL, the operation will apply to all bytes in the file. If the *size*, *stride*, and *element_cnt* fields of the **sio_file_io_list_t** pointed to by the op_data field are all zero then the operation applies to the set of bytes beginning at the offset specified in the *offset* field of the **sio_file_io_list_t** and ending at the end of the file.

Result Values

SIO_SUCCESS
The regions have been refreshed.

SIO_ERR_INVALID_FILE_LIST
op_data is not **NULL** or a pointer to a valid **sio_file_io_list_t**.

[11] The file system may satisfy this requirement by explicitly validating all cached data in the specified region(s) with the server, or by ejecting the specified blocks from the cache entirely.

13.7 SIO_CTL_Sync

Purpose

Force dirty data to stable storage.

Affects

Blocks written via the file descriptor.

Parameter Type

None

Description

This operation causes all dirty blocks associated with the file descriptor to be written to stable storage. The meaning of "stable storage" is implementation specific – it may be the disk, non-volatile memory, or another mechanism that provides greater reliability than the volatile memory in the node caching the blocks. **SIO_CTL_Sync** performs a superset the operations performed by **SIO_CTL_Propagate**.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_IO_FAILED
A physical I/O error caused the operation to fail.

13.8 SIO_CTL_GetLayout, SIO_CTL_SetLayout

Purpose

Get or set the layout of the file data on the storage system.

Affects

Underlying file.

Parameter Type

Pointer to a **sio_layout_t**.

Description

These operations allow the layout of a file's data on the underlying storage system to be queried and modified.

The control **SIO_CTL_GetLayout** will return the layout for the underlying file, while **SIO_CTL_SetLayout** will set the layout, if possible. Implementations may choose to ignore **SIO_CTL_SetLayout** entirely, returning **SIO_ERR_OP_UNSUPPORTED**.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_OP_ONLY_AT_CREATE
The implementation only supports **SIO_CTL_SetLayout** when a file is being created.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the operation.

SIO_ERR_OP_UNSUPPORTED
The operation is not supported by the system.

13.9 SIO_CTL_GetLabel, SIO_CTL_SetLabel

Purpose

Get or set the file's label.

Affects

Underlying file.

Parameter Type

Pointer to a **sio_label_t**.

Description

These operations allow the label associated with a file to be set and retrieved. A file's label is not interpreted by the file system. The intent is for applications to store descriptive information about a file in the a file's label, rather than in the file itself. That removes the need for file headers and the inefficiencies that go with them.

The maximum size of a file's label is **SIO_MAX_LABEL_LEN**, the value of which is implementation-specific. It is guaranteed, however, to be at least as big as **SIO_MAX_NAME_LEN**, allowing any legal file name to fit in a label. This allows descriptive information that is too large to fit in a label to be stored in an auxiliary file whose name can be stored in the label of the file being described.

For descriptive labels to be portable across implementations they must be no larger than the minimum allowed value for **SIO_MAX_LABEL_LEN**.

When performing **SIO_CTL_SetLabel**, the *data* field of the **sio_label_t** must contain a pointer to a buffer, the *length* field must contain the length of that buffer. If the length given is greater than **SIO_MAX_LABEL_LEN**, **SIO_ERR_INVALID_LABEL** will be returned and the operation will fail. After a **SIO_CTL_SetLabel** operation successfully completes, the length of the file's label will be equal to *length*, and the file's label data will be the same as the contents of the buffer.

When performing **SIO_CTL_GetLabel**, the *data* field of the **sio_label_t** must contain a pointer to a buffer to be filled in with the file's current label data, and the *length* field must contain the size of that buffer. If the buffer is too small to contain the label, the **SIO_ERR_INVALID_LABEL** error code will be returned, *length* will be set to the actual length of the label, and the contents of the data buffer will be unspecified. If the buffer is at least as large as the current file label, **SIO_SUCCESS** will be returned, *length* will be set to the actual length of the label (as set by a previous call to **SIO_CTL_SetLabel**, or to zero if the file's label has never been set), and the data buffer will be filled with that many bytes of label data. If the buffer is larger than the label, the contents of the bytes in the buffer following the label are unspecified.

Result Values

SIO_SUCCESS
The operation succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the operation.

SIO_ERR_INVALID_LABEL
The length of the new label being set exceeds **SIO_MAX_LABEL_LEN**, or the length of the label being retrieved exceeds the size of the application-provided buffer.

SIO_ERR_IO_FAILED
A physical I/O error caused the operation to fail.

SIO_ERR_NO_SPACE
The system needs to increase the amount of storage used by the file but cannot.

13.10 SIO_CTL_GetConsistencyUnit

Purpose

Get the size of the cache consistency unit.

Affects

File system.

Parameter Type

Pointer to a **sio_size_t**.

Description

This operation returns the size of the cache consistency unit. The consistency unit defines the granularity of cache consistency under weak caching, as described in Section 12.

Result Values

SIO_SUCCESS
The operation succeeded.

14 Extension Support

Support for querying the presence of extensions is part of the basic API, and must be implemented by all conforming implementations, even if no extensions are supported by an implementation. Applications may determine either statically (described in Section 14.1.1) or dynamically (via the **sio_query_extension()** function, described in Section 14.2) whether or not an extension is supported by the implementation of the API. Sample code indicating the proper way to check for the presence of extensions is included in Section14.3.

14.1 Static Constants

14.1.1 Extension Support Constants

Applications may statically determine via constants which extensions are supported by a given implementation. For each extension that an implementation is capable of supporting, the implementation should define a constant which indicates that the extension is supported, that it is not, or that the support status cannot be determined during compilation. These constants are of the form **SIO_EXT_*NAME*_SUPPORTED**, where *NAME* is the name of the extension. Each of these constants must be set to one of the following values:

SIO_EXT_ABSENT (equal to 0) The extension is not supported.

SIO_EXT_PRESENT The extension is supported.

SIO_EXT_MAYBE The extension might be supported. A dynamic check must be used to make a final determination.

The **SIO_EXT_ABSENT** constant must be zero so that existence of extensions which the implementation is completely unaware of can be checked.[12] The values of the other constants are unspecified.

If the static constant for an extension is equal to **SIO_EXT_ABSENT**, then the application cannot depend on any of the functions or definitions that are a part of the extension (including the extension ID) being present. If the static constant is **SIO_EXT_PRESENT** or **SIO_EXT_MAYBE**, then the functions and definitions that are a part of the extension will be present. In the case of **SIO_EXT_MAYBE**, the functions and definitions may be usable only if the extension is determined to be available at run-time.

The definition of **SIO_EXT_ABSENT** allows for implementations to conform to this API without requiring updates for any new extensions which may be added in the future. The **SIO_EXT_MAYBE** value allows for binary compatibility across different versions of an implementation that support different sets of extensions.

[12]The C preprocessor will expand an unknown definition as zero when used in preprocessor directives, and this allows undefined extension support macros to match **SIO_EXT_ABSENT**.

14.1.2 Extension Identifiers

Extension identifiers are constants of the form **SIO_EXT_***NAME*, where *NAME* is the name of the extension. Extension identifiers with names of the form **SIO_EXT_VEND_***NAME* are reserved for use by vendors, and all other extension names are reserved for future use by this API.

An implementation must define an extension identifier for each extension which is supported or may be supported by that implementation as determined by the value of the extension's **SIO_EXT_***NAME***_SUPPORTED** constant described in Section 14.1.1. Extension identifiers can be given to **sio_query_extension()** to check whether or not the extensions in question are actually available. [13]

[13] It is not necessary to call **sio_query_extension()** for extensions whose extension support constants indicate that they are present, but it is safe to do so and **sio_query_extension()** must indicate that those extensions are supported.

14.2 sio_query_extension

Purpose

Determine whether or not an extension is supported.

Syntax

#include <sio_fs.h>

sio_return_t sio_query_extension(sio_extension_id_t *ExtID***);**

Parameters

ExtID Extension identifier of extension being queried.

Description

This function takes an extension identifier and returns **SIO_SUCCESS** if the extension is supported by this implementation, or **SIO_ERR_INVALID_EXTENSION** if the extension is not supported, or if the identifier is not recognized as valid.

Return Codes

SIO_SUCCESS
The extension is supported by the implementation.

SIO_ERR_INVALID_EXTENSION
ExtID contains an invalid or unsupported extension ID.

14.3 Sample Code to Check for Extension Presence

A code fragment which queries the presence of an extension might look like:

```
    int fooext_is_present;
    sio_return_t rc;

#if SIO_EXT_FOO_SUPPORTED == SIO_EXT_ABSENT
    fooext_is_present = 0;
#elif SIO_EXT_FOO_SUPPORTED == SIO_EXT_PRESENT
    fooext_is_present = 1;
#else /* SIO_EXT_FOO_SUPPORTED == SIO_EXT_MAYBE */
    rc = sio_query_extension(SIO_EXT_FOO);
    switch(rc) {
      case SIO_SUCCESS:
        fooext_is_present = 1;
        break;
      case SIO_ERR_INVALID_EXTENSION:
        fooext_is_present = 0;
        break;
      default:
        fooext_is_present = 0;
        printf("can't determine if extension foo is present (%s)\n",
          sio_error_string(rc));
    }
#endif /* SIO_EXT_FOO_SUPPORTED == SIO_EXT_ABSENT */
```

 Typeset at 13:53 on September 3, 1996

15 Extension: Collective I/O

Static Constant: SIO_EXT_COLLECTIVE_SUPPORTED

Extension ID: SIO_EXT_COLLECTIVE

15.1 Motivation

As demonstrated by Kotz et al., collective I/O allows for a distributed batching process which can greatly enhance I/O performance in a parallel file system. Semantically, by declaring an I/O or set of I/Os to be part of a single, collective I/O, an application is indicating to the file system that the relative ordering of the components of the collective I/O is irrelevant, since no portion of the application awaiting a component of the collective I/O can make any progress until the entirety of the collective I/O completes. File systems can take advantage of this to drastically reorder I/O components to reduce overall latency, at the potential cost of increasing the latency of component I/Os (the constraint which prevents this optimization from occurring in the standard case).

15.2 High Level Look

To initiate a collective I/O one task of the application requests that a new collective I/O handle be created. This is what we refer to as "defining" the collective I/O. At this time, the application indicates the number of participants, whether the collective I/O is a read or write operation (we do not allow collective mixed read/writes), the number of iterations of the collective I/O, and optionally indicates what portions of the file will be operated on. Specification of file regions at define time provides (ordered) file access hints which, if properly given, allow the file system to implement performance optimizations.

Each participant "joins" an iteration of the collective I/O by providing the handle created by the define operation, the file descriptor, the portions of

the file they wish to read or write, the source/destination memory locations, their participant identifier, and a sequence number indicating which iteration of the collective I/O they are joining.

Note that the application will generally need to pass the handle from the task that defined the collective I/O to any other tasks that participate in the I/O. A single task may participate multiple times in a given collective I/O iteration by joining that iteration multiple times using different participant numbers. Prior to joining a collective I/O operation, a task must open the file being accessed so a file descriptor for the file is available for use with the join call.

15.3 New Data Types

15.3.1 sio_coll_handle_t

This is a 64-bit integral type used as an abstract handle to represent a collective I/O. We explicitly define the format and size of this datatype because applications will need to use their own communications mechanisms to pass these among tasks on different nodes, and therefore need to be aware of size and network ordering issues.

15.3.2 sio_coll_participant_t

This is an unsigned integral type with the range [0... **SIO_MAX_COLL_PARTICIPANTS**] which is used in the definition of a collective I/O operation to specify the number of participants, and in the collective I/O join to identify the participant joining the collective I/O iteration.

These values have no meaning or permanence beyond the collective I/O in which they are used.

15.3.3 sio_coll_iteration_t

This is an unsigned integral type with the range [0... **SIO_MAX_COLL_ITERATIONS**] which is used in the definition of a collective I/O operation to specify the number of iterations, and in the collective I/O join to identify the iteration being joined.

15.4 New Range Constants

15.4.1 SIO_MAX_COLL_ITERATIONS

This constant specifies the maximum number of iterations that a collective I/O can describe. The minimum value is 1, and the recommended value is 128.

15.4.2 SIO_MAX_COLL_PARTICIPANTS

This constant specifies the maximum number of participants that can take part in a collective I/O. The minimum value is 16, but the recommended value is at least 256.

15.4.3 SIO_MAX_COLL_OUTSTANDING

This constant specifies the maximum number of outstanding collective I/O requests that one task can have at any given time. The minimum value is 1, and the recommended value is at least 512.

15.5 New Functions

Two new functions are added by the collective I/O extension: **sio_coll_define()** and **sio_coll_join()**, which are described in Sections 15.5.1 and 15.5.2, respectively.

15.5.1 sio_coll_define

Purpose

Define a new collective I/O and get a handle to refer to it.

Syntax

```
#include <sio_fs.h>

sio_return_t sio_coll_define(int FileDescriptor,
                             sio_coll_iteration_t NumIterations,
                             const sio_file_io_list_t *FileList,
                             sio_count_t FileListLength,
                             sio_size_t IterationStride,
                             sio_mode_t ReadWrite,
                             sio_coll_participant_t NumParticipants,
                             sio_coll_handle_t *Handle);
```

Parameters

FileDescriptor The file descriptor of an open file.

NumIterations The number of times the collective I/O will be repeated.

FileList Specification of file data to be read or written.

FileListLength Number of elements in *FileList*. This may be zero.

IterationStride A value that modifies the location of the file data to be read or written as specified in *FileList* based on the iteration in progress.

ReadWrite One of **SIO_MODE_READ** or **SIO_MODE_WRITE**.

NumParticipants The number of participants in each iteration of the collective I/O.

Handle On success, returns the handle of the newly-defined collective I/O.

Description

This interface creates a new handle for a collective I/O, and returns it in *Handle*. The *NumIterations* parameter indicates the number of times the collective I/O will be performed. The application programmer may choose to disclose the portions of the file which will be affected in *FileList*, or *FileListLength* may be zero in which case the file system must wait for a participant to call **sio_coll_join()** before its workload is known.

In cases where the collective I/O will be performed more than once and the application programmer indicates what portions of the file will be operated on, it is often true that the access patterns for each iteration are identical except for their offsets from the beginning of the file, and that the offsets are based on the iteration being performed. The *IterationStride* parameter lets the programmer express these common cases without having to separate them into individual collective I/O operations. If i is the iteration number (starting at iteration 0), the *offset* field in each **sio_file_io_list_t** structure of the *FileList* parameter would have the value:

$$\mathit{offset}_i = \mathit{offset}_0 + (i \times \mathit{IterationStride})$$

For example, if *FileList* has two entries with the values (*offset*=0, *size*=2, *stride*=3, *element_cnt*=4) and (*offset*=100, *size*=5, *stride*=0, *element_cnt*=1), the programmer is hinting that the first iteration will access bytes (0, 1, 3, 4, 6, 7, 9, 10, 100, 101, 103, 104, 105) in the file. If *IterationStride* is zero, the second iteration will access the same bytes. However, if *IterationStride* is 50, the second iteration will access bytes (50, 51, 53, 54, 56, 57, 59, 60, 150, 151, 153, 154, 155) – the *offset* components of the *FileList* structures are adjusted based on the iteration (1) and the *IterationStride* (50).

Note that **sio_coll_join()** must always be called by each participant and must provide a *FileList* for that participant's portion of the collective I/O, whether or not *FileListLength* is zero in **sio_coll_define()**. Providing a description of the entire operation in *FileList* simply provides a way for the file system to optimize scheduling of the transfer.

Return Codes

SIO_SUCCESS
The function succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the I/O.

SIO_ERR_INVALID_DESCRIPTOR
FileDescriptor does not refer to a valid file descriptor created by **sio_open()**.

SIO_ERR_INVALID_FILE_LIST
The file regions described by *FileList* are invalid, e.g. they contain illegal offsets.

SIO_ERR_MAX_COLL_ITERATIONS_EXCEEDED
The number of iterations described by *NumIterations* exceeds the maximum allowed as defined by **SIO_MAX_COLL_ITERATIONS**.

SIO_ERR_MAX_COLL_PARTICIPANTS_EXCEEDED
The number of participants described by *NumParticipants* exceeds the maximum allowed as defined by **SIO_MAX_COLL_PARTICIPANTS**.

15.5.2 sio_coll_join

Purpose

Initiate an asynchronous transfer as part of a collective I/O.

Syntax

```
#include <sio_fs.h>

sio_return_t sio_coll_join(int FileDescriptor,
                           sio_coll_handle_t Handle,
                           sio_coll_participant_t Participant,
                           sio_coll_iteration_t Iteration,
                           const sio_file_io_list_t *FileList,
                           sio_count_t FileListLength,
                           const sio_mem_io_list_t *MemoryList,
                           sio_count_t MemoryListLength,
                           sio_async_handle_t *AsyncHandle);
```

Parameters

FileDescriptor The file descriptor of the open file where the collective I/O is being performed.

Handle The handle provided by **sio_coll_define()** for this collective operation.

Participant The identifier for this participant. This is a number in the range $[0 \ldots (NumParticipants - 1)]$, where *NumParticipants* is the number of participants that was provided to **sio_coll_define()**.

Iteration Which iteration of the collective I/O the participant is joining.

FileList Specification of file data to be read or written by this participant.

FileListLength Number of elements in *FileList*.

MemoryList Memory locations read or written by this I/O component.

MemoryListLength Number of elements in *MemoryList*.

AsyncHandle Handle returned by the operation, which can be used later to determine the status of the I/O, to wait for its completion, or to cancel it.

Description

This interface initiates a component of a collective I/O. At this point, the file system may immediately begin transferring data to or from these memory locations, or it may choose to wait for other participants to join the collective I/O. The number of participants in each iteration of the collective I/O must equal the *NumParticipants* specified to **sio_coll_define()**, i.e. **sio_coll_join()** must be called *NumParticipants* times for each iteration. **sio_coll_join()** returns immediately and **sio_async_status_any()** or **sio_async_cancel_all()** must be called with the *AsyncHandle* to complete or cancel the operation.

Note that calls to **sio_async_status_any()** or **sio_async_cancel_all()** reflect only this participant's portion of this iteration of the collective I/O, as identified by the value of *AsyncHandle*. Also, calls to the **sio_async_status_any()** and **sio_async_cancel_all()** may contain multiple *AsyncHandle*s, but the *AsyncHandle*s returned by the **sio_coll_join()** may not be mixed with *AsyncHandle*s returned by **sio_async_sg_read()** or **sio_async_sg_write()** functions in the same call.

To clarify some of the parameters a bit further, the *FileDescriptor* parameter must refer to the same file as was specified by the *FileDescriptor* in the **sio_coll_define()** for this collective operation. However, the actual *FileDescriptor* value may differ from the one in the **sio_coll_define()** because the task making the join call may be different from the task that defined the collective operation.

If the **sio_coll_define()** for this collective operation contained information about the bytes that would be accessed in its *FileList* parameter, then to realize performance gains the *FileList* parameter in this **sio_coll_join()** call should contain bytes that appeared in the original **sio_coll_define()** *FileList* parameter. If this is not the case, or if the **sio_coll_define()** did not contain file region information, the bytes specified in the **sio_coll_join()** *FileList* parameter will still be read or written, but potentially with poorer performance.

Finally, note that there is no parameter in the **sio_coll_join()** call corresponding to the **sio_coll_define()** parameter *IterationStride*. In the join, it is the responsibility of the application programmer to adjust the *FileList* offset values as appropriate for the iteration being joined.

Return Codes

SIO_SUCCESS
The function succeeded.

SIO_ERR_INCORRECT_MODE
The mode of the file descriptor does not permit the I/O.

SIO_ERR_INVALID_DESCRIPTOR
FileDescriptor does not refer to a valid file descriptor created by **sio_open()**, or does not refer to the file specified to **sio_call_define()** when the collective I/O was created.

SIO_ERR_INVALID_FILE_LIST
The file regions described by *FileList* are invalid, e.g. they contain illegal offsets. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

SIO_ERR_INVALID_HANDLE
Handle is not the handle for a collective I/O.

SIO_ERR_INVALID_ITERATION
Iteration is not valid, either because it is greater than the number of iterations specified when the collective I/O was created or between the task already joined that iteration of the I/O.

SIO_ERR_INVALID_MEMORY_LIST
The memory regions described by *MemoryList* are invalid, e.g. they contain illegal addresses. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

SIO_ERR_INVALID_PARTICIPANT
Participant is not valid because it is greater than the number of participants specified when the collective I/O was created.

SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED
The I/O request could not be initiated because doing so would cause the calling task's number of outstanding asynchronous I/Os to exceed the limit.

SIO_ERR_UNEQUAL_LISTS
The number of bytes in *MemoryList* and *FileList* are not equal. Implementations may defer returning this error until **sio_async_status_any()** is invoked on the I/O.

16 Extension: Fast Copy

Static Constant: SIO_EXT_FAST_COPY_SUPPORTED

Extension ID: SIO_EXT_FAST_COPY

This extension provides a low-level versioning mechanism by allowing an efficient "snapshot" of a file's current contents to be created. This is done via the **sio_control()** operation **SIO_CTL_FastCopy**.

The **SIO_CTL_FastCopy** control operation creates snapshots by replacing the contents of a parallel file (created and opened with **sio_open()**), with the contents of the file being duplicated. Since snapshots are normal parallel files, they can be accessed in all of the ways that parallel files can be accessed. That is, snapshots created by **SIO_CTL_FastCopy** can be read, written, operated on by controls, etc.

If a higher-level file system library is using **SIO_CTL_FastCopy** to provide versioning support, that library is responsible for managing the translation between its notion of versions and that provided by the **SIO_CTL_FastCopy** mechanism. For instance, the higher-level library must translate between the file name and version number that the application supplies and the actual parallel name for that snapshot. The higher-level library must also enforce its own version reference semantics (perhaps preventing write access to old versions of the file, or taking other actions as necessary).

16.1 SIO_CTL_FastCopy

Purpose

Efficiently copy the contents of one file into another.

Affects

Underlying file.

Parameter Type

Pointer to an **int** which is a file descriptor for the open parallel file to be used as the source of the efficient copy operation.

Description

This operation performs an efficient copy of the contents of one parallel file into another. The source file descriptor is specified by the **int** pointed to by the *op_data* member of the **sio_control_t**. The destination file is specified by the *Name* argument to **sio_open()** or by the *FileDescriptor* argument to **sio_control()**.

The implementation of the efficient copy operation performed by this function is intended to use copy-on-write or similar techniques to minimize data duplication.

If the **SIO_CTL_FastCopy** operation fails or is not supported, an error will be returned and the source and destination files will be unmodified.

Effects of Successful Operation on the Source File

The source file's data are unmodified by the **SIO_CTL_FastCopy** operation.

The source file's physical size at the conclusion of the **SIO_CTL_FastCopy** operation is unspecified.

None of the source file's other file or file descriptor attributes (as defined by this API) are modified by the **SIO_CTL_FastCopy** operation.

If vendors define new attributes, the effect of **SIO_CTL_FastCopy** on the source file with respect to those attributes should be specified.

Hints about expected use of the source file are unmodified by the **SIO_CTL_FastCopy** operation.

Effects of Successful Operation on the Destination File

The destination file's logical size is set to the source file's logical size, and the destination file's contents are made to appear identical (e.g. if accessed with **sio_sg_read()**) to those of the source file. If **SIO_CTL_SetSize** is specified in the same set of control operations as **SIO_CTL_FastCopy**, the resulting size of the destination file is undefined.

The destination file's physical size at the conclusion of the **SIO_CTL_FastCopy** operation is unspecified.

The destination file's label is made identical to the source file's label.

The destination file's other file attributes (preallocation and layout) are not affected.

None of the destination's file descriptor attributes (caching mode and consistency unit) are affected. Note that if a weak client caching mode is in use on the destination file, the destination file's new contents may need to be propagated (with **SIO_CTL_Propagate**) before they can be used by other clients.

If vendors define new attributes, the effect of **SIO_CTL_FastCopy** on the destination file with respect to those attributes should be specified.

The effect of the **SIO_CTL_FastCopy** operation on hints about expected use of the destination file is unspecified. Portable applications or libraries that wish to hint about future accesses to the destination file should cancel all outstanding hints on the destination file after performing a **SIO_CTL_FastCopy** operation and then reissue hints as appropriate.

Result Values

SIO_SUCCESS
: The function succeeded.

SIO_ERR_INVALID_DESCRIPTOR
: The file descriptor for the source file is invalid.

SIO_ERR_NO_SPACE
: There isn't enough free space to perform a fast copy.

SIO_ERR_OP_UNSUPPORTED
: Fast copy is not supported by the implementation for files with the attributes of the source file and/or destination file.

Acknowledgments

Many of the ideas presented in the 0.1 draft were developed in discussions with many different people. Among these are Dror Feitelson, Yarsun Hsu, and Marc Snir of IBM Research, Bob Curran, Joe Kavaky, and Jeff Lucash of IBM Power Parallel Division, and Daniel Stodolsky of CMU, David Kotz of Dartmouth, and David Payne and Brad Rullman of Intel SSD. Also contributing were members of the SIO community as a whole who participated in the discussions we had in the first half of 1995.

The second draft, version 0.2, reflected comments made by Dror Feitelson, Marc Snir, Jeff Lucash, and Bob Curran of IBM.

The third draft, version 0.3, incorporating asynchronous and return-by-reference interface variants and client caching control, reflects comments from Adam Beguelin, Dave O'Halleron, Jaspal Subhlok, and Thomas Stricker of CMU, December 1995.

The fourth draft, version 0.41, was presented to the SIO Operating Systems Working Group at Princeton on 8 February 1996.

The fifth draft, version 0.52, was presented to the SIO technical committee meeting at Chicago on 2 April 1996. It incorporates comments and results of discussion from the Princeton workshop and specific detailed comments from Tom Cormen of Dartmouth.

The sixth draft, version 0.54, was presented at the SIO technical meeting held at Argonne National Laboratory on May 13-14, 1996.

The seventh draft, version 0.60, was presented at the meeting of the SIO Operating Systems Working Group, held at Carnegie Mellon University on July 2, 1996.

The eighth and ninth drafts, 0.62 and 0.63 were presented and discussed at the meeting of the SIO Operating Systems Working Group at Princeton University on August 8-9, 1996.

Version .66 was reviewed by e-mail, August 21-31, 1996.

Members of the SIO Performance Evaluation working group at UIUC reviewed and commented on the API beginning with the fifth draft. In particular, Ruth Aydt, Andrew Chien, Chris Elford, Tara Madhyastha, Dan Reed, Huseyin Simitci, and Evgenia Smirni contributed to the discussions and suggestions put forth by the UIUC group.

A Result codes (for sio_return_t)

This appendix describes some error and return codes that the parallel file system may wish to return. As discussed in the Data Types section, implementors should feel free to add whatever additional codes they see fit, and should make **sio_error_string()** aware of them.

SIO_SUCCESS
: The operation completed successfully. The value of **SIO_SUCCESS** must always be 0.

SIO_ERR_ALREADY_EXISTS
: The file name to be created already exists.

SIO_ERR_CONTROL_FAILED
: One or more of the control operations requested by **sio_control()**, **sio_open()**, or **sio_test()** was unsuccessful.

SIO_ERR_CONTROL_NOT_ATTEMPTED
: A control operation requested by **sio_control()**, **sio_open()**, or **sio_test()** was not attempted.

SIO_ERR_CONTROL_NOT_ON_TEST
: The control operation cannot be used with **sio_test()**.

SIO_ERR_CONTROL_WOULD_HAVE_SUCCEEDED
: The control operation would have succeeded but the function performing the control failed.

SIO_ERR_CONTROLS_CLASH
: The list of controls contains combinations of operations that are incompatible.

SIO_ERR_FILE_NOT_FOUND
: The specified file did not exist.

SIO_ERR_FILE_OPEN
: The operation failed because the file was open.

SIO_ERR_INCORRECT_MODE
: The mode of the file descriptor does not permit the operation or function.

SIO_ERR_INVALID_CLASS
: The hint class is not valid.

SIO_ERR_INVALID_DESCRIPTOR
: A file descriptor argument was not a valid parallel file descriptor.

SIO_ERR_INVALID_EXTENSION
: An invalid extension identifier was given, or the indicated extension is not supported.

SIO_ERR_INVALID_FILE_LIST
: The file list argument is invalid (e.g contains illegal offsets).

SIO_ERR_INVALID_FILENAME
: A file name argument did not contain a legal file name (e.g. it was too long).

SIO_ERR_INVALID_HANDLE
: A handle argument does not contain a valid handle.

SIO_ERR_INVALID_ITERATION
: The iteration argument is invalid.

SIO_ERR_INVALID_MEMORY_LIST
: The memory list argument is invalid (e.g. contains an illegal address).

SIO_ERR_INVALID_PARTICIPANT
: The participant number provided is not valid because it is greater than the number of participants specified when the collective I/O was created.

SIO_ERR_IO_CANCELED
: An asynchronous I/O did not complete because it was canceled while in progress.

SIO_ERR_IO_FAILED
: A physical I/O error occurred.

SIO_ERR_IO_IN_PROGRESS
An asynchronous I/O has not yet completed.

SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED
The I/O request could not be initiated because doing so would cause the calling task's number of outstanding asynchronous I/Os to exceed the limit.

SIO_ERR_MAX_COLL_ITERATIONS_EXCEEDED
The number of iterations specified for a collective I/O exceeds the limit.

SIO_ERR_MAX_COLL_OUTSTANDING_EXCEEDED
The I/O request could not be initiated because doing so would cause the calling task's number of outstanding collective I/O's to exceed the limit.

SIO_ERR_MAX_COLL_PARTICIPANTS_EXCEEDED
The number of participants specified for a collective I/O exceeds the limit.

SIO_ERR_MAX_OPEN_EXCEEDED
The file could not be opened because doing so would cause the calling task's number of open files to exceed the limit.

SIO_ERR_MIXED_COLL_AND_ASYNC
The implementation does allow asynchronous I/O handles created by **sio_coll_define()** to be passed to functions in the same list as handles from **sio_async_sg_read()** and **sio_async_sg_write()**.

SIO_ERR_NO_SPACE
An operation that would allocate more storage to a file failed because no storage could be allocated.

SIO_ERR_ONLY_AT_CREATE
The control operation may only be specified during a call to **sio_open()** which is creating a file.

SIO_ERR_ONLY_AT_OPEN
The control operation may only be specified during a call to **sio_open()**.

SIO_ERR_OP_UNSUPPORTED
The parallel file system has elected to not support this interface. Note that some interfaces may not be supported, but implementations can choose to return **SIO_SUCCESS** for all cases instead.

SIO_ERR_UNEQUAL_LISTS
The number of bytes in the memory and file lists arguments to an I/O operation are not the same.

B Sample Derived Interfaces

This section describes some simple interfaces which could easily be created using the interfaces provided by this API. These derived interfaces are *not* a part of this API, and are intended only as examples of interfaces which could be provided by high level libraries.

If a high level library provides interfaces similar (or identical) to the sample interfaces presented here, those interfaces should be named in accordance with the rest of the interfaces provided by that library. In other words, *use of the names given here is strongly discouraged.*

B.1 Synchronous I/O

Routines

```
sio_return_t sample_read(int FileDescriptor,
                         sio_addr_t BufferPointer,
                         sio_offset_t Offset, sio_size_t Count,
                         sio_transfer_len_t *BytesRead);

sio_return_t sample_write(int FileDescriptor,
                          sio_addr_t BufferPointer,
                          sio_offset_t Offset, sio_size_t Count,
                          sio_transfer_len_t *BytesWritten);

sio_return_t sample_read_io_list(int FileDescriptor,
                                 sio_addr_t BufferPointer,
                                 sio_file_io_list_t *FileList,
                                 sio_count_t FileListLength,
                                 sio_transfer_len_t *BytesRead);

sio_return_t sample_write_io_list(int FileDescriptor,
                                  sio_addr_t BufferPointer,
                                  sio_file_io_list_t *FileList,
                                  sio_count_t FileListLength,
                                  sio_transfer_len_t *BytesWritten);

sio_return_t sample_read_mem_list(int FileDescriptor,
                                  sio_mem_io_list_t *MemoryList,
                                  sio_count_t MemoryListLength
                                  sio_offset_t Offset,
                                  sio_transfer_len_t *BytesRead);

sio_return_t sample_write_mem_list(int FileDescriptor,
                                   sio_mem_io_list_t *MemoryList,
                                   sio_count_t MemoryListLength
                                   sio_offset_t Offset,
                                   sio_transfer_len_t *BytesWritten);
```

Parameters

FileDescriptor The file descriptor of an open parallel file.

BufferPointer Memory address of contiguous buffer containing data to be written or to contain data being read.

Offset Starting file offset from which to read or at which to write.

Count Number of bytes to read or write.

BytesRead Number of bytes actually read.

BytesWritten Number of bytes actually written.

FileList Description of strided regions within the file.

FileListLength Number of valid elements to use in *FileList*.

MemoryList Description of strided regions within the memory buffer.

MemoryListLength Number of valid elements to use in *MemoryList*.

Description

These functions would provide a simplified synchronous I/O interface. They may be implemented as wrappers which would convert the given arguments into **sio_mem_io_list_t** and **sio_file_io_list_t** structures (as necessary) and invoke **sio_sg_read()** or **sio_sg_write()**.

The functions **sample_read()** and **sample_write()** would transfer data between a single contiguous memory buffer and a single contiguous region of the file. The functions **sample_read_io_list()** and **sample_write_io_list()** would use a single contiguous memory buffer, but a strided region within the file. Similarly, **sample_read_mem_list()** and **sample_write_mem_list()** would use a contiguous file region, but a strided region within the memory buffer.

B.2 Asynchronous I/O

Routines

```
sio_return_t sample_async_read(int FileDescriptor,
                               sio_addr_t BufferPointer,
                               sio_offset_t Offset,
                               sio_size_t Count,
                               sio_async_handle_t *Handle);

sio_return_t sample_async_write(int FileDescriptor,
                                sio_addr_t BufferPointer,
                                sio_offset_t Offset,
                                sio_size_t Count,
                                sio_async_handle_t *Handle);

sio_return_t sample_async_read_io_list(int FileDescriptor,
                                       sio_addr_t BufferPointer,
                                       sio_file_io_list_t *FileList,
                                       sio_count_t FileListLength
                                       sio_async_handle_t *Handle);

sio_return_t sample_async_write_io_list(int FileDescriptor,
                                        sio_addr_t BufferPointer,
                                        sio_file_io_list_t *FileList,
                                        sio_count_t FileListLength
                                        sio_async_handle_t *Handle);

sio_return_t sample_async_read_mem_list(int FileDescriptor,
                                        sio_mem_io_list_t *MemoryList,
                                        sio_count_t MemoryListLength
                                        sio_offset_t Offset,
                                        sio_async_handle_t *Handle);

sio_return_t sample_async_write_mem_list(int FileDescriptor,
                                         sio_mem_io_list_t *MemoryList,
                                         sio_count_t MemoryListLength
                                         sio_offset_t Offset,
                                         sio_async_handle_t *Handle);
```

Parameters

FileDescriptor The file descriptor of an open parallel file.

BufferPointer Memory address of contiguous buffer containing data to be written or to contain data being read.

Offset Starting file offset from which to read or at which to write.

Count Number of bytes to read or write.

BytesRead Number of bytes actually read.

BytesWritten Number of bytes actually written.

FileList Description of strided regions within the file.

FileListLength Number of valid elements to use in *FileList*.

MemoryList Description of strided regions within the memory buffer.

MemoryListLength Number of valid elements to use in *MemoryList*.

Handle Handle for asynchronous I/O that can later be used to test its status.

Description

These routines would provide a simplified asynchronous I/O interface. They may be implemented as wrappers which would convert the given arguments into **sio_mem_io_list_t** and **sio_file_io_list_t** structures (as necessary) and invoke **sio_async_sg_read()** or **sio_async_sg_write()**.

These functions would take arguments similar to those given to the simplified synchronous functions, and perform similar actions.

B.3 Cache Consistency

Functions

sio_return_t sample_propagate(int *FileDescriptor*,
sio_offset_t *Offset*,
sio_size_t *Length***);**

sio_return_t sample_refresh(int *FileDescriptor*,
sio_offset_t *Offset*,
sio_size_t *Length***);**

Parameters

FileDescriptor File descriptor to which cache consistency action applies.

Offset Starting file offset affected by consistency action.

Length Number of bytes affected by consistency action.

Description

These functions would perform cache consistency actions on the specified region of the file associated with the given file descriptor. It may be implemented as wrappers which would invoke **sio_control()** to perform the appropriate **SIO_CTL_Propagate** or **SIO_CTL_Refresh** operation.