

Proposal for a Common Parallel File System Programming Interface 1.0

Peter Corbett¹, Jean-Pierre Prost¹, Chris Demetriou,
Garth Gibson, Erik Riedel, Jim Zelenka, Yuqun Chen²,
Ed Felten², Kai Li², John Hartman³, Larry Peterson³,
Brian Bershad⁴, Alec Wolman⁴, Ruth Ayt⁵

October 1996
CMU-CS-96-193

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Also appears as Tech. Report CACR-130, Scalable I/O Initiative, Caltech
Center for Advanced Computing Research, Pasadena, CA, November 1996.

¹IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

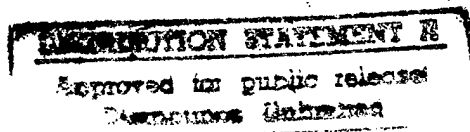
²Department of Computer Science
Princeton University
Princeton, NJ 08544

³Department of Computer Science
The University of Arizona
Tucson, AZ 85721

⁴Computer Science & Engineering
University of Washington
Seattle, WA 98195

⁵Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

This research is sponsored by DARPA through the Scalable I/O Initiative, and
issued by the US Army at Fort Huachuca under contract DABT63-94-C-0049.
The views and conclusions contained in this document are those of the authors
and should not be interpreted as representing the official policies, either expressed
or implied, of any supporting organization or the U.S. Government.



DTIC QUALITY INSPECTED 8

19970519 031

[DTIC QUALITY INSPECTED 3]

Keywords: Parallel File Systems, Low-Level Interface, Scalable Input/Output,
High-Performance Computing

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

1	Introduction	7
1.1	Independent Messaging and Minimal Synchronization	8
1.2	No Shared File Pointers	9
1.3	Scatter-Gather Transfers	10
1.4	Asynchronous I/O	10
1.5	I/O Controls	10
1.6	Client Caching	11
1.7	File Access Pattern Hints	12
1.8	Extensions to this API	12
1.9	Collective I/O	13
1.10	Checkpoints and File Versioning	13
1.11	File Names and Access Protection	14
1.12	File Labels	14
2	Document Conventions	15
2.1	Typesetting Conventions	15
2.2	Definition of Terms	15
2.3	How to Read this Document	17
3	The <code>sio_fs.h</code> Include File	19
4	Data Types	21
4.1	File Descriptor	21
4.2	File Name	21
4.3	Memory Address	22
4.4	<code>sio_async_flags_t</code>	22
4.5	<code>sio_async_handle_t</code>	22
4.6	<code>sio_async_status_t</code>	22
4.7	<code>sio_caching_mode_t</code>	23
4.8	<code>sio_control_t</code>	23
4.9	<code>sio_control_flags_t</code>	23
4.10	<code>sio_control_op_t</code>	24
4.11	<code>sio_count_t</code>	24
4.12	<code>sio_extension_id_t</code>	24

4.13	sio_file_io_list_t	24
4.14	sio_hint_t	25
4.15	sio_hint_class_t	25
4.16	sio_hint_flags_t	26
4.17	sio_label_t	26
4.18	sio_layout_t	26
4.19	sio_layout_algorithm_t	27
4.20	sio_layout_flags_t	28
4.21	sio_mem_io_list_t	28
4.22	sio_mode_t	28
4.23	sio_offset_t	29
4.24	sio_return_t	29
4.25	sio_size_t	30
4.26	sio_transfer_len_t	30
5	Range Constants	31
5.1	SIO_MAX_ASYNC_OUTSTANDING	31
5.2	SIO_MAX_COUNT	31
5.3	SIO_MAX_LABELLEN	31
5.4	SIO_MAX_NAMELEN	31
5.5	SIO_MAX_OFFSET	32
5.6	SIO_MAX_OPEN	32
5.7	SIO_MAX_SIZE	32
5.8	SIO_MAX_TRANSFERLEN	32
6	File Attributes	33
6.1	File Sizes	33
6.2	File Label	34
6.3	File Layout	34
7	Error Reporting	37
7.1	sio_error_string	38
8	Basic Operations	39
8.1	sio_control	40
8.2	sio_open	42
8.3	sio_close	45

CONTENTS	3
8.4 sio_unlink	46
8.5 sio_test	47
8.6 sio_rename	50
9 Synchronous File I/O	53
9.1 sio_sg_read, sio_sg_write	55
10 Asynchronous File I/O	59
10.1 sio_async_sg_read, sio_async_sg_write	60
10.2 sio_async_status_any	63
10.3 sio_async_cancel_all	66
11 File Access Pattern Hints	69
11.1 Ordered Hints	71
11.2 Unordered Hints	73
11.3 sio_hint, sio_hint_by_name	75
12 Client Cache Control	77
13 Control Operations	81
13.1 SIO_CTL_GetSize, SIO_CTL_SetSize	82
13.2 SIO_CTL_GetAllocation	83
13.3 SIO_CTL_GetPreallocation, SIO_CTL_SetPreallocation	84
13.4 SIO_CTL_GetCachingMode, SIO_CTL_SetCachingMode	86
13.5 SIO_CTL_Propagate	87
13.6 SIO_CTL_Refresh	89
13.7 SIO_CTL_Sync	90
13.8 SIO_CTL_GetLayout, SIO_CTL_SetLayout	91
13.9 SIO_CTL_GetLabel, SIO_CTL_SetLabel	92
13.10 SIO_CTL_GetConsistencyUnit	94
14 Extension Support	95
14.1 Static Constants	96
14.1.1 Extension Support Constants	96
14.1.2 Extension Identifiers	97
14.2 sio_query_extension	98
14.3 Sample Code to Check for Extension Presence	99

15 Extension: Collective I/O	101
15.1 Motivation	101
15.2 High Level Look	101
15.3 New Data Types	103
15.3.1 sio_coll_handle_t	103
15.3.2 sio_coll_participant_t	103
15.3.3 sio_colliteration_t	103
15.4 New Range Constants	104
15.4.1 SIO_MAX_COLLITERATIONS	104
15.4.2 SIO_MAX_COLLPARTICIPANTS	104
15.4.3 SIO_MAX_COLL_OUTSTANDING	104
15.5 New Functions	105
15.5.1 sio_coll_define	106
15.5.2 sio_coll_join	109
16 Extension: Fast Copy	113
16.1 SIO_CTL_FastCopy	114
A Result codes (for sio_return_t)	119
B Sample Derived Interfaces	123
B.1 Synchronous I/O	124
B.2 Asynchronous I/O	126
B.3 Cache Consistency	128

1 Context

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

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

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

22 Perhaps unavoidably, this document is more about the description of inter-
23 faces than it is about their rationalizations. We apologize in advance for your
24 many unanswered questions.

25 1 Introduction

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

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

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

54 Our two-level approach arises from the conflicting goals of some aspects of
 55 different extended interfaces. For example, in a discussion of the commonal-
 56 ities between IBM's PIOFS and Intel's PFS in February 1995, we identified
 57 little more than the basic UNIX functions in common. Largely this is be-

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

58 cause IBM had chosen to support the concept of dynamic partitioning and
59 subfiles, while Intel supported a set of file modes to define the semantics of
60 parallel access. Our two-level approach moves the implementation of the spe-
61 cial character of these parallel file systems (Intel I/O modes or IBM subfiles)
62 to high-level libraries and proposes a low-level interface capable of efficiently
63 supporting both of these and other specialized parallel file system function
64 sets. The approach follows CMU's December 1994 suggestion, in that the
65 new interfaces are low level, but are powerful for implementing high-level
66 parallel I/O libraries.

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

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

83 1.1 Independent Messaging and 84 Minimal Synchronization

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

²We 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.

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

97 Another guiding principle in the design of this API is to discourage unnec-
98 essary synchronization of the client applications or of the client agents of
99 the parallel file system. To this end, this API is designed to admit efficient
100 low-level parallel file system implementations which restrict internal commu-
101 nication to a single client and the parallel file system server(s) responsible
102 for a particular file. That is, this API does not require that client agents of
103 the parallel file system directly communicate. This means that a compliant
104 parallel file system implementation need not provide coherent distributed
105 shared memory, shared file pointer synchronization, or collective I/O bar-
106 rier synchronization. As described below, distributed shared memory may
107 be avoided with application-managed weakly consistent caches and collective
108 I/O barrier synchronization can be made implicit by requiring the applica-
109 tion to distribute an opaque collective I/O handle defined by the parallel file
110 system.

111 1.2 No Shared File Pointers

112 One of the original points of disagreement in the development of the API was
113 support for shared file pointers. Some parallel file systems exploit shared file
114 pointers extensively while others avoid this implicit synchronization as much
115 as possible. The position of this API is similar to the latter: that shared
116 file pointers can require extensive synchronization of the client agents of the
117 parallel file system; that they implicitly synchronize the application's tasks;
118 and that they can easily lead to excessive synchronization, slowing the appli-
119 cation. Further, we contend that if this level of application synchronization
120 is valuable, it should be provided by the higher level parallel file system li-
121 braries which may have access to peer-to-peer messaging systems and can be

122 customized to specific applications' needs. For these reasons this API does
123 not support shared file pointers; in fact, it does not support file pointers at
124 all, requiring the offsets for all I/O operations to be explicitly provided.

125 1.3 Scatter-Gather Transfers

126 Batching transfers is a powerful strategy for improving performance. A par-
127 allel file system implementation can be expected to try to batch accesses to
128 the disk, transfers between machine nodes, and buffer manipulations. Tra-
129 ditional UNIX read-write interfaces transfer contiguous file regions and con-
130 tiguous memory regions, dramatically reducing batching opportunities for ap-
131 plications that manipulate large, non-contiguous data regions. Correspond-
132 ingly, a principle extension for high-performance file systems is the compact
133 representation of transfers of non-contiguous regions, commonly known as
134 scatter-gather. In the core of this API proposal, the expressive power of
135 scatter-gather is limited to a list of strided (vector) regions. ³

136 1.4 Asynchronous I/O

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

139 1.5 I/O Controls

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

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

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

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

154 1.6 Client Caching

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

171 Note that this API makes no requirement that a low-level parallel file sys-
172 tem implementation control or even detect unintentional read-write sharing,
173 that is, read-write sharing by tasks that are parts of multiple uncoordinated
174 parallel applications. In situations like this, which are common to many file
175 systems, the atomicity of file creation can be used by higher level tools to
176 provide simple advisory locks by using the existence of a file to signify a held
177 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.

254 2 Document Conventions

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

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

263 2.1 Typesetting Conventions

264 Type definitions, functions definitions, and constants (including control op-
265 eration identifiers) are typeset in the **bold** font.

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

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

270 2.2 Definition of Terms

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

277 Throughout this document, several words or phrases are used to indicate
278 how given functionality must be used or implemented. For clarity, they are
279 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

313 different ways. Unspecified behavior must not have harmful permanent
314 effects on the application or its data, and should be documented in in-
315 dividual implementations' documentation. Portable applications must
316 not rely on unspecified behavior causing the same results on multiple
317 file system implementations.

318 **2.3 How to Read this Document**

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

322 **3 The `sio_fs.h` Include File**

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

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

333 4 Data Types

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

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

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

345 4.1 File Descriptor

346 All file descriptors are described as being of type **int**, primarily for compati-
 347 bility with other systems (including UNIX) which use **ints** as file descriptors.
 348 A task may have up to **SIO_MAX_OPEN** parallel files open at any given
 349 time.

350 4.2 File Name

351 All file names are character strings terminated by a byte with the value
 352 zero, and are described being of type **const char ***. (They must never be
 353 modified by the system, and thus are **const**.) File names must not be longer
 354 than **SIO_MAX_NAME_LEN** characters, including the terminating zero
 355 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).

380 4.7 **sio_caching_mode_t**

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

383 4.8 **sio_control_t**

```
384     typedef struct {  
385         sio_control_flags_t flags;  
386         sio_control_op_t op_code;  
387         void *op_data;  
388         sio_return_t result;  
389     } sio_control_t;
```

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

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

397 4.9 **sio_control_flags_t**

398 This is an unsigned integral type used as a set of bits. Cur-
399 rently it can contain one of **SIO_CONTROL_MANDATORY** or
400 **SIO_CONTROL_OPTIONAL**. These flags indicate whether failure of
401 this control operation will cause the entire set of control operations to fail,
402 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

426 sometimes called a “strided” access pattern. Common matrix decompositions
 427 can be described with such data structures.

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

433 4.14 *sio_hint_t*

```

434     typedef struct {
435         sio_hint_flags_t flag;
436         sio_file_io_list_t *io_list;
437         sio_count_t list_len;
438         void *arg;
439         sio_size_t arg_len;
440     } sio_hint_t;
```

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

448 4.15 *sio_hint_class_t*

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

454 and **SIO_HINT_CLASS_UNORDERED** constants to describe manda-
 455 tory hint classes, and reserves constants whose names begin with with
 456 **SIO_HINT_CLASS_VEND_** for use by vendors. See Section 11 for more
 457 details about hints and hint classes.

458 4.16 **sio_hint_flags_t**

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

462 4.17 **sio_label_t**

```
463     typedef struct {
464         sio_size_t size;
465         void *data;
466     } sio_label_t;
```

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

472 4.18 **sio_layout_t**

```
473     typedef struct {
474         sio_layout_flags_t flags;
475         sio_count_t stripe_width;
476         sio_size_t stripe_depth;
477         sio_layout_algorithm_t algorithm;
478         void * algorithm_data;
```

```
479     } sio_layout_t;
```

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

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

492 4.19 **sio_layout_algorithm_t**

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

502 Layout algorithm names beginning with
 503 **SIO_LAYOUT_ALGORITHM_VEND_** are reserved for use by vendors.
 504 All other names beginning with **SIO_LAYOUT_ALGORITHM_** are re-
 505 served for future use by this API.

506 4.20 `sio_layout_flags_t`

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

511 4.21 `sio_mem_io_list_t`

```
512     typedef struct {
513         void *addr;
514         sio_size_t size;
515         sio_size_t stride;
516         sio_count_t element_cnt;
517     } sio_mem_io_list_t;
```

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

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

530 4.22 `sio_mode_t`

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

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

536 4.23 `sio_offset_t`

537 This is a signed integral type whose absolute value is in the range
538 `[0...SIO_MAX_OFFSET]`.⁴ This type is signed to allow an offset vari-
539 able to be decremented in a loop, and have the loop terminate when the
540 variable becomes negative.

541 4.24 `sio_return_t`

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

545 Other values indicate specific errors which have been encountered in pro-
546 cessing this request (the enumeration of standard error codes is included in
547 Appendix A). Error code names beginning with **SIO_ERR_VEND_** may be
548 used by vendors for vendor-specific error codes. All other error code names,
549 beginning with **SIO_ERR** are reserved for future use by this API. At least
550 16384 error codes (including 0, for **SIO_SUCCESS**) must be available for
551 use by this API.

⁴We 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.

⁵An 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.

552 4.25 `sio_size_t`

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

557 4.26 `sio_transfer_len_t`

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

563 5 Range Constants

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

567 5.1 SIO_MAX_ASYNC_OUTSTANDING

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

571 5.2 SIO_MAX_COUNT

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

575 5.3 SIO_MAX_LABEL_LEN

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

580 5.4 SIO_MAX_NAME_LEN

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

583 5.5 SIO_MAX_OFFSET

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

586 5.6 SIO_MAX_OPEN

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

591 5.7 SIO_MAX_SIZE

592 This constant specifies the maximum size in bytes of a variety of objects
593 in the API. The minimum value is $2^{31} - 1$, and the recommended value is
594 $2^{63} - 1$.

595 5.8 SIO_MAX_TRANSFER_LEN

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

601 6 File Attributes

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

607 6.1 File Sizes

608 The **logical size** of an SIO file is the number of bytes from the begin-
609 ning of the file (offset zero) to the end of the file (the largest offset from
610 which data can be read successfully). The file may contain regions which
611 have not yet been written (referred to as “holes”), which are read as ze-
612 ros. The logical size can be increased or decreased with the control oper-
613 ation **SIO_CTL_SetSize** (see Section 13). Decreasing the logical size via
614 **SIO_CTL_SetSize** corresponds to truncating the file, and increasing it cre-
615 ates a hole extending from the previous end of file to the new end of file. A
616 file’s logical size can also be increased by writing data past the current end
617 of file.

618 The **physical size** of an SIO file is the amount of physical storage in bytes
619 allocated to store the file data (excluding metadata). It may be different
620 from the logical size of the file because of fixed size allocation blocks and be-
621 cause each implementation has the freedom to store data in any appropriate
622 manner, including not storing the content of holes and the use of data com-
623 pression techniques. The user has no direct control over the file’s physical
624 size.

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

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

632 6.2 File Label

633 The **file label** of an SIO file is a part of the file's metadata that is acces-
634 sible to the user for storing descriptive information about the file without
635 keeping a header in the file itself. Labels are intended to support interoper-
636 ability by associating information about a file's representation (including
637 file type, version, writing application, etc) with the file itself. Labels are not
638 necessarily the same length in all implementations, but must always be long
639 enough to record a maximum length file name for that implementation. This
640 allows representation information too large to fit in a file label to be stored
641 in a separate file named in the file label. The size of a label is given in the
642 **sio_label_t** containing the label. This size is at least as large as an im-
643 plementation's longest name which must be at least 256 bytes. The maxi-
644 mum size of a label in any specific implementation is given by the constant
645 **SIO_MAX_LABEL_LENGTH** and is recommended to be at least 1024
646 bytes.

647 6.3 File Layout

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

656 A given file layout consists of the number of parallel storage devices over
657 which the file data are striped, the number of contiguous bytes constituting
658 each striping unit, and the algorithm which specifies the striping pattern of
659 the striping units. For example, a simple striping pattern on four storage

660 devices using a striping unit of 1024 bytes would look like the following (the
 661 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
⋮	⋮	⋮	⋮

662

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

669 7 Error Reporting

670 To make it easier for applications to deal with SIO error codes, the function
671 **sio_error_string()** is provided. This function takes a **sio_return_t** value
672 and returns a **const char ***. The **sio_error_string** function maps error codes
673 to meaningful error strings. When passed an error code that is not defined
674 by the implementation, **sio_error_string()** must return a string indicating
675 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.

692 8 Basic Operations

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

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

702 Three of the functions defined in this section, **sio_open()**, **sio_control()**,
703 and **sio_test()**, allow the application to specify a set of controls to be applied
704 to a file. Because **sio_control()** provides the simplest introduction to the
705 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:

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

744 If the **sio_control()** operation fails for any reason, then all of the
745 control operations in *Ops* are annulled, that is, they have no per-
746 manent effect on the file system. If **sio_control()** fails, none of
747 the controls will have their *result* field set to **SIO_SUCCESS**. In
748 this case, the implementation may set the result field of a partic-
749 ular control to a control-specific error code if that control would
750 have failed or if the control caused the **sio_control()** to fail, or to
751 **SIO_ERR_CONTROL_WOULD_HAVE_SUCCEEDED** if that
752 control would have succeeded had the **sio_control()** operation not
753 failed, or to **SIO_ERR_CONTROL_NOT_ATTEMPTED** if the
754 **sio_control()** failed before the implementation checked whether or not
755 that control would have succeeded.

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

757 Return Values

758 **SIO_SUCCESS**

759 All mandatory control operations succeeded.

760 **SIO_ERR_CONTROL_FAILED**

761 At least one of the mandatory control operations failed.

762 **SIO_ERR_CONTROLS_CLASH**

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

768 **SIO_ERR_INVALID_DESCRIPTOR**

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

770 8.2 sio_open

771 Purpose

772 Open a file for reading and/or writing.

773 Syntax

```
774 #include <sio.fs.h>
775 sio_return_t sio_open(int *FileDescriptorPtr, const char *Name,
776                      sio_mode_t Mode,
777                      sio_control_t *ControlOps,
778                      sio_count_t ControlOpCount);
```

779 Parameters

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

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

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

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

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

791 Description

792 This function takes a logical file name, and produces a file de-
793 scriptor which supports reading and/or writing, depending on the
794 value of *Mode*. If the named file does not exist and *Mode* has the
795 **SIO_MODE_CREATE** bit set, then the file will be created; if
796 the bit is not set then **SIO_ERR_FILE_NOT_FOUND** will be re-
797 turned. If **SIO_MODE_CREATE** is set and the file already exists,
798 **SIO_ERR_ALREADY_EXISTS** will be returned.

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

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

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

811 Return Codes

812 **SIO_SUCCESS**

813 The open succeeded.

814 **SIO_ERR_ALREADY_EXISTS**

815 **SIO_MODE_CREATE** was specified and the file already exists.

816 **SIO_ERR_CONTROL_FAILED**

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

819 **SIO_ERR_CONTROLS_CLASH**

820 Some of the mandatory control operations specified are incompat-
821 ible with each other and cannot be performed together by this
822 implementation.

823 **SIO_ERR_FILE_NOT_FOUND**

824 The file did not exist and **SIO_MODE_CREATE** was not spec-
825 ified.

826 **SIO_ERR_INVALID_FILENAME**

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

828 **SIO_ERR_IO_FAILED**

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

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

833 8.3 **sio_close**

834 **Purpose**

835 Close a previously opened file.

836 **Syntax**

```
837 #include <sio.fs.h>  
838 sio_return_t sio_close(int FileDescriptor);
```

839 **Parameters**

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

841 **Description**

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

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

854 **Return Codes**

855 **SIO_SUCCESS**

856 The close succeeded.

857 **SIO_ERR_INVALID_DESCRIPTOR**

858 The *FileDescriptor* parameter does not refer to a valid file descrip-
859 tor previously returned by **sio_open()**.

860 **SIO_ERR_IO_FAILED**

861 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.

889 8.5 *sio_test*

890 Purpose

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

893 Syntax

```
894 #include <sio_fs.h>
895 sio_return_t sio_test(const char *Name, sio_mode_t Mode,
896                      sio_control_t *ControlOps,
897                      sio_count_t ControlOpCount);
```

898 Parameters

899 *Name* The name of the target file.

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

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

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

906 Description

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

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

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

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

918 **SIO_CTL_GetPreallocation** **SIO_CTL_GetLayout**
 919 **SIO_CTL_GetLabel** **SIO_CTL_GetConsistencyUnit**.

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

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

926 If this function fails for any reason, then the result codes of the indi-
 927 vidual *Ops* are set in the same manner that **sio_open()** sets the result
 928 codes of its *Ops*.

929 Return Codes

930 **SIO_SUCCESS**
 931 The test succeeded.

932 **SIO_ERR_ALREADY_EXISTS**
 933 **SIO_MODE_CREATE** was specified and the file already exists.

934 **SIO_ERR_CONTROL_FAILED**
 935 At least one of the mandatory control operations would have
 936 failed.

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

940 **SIO_ERR_CONTROLS_CLASH**
 941 Some of the mandatory control operations specified are incompat-
 942 ible with each other and cannot be performed together by this
 943 implementation.

944 **SIO_ERR_FILE_NOT_FOUND**
 945 The file did not exist and **SIO_MODE_CREATE** was not spec-
 946 ified.

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

SIO_ERR_IO_FAILED

949

950

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

951

SIO_ERR_MAX_OPEN_EXCEEDED

952

953

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.

983

SIO_ERR_IO_FAILED

984

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.

1018 Note that the file system need not access the file or memory in canonical
1019 order. Data can be accessed in the file or memory in any sequence as preferred
1020 by the file system to optimize performance. The canonical sequence of file
1021 regions is used only to compute the association of the file data with memory
1022 regions.

1023 If the source list (i.e. the memory buffer during a write or the file buffer
1024 during a read) contains the same region more than once then its data will
1025 be copied into the destination buffer multiple times. If the destination list
1026 contains the same region more than once then the resulting contents of the
1027 duplicated region are undefined.⁶

1028 Applications must not access an I/O operation's memory buffer while the
1029 operation is in progress. For example, a thread in a multi-threaded appli-
1030 cation must not read or write a buffer while another thread has an I/O in
1031 progress using the same buffer. Failure to avoid such accesses may corrupt
1032 the task and/or file in undefined ways, including leaving the contents of the
1033 file corrupted or causing the task to fault. Applications that wish to share
1034 I/O buffers between threads must explicitly synchronize the threads' accesses
1035 to those buffers.

1036 It is expected that many users of this API will desire simpler interfaces to
1037 this functionality. In addition to the basic POSIX interfaces, the interfaces in
1038 Appendix B are easily built on the interfaces provided in this API. These, or
1039 similar simplified interfaces, could easily be provided by a high-level library,
1040 and are not defined by this API.

⁶No 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.

1041 **9.1 *sio_sg_read, sio_sg_write***1042 **Purpose**

1043 Transfer data between a file and memory.

1044 **Syntax**

```
1045     #include <sio.fs.h>
1046     sio_return_t sio_sg_read(int FileDescriptor,
1047                             const sio_file_io_list_t *FileList,
1048                             sio_count_t FileListLength,
1049                             const sio_mem_io_list_t *MemoryList,
1050                             sio_count_t MemoryListLength,
1051                             sio_transfer_len_t *TotalTransferred);
1052     sio_return_t sio_sg_write(int FileDescriptor,
1053                               const sio_file_io_list_t *FileList,
1054                               sio_count_t FileListLength,
1055                               const sio_mem_io_list_t *MemoryList,
1056                               sio_count_t MemoryListLength,
1057                               sio_transfer_len_t *TotalTransferred);
```

1058 **Parameters**

1059 *FileDescriptor* The file descriptor of an open file.

1060 *FileList* Specification of file data to be read or written.

1061 *FileListLength* Number of elements in *FileList*.

1062 *MemoryList* Specification of the memory buffer containing data to be
1063 read or written.

1064 *MemoryListLength* Number of elements in *MemoryList*.

1065 *TotalTransferred* Used to return the total number of bytes read or writ-
1066 ten.

1067 **Description**

1068 These functions move data between a list of file locations and a list
1069 of memory locations. All I/O must be done to a single file, in the
1070 *FileDescriptor* argument.

1071 The mapping between the collection of file regions specified by *FileList*
 1072 and the collection of memory byte regions specified by *MemoryList*
 1073 is in matching indices in the canonical ordering of the corresponding
 1074 **sio_file_io_list_t** and **sio_mem_io_list_t**.

1075 If the total transfer cannot be completed because a file address is not
 1076 valid (i.e. reading beyond the end of the file), these interfaces will
 1077 complete successfully, and return in *TotalTransferred* the index of the
 1078 first byte in the canonical ordering that could not be transferred (fol-
 1079 lowing the UNIX example); bytes preceding this index in the canonical
 1080 ordering have been transferred successfully and bytes following (and
 1081 including) it may or may not have been transferred successfully.

1082 Implementations may return a value less than the actual amount trans-
 1083 ferred if the operation was not successful; in particular, an implemen-
 1084 tation may indicate that zero bytes were transferred successfully on all
 1085 failures.

1086 Return Codes

1087 **SIO_SUCCESS**

1088 The function succeeded.

1089 **SIO_ERR_INCORRECT_MODE**

1090 The mode of the file descriptor does not permit the I/O.

1091 **SIO_ERR_INVALID_DESCRIPTOR**

1092 *FileDescriptor* does not refer to a valid file descriptor.

1093 **SIO_ERR_INVALID_FILE_LIST**

1094 The file regions described by *FileList* are invalid, e.g. they contain
 1095 illegal addresses.

1096 **SIO_ERR_INVALID_MEMORY_LIST**

1097 The memory regions described by *MemoryList* are invalid, e.g.
 1098 they contain illegal addresses.

1099 **SIO_ERR_IO_FAILED**

1100 A physical I/O error caused the function to fail.

1101 **SIO_ERR_NO_SPACE**

1102 The file system ran out of space while trying to extend the file.

1103 **SIO_ERR_UNEQUAL_LISTS**

1104 The number of bytes in *MemoryList* and *FileList* are not equal.

1105 10 Asynchronous File I/O

1106 Asynchronous I/O allows a single-threaded task to issue concur-
1107 rent I/O requests. The parallel file system supports up to
1108 **SIO_MAX_ASYNC_OUTSTANDING** (see Section 5.1) asynchronous
1109 I/Os at a time for each task. Asynchronous I/O functions merely initiate an
1110 I/O, returning to the task a handle that may be used by the task to wait for
1111 the I/O to complete, to check its status of the I/O, or to cancel the I/O.

1112 These handles are of type **sio_async_handle_t**, which is an opaque type
1113 defined by the system. Only the task that issued the asynchronous I/O is
1114 able to use the **sio_async_handle_t** associated with the I/O to retrieve the
1115 status of or cancel the I/O. Other tasks that wish to retrieve the status of or
1116 cancel an I/O must contact the task that initiated the I/O.

1117 10.1 sio_async_sg_read, sio_async_sg_write

1118 Purpose

1119 Asynchronously transfer data between a file and memory.

1120 Syntax

```
1121 #include <sio_fs.h>
1122 sio_return_t sio_async_sg_read(int FileDescriptor,
1123                               const sio_file_io_list_t *FileList,
1124                               sio_count_t FileListLength,
1125                               const sio_mem_io_list_t *MemoryList,
1126                               sio_count_t MemoryListLength,
1127                               sio_async_handle_t *Handle);
1128 sio_return_t sio_async_sg_write(int FileDescriptor,
1129                                const sio_file_io_list_t *FileList,
1130                                sio_count_t FileListLength,
1131                                const sio_mem_io_list_t *MemoryList,
1132                                sio_count_t MemoryListLength,
1133                                sio_async_handle_t *Handle);
```

1134 Parameters

1135 *FileDescriptor* The file descriptor of an open file.

1136 *FileList* Specification of file data to be read or written.

1137 *FileListLength* Number of elements in *FileList*.

1138 *MemoryList* Specification of the memory buffer containing data to be
1139 read or written.

1140 *MemoryListLength* Number of elements in *MemoryList*.

1141 *Handle* Handle returned by the operation, which can be used later to
1142 determine the status of the I/O, to wait for its completion, or to
1143 cancel it.

1144 Description

1145 These functions behave similarly to **sio_sg_read()** and **sio_sg_write()**.
 1146 A successful return, however, indicates only that the I/O has been
 1147 queued for processing by the parallel file system.

1148 *Handle* is a task-specific value which may be used to poll for comple-
 1149 tion, block until the I/O completes, or cancel the I/O. The handle re-
 1150 mains valid until either the task completes, or **sio_async_status_any()**
 1151 indicates that the I/O transfer associated with *Handle* is no longer
 1152 in progress. While a handle is valid it counts towards the
 1153 **SIO_MAX_ASYNC_OUTSTANDING** asynchronous I/Os that a
 1154 task may have.

1155 As in synchronous I/O, applications must neither access nor modify the
 1156 contents of a memory buffer while an asynchronous I/O is in progress
 1157 on that buffer. Doing so may leave the buffer and/or the file in an
 1158 undefined state, and may cause the task to fault. See Section 9 for
 1159 details.

1160 Return Codes

1161 **SIO_SUCCESS**

1162 The function succeeded.

1163 **SIO_ERR_INCORRECT_MODE**

1164 The mode of the file descriptor does not allow the I/O.

1165 **SIO_ERR_INVALID_DESCRIPTOR**

1166 *FileDescriptor* does not refer to a valid file descriptor.

1167 **SIO_ERR_INVALID_FILE_LIST**

1168 The file regions described by *FileList* are invalid, e.g. they contain
 1169 illegal addresses. Implementations may defer returning this error
 1170 until **sio_async_status_any()** is invoked on the I/O.

1171 **SIO_ERR_INVALID_MEMORY_LIST**

1172 The memory regions described by *MemoryList* are invalid, e.g.
 1173 they contain illegal addresses. Implementations may defer return-
 1174 ing this error until **sio_async_status_any()** is invoked on the
 1175 I/O.

1176 **SIO_ERR_IO_FAILED**

1177 A physical I/O error caused the function to fail.

1178 SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED

1179 The I/O request could not be initiated because doing so would
1180 cause the calling task's number of outstanding asynchronous I/Os
1181 to exceed the limit.

1182 SIO_ERR_NO_SPACE

1183 The file system ran out of space while trying to extend the
1184 file. Implementations may defer returning this error until
1185 *sio_async_status_any()* is invoked on the I/O.

1186 SIO_ERR_UNEQUAL_LISTS

1187 The number of bytes in *MemoryList* and *FileList* are not
1188 equal. Implementations may defer returning this error until
1189 *sio_async_status_any()* is invoked on the I/O.

1190 **10.2 *sio_async_status_any***1191 **Purpose**

1192 Get the status of asynchronous I/Os.

1193 **Syntax**

```

1194     #include <sio.fs.h>
1195     sio_return_t sio_async_status_any(
1196                                     sio_async_handle_t *HandleList,
1197                                     sio_count_t HandleListLength,
1198                                     sio_count_t *Index,
1199                                     sio_async_status_t *Status,
1200                                     sio_async_flags_t Flags);

```

1201 **Parameters**

1202 *HandleList* An array of **sio_async_handle_t**s identifying the asyn-
 1203 chronous I/Os for which status is desired.

1204 *HandleListLength* The number of elements in *HandleList*.

1205 *Index* Used to return the index of handle within *HandleList* for which
 1206 status is returned.

1207 *Status* Pointer to an **sio_async_status_t** to be filled in.

1208 *Flags* Determines whether or not the operation blocks or returns im-
 1209 mediately.

1210 **Description**

1211 This function retrieves the status of one of the asynchronous I/Os spec-
 1212 ified by *HandleList*. The index of the handle within *HandleList* for
 1213 which the status is returned is stored in *Index*. The system may return
 1214 the status for any of the handles, provided that if any of the I/Os are
 1215 complete or canceled, then the status for one of these I/Os is returned
 1216 and not the status of an I/O that is still in progress.

1217 It is important to note that once the status for an I/O indi-
 1218 cates that the I/O is no longer in progress (i.e. it completed
 1219 or was canceled) the handle for the I/O is no longer valid. If

1220 it is subsequently passed to `sio_async_status_any()` the value
 1221 **SIO_ERR_INVALID_HANDLE** will be returned if the handle is
 1222 still invalid, otherwise the status of the new asynchronous I/O will be
 1223 returned if the handle has been reused.

1224 The task may place a dummy handle in the **HandleList** by setting the
 1225 entry to **SIO_ASYNC_DUMMY_HANDLE**. The system ignores a
 1226 handle with this value, allowing the task to retrieve the status for a set
 1227 of handles using the same *HandleList* array, by replacing the handle for
 1228 the I/O just finished with the dummy value.

1229 If the *Flags* parameter includes **SIO_ASYNC_BLOCKING**, this
 1230 function will not return until at least one of the I/Os has completed. If
 1231 it includes **SIO_ASYNC_NONBLOCKING**, this function returns
 1232 immediately, regardless of whether or not one of the I/Os has com-
 1233 pleted.

1234 *Note to implementors:* When an I/O is canceled the *count* field in *Status*
 1235 will contain the number of bytes guaranteed to have been transferred
 1236 prior to the cancellation. Implementations may always set this value
 1237 to zero, indicating that none of the bytes are guaranteed to have been
 1238 transferred.

1239 Status Results

1240 The following values are returned in the *result* field of the *Status* struc-
 1241 ture, indicating the status of the I/O:

1242 SIO_SUCCESS

1243 The I/O has completed or been canceled. The *count* field contains
 1244 the number of bytes transferred.

1245 SIO_ERR_INVALID_FILE_LIST

1246 The file regions described by the *FileList* parameter passed to the
 1247 function that initiated the I/O are invalid, e.g. they contain illegal
 1248 addresses.

1249 SIO_ERR_INVALID_MEMORY_LIST

1250 The memory regions described by the *MemoryList* parameter
 1251 passed to the function that initiated the I/O are invalid, e.g. they
 1252 contain illegal addresses.

1253 **SIO_ERR_IO_CANCELED**

1254 The I/O was canceled without completing. The *count* field con-
1255 tains the number of bytes *guaranteed* to have been transferred
1256 successfully prior to the cancellation. Implementations may set
1257 *count* to zero.

1258 **SIO_ERR_IO_FAILED**

1259 A physical I/O error caused the function to fail.

1260 **SIO_ERR_IO_IN_PROGRESS**

1261 The I/O is still in progress.

1262 **SIO_ERR_MIXED_COLL_AND_ASYNC**

1263 The implementation does not support mixing of asynchronous and
1264 collective I/O handles, and a mix of handle types was supplied.

1265 **SIO_ERR_NO_SPACE**

1266 The file system ran out of space while trying to extend the file.

1267 **SIO_ERR_UNEQUAL_LISTS**

1268 The size of the memory buffer doesn't match size of the file regions
1269 to be accessed.

1270 **Return Values**1271 **SIO_SUCCESS**

1272 An I/O has completed or been canceled, the index and result of
1273 which are stored in *Index* and *Status*, respectively.

1274 **SIO_ERR_INVALID_HANDLE**

1275 At least one of the elements of *HandleList* is neither a valid handle
1276 for an asynchronous I/O nor a dummy handle. *Index* will contain
1277 the index of one of the invalid handles.

1278 **SIO_ERR_IO_IN_PROGRESS**

1279 All I/Os are still in progress.

1280 10.3 **sio_async_cancel_all**

1281 **Purpose**

1282 Request that a collection of asynchronous I/Os be canceled.

1283 **Syntax**

```
1284 #include <sio_fs.h>
1285 sio_return_t sio_async_cancel_all(
1286                                     sio_async_handle_t *HandleList,
1287                                     sio_count_t HandleListLength);
```

1288 **Parameters**

1289 *HandleList* An array of **sio_async_handle_t**s identifying the asyn-
1290 chronous I/Os to be canceled.

1291 *HandleListLength* The number of elements in *HandleList*.

1292 **Description**

1293 This function is used to request that asynchronous I/Os be canceled.
1294 It is not guaranteed that the I/O will not complete in full or in part;
1295 an implementation may ignore cancel requests. A canceled read leaves
1296 the contents of the I/O's memory buffer undefined. Likewise, if a write
1297 is canceled, the contents of the regions of the file regions being written
1298 are undefined.

1299 The status of a canceled request remains available until an
1300 **sio_async_status_any()** reports its completion. An application
1301 should test for this status or its maximum outstanding asynchronous
1302 I/Os will appear to diminish.

1303 *Note to implementors:*

1304 An implementation may ignore cancellation requests altogether. In this
1305 case a call to **sio_async_status_any()** on an I/O that whose cancel-
1306 lation was requested should return the normal, uncanceled completion
1307 status of the I/O.

1308 *Note to implementors:* Implementations are encouraged to avoid
1309 reusing the same handles for different asynchronous I/Os within the

1310 same task. A handle becomes invalid once the I/O is no longer in
1311 progress and its status has been retrieved, but bugs may cause a task
1312 to use such an invalid handle. If the system has reassigned the handle
1313 to a new I/O the task will end up affecting the new I/O, instead of
1314 getting an invalid handle error. Although this behavior is caused by
1315 a bug in the application, avoiding reuse of handles will help track the
1316 problem.

1317 **Return Values**

1318 **SIO_SUCCESS**

1319 The request for cancellation was accepted. This does not mean
1320 that the I/Os were actually canceled.

1321 **SIO_ERR_INVALID_HANDLE**

1322 One of the elements in *HandleList* is not a valid handle for an
1323 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 (ac-

1357 cesses that have been hinted, but will not actually be performed) and should
1358 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

1391 previously issued by this task for which no corresponding for transfer
1392 request has occurred.

1393 A previously issued hint's profile "matches" the current hint's pro-
1394 file if the hints pertain to the same file, and the regions specified by
1395 the *io_list* entry in the **sio_hint_t** structures are the same and the
1396 **SIO_HINT_READ** or **SIO_HINT_WRITE** flag matches.

1397 No more than one of these flags may be specified for each hint.

1398 *Note to implementors:* Implementations are not required to keep track
1399 of "outstanding" hints. The concept of "outstanding" only describes
1400 the application's intent in issuing the hint, and does not describe the
1401 implementation's behavior. In implementations that do not keep track
1402 of "outstanding" hints the **SIO_HINT_CANCEL_NEXT** hint may
1403 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:

1435 SIO_HINT_SEQUENTIAL

1436 The entire region will be accessed in non-overlapping blocks whose
1437 starting offsets increase monotonically.

1438 SIO_HINT_REVERSE

1439 The entire region will be accessed in non-overlapping blocks whose
1440 starting offsets decrease monotonically.

1441 SIO_HINT_RANDOM_COMPLETE

1442 Accesses in the region will have starting addresses and sizes that
1443 vary without pattern but the entire region will be accessed.

1444 SIO_HINT_RANDOM_PARTIAL

1445 Accesses in the region will have starting addresses and sizes that
1446 vary without pattern and the entire region may not be accessed.

1447 SIO_HINT_NO_FURTHER_USE

1448 No further accesses are expected in the region.

1449 SIO_HINT_WILL_USE

1450 All data in the region will be accessed although no explicit pattern
1451 can be predicted or excluded.⁷

1452 Exactly one of these flags must be specified for each hint.

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

1453 **11.3 *sio_hint*, *sio_hint_by_name***1454 **Purpose**

1455 Issue a set of predictions about the future accesses of this task.

1456 **Syntax**

```

1457     #include <sio.fs.h>
1458     sio_return_t sio_hint(int FileDescriptor,
1459                           sio_hint_class_t HintClass,
1460                           const sio_hint_t *Hints,
1461                           sio_count_t HintCount);
1462     sio_return_t sio_hint_by_name(const char *FileName,
1463                                   sio_hint_class_t HintClass,
1464                                   const sio_hint_t *Hints,
1465                                   sio_count_t HintCount);

```

1466 **Parameters**

1467 *FileDescriptor* The file descriptor of an open file to which these hints
 1468 apply.

1469 *FileName* The name of a file, not necessarily an open file, to which
 1470 these hints apply.

1471 *HintClass* The class of the hints being issued.

1472 *Hints* An array of file access pattern hints.

1473 *HintCount* The number of entries in the *Hints* array.

1474 **Description**

1475 This function reports the application's knowledge of future access pat-
 1476 terns to the file system. The purpose of issuing this information is to
 1477 enable optimizations in the dynamic behavior of the parallel file sys-
 1478 tem. This knowledge is expressed as a set of hints, all from the same
 1479 hint class. The interpretation of mixtures of hint types from differ-
 1480 ent hint classes is unspecified. Hints can be applied to an open file
 1481 using *sio_hint()*, or to a named file (which may not be open) using

1482 **sio_hint_by_name()**. Each **sio_hint_t** structure in the *Hints* array
 1483 describes a hint type applied to a list of file regions and optionally
 1484 hint-specific arguments.

1485 If the *size*, *stride*, and *element_cnt* fields for a particular
 1486 **sio_file_io_list_t** in a hint are all zero, then the region being specified
 1487 begins at the offset given by the *offset* field of that **sio_file_io_list_t**
 1488 and continues until the end of the file. The entire contents of a file are
 1489 specified as the region whenever an **sio_file_io_list_t** contains zero in
 1490 the four fields: *offset*, *size*, *stride* and *element_cnt*.

1491 The implementation may not act on any specific hint or on any hints
 1492 at all.

1493 Return Codes

1494 **SIO_SUCCESS**

1495 The function succeeded.

1496 **SIO_ERR_FILE_NOT_FOUND**

1497 The specified file did not exist.

1498 **SIO_ERR_HINT_TYPES_CLASH**

1499 The class of this hint differs from the class of another hint previ-
 1500 ously issued for the same file region.⁸

1501 **SIO_ERR_INVALID_CLASS**

1502 The hint class given in *HintClass* is not a valid hint class.

1503 **SIO_ERR_INVALID_DESCRIPTOR**

1504 *FileDescriptor* does not refer to a valid file descriptor created by
 1505 **sio_open()**.

1506 **SIO_ERR_INVALID_FILENAME**

1507 The name given by *FileName* is invalid.

⁸As 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.

1508 12 Client Cache Control

1509 The basic API includes facilities to control caching of data in client memory.
 1510 The caching interfaces are specified such that it is a valid implementation
 1511 strategy to simply ignore all cache control calls. The only requirement of
 1512 an implementation that ignores these calls is that it must provide strongly
 1513 consistent semantics.

1514 The client caching mode of an SIO file may be specified by including the
 1515 **SIO_CTL_SetCachingMode** control operation when making **sio_open()**
 1516 or **sio_control()** calls.

1517 This API specifies client caching modes with the type **sio_caching_mode_t**,
 1518 which can have the following values:

1519 **SIO_CACHING_NONE**

1520 Completely disable client caching.

1521 **SIO_CACHING_STRONG**

1522 Allow strongly-consistent client caching. The file system may choose
 1523 to provide caching with strong sequential consistency, or provide no
 1524 caching at all.

1525 **SIO_CACHING_WEAK**

1526 Allow weakly-consistent client caching. The file system may provide no
 1527 client caching, strongly-consistent client caching, or weakly-consistent
 1528 client caching.

1529 Caching mode names beginning with **SIO_CACHING_** are reserved for
 1530 future use by this API. Vendors may define their own caching modes by
 1531 naming them with the prefix **SIO_CACHING_VEND_**.

1532 An SIO parallel file system implementation's default client caching
 1533 mode must provide sequential consistency. That is, it must be either
 1534 **SIO_CACHING_NONE**, **SIO_CACHING_STRONG**, or a vendor-
 1535 defined mode that provides strong sequential consistency.

1536 If client caching is not disabled by using a caching mode of
1537 **SIO_CACHING_NONE**, the file system on a client node is free to main-
1538 tain local copies of file data for both read and write operations.

1539 In a system with strongly-consistent caching, every write forces the client
1540 node to immediately make the file system aware that the file has changed.
1541 This also requires that client nodes either check the validity of cached data
1542 before providing them to applications to satisfy a read, or be notified when-
1543 ever cached or potentially cached data have changed.

1544 On the other hand, weakly-consistent client caching allows the file system to
1545 avoid the messaging and bookkeeping which a sequentially consistent caching
1546 mode mandates, while providing the application with the benefits of caching.
1547 With this form of caching, client nodes may defer exposing all or part of a set
1548 of changes to a file until instructed otherwise by the application. Likewise,
1549 a client node need not confirm the validity of cached data with the server
1550 unless explicitly instructed to do so by the application.

1551 An application informs the file system that data written on a file descriptor
1552 should become visible to other readers via the **SIO_CTL_Propagate** control
1553 operation. If the changed data have not already been exposed to the rest
1554 of the file system, this is done so immediately. Note that all, none, or part
1555 of this changed data may already have been exposed to the rest of the file
1556 system.

1557 Likewise, an application informs the file system that locally cached data may
1558 be stale using the **SIO_CTL_Refresh** control operation. Reads of refreshed
1559 regions of a file are guaranteed to yield either the most current available data,
1560 or data that were not stale at the time of the most recent refresh operation.
1561 That is to say, if the data returned by the read are stale, it was made so *after*
1562 the refresh.

1563 It is assumed that applications using weakly-consistent client caching either
1564 do not share data between nodes, or provide their own internal synchroniza-
1565 tion to coordinate when nodes must propagate and refresh data.

1566 Thus, the way in which a node **A** would write data which are then read by
1567 a node **B** is:

1568 **A** writes data to region **R**

1569 **A** propagates data in region **R**

1570 (Implicit:) **A** and **B** synchronize; **B** becomes aware that new data in region
1571 **R** are available

1572 **B** refreshes data in region **R**

1573 **B** reads data in region **R**

1574 The granularity of caching is known as the *consistency unit*. This defines both
1575 the size and the alignment of the blocks of data within the file for which the
1576 file system insures that all non-conflicting writes are merged into the file.
1577 Tasks on different nodes cannot use weak consistency and achieve consis-
1578 tent parallel updates within a single consistency unit. Any conflicting writes
1579 within a single consistency unit will be resolved by an arbitrary selection
1580 of a winning writer when the data arrive at a server. The size of the con-
1581 sistency unit is implementation specific, and is represented by the constant
1582 **SIO_CACHE_CONSISTENCY_UNIT**. Additionally, the control oper-
1583 ation **SIO_CTL_GetConsistencyUnit** can be used to retrieve the consis-
1584 tency unit for a file descriptor.⁹ Applications should not make any assump-
1585 tions about the size of the consistency unit; it may vary between individual
1586 bytes, cache lines, pages, and file blocks depending upon the implementation
1587 of the file system.

1588 The motivation for providing weakly-consistent client caching as an option
1589 within the parallel file system is to allow parallel applications that could ben-
1590 efit from a decrease in the total amount of data being transferred between
1591 clients and servers to exercise relatively fine-grained control over the consis-
1592 tency of their local caches. **SIO_CTL_Propagate** and **SIO_CTL_Refresh**
1593 operations can be piggy-backed onto synchronization steps that already ex-
1594 ist in parallel applications. These primitives allow application programmers
1595 and toolkit developers the mechanisms necessary to ensure consistency of the
1596 local parallel file system cache, without requiring the parallel file system to
1597 enforce any consistency model itself.

1598 This implementation of weakly-consistent caching is only intended to cope
1599 with sharing among the tasks of a parallel application. To avoid unintended

⁹Currently, 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.

1600 sharing among independent applications, traditional methods based on de-
 1601 tecting conflicts at open time and disabling caching or resorting to strongly-
 1602 consistent caching may be used.

1603 Some implementations may choose not to provide weak client cache consis-
 1604 tency by ignoring a **SIO_CTL_SetCachingMode** operation that specifies
 1605 the **SIO_CACHING_WEAK** mode, as well as the **SIO_CTL_Propagate**
 1606 and **SIO_CTL_Refresh**
 1607 operations. In this case, the **SIO_CTL_GetCachingMode** should re-
 1608 turn a value of **SIO_CACHING_NONE**, **SIO_CACHING_STRONG**,
 1609 or a sequentially-consistent vendor-defined caching mode as appropriate, and
 1610 **SIO_CTL_Propagate** and **SIO_CTL_Refresh** should always return suc-
 1611 cess. (This way, an application which can tolerate weakly-consistent caching
 1612 will not see extraneous errors in its absence.¹⁰)

1613 Note that client caching is controlled on a per-file descriptor basis, so it
 1614 is possible to have a file opened with one client caching mode on one file
 1615 descriptor and with a different mode on another file descriptor.

1616 Descriptions of the **SIO_CTL_GetCachingMode**,
 1617 **SIO_CTL_SetCachingMode**, **SIO_CTL_Propagate**,
 1618 **SIO_CTL_Refresh**, and **SIO_CTL_GetConsistencyUnit** control oper-
 1619 ations are given in Section 13.

1620 *Note to implementors:* The routine **sio_close()** implicitly performs a
 1621 **SIO_CTL_Propagate** on the file descriptor. This causes all cached writes
 1622 to be exposed to the file system at the time the file is closed, if they have
 1623 not been already.

¹⁰Since 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.

1624 13 Control Operations

1625 This section describes the file control operations that can be performed using
1626 the functions **sio_control()**, **sio_open()**, **sio_test()**.

1627 These control operations allow properties of files, file descriptors, and the file
1628 system to be set and retrieved.

1629 Control operations are performed by invoking **sio_open()**, **sio_control()**, or
1630 **sio_test()** with the list of operations to be performed. Each operation de-
1631 scription, an **sio_control_t**, includes the code of the operation to be per-
1632 formed, a pointer to the data to be manipulated by that operation, and
1633 space for a result code. In the following sections, information is provided
1634 about the various operation codes that must be implemented by file systems
1635 that conform to this API.

1636 Operation names beginning with **SIO_CTL_** are reserved for use by this
1637 API. Operation names beginning with **SIO_CTL_VEND_** may be used by
1638 vendors to define vendor-specific operations.

1639 13.1 SIO_CTL_GetSize, SIO_CTL_SetSize

1640 Purpose

1641 Get or set the file's logical size.

1642 Affects

1643 Open file

1644 Parameter Type

1645 Pointer to a `sio_offset_t`.

1646 Description

1647 Applications may query and adjust the logical size (see Section 6.1)
1648 of a file using these control operations. The **SIO_CTL_SetSize** op-
1649 eration causes the logical size of the file to be set to the value in the
1650 `sio_offset_t` pointed to by the *op_data* field of the `sio_control_t`. Set-
1651 ting a file's logical size may change the amount of storage that the file
1652 uses, but is not guaranteed to do so. An application wishing to preal-
1653 locate storage for a file should use the **SIO_CTL_SetPreallocation**
1654 control operation.

1655 The **SIO_CTL_GetSize** operation causes the logical size of the file
1656 being operated on to be placed in the `sio_offset_t` pointed to by the
1657 *op_data* member of the `sio_control_t`.

1658 Result Values

1659 SIO_SUCCESS

1660 The operation succeeded.

1661 SIO_ERR_INCORRECT_MODE

1662 The mode of the file descriptor does not permit the operation.

1663 SIO_ERR_IO_FAILED

1664 A physical I/O error caused the operation to fail.

1665 SIO_ERR_NO_SPACE

1666 The system needs to increase the amount of storage used by the
1667 file but cannot.

1668 **13.2 SIO_CTL_GetAllocation**1669 **Purpose**

1670 Get the file's physical size.

1671 **Affects**

1672 Underlying file.

1673 **Parameter Type**1674 Pointer to a **sio_offset_t**.1675 **Description**

1676 The **SIO_CTL_GetAllocation** operation causes file's physical size
1677 (see Section 6.1) to be placed in the **sio_offset_t** pointed to by the
1678 *op_data* field of the **sio_control_t**.

1679 **Result Values**1680 **SIO_SUCCESS**

1681 The operation succeeded.

1682 **SIO_ERR_INCORRECT_MODE**

1683 The mode of the file descriptor does not permit the operation.

1684 **SIO_ERR_IO_FAILED**

1685 A physical I/O error caused the operation to fail.

1686 13.3 SIO_CTL_GetPreallocation, 1687 SIO_CTL_SetPreallocation

1688 Purpose

1689 Get or set amount of space preallocated for the file.

1690 Affects

1691 Underlying file.

1692 Parameter Type

1693 Pointer to a `sio_offset_t`.

1694 Description

1695 The **SIO_CTL_GetPreallocation** operation causes the amount of
1696 space preallocated (see Section 6.1) for the file being operated on to
1697 be placed in the `sio_offset_t` pointed to by the *op_data* field of the
1698 `sio_control_t`.

1699 The **SIO_CTL_SetPreallocation** operation causes the amount of
1700 space preallocated for the file being operated on to be set to the value
1701 in the `sio_offset_t` pointed to by the *op_data* field of the `sio_control_t`.
1702 A preallocation applies to an open file and will be reset to zero when
1703 the file is closed. While open, writes by other tasks that extend the
1704 physical size of the file may reduce the unconsumed preallocation.

1705 If either the **SIO_CTL_GetPreallocation** operation or the
1706 **SIO_CTL_SetPreallocation** operation is supported, both must be
1707 supported.

1708 Result Values

1709 SIO_SUCCESS

1710 The operation succeeded.

1711 SIO_ERR_INCORRECT_MODE

1712 The mode of the file descriptor does not permit the operation.

1713 SIO_ERR_IO_FAILED

1714 A physical I/O error caused the operation to fail.

1715 **SIO_ERR_NO_SPACE**

1716 There isn't enough free space in the system to satisfy the request.

1717 **SIO_ERR_OP_UNSUPPORTED**

1718 The operation is not supported by the system.

1719 13.4 SIO_CTL_GetCachingMode, 1720 SIO_CTL_SetCachingMode

1721 Purpose

1722 Get or set the file's caching mode.

1723 Affects

1724 File descriptor.

1725 Parameter Type

1726 Pointer to a `sio_caching_mode_t`.

1727 Description

1728 The `SIO_CTL_GetCachingMode` operation causes the caching
1729 mode of the file descriptor to be placed in the `sio_caching_mode_t`
1730 pointed to by the *op_data* field of the `sio_control_t`.

1731 The `SIO_CTL_SetCachingMode` operation causes the caching mode
1732 of the file descriptor to be set to the value of the `sio_caching_mode_t`
1733 pointed to by the *op_data* field of the `sio_control_t`. SIO implementa-
1734 tions which provide support for multiple caching modes may elect not
1735 to provide support for changing the caching mode of an open file.

1736 Result Values

1737 SIO_SUCCESS

1738 The operation succeeded.

1739 SIO_ERR_INCORRECT_MODE

1740 The mode of the file descriptor does not permit the operation.

1741 SIO_ERR_ONLY_AT_OPEN

1742 The system does not allow the caching mode of an open file to
1743 be changed. Caching modes can only be changed as part of
1744 `sio_open()`.

1745 SIO_ERR_OP_UNSUPPORTED

1746 The system does not support `SIO_CTL_SetCachingMode`.

1747 13.5 SIO_CTL_Propagate

1748 Purpose

1749 Force locally cached writes to be made visible to other nodes.

1750 Affects

1751 Cached writes associated with file descriptor.

1752 Parameter Type

1753 Pointer to a `sio_file_io_list_t`.

1754 Description

1755 This operation allows a task to force the parallel file system to make
1756 any data associated with a particular set of byte ranges visible to other
1757 nodes in the system (see Section 12 for information about why this
1758 might be necessary), as specified by the `sio_file_io_list_t` pointed to
1759 by the `op_data` field of the control request. If `op_data` is `NULL`, the
1760 propagation will apply to all bytes in the file. If the `size`, `stride`, and
1761 `element_cnt` fields of the `sio_file_io_list_t` pointed to by the `op_data`
1762 field are all zero, then the set of bytes to be propagated begins at the
1763 offset specified in the `offset` field of the `sio_file_io_list_t` and continues
1764 until the end of the file.

1765 This operation only affects those bytes written via the given file descrip-
1766 tor; if an application writes to a file using more than one file descriptor,
1767 it must perform a propagate operation on each of them to guarantee
1768 the dirty data become visible to other nodes. While it is guaranteed
1769 after a propagate operation completes that all locally cached writes for
1770 the specified file regions have been exposed to the rest of the file sys-
1771 tem, it is *not* guaranteed that some or all the changed data was not
1772 visible to the rest of the file system *prior to* the propagate. That is,
1773 weakly-consistent client caching implies only that cached writes will be
1774 exposed to the rest of the file system *no later than* the completion of
1775 the propagate operation.

1776 Result Values

1777 **SIO_SUCCESS**

1778 The results of all writes on this file descriptor in the specified
1779 region(s) have been exposed to the rest of the file system.

1780 **SIO_ERR_INVALID_FILE_LIST**

1781 `op_data` is not **NULL** nor a pointer to a valid `sio_file_io_list_t`.

1782 13.6 SIO_CTL_Refresh

1783 Purpose

1784 Inform the file system that locally cached data may be invalid.

1785 Affects

1786 Blocks in client's cache containing data for this file.

1787 Parameter Type

1788 Pointer to a `sio_file_io_list_t`.

1789 Description

1790 This operation informs the parallel file system that data cached for
 1791 a file may be stale, that is, superseded by more recent writes (see
 1792 Section 12 for information about why this might be necessary). Future
 1793 reads to the specified client region(s) are guaranteed to not yield data
 1794 that were stale at the time the refresh operation began.¹¹ File region(s)
 1795 are specified by the `sio_file_io_list_t` pointed to by the `op_data` field
 1796 of the control request. If `op_data` is NULL, the operation will apply
 1797 to all bytes in the file. If the `size`, `stride`, and `element_cnt` fields of the
 1798 `sio_file_io_list_t` pointed to by the `op_data` field are all zero then the
 1799 operation applies to the set of bytes beginning at the offset specified
 1800 in the `offset` field of the `sio_file_io_list_t` and ending at the end of the
 1801 file.

1802 Result Values

1803 SIO_SUCCESS

1804 The regions have been refreshed.

1805 SIO_ERR_INVALID_FILE_LIST

1806 `op_data` is not NULL or a pointer to a valid `sio_file_io_list_t`.

¹¹The 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.

1807 13.7 SIO_CTL_Sync

1808 Purpose

1809 Force dirty data to stable storage.

1810 Affects

1811 Blocks written via the file descriptor.

1812 Parameter Type

1813 None

1814 Description

1815 This operation causes all dirty blocks associated with the file descriptor
1816 to be written to stable storage. The meaning of “stable storage” is
1817 implementation specific – it may be the disk, non-volatile memory, or
1818 another mechanism that provides greater reliability than the volatile
1819 memory in the node caching the blocks. **SIO_CTL_Sync** performs a
1820 superset the operations performed by **SIO_CTL_Propagate**.

1821 Result Values

1822 SIO_SUCCESS

1823 The operation succeeded.

1824 SIO_ERR_IO_FAILED

1825 A physical I/O error caused the operation to fail.

1826 **13.8 SIO_CTL_GetLayout, SIO_CTL_SetLayout**1827 **Purpose**

1828 Get or set the layout of the file data on the storage system.

1829 **Affects**

1830 Underlying file.

1831 **Parameter Type**1832 Pointer to a **sio_layout_t**.1833 **Description**1834 These operations allow the layout of a file's data on the underlying
1835 storage system to be queried and modified.1836 The control **SIO_CTL_GetLayout** will return the layout for the un-
1837 derlying file, while **SIO_CTL_SetLayout** will set the layout, if pos-
1838 sible. Implementations may choose to ignore **SIO_CTL_SetLayout**
1839 entirely, returning **SIO_ERR_OP_UNSUPPORTED**.1840 **Result Values**1841 **SIO_SUCCESS**

1842 The operation succeeded.

1843 **SIO_ERR_OP_ONLY_AT_CREATE**1844 The implementation only supports **SIO_CTL_SetLayout** when
1845 a file is being created.1846 **SIO_ERR_INCORRECT_MODE**

1847 The mode of the file descriptor does not permit the operation.

1848 **SIO_ERR_OP_UNSUPPORTED**

1849 The operation is not supported by the system.

1850 13.9 SIO_CTL_GetLabel, SIO_CTL_SetLabel

1851 Purpose

1852 Get or set the file's label.

1853 Affects

1854 Underlying file.

1855 Parameter Type

1856 Pointer to a `sio_label_t`.

1857 Description

1858 These operations allow the label associated with a file to be set and
1859 retrieved. A file's label is not interpreted by the file system. The intent
1860 is for applications to store descriptive information about a file in the a
1861 file's label, rather than in the file itself. That removes the need for file
1862 headers and the inefficiencies that go with them.

1863 The maximum size of a file's label is `SIO_MAX_LABEL_LEN`, the
1864 value of which is implementation-specific. It is guaranteed, however,
1865 to be at least as big as `SIO_MAX_NAME_LEN`, allowing any legal
1866 file name to fit in a label. This allows descriptive information that is
1867 too large to fit in a label to be stored in an auxiliary file whose name
1868 can be stored in the label of the file being described.

1869 For descriptive labels to be portable across implementations
1870 they must be no larger than the minimum allowed value for
1871 `SIO_MAX_LABEL_LEN`.

1872 When performing `SIO_CTL_SetLabel`, the *data* field of the
1873 `sio_label_t` must contain a pointer to a buffer, the *length* field must
1874 contain the length of that buffer. If the length given is greater than
1875 `SIO_MAX_LABEL_LEN`, `SIO_ERR_INVALID_LABEL` will be
1876 returned and the operation will fail. After a `SIO_CTL_SetLabel` op-
1877 eration successfully completes, the length of the file's label will be equal
1878 to *length*, and the file's label data will be the same as the contents of
1879 the buffer.

1880 When performing **SIO_CTL_GetLabel**, the *data* field of the
1881 **sio_label_t** must contain a pointer to a buffer to be filled in with
1882 the file's current label data, and the *length* field must contain the size
1883 of that buffer. If the buffer is too small to contain the label, the
1884 **SIO_ERR_INVALID_LABEL** error code will be returned, *length*
1885 will be set to the actual length of the label, and the contents of the
1886 data buffer will be unspecified. If the buffer is at least as large as
1887 the current file label, **SIO_SUCCESS** will be returned, *length* will
1888 be set to the actual length of the label (as set by a previous call to
1889 **SIO_CTL_SetLabel**, or to zero if the file's label has never been set),
1890 and the data buffer will be filled with that many bytes of label data.
1891 If the buffer is larger than the label, the contents of the bytes in the
1892 buffer following the label are unspecified.

1893 Result Values

1894 **SIO_SUCCESS**

1895 The operation succeeded.

1896 **SIO_ERR_INCORRECT_MODE**

1897 The mode of the file descriptor does not permit the operation.

1898 **SIO_ERR_INVALID_LABEL**

1899 The length of the new label being set exceeds
1900 **SIO_MAX_LABEL_LEN**, or the length of the label being re-
1901 trieved exceeds the size of the application-provided buffer.

1902 **SIO_ERR_IO_FAILED**

1903 A physical I/O error caused the operation to fail.

1904 **SIO_ERR_NO_SPACE**

1905 The system needs to increase the amount of storage used by the
1906 file but cannot.

1907 13.10 SIO_CTL_GetConsistencyUnit

1908 Purpose

1909 Get the size of the cache consistency unit.

1910 Affects

1911 File system.

1912 Parameter Type

1913 Pointer to a **sio_size_t**.

1914 Description

1915 This operation returns the size of the cache consistency unit. The
1916 consistency unit defines the granularity of cache consistency under weak
1917 caching, as described in Section 12.

1918 Result Values

1919 SIO_SUCCESS

1920 The operation succeeded.

1921 14 Extension Support

1922 Support for querying the presence of extensions is part of the basic API,
1923 and must be implemented by all conforming implementations, even if no
1924 extensions are supported by an implementation. Applications may deter-
1925 mine either statically (described in Section 14.1.1) or dynamically (via the
1926 **sio_query_extension()** function, described in Section 14.2) whether or not
1927 an extension is supported by the implementation of the API. Sample code
1928 indicating the proper way to check for the presence of extensions is included
1929 in Section 14.3.

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.

¹²The 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**.

1959 **14.1.2 Extension Identifiers**

1960 Extension identifiers are constants of the form **SIO_EXT_NAME**, where
1961 *NAME* is the name of the extension. Extension identifiers with names of
1962 the form **SIO_EXT_VEND_NAME** are reserved for use by vendors, and all
1963 other extension names are reserved for future use by this API.

1964 An implementation must define an extension identifier for each extension
1965 which is supported or may be supported by that implementation as deter-
1966 mined by the value of the extension's **SIO_EXT_NAME_SUPPORTED**
1967 constant described in Section 14.1.1. Extension identifiers can be given to
1968 **sio_query_extension()** to check whether or not the extensions in question
1969 are actually available.¹³

¹³It 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.

1970 14.2 sio_query_extension

1971 Purpose

1972 Determine whether or not an extension is supported.

1973 Syntax

1974 `#include <sio_fs.h>`

1975 `sio_return_t sio_query_extension(sio_extension_id_t ExtID);`

1976 Parameters

1977 *ExtID* Extension identifier of extension being queried.

1978 Description

1979 This function takes an extension identifier and returns
1980 **SIO_SUCCESS** if the extension is supported by this implementa-
1981 tion, or **SIO_ERR_INVALID_EXTENSION** if the extension is not
1982 supported, or if the identifier is not recognized as valid.

1983 Return Codes

1984 **SIO_SUCCESS**

1985 The extension is supported by the implementation.

1986 **SIO_ERR_INVALID_EXTENSION**

1987 *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:

```
1990     int fooext_is_present;
1991     sio_return_t rc;
1992
1993     #if SIO_EXT_FOO_SUPPORTED == SIO_EXT_ABSENT
1994         fooext_is_present = 0;
1995     #elif SIO_EXT_FOO_SUPPORTED == SIO_EXT_PRESENT
1996         fooext_is_present = 1;
1997     #else /* SIO_EXT_FOO_SUPPORTED == SIO_EXT_MAYBE */
1998         rc = sio_query_extension(SIO_EXT_FOO);
1999         switch(rc) {
2000             case SIO_SUCCESS:
2001                 fooext_is_present = 1;
2002                 break;
2003             case SIO_ERR_INVALID_EXTENSION:
2004                 fooext_is_present = 0;
2005                 break;
2006             default:
2007                 fooext_is_present = 0;
2008                 printf("can't determine if extension foo is present (%s)\n",
2009                     sio_error_string(rc));
2010         }
2011     #endif /* SIO_EXT_FOO_SUPPORTED == SIO_EXT_ABSENT */
```


2012 **15 Extension: Collective I/O**

2013 **Static Constant: SIO_EXT_COLLECTIVE_SUPPORTED**

2014 **Extension ID: SIO_EXT_COLLECTIVE**

2015 **15.1 Motivation**

2016 As demonstrated by Kotz et al., collective I/O allows for a distributed batch-
2017 ing process which can greatly enhance I/O performance in a parallel file sys-
2018 tem. Semantically, by declaring an I/O or set of I/Os to be part of a single,
2019 collective I/O, an application is indicating to the file system that the relative
2020 ordering of the components of the collective I/O is irrelevant, since no por-
2021 tion of the application awaiting a component of the collective I/O can make
2022 any progress until the entirety of the collective I/O completes. File systems
2023 can take advantage of this to drastically reorder I/O components to reduce
2024 overall latency, at the potential cost of increasing the latency of component
2025 I/Os (the constraint which prevents this optimization from occurring in the
2026 standard case).

2027 **15.2 High Level Look**

2028 To initiate a collective I/O one task of the application requests that a new
2029 collective I/O handle be created. This is what we refer to as “defining” the
2030 collective I/O. At this time, the application indicates the number of partic-
2031 ipants, whether the collective I/O is a read or write operation (we do not
2032 allow collective mixed read/writes), the number of iterations of the collective
2033 I/O, and optionally indicates what portions of the file will be operated on.
2034 Specification of file regions at define time provides (ordered) file access hints
2035 which, if properly given, allow the file system to implement performance
2036 optimizations.

2037 Each participant “joins” an iteration of the collective I/O by providing the
2038 handle created by the define operation, the file descriptor, the portions of

2039 the file they wish to read or write, the source/destination memory locations,
2040 their participant identifier, and a sequence number indicating which iteration
2041 of the collective I/O they are joining.

2042 Note that the application will generally need to pass the handle from the
2043 task that defined the collective I/O to any other tasks that participate in the
2044 I/O. A single task may participate multiple times in a given collective I/O
2045 iteration by joining that iteration multiple times using different participant
2046 numbers. Prior to joining a collective I/O operation, a task must open the
2047 file being accessed so a file descriptor for the file is available for use with the
2048 join call.

2049 15.3 New Data Types

2050 15.3.1 sio_coll_handle_t

2051 This is a 64-bit integral type used as an abstract handle to represent a col-
2052 lective I/O. We explicitly define the format and size of this datatype because
2053 applications will need to use their own communications mechanisms to pass
2054 these among tasks on different nodes, and therefore need to be aware of size
2055 and network ordering issues.

2056 15.3.2 sio_coll_participant_t

2057 This is an unsigned integral type with the range
2058 [0...SIO_MAX_COLL_PARTICIPANTS] which is used in the definition
2059 of a collective I/O operation to specify the number of participants, and in
2060 the collective I/O join to identify the participant joining the collective I/O
2061 iteration.

2062 These values have no meaning or permanence beyond the collective I/O in
2063 which they are used.

2064 15.3.3 sio_coll_iteration_t

2065 This is an unsigned integral type with the range
2066 [0...SIO_MAX_COLL_ITERATIONS] which is used in the definition of
2067 a collective I/O operation to specify the number of iterations, and in the
2068 collective I/O join to identify the iteration being joined.

2069 15.4 New Range Constants

2070 15.4.1 SIO_MAX_COLL_ITERATIONS

2071 This constant specifies the maximum number of iterations that a collective
2072 I/O can describe. The minimum value is 1, and the recommended value is
2073 128.

2074 15.4.2 SIO_MAX_COLL_PARTICIPANTS

2075 This constant specifies the maximum number of participants that can take
2076 part in a collective I/O. The minimum value is 16, but the recommended
2077 value is at least 256.

2078 15.4.3 SIO_MAX_COLL_OUTSTANDING

2079 This constant specifies the maximum number of outstanding collective I/O
2080 requests that one task can have at any given time. The minimum value is 1,
2081 and the recommended value is at least 512.

2082 15.5 New Functions

2083 Two new functions are added by the collective I/O extension:
2084 **sio_coll_define()** and **sio_coll_join()**, which are described in Sections 15.5.1
2085 and 15.5.2, respectively.

2086 **15.5.1 sio_coll_define**2087 **Purpose**

2088 Define a new collective I/O and get a handle to refer to it.

2089 **Syntax**

```

2090 #include <sio_fs.h>
2091 sio_return_t sio_coll_define(int FileDescriptor,
2092                             sio_coll_iteration_t NumIterations,
2093                             const sio_file_io_list_t *FileList,
2094                             sio_count_t FileListLength,
2095                             sio_size_t IterationStride,
2096                             sio_mode_t ReadWrite,
2097                             sio_coll_participant_t NumParticipants,
2098                             sio_coll_handle_t *Handle);

```

2099 **Parameters**

2100 *FileDescriptor* The file descriptor of an open file.

2101 *NumIterations* The number of times the collective I/O will be repeated.

2102 *FileList* Specification of file data to be read or written.

2103 *FileListLength* Number of elements in *FileList*. This may be zero.

2104 *IterationStride* A value that modifies the location of the file data to be
 2105 read or written as specified in *FileList* based on the iteration in
 2106 progress.

2107 *ReadWrite* One of **SIO_MODE_READ** or **SIO_MODE_WRITE**.

2108 *NumParticipants* The number of participants in each iteration of the
 2109 collective I/O.

2110 *Handle* On success, returns the handle of the newly-defined collective
 2111 I/O.

2112 **Description**

2113 This interface creates a new handle for a collective I/O, and returns it in
 2114 *Handle*. The *NumIterations* parameter indicates the number of times

2115 the collective I/O will be performed. The application programmer may
 2116 choose to disclose the portions of the file which will be affected in
 2117 *FileList*, or *FileListLength* may be zero in which case the file system
 2118 must wait for a participant to call **sio_coll_join()** before its workload
 2119 is known.

2120 In cases where the collective I/O will be performed more than once and
 2121 the application programmer indicates what portions of the file will be
 2122 operated on, it is often true that the access patterns for each iteration
 2123 are identical except for their offsets from the beginning of the file,
 2124 and that the offsets are based on the iteration being performed. The
 2125 *IterationStride* parameter lets the programmer express these common
 2126 cases without having to separate them into individual collective I/O
 2127 operations. If *i* is the iteration number (starting at iteration 0), the
 2128 *offset* field in each **sio_file_io_list_t** structure of the *FileList* parameter
 2129 would have the value:

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

2130
 2131 For example, if *FileList* has two entries with the values (*offset*=0,
 2132 *size*=2, *stride*=3, *element_cnt*=4) and (*offset*=100, *size*=5, *stride*=0,
 2133 *element_cnt*=1), the programmer is hinting that the first iteration will
 2134 access bytes (0, 1, 3, 4, 6, 7, 9, 10, 100, 101, 103, 104, 105) in the
 2135 file. If *IterationStride* is zero, the second iteration will access the same
 2136 bytes. However, if *IterationStride* is 50, the second iteration will access
 2137 bytes (50, 51, 53, 54, 56, 57, 59, 60, 150, 151, 153, 154, 155) – the
 2138 *offset* components of the *FileList* structures are adjusted based on the
 2139 iteration (1) and the *IterationStride* (50).

2140 Note that **sio_coll_join()** must always be called by each participant
 2141 and must provide a *FileList* for that participant's portion of the col-
 2142 lective I/O, whether or not *FileListLength* is zero in **sio_coll_define()**.
 2143 Providing a description of the entire operation in *FileList* simply pro-
 2144 vides a way for the file system to optimize scheduling of the transfer.

2145 Return Codes

2146 SIO_SUCCESS

2147 The function succeeded.

2148 **SIO_ERR_INCORRECT_MODE**

2149 The mode of the file descriptor does not permit the I/O.

2150 **SIO_ERR_INVALID_DESCRIPTOR**

2151 *FileDescriptor* does not refer to a valid file descriptor created by
2152 **sio_open()**.

2153 **SIO_ERR_INVALID_FILE_LIST**

2154 The file regions described by *FileList* are invalid, e.g. they contain
2155 illegal offsets.

2156 **SIO_ERR_MAX_COLL_ITERATIONS_EXCEEDED**

2157 The number of iterations described
2158 by *NumIterations* exceeds the maximum allowed as defined by
2159 **SIO_MAX_COLL_ITERATIONS**.

2160 **SIO_ERR_MAX_COLL_PARTICIPANTS_EXCEEDED**

2161 The number of participants described
2162 by *NumParticipants* exceeds the maximum allowed as defined by
2163 **SIO_MAX_COLL_PARTICIPANTS**.

2164 **15.5.2 sio_coll_join**2165 **Purpose**

2166 Initiate an asynchronous transfer as part of a collective I/O.

2167 **Syntax**

```

2168 #include <sio_fs.h>
2169 sio_return_t sio_coll_join(int FileDescriptor,
2170                             sio_coll_handle_t Handle,
2171                             sio_coll_participant_t Participant,
2172                             sio_coll_iteration_t Iteration,
2173                             const sio_file_io_list_t *FileList,
2174                             sio_count_t FileListLength,
2175                             const sio_mem_io_list_t *MemoryList,
2176                             sio_count_t MemoryListLength,
2177                             sio_async_handle_t *AsyncHandle);

```

2178 **Parameters**

2179 *FileDescriptor* The file descriptor of the open file where the collective
 2180 I/O is being performed.

2181 *Handle* The handle provided by **sio_coll_define()** for this collective
 2182 operation.

2183 *Participant* The identifier for this participant. This is a number in the
 2184 range $[0 \dots (NumParticipants - 1)]$, where *NumParticipants* is the
 2185 number of participants that was provided to **sio_coll_define()**.

2186 *Iteration* Which iteration of the collective I/O the participant is joining.

2187 *FileList* Specification of file data to be read or written by this partici-
 2188 pant.

2189 *FileListLength* Number of elements in *FileList*.

2190 *MemoryList* Memory locations read or written by this I/O component.

2191 *MemoryListLength* Number of elements in *MemoryList*.

2192 *AsyncHandle* Handle returned by the operation, which can be used
 2193 later to determine the status of the I/O, to wait for its completion,
 2194 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 *AsyncHandles*, but the *AsyncHandles* returned by the **sio_coll_join()** may not be mixed with *AsyncHandles* 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

2230 the *FileList* offset values as appropriate for the iteration being joined.

2231 Return Codes

2232 SIO_SUCCESS

2233 The function succeeded.

2234 SIO_ERR_INCORRECT_MODE

2235 The mode of the file descriptor does not permit the I/O.

2236 SIO_ERR_INVALID_DESCRIPTOR

2237 *FileDescriptor* does not refer to a valid file descriptor cre-
2238 ated by `sio_open()`, or does not refer to the file specified to
2239 `sio_call_define()` when the collective I/O was created.

2240 SIO_ERR_INVALID_FILE_LIST

2241 The file regions described by *FileList* are invalid, e.g. they contain
2242 illegal offsets. Implementations may defer returning this error
2243 until `sio_async_status_any()` is invoked on the I/O.

2244 SIO_ERR_INVALID_HANDLE

2245 *Handle* is not the handle for a collective I/O.

2246 SIO_ERR_INVALID_ITERATION

2247 *Iteration* is not valid, either because it is greater than the num-
2248 ber of iterations specified when the collective I/O was created or
2249 between the task already joined that iteration of the I/O.

2250 SIO_ERR_INVALID_MEMORY_LIST

2251 The memory regions described by *MemoryList* are invalid, e.g.
2252 they contain illegal addresses. Implementations may defer return-
2253 ing this error until `sio_async_status_any()` is invoked on the
2254 I/O.

2255 SIO_ERR_INVALID_PARTICIPANT

2256 *Participant* is not valid because it is greater than the number of
2257 participants specified when the collective I/O was created.

2258 SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED

2259 The I/O request could not be initiated because doing so would
2260 cause the calling task's number of outstanding asynchronous I/Os
2261 to exceed the limit.

SIO_ERR_UNEQUAL_LISTS

2262 The number of bytes in *MemoryList* and *FileList* are not
2263 equal. Implementations may defer returning this error until
2264 *sio_async_status_any()* is invoked on the I/O.
2265

2266 16 Extension: Fast Copy

2267 Static Constant: **SIO_EXT_FAST_COPY_SUPPORTED**

2268 Extension ID: **SIO_EXT_FAST_COPY**

2269 This extension provides a low-level versioning mechanism by allowing an
2270 efficient “snapshot” of a file’s current contents to be created. This is done
2271 via the **sio_control()** operation **SIO_CTL_FastCopy**.

2272 The **SIO_CTL_FastCopy** control operation creates snapshots by replacing
2273 the contents of a parallel file (created and opened with **sio_open()**), with
2274 the contents of the file being duplicated. Since snapshots are normal parallel
2275 files, they can be accessed in all of the ways that parallel files can be accessed.
2276 That is, snapshots created by **SIO_CTL_FastCopy** can be read, written,
2277 operated on by controls, etc.

2278 If a higher-level file system library is using **SIO_CTL_FastCopy** to pro-
2279 vide versioning support, that library is responsible for managing the
2280 translation between its notion of versions and that provided by the
2281 **SIO_CTL_FastCopy** mechanism. For instance, the higher-level library
2282 must translate between the file name and version number that the appli-
2283 cation supplies and the actual parallel name for that snapshot. The higher-
2284 level library must also enforce its own version reference semantics (perhaps
2285 preventing write access to old versions of the file, or taking other actions as
2286 necessary).

2287 16.1 SIO_CTL_FastCopy

2288 Purpose

2289 Efficiently copy the contents of one file into another.

2290 Affects

2291 Underlying file.

2292 Parameter Type

2293 Pointer to an **int** which is a file descriptor for the open parallel file to
2294 be used as the source of the efficient copy operation.

2295 Description

2296 This operation performs an efficient copy of the contents of one par-
2297 allel file into another. The source file descriptor is specified by the
2298 **int** pointed to by the *op_data* member of the **sio_control.t**. The desti-
2299 nation file is specified by the *Name* argument to **sio_open()** or by the
2300 *FileDescriptor* argument to **sio_control()**.

2301 The implementation of the efficient copy operation performed by this
2302 function is intended to use copy-on-write or similar techniques to min-
2303 imize data duplication.

2304 If the **SIO_CTL_FastCopy** operation fails or is not supported, an
2305 error will be returned and the source and destination files will be un-
2306 modified.

2307 Effects of Successful Operation on the Source File

2308 The source file's data are unmodified by the **SIO_CTL_FastCopy**
2309 operation.

2310 The source file's physical size at the conclusion of the
2311 **SIO_CTL_FastCopy** operation is unspecified.

2312 None of the source file's other file or file descriptor attributes (as defined
2313 by this API) are modified by the **SIO_CTL_FastCopy** operation.

2314 If vendors define new attributes, the effect of **SIO_CTL_FastCopy** on
2315 the source file with respect to those attributes should be specified.

2316 Hints about expected use of the source file are unmodified by the
2317 **SIO_CTL_FastCopy** operation.

2318 **Effects of Successful Operation on the Destination File**

2319 The destination file's logical size is set to the source file's logical
2320 size, and the destination file's contents are made to appear identical
2321 (e.g. if accessed with **sio_sg_read()**) to those of the source file. If
2322 **SIO_CTL_SetSize** is specified in the same set of control operations as
2323 **SIO_CTL_FastCopy**, the resulting size of the destination file is un-
2324 defined.

2325 The destination file's physical size at the conclusion of the
2326 **SIO_CTL_FastCopy** operation is unspecified.

2327 The destination file's label is made identical to the source file's label.

2328 The destination file's other file attributes (preallocation and layout) are
2329 not affected.

2330 None of the destination's file descriptor attributes (caching mode and
2331 consistency unit) are affected. Note that if a weak client caching mode
2332 is in use on the destination file, the destination file's new contents may
2333 need to be propagated (with **SIO_CTL_Propagate**) before they can
2334 be used by other clients.

2335 If vendors define new attributes, the effect of **SIO_CTL_FastCopy** on
2336 the destination file with respect to those attributes should be specified.

2337 The effect of the **SIO_CTL_FastCopy** operation on hints about ex-
2338 pected use of the destination file is unspecified. Portable applications
2339 or libraries that wish to hint about future accesses to the destination
2340 file should cancel all outstanding hints on the destination file after per-
2341 forming a **SIO_CTL_FastCopy** operation and then reissue hints as
2342 appropriate.

2343 **Result Values**

2344 **SIO_SUCCESS**

2345 The function succeeded.

2346 **SIO_ERR_INVALID_DESCRIPTOR**

2347 The file descriptor for the source file is invalid.

SIO_ERR_NO_SPACE

2348

2349

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

SIO_ERR_OP_UNSUPPORTED

2350

2351

2352

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 Carnegie Mellon, 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'Hallaron, Jaspal Subhlok, and Thomas Stricker of Carnegie Mellon, 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 13-14 May 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 2 July 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 8-9 August 1996.

Version 0.66 was reviewed by e-mail, 21-31 August 1996.

2382 Members of the SIO Performance Evaluation working group at UIUC re-
2383 viewed and commented on the API beginning with the fifth draft. In par-
2384 ticular, Andrew Chien, Chris Elford, Tara Madhyastha, Dan Reed, Huseyin
2385 Simitci, and Evgenia Smirni contributed to the discussions and suggestions
2386 put forth by the Illinois group.

2387 Version 1.0 was released to the parallel computing community for comment
2388 on 1 October 1996.

2389 **A Result codes (for `sio_return_t`)**

2390 This appendix describes some error and return codes that the parallel file
2391 system may wish to return. As discussed in the Data Types section, imple-
2392 mentors should feel free to add whatever additional codes they see fit, and
2393 should make `sio_error_string()` aware of them.

2394 **SIO_SUCCESS**

2395 The operation completed successfully. The value of **SIO_SUCCESS**
2396 must always be 0.

2397 **SIO_ERR_ALREADY_EXISTS**

2398 The file name to be created already exists.

2399 **SIO_ERR_CONTROL_FAILED**

2400 One or more of the control operations requested by `sio_control()`,
2401 `sio_open()`, or `sio_test()` was unsuccessful.

2402 **SIO_ERR_CONTROL_NOT_ATTEMPTED**

2403 A control operation requested by `sio_control()`, `sio_open()`, or
2404 `sio_test()` was not attempted.

2405 **SIO_ERR_CONTROL_NOT_ON_TEST**

2406 The control operation cannot be used with `sio_test()`.

2407 **SIO_ERR_CONTROL_WOULD_HAVE_SUCCEEDED**

2408 The control operation would have succeeded but the function perform-
2409 ing the control failed.

2410 **SIO_ERR_CONTROLS_CLASH**

2411 The list of controls contains combinations of operations that are in-
2412 compatible.

2413 **SIO_ERR_FILE_NOT_FOUND**

2414 The specified file did not exist.

2415 **SIO_ERR_FILE_OPEN**

2416 The operation failed because the file was open.

2417 SIO_ERR_INCORRECT_MODE

2418 The mode of the file descriptor does not permit the operation or func-
2419 tion.

2420 SIO_ERR_INVALID_CLASS

2421 The hint class is not valid.

2422 SIO_ERR_INVALID_DESCRIPTOR

2423 A file descriptor argument was not a valid parallel file descriptor.

2424 SIO_ERR_INVALID_EXTENSION

2425 An invalid extension identifier was given, or the indicated extension is
2426 not supported.

2427 SIO_ERR_INVALID_FILE_LIST

2428 The file list argument is invalid (e.g. contains illegal offsets).

2429 SIO_ERR_INVALID_FILENAME

2430 A file name argument did not contain a legal file name (e.g. it was too
2431 long).

2432 SIO_ERR_INVALID_HANDLE

2433 A handle argument does not contain a valid handle.

2434 SIO_ERR_INVALID_ITERATION

2435 The iteration argument is invalid.

2436 SIO_ERR_INVALID_MEMORY_LIST

2437 The memory list argument is invalid (e.g. contains an illegal address).

2438 SIO_ERR_INVALID_PARTICIPANT

2439 The participant number provided is not valid because it is greater than
2440 the number of participants specified when the collective I/O was cre-
2441 ated.

2442 SIO_ERR_IO_CANCELED

2443 An asynchronous I/O did not complete because it was canceled while
2444 in progress.

2445 SIO_ERR_IO_FAILED

2446 A physical I/O error occurred.

2447 **SIO_ERR_IO_IN_PROGRESS**

2448 An asynchronous I/O has not yet completed.

2449 **SIO_ERR_MAX_ASYNC_OUTSTANDING_EXCEEDED**

2450 The I/O request could not be initiated because doing so would cause
2451 the calling task's number of outstanding asynchronous I/Os to exceed
2452 the limit.

2453 **SIO_ERR_MAX_COLL_ITERATIONS_EXCEEDED**

2454 The number of iterations specified for a collective I/O exceeds the limit.

2455 **SIO_ERR_MAX_COLL_OUTSTANDING_EXCEEDED**

2456 The I/O request could not be initiated because doing so would cause
2457 the calling task's number of outstanding collective I/O's to exceed the
2458 limit.

2459 **SIO_ERR_MAX_COLL_PARTICIPANTS_EXCEEDED**

2460 The number of participants specified for a collective I/O exceeds the
2461 limit.

2462 **SIO_ERR_MAX_OPEN_EXCEEDED**

2463 The file could not be opened because doing so would cause the calling
2464 task's number of open files to exceed the limit.

2465 **SIO_ERR_MIXED_COLL_AND_ASYNC**

2466 The implementation does allow asynchronous I/O handles created by
2467 **sio_coll_define()** to be passed to functions in the same list as handles
2468 from **sio_async_sg_read()** and **sio_async_sg_write()**.

2469 **SIO_ERR_NO_SPACE**

2470 An operation that would allocate more storage to a file failed because
2471 no storage could be allocated.

2472 **SIO_ERR_ONLY_AT_CREATE**

2473 The control operation may only be specified during a call to **sio_open()**
2474 which is creating a file.

2475 **SIO_ERR_ONLY_AT_OPEN**

2476 The control operation may only be specified during a call to
2477 **sio_open()**.

2478 SIO_ERR_OP_UNSUPPORTED

2479 The parallel file system has elected to not support this interface. Note
2480 that some interfaces may not be supported, but implementations can
2481 choose to return **SIO_SUCCESS** for all cases instead.

2482 SIO_ERR_UNEQUAL_LISTS

2483 The number of bytes in the memory and file lists arguments to an I/O
2484 operation are not the same.

2485 B Sample Derived Interfaces

2486 This section describes some simple interfaces which could easily be created
2487 using the interfaces provided by this API. These derived interfaces are *not* a
2488 part of this API, and are intended only as examples of interfaces which could
2489 be provided by high level libraries.

2490 If a high level library provides interfaces similar (or identical) to the sample
2491 interfaces presented here, those interfaces should be named in accordance
2492 with the rest of the interfaces provided by that library. In other words, *use*
2493 *of the names given here is strongly discouraged.*

2494 B.1 Synchronous I/O

2495 Routines

```

2496     sio_return_t sample_read(int FileDescriptor,
2497                               sio_addr_t BufferPointer,
2498                               sio_offset_t Offset, sio_size_t Count,
2499                               sio_transfer_len_t *BytesRead);

2500     sio_return_t sample_write(int FileDescriptor,
2501                                sio_addr_t BufferPointer,
2502                                sio_offset_t Offset, sio_size_t Count,
2503                                sio_transfer_len_t *BytesWritten);

2504     sio_return_t sample_read_io_list(int FileDescriptor,
2505                                       sio_addr_t BufferPointer,
2506                                       sio_file_io_list_t *FileList,
2507                                       sio_count_t FileListLength,
2508                                       sio_transfer_len_t *BytesRead);

2509     sio_return_t sample_write_io_list(int FileDescriptor,
2510                                       sio_addr_t BufferPointer,
2511                                       sio_file_io_list_t *FileList,
2512                                       sio_count_t FileListLength,
2513                                       sio_transfer_len_t *BytesWritten);

2514     sio_return_t sample_read_mem_list(int FileDescriptor,
2515                                       sio_mem_io_list_t *MemoryList,
2516                                       sio_count_t MemoryListLength,
2517                                       sio_offset_t Offset,
2518                                       sio_transfer_len_t *BytesRead);

2519     sio_return_t sample_write_mem_list(int FileDescriptor,
2520                                       sio_mem_io_list_t *MemoryList,
2521                                       sio_count_t MemoryListLength,
2522                                       sio_offset_t Offset,
2523                                       sio_transfer_len_t *BytesWritten);

```

2524 Parameters

2525 *FileDescriptor* The file descriptor of an open parallel file.

2526 *BufferPointer* Memory address of contiguous buffer containing data to
 2527 be written or to contain data being read.

2528 *Offset* Starting file offset from which to read or at which to write.

2529 *Count* Number of bytes to read or write.

2530 *BytesRead* Number of bytes actually read.

2531 *BytesWritten* Number of bytes actually written.

2532 *FileList* Description of strided regions within the file.

2533 *FileListLength* Number of valid elements to use in *FileList*.

2534 *MemoryList* Description of strided regions within the memory buffer.

2535 *MemoryListLength* Number of valid elements to use in *MemoryList*.

2536 Description

2537 These functions would provide a simplified synchronous I/O interface.
 2538 They may be implemented as wrappers which would convert the given
 2539 arguments into `sio_mem_io_list_t` and `sio_file_io_list_t` structures (as
 2540 necessary) and invoke `sio_sg_read()` or `sio_sg_write()`.

2541 The functions `sample_read()` and `sample_write()` would transfer
 2542 data between a single contiguous memory buffer and a single con-
 2543 tiguous region of the file. The functions `sample_read_io_list()`
 2544 and `sample_write_io_list()` would use a single contiguous mem-
 2545 ory buffer, but a strided region within the file. Similarly,
 2546 `sample_read_mem_list()` and `sample_write_mem_list()` would use
 2547 a contiguous file region, but a strided region within the memory buffer.

2548

2549

[illegible]

2580 **Parameters**

- 2581 *FileDescriptor* The file descriptor of an open parallel file.
- 2582 *BufferPointer* Memory address of contiguous buffer containing data to
2583 be written or to contain data being read.
- 2584 *Offset* Starting file offset from which to read or at which to write.
- 2585 *Count* Number of bytes to read or write.
- 2586 *BytesRead* Number of bytes actually read.
- 2587 *BytesWritten* Number of bytes actually written.
- 2588 *FileList* Description of strided regions within the file.
- 2589 *FileListLength* Number of valid elements to use in *FileList*.
- 2590 *MemoryList* Description of strided regions within the memory buffer.
- 2591 *MemoryListLength* Number of valid elements to use in *MemoryList*.
- 2592 *Handle* Handle for asynchronous I/O that can later be used to test its
2593 status.

2594 **Description**

- 2595 These routines would provide a simplified asynchronous I/O interface.
- 2596 They may be implemented as wrappers which would convert the given
- 2597 arguments into `sio_mem_io_list_t` and `sio_file_io_list_t` structures (as
- 2598 necessary) and invoke `sio_async_sg_read()` or `sio_async_sg_write()`.
- 2599 These functions would take arguments similar to those given to the
- 2600 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.