Quantum

StorNext File System API Guide

Quantum StorNext 4.2.1.0.1

6-67399-03 Rev A

Quantum StorNext 4.2.1.0.1 File System API Guide, 6-67399-03 Rev A, February 2012, Product of USA.

Contents

Chapter 1

Introduction

About This Guide

This guide contains information and instructions necessary to use the StorNext APIs (SNAPI). This guide is intended for system administrators, programmers, and anyone interested in learning about using the StorNext APIs.

The StorNext API guide is divided into the following chapters:

- StorNext File System APIs
- File System API Example

StorNext API (SNAPI) contains StorNext File System APIs which can be used to make calls from third-party applications, resulting in enhanced operations between third-party applications and StorNext.

Note: StorNext Storage Manager APIs are described in the *StorNext Storage Manager API Guide*.

Installing StorNext APIs

The StorNext File System APIs are automatically installed when you install the StorNext software.

Running APIs Remotely

StorNext File System APIs run locally on the client and the client's mounted file systems. They interact with the remote MDC indirectly, through the StorNext client.

Explanation of Warnings, Cautions and Notes

The following cautions, and notes appear throughout this document to highlight important information.

Caution: Indicates a situation that may cause possible damage to equipment, loss of data, or interference with other equipment.

Note: Indicates important information that helps you make better use of your system.

Quantum Service and Support

More information about this product is available on the Quantum Service and Support website at www.quantum.com/ServiceandSupport. The Quantum Service and Support website contains a collection of information, including answers to frequently asked questions (FAQs). You can also access software, firmware, and drivers through this site.

For further assistance, or if training is desired, contact the Quantum Technical Assistance Center:

North America	1+800-284-5101 Option 5
EMEA	00800 999 3822
Online Service and Support	www.quantum.com/OSR
Worldwide Web	www.quantum.com/ServiceandSupport

(Local numbers for specific countries are listed on the Quantum Service and Support Website.)

Chapter 2
StorNext File System APIs

This chapter describes the Application Programming Interfaces (APIs) that are available for StorNext File System. These file system APIs are automatically installed with the StorNext software and do not need to be installed separately.

The file system APIs are grouped into the following categories:

- Allocation and Space Management APIs
- Quality of Service and Real Time I/O APIs
- File System Configuration and Location Management APIs
- Access Management APIs

Appendix A at the end of this guide provides a test API sample program that illustrates how to use many of the StorNext File System APIs described in this guide.

Most of these APIs take two parameters: a request structure and a reply structure. On Windows, these correspond to the *inbuffer* and *outbuffer* parameters of DeviceIoControl(). On UNIX, the two structures (request and response) are consolidated into a single union so that we can still use the ioctl() and fcntl() interface. The functionality remains the same.

For example, the UNIX definition of the call to allocate space would be:

```
typedef union allocspacereqreply {
AllocSpaceReq_t req;
AllocSpaceReply_t reply;
} AllocSpaceReqReply_t;
```

Note: On Windows systems no helper library is distributed, and the ioctls must be called directly.

Field names for request structures are prefaced with xq_ where <x> is representative of the structure's name. Field names for reply structures are similarly prefaced with xr_.

All structures are 64-bit aligned and use the following typedefs on Windows:

```
typedef UCHARuint8_t;
typedef USHORTuint16_t;
typedef ULONGuint32_t;
typedef ULONGLONGuint64_t;
typedef CHARint8_t;
typedef SHORTint16_t;
typedef LONGint32_t;
typedef LONGLONGint64_t;
```

The control code definitions are platform dependent, and are defined in a separate file for each platform. The version number is encoded in each control code so that individual calls can be modified without affecting the rest of the interface. The macro name (for example, CvApi_PunchHole,) is the same on all platforms.

Platform	Filename	Interface Routine
Linux	cv_linux_ioctl.h	ioctl(2)
Solaris	cv_sol_ioctl.h	ioctl(2)
Irix	cv_irix_ioctl.h	fcntl(2)
Windows	cv_nt_ioctl.h	DeviceIoControl()

Except where noted, all calls return 0 (zero) on success, and a standard platform-specific error code on error. On UNIX, it is a standard errno. On Windows, it is one of the status codes listed in ddk\inc Windows Driver Development Kit. Internally, StorNext maps all error codes to a platform-independent value and only maps to platform error codes just before returning to the user.

In this document the following error codes are defined:

```
VOP_ENOENT2
VOP_EACCESS5
VOP_EXIST7
VOP_EINVAL11
VOP_ENOSPC12
VOP_EFAULT19
```

These error codes are mapped to their closest platform-specific error. For example, the error VOP_ENOENT maps to ENOENT on most UNIX platforms, and the Windows error code STATUS_OBJECT_NAME_NOT_FOUND on Windows (which may map to a Win32 definition such as ERROR_NOT_FOUND).

All offsets and sizes are given in bytes and are rounded up, if necessary, to the next file system block size.

For calls that return variable length buffers (such as extent lists), the call will return the total number of items available, as well as the number returned in this particular call. If more data is available than can be returned in the user's buffer, as much as possible is copied into the buffer, and no error is returned. For subsequent calls, the user can specify a different starting location (ordinal for stripe groups, starting offset for extents). This is similar to the getdents/getdirentries semantic on UNIX. Note that if the list is changing while the user is attempting to retrieve it, inconsistent results may be returned. When there are no more entries available, ENOENT is returned.

In this document, the word *handle* is synonymous with a *file descriptor* in UNIX.

Note: StorNext file system API names are preceded with "CvApi" because they were inherited from CVFS.

Allocation and Space Management APIs

These APIs allow you to control how data is written to StorNext, resulting in faster writes and more efficient allocation of capacity.

CvApi_AllocSpace

This API allocates extent space in a file. (This API is scheduled for deprecation and will not be supported in future StorNext releases.)

Handle

Target file.

Notes

This call attempts to allocate disk space of the requested size, starting at the requested file-relative offset. Commonly, this results in a single extent being allocated. However, if the file system free space is fragmented, up to 24 extents may be allocated. In addition, if the entire requested space cannot be allocated by adding 24 new extents, the API performs only a partial allocation and still returns a successful status. Therefore, to reliably determine the actual amount of space allocated by CvApi_AllocSpace, applications must track the number of file blocks using the UNIX fstat(2) system call or through the CvApi_CvFstat API.

If the caller specifies an offset to begin allocation, the call allocates space at the offset, rounded up to a file system block size. If any allocation exists that maps even a portion of <offset + size>, the call returns EXISTS. If no offset is specified, the call allocates the requested size beginning at the next file system block boundary beyond the current end of file. The number of bytes is rounded up to a file system block size. In both cases, the file size is updated if the allocation causes an increase in the current end of file.

If the affinity is specified, this sets the affinity for this and all future allocations. In this way, setting the affinity is “sticky.” If the affinity is already set in the file, setting the affinity in this call has no affect. To get/set the affinity, see the CvApi_GetAffinity / CvApi_SetAffinity calls. The allocation will be made *exclusively* only from stripe groups that have the matching affinity.

The caller can also specify that the extent information be loaded into the client extent mapping tables. This eliminates a subsequent trip to the FSM to retrieve extent information for the range mapped by this call.

All byte sizes and offsets are rounded up to the nearest file system block size. The call returns the actual allocated size and offset.

CvApi_AllocSpace is scheduled for deprecation in a future release. You should use the new function CvApi_VerifyAlloc instead.

Structure

```
typedef struct _AllocSpaceReq {
       uint64_taq_size;
       uint64_taq_offset;
       uint64_taq_affinitykey;

       uint32_taq_flags;
              #define ALLOC_OFFSETzx01
              #define ALLOC_LOAD_EXT0x02
              #define ALLOC_STRIPE_ALIGN0x04
              #define ALLOC_AFFINITY0x08
              #define ALLOC_KEEPSIZE0x10
              #define ALLOC_PERFECTFIT0x20
              #define ALLOC_SETSIZE0x40


       uint32_taq_pad1;

} AllocSpaceReq_t;

typedef struct _AllocSpaceReply {
       uint64_tar_size;
       uint64_tar_offset;
} AllocSpaceReply_t;
```

UNIX `ioctl` structure:

```
typedef union _allocspacereqreply {
       AllocSpaceReq_treq;
       AllocSpaceReply_treply
} AllocSpaceReqReply_t;
```

reQuest Fields

aq_size	Size in bytes to allocate. Must not be zero. If not a multiple of the file system block size, it is rounded up to the nearest file system block size.
aq_offset	If ALLOC_OFFSET is set, the aq_size bytes (rounded up to the nearest file system block size) is allocated starting at file byte offset

	aq_offset (rounded up to the nearest file system block size). If ALLOC_OFFSET is clear, the offset is ignored when allocating space. The space is allocated at the end of the file.
aq_affinitykey	64 bit affinity "key" identifying the affinity for the allocation. Valid only if ALLOC_AFFINITY is set. This forces the space to be allocated exclusively from stripe groups with a matching affinity key.
aq_flags	Control Flags:
ALLOC_OFFSET	aq_offset is valid.
ALLOC_LOAD_EXT	Load the extent struct mapping on the client and add it to the client file system. Note: in case where multiple extents are allocated, only the first extent is loaded.
ALLOC_STRIPE_ALIGN	Allocate space starting at a stripe boundary.
ALLOC_AFFINITY	aq_affinitykey is valid.
ALLOC_KEEPSIZE	Do not update the size of the file, even if extending it.
ALLOC_PERFECTFIT	Follow the 'perfect fit' rules for allocation.
ALLOC_SETSIZE	Update the file size, and broadcast the size change to other clients. The default is to only broadcast the new number of blocks and the fact the extent list has changed (which causes clients to flush their extent lists).

Reply Fields

ar_size	Actual size of the allocation that was attempted.
ar_offset	File relative offset of allocated space.

Error Returns

VOP_ENOSPC	Insufficient space in the file system to satisfy the request.

VOP_EINVAL	Invalid affinity key.
VOP_EXISTS	An allocation already exists that maps some or all of the specified offset and length.

CvApi_GetPerfectFitStatus

This API determines whether a file has been marked for PerfectFit allocations.

Handle

Handle for the file being queried.

Notes

This API is used by the `snfsdefrag` application to ensure that files with PerfectFit allocations continue to have PerfectFit allocations after the files are defragmented.

Structure

```
typedef struct _GetPerfectFitStatusReply {
       uint32_t    pr_status;
       uint32_t    pr_pad1;
} GetPerfectFitStatusReply_t;
```

UNIX ioctl structure:

None. Use GetPerfectFitStatusReply_t directly.

Reply Fields:

pr_status	If 1, the file has the PerfectFit bit set. If 0, the file does not.

Error Returns

VOP_ENOENT	The file doesn't exist.
VOP_EINVAL	The file is not a “regular” file.
Other	Communications failure with the FSM.

CvApi_PunchHole

This API punches a hole in the file.

Handle

Target file.

Notes

This call punches a hole in the file, adjusting the allocation map of the file to indicate that no data blocks are allocated for the indicated range. The granularity of access is the file system block size. The byte offsets are rounded down to the beginning of the block containing the specified byte offset. The actual offsets used are returned; the starting offset will always be file system block aligned.

If zero is specified as the ending offset, a hole is punched to the end of the file.

Structure

```
typedef struct _PunchHoleReq {
        uint64_tpq_start;
        uint64_tpq_end;     /* Inclusive */
} PunchHoleReq_t;

typedef struct _PunchHoleReply {
        uint64_tpr_start;
        uint64_tpr_end;
        uint64_tpr_nblocks;
        uint64_tpr_blksfreed;
} PunchHoleReply_t;
```

UNIX *ioctl* structure:

```
typedef union _punchholereqrep {
        PunchHoleReq_treq;
        PunchHoleReply_treply;
} PunchHoleReqReply_t;
```

reQuest fields

pq_start Starting byte offset of hole.

pq_end	Inclusive ending byte offset. A value of zero means punch to EOF.

Reply Fields

pr_start	Starting byte offset where hole was created.
pr_end	Inclusive ending byte offset where hole was created.
pr_nblocks	Number of blocks currently allocated to the file.
pr_blksfreed	Number of blocks freed.

CvApi_SetFileSize

This API sets the size of a file without zeroing pre-allocated data.

Handle

Target file.

Notes

The effect of this call is very similar to making the ftruncate(2) system call, except that when the file size is being extended, any existing blocks between the old EOF and the new EOF are not zeroed regardless of whether the SNFS “sparse” mount option is enabled or not. CvApi_SetFileSize is not currently supported on the Apple Xsan clients.

Structure

```
typedef struct _setfilesizereq {
       uint64_t sq_size;
} SetFileSizeReq_t;
```

No reply structure

UNIX *ioctl* structure.

None. Use SetFileSizeReq_t directly.

reQuest Fields

sq_size New file size.

Error Returns

VOP_EFAULT	Bad buffer offset.
VOP_EPERM	File is not writable by the caller.
VOP_EPERM	The user is not superuser and the “protect alloc” mount option is enabled.
Other	Communications failure with the FSM.

CvAPI_VerifyAlloc

This API allocates blocks within a file at the given offset.

Handle

The target file.

Notes

This call allocates all extents needed to fill the given range within a file. If any portion of the given range is already allocated, that portion is skipped. All additional space needed to form a completely allocated range is then filled in and a successful return status is set. If the given range is completely allocated, no allocations will be done and a successful status will be returned.

In order to ensure that only the requested allocation and no more is provided, use the ALLOC_NOMORETHAN flag. If this flag is not used the allocation might be rounded to optimize allocations.

In the case of an allocation failure, no space will be allocated and an error will be returned.

CvAPI_VerifyAlloc does not modify the file's size.

Structure

```
typedef struct  _VerifyAllocReq {
       uint64_tvq_size;
```

```
        uint64_tvq_offset;

        uint32_tvq_flags;

#define ALLOC_STRIPE_ALIGN0x04
#define ALLOC_NOMORETHAN0x20

        uint32_tvq_pad1;

} VerifyAllocReq_t;
```

No reply structure

Unix ioctl structure:

None. Use VerifyAllocReq_t directly.

reQuest Fields

vq_size	Size, in bytes, of the allocation request.
vq_offset	Offset, in bytes, of the start of the allocation range.
vq_flags	Flags to affect allocation behavior.

Error Returns

VOP_ENOSPC Not enough free space from which to allocate.

Note: This error is returned for files on both managed and unmanaged file systems and does not trigger emergency truncation on the StorNext Storage Manager.

VOP_EINVAL	Invalid arguments.
Other	Communications failure with the FSM.

Quality of Service and Real Time I/O APIs

This section describes the file system APIs that pertain to quality of service and real time IO.

CvApi_DisableRtio

This API disables (clears) the file's real-time attribute, making all further I/Os gated (if the stripe group is still in real-time mode).

Handle

Target file.

Notes

Files are also ungated by closing.

Structure

None.

CvApi_EnableRtio

This API puts a file handle into real time (ungated) mode.

Handle

Target file.

Notes

Ungated file handles are allowed full, unfettered (ungated) access to the SAN. Handles remain ungated until explicitly disabled or closed.

It is important to note that gating occurs on a handle basis, not a file basis. It is therefore possible for multiple threads with multiple handles

to be sharing a file, and for some of them to receive ungated (real time) access, while the remainder is gated.

It is not necessary to explicitly enable RTIO on a handle via this call if the handle refers to a regular file and the handle was specified in the call to CvApi_SetRtio.

Structure

Optional: If the RtReq_t structure is provided as an argument, the rq_flags field is queried to determine if extents should be pre-loaded. See CvApi_SetRtio for more information.

Reply Fields

pr_extent	Extent information for the target offset.
pr_breadth	Stripe breadth. This is the amount of data written on each disk.
pr_depth	Stripe depth. This is the number of disks in a stripe.
pr_voloffset	Amount to skip in basic blocks from start of volume (size of disk label info).
pr_blkoffset	Device block offset from beginning of disk.
pr_edev	Unit object pointer. Fairly useless outside of the kernel.

CvApi_GetRtio

This API retrieves the real time parameters for a stripe group.

Handle

Any file in the file system.

Notes

This API returns the real time parameters for a stripe group. The parameters are in I/Os/sec.

Structure

This uses the RtReq_t structure with the RT_GET flag to request the real-time parameters for a stripe group.

```
typedef struct _rtqueryreply {
        uint32_t  rrq_sgid;
        uint32_t  rrq_pad;

        int32_t  rrq_limit;
        int32_t  rrq_cur;

        int32_t  rrq_nrtio_hint;
        uint32_t  rrq_nrtio_clients;
} RtQueryReply_t;
```

UNIX ioctl structure

```
typedef union rtqueryreqrep {
        RtReq_t req;
        RtQueryReply_t reply;
} RtQueryReqReply_t;
```

reQuest Fields

See SETRTIO

Reply Fields

rrq_sgid	Stripe group number.
rrq_limit	Configured real time I/O limit, in I/Os/sec.
rrq_cur	Current amount of real time I/O committed to clients.
rrq_nrtio_hint	Amount of non-real time I/O a client is most likely to obtain when requesting a non-real time I/O token.
rrq_nrtio_clients	Number of clients with outstanding non-real time I/O tokens.

CvApi_GetRtio_V3

This API retrieves the real time (rtio) and non-real time reservation (rvio) parameters for a stripe group.

Handle

Any file in the file system.

Notes

This API returns the real time and non-real time reservation (rvio) parameters for a stripe group. The parameters are in I/Os/sec. This API is intended to replace CvApi_GetRtio for StorNext releases since 3.5.0 which support the rvio feature.

Structure

This uses the RtReq_t structure with the RT_GET flag to request the real time parameters for a stripe group. rq_sgid in RtReq_t is set to the stripe group ordinal number.

```
typedef struct _rtqueryreply_v3 {
    uint32_t    rrq_sgid;
    uint32_t    rrq_state;
       #define RT_STATE_NONREALTIME  0x01
       #define RT_STATE_REALTIME     0x02
       #define RT_STATE_REQUEST      0x04
       #define RT_STATE_TIMEOUT      0x10
       #define RT_STATE_ALL_UNGATED  0x20
    int32_t     rrq_limit;
    int32_t     rrq_rtiocur;
    int32_t     rrq_rviocur;
    int32_t     rrq_rvioreq;
    int32_t     rrq_nrtio_hint;
    uint32_t    rrq_nrtio_clients;
    uint32_t    rrq_rvio_clients;
} RtQueryReply_v3_t;
```

UNIX ioctl structure

```
typedef union rtqueryreqrep {
RtReq_t req;
RtQueryReply_t reply;
```

```
RtQueryReply_t reply_v3;
} RtQueryReqReply_t;
```

reQuest Fields

See SETRTIO

Reply Fields

rrq_sgid	Stripe group ordinal number.
rrq_state	The state of real time state machine.
rrq_limit	Configured real time I/O limit, in I/Os/sec.
rrq_rtiocur	Current amount of real time I/Os committed to clients.
rrq_rviocur	Current amount of non-real time reservation I/Os (rvio) committed to clients.
rrq_rvioreq	Current amount of non-real time reservation I/Os (rvio) requested from clients.
rrq_nrtio_hint	Amount of non-real time I/O a client is most likely to obtain when requesting a non-real time I/O token.
rrq_nrtio_clients	Number of clients with outstanding non-real time I/O tokens.
rrq_rvio_clients	Number of clients that have requested non-real time reservation IOs.

CvApi_QosClientStats

This API gets the QOS statistics for all connected clients.

Handle

Target file.

Notes

This call obtains the amount of real time and non-real time currently allocated for each client in the SAN.

The number of client connections the FSM can support is fixed by the configuration file. To retrieve the number of client connections available, the caller should first specify zero as the maximum number of clients. This returns the size of the connection table on the FSM. The caller can then allocate an array large enough to hold the QOS information for all clients, and pass the array in a second call.

Structure

Following is the structure that returns information about each connected client. For QOS purposes, only clients that have the QSTAT_CLIENT flag set will have any valid QOS information. The QSTAT_VALID flag means the entry in the connection table is valid, and the QSTAT_ADM flag means the connection is an “administrative tap.”

```
typedef struct rtclientstat {
       uint32_t  c_ipaddr;
       uint32_t  c_rtios;
       uint32_t  c_nonrtios;
       uint32_t  c_flags;
              #define QSTAT_VALID0x1/* entry valid */
              #define QSTAT_ADM0x2  /* adm connection */
              #define QSTAT_CLIENT0x4/* client connection */
} RtClientStat_t;

typedef struct _rtclientstatreq {
       uint32_t         cq_nclients;
       uint32_t         cq_sg;
       CvUserAddr_t  cq_buf; /* RtClientStat_t */
} RtClientStatReq_t;

typedef struct _rtclientstatreply {
       uint32_tcr_maxclients;/* max entries on fsm */
       uint32_tcr_pad;
} RtClientStatReply_t;

typedef union rtclientstatreqrep {
       RtClientStatReq_t req;
       RtClientStatReply_t reply;
} RtClientStatReqReply_t;
```

reQuest Fields

cq_nclients	Number of RtClientStat_t entries in the .cq_buf field. If this field is zero, then the call will return the max number of clients in the FSM connection table. This information can be used on subsequent calls to size the request buffer.
cq_sg	Stripe group to query.
cq_buf	Address of RtClientStat_t buffer where information will be deposited.

Reply Fields

cr_maxclients	Maximum number of entries in the FSM connection table. Callers can use this information to size their buffer to the largest possible size.

Error Returns

VOP_EFAULT	Bad buffer offset.
Other	Communications failure with the FSM.

CvApi_QosClientStats_v3

This API gets the QOS statistics, including rtio, rvio and non-rtio, for all connected clients.

Handle

Target file.

Notes

This call obtains the real time and non-real time parameters for each client in the SAN.

This API is intended to replace API CvApi_QosClientStats for StorNext releases since 3.5.0 which support the rvio feature.

The maximum number of client connections the FSM can support is fixed by the configuration file. To retrieve the number of client connections available, the caller should first specify zero as the

maximum number of clients. This returns the size of the connection table on the FSM. The caller can then allocate an array large enough to hold the QOS information for all clients, and pass the array in a second call.

Structure

Following is the structure that returns information about each connected client. For QOS purposes, only clients that have the QSTAT_CLIENT flag set will have any valid QOS information. The QSTAT_VALID flag means the entry in the connection table is valid, and the QSTAT_ADM flag means the connection is an "administrative tap."

```
typedef struct rtclientstat_v3 {
    uint32_t c_ipaddr;
    uint32_t c_rtios;
    uint32_t c_nonrtios;
    uint32_t c_rvios_req;
    uint32_t c_flags;
} RtClientStat_v3_t;

typedef struct _rtclientstatreq {
    uint32_t cq_nclients;
    uint32_t cq_sg;
    CvUserAddr_t cq_buf; /* RtClientStat_t */
} RtClientStatReq_t;

typedef struct _rtclientstatreply {
    uint32_t cr_maxclients; /* max entries on fsm */
    uint32_t cr_pad;
} RtClientStatReply_t;
```

UNIX ioctl structure

```
typedef union rtclientstatreqrep {
    RtClientStatReq_t req;
    RtClientStatReply_t reply;
} RtClientStatReqReply_t;
```

reQuest Fields

cq_nclients — Number of RtClientStat_t entries in the .cq_buf field. If this field is zero, the call will return the maximum number of

	clients in the FSM connection table. This information can be used on subsequent calls to size the request buffer.
cq_sg	Stripe group to query.
cq_buf	Address of RtClientStat_v3_t buffer where information will be deposited.

Reply Fields

cr_maxclients	Maximum number of entries in the FSM connection table. Callers can use this information to size their buffer to the largest possible size.

Error Returns

VOP_EFAULT	Bad buffer offset.
Other	Communications failure with the FSM.

CvApi_SetRtio

This API requests real time IO on an individual stripe group or file. See also the Platform Dependencies section for this API.

Handle

Root directory. Affects all files on stripe group.

or

Handle to target file. Affects only specific target handle/file descriptor.

Notes

This call enables real time and non-real time parameters on a stripe group basis. RTIO is on an individual stripe group, not file system basis, since stripe groups can have very different access characteristics and can be used for very different file types (for example, audio versus video).

The caller specifies the maximum number of IOs per second or MB/sec. that they expect to utilize on the stripe group. Either one, but not both, may be specified. The number of IOs per second can be converted into MB/sec. by dividing the MB/sec. rate by the block size, stripe width, and stripe depth.

For example:Given an 8 disk stripe group with a *StripeBreadth* of 16 and an *FsBlockSize* of 4k, a requested rate of 50 MB/sec. would be (50mb/sec. * 1024k) / (8 * 16 *4k) = 100/sec.

The FSM may grant some amount less than requested unless the RT_MUST flag is set.

Once the FSM has returned a non-zero value in the reply structure, the partition group is in real-time mode. If the caller chooses to not accept the values returned by the FSM, it is the caller's responsibility to disable RTIO on the stripe group.

If the handle/file descriptor is the handle to the root of the file system ("\"),the call affects all files on the designated stripe group. Closing the root handle will not disable RTIO on the stripe group. It must be implicitly cleared by specifying zero in either the rq_rtios or the rq_rtmb fields and setting the RT_CLEAR flag.

Setting the RT_CLEAR flag with a non-zero value in either rq_rtios or rq_rtmb only releases the amount specified; specifying zero in those fields completely disables real time IO on the stripe group.

If the handle/file descriptor is for a regular file in the stripe group, the call has slightly different semantics. Using a target handle is equivalent to specifying RTIO on the root directory, followed by a call to put the handle into real time mode (CvApi_EnableRtio). All other non-real time IO handles on the stripe group will be gated, as when specifying RTIO on the root directory. RTIO will be disabled when the handle is closed, either explicitly or implicitly, and the bandwidth returned to the system.

Note: If a file is opened with multiple handles, each specifying a different amount of real time IO, the RTIO be released only when the last handle has been closed. No RTIO will be released until the last handle has been closed.

The mode of specifying a target file allows non-cooperating applications to request differing amounts of real time IO on the same stripe group. Upon successful return from the call, the target handle is in real time (ungated) mode; no further calls need be made. All accesses to other non-real time handles will be gated. The additional semantic difference is that RTIO is returned to the system when the handle is closed.

If the handle/file descriptor is for a regular file and the RT_NOLOAD flag is not set, all extents for the file are preloaded into the file system. This cuts down on cold-start overhead. However, if the file has many extents, this operation can take a long time to complete.

For both modes, if the FSM is rebooted or is reset, the client file system will attempt to re-negotiate the real time IO requirements with the FSM. This may introduce a period of instability. It may not be possible to guarantee the same bandwidth requests as before, due to request ordering during the recovery period. If the same amount of real-time bandwidth cannot be obtained during recovery processing, the next access to a handle that is real time mode will fail and an event will be logged in the system log.

If the handle/file descriptor is for the root of the file system and the RT_ABSOLUTE flag is set, the system adjusts the amount of currently allocated RTIO to bring it in line with the request. This is useful for systems that are continually adjusting the amount of RTIO available based on external criteria (such as a video stream bit rate).

Note: The following extent crossing functionality will be implemented in a future release.

File-based RTIO encompasses the entire file. Since the file can have multiple extents, it is possible that it will cross stripe groups. If the RT_SEQ bit is set in rq_flags field, the client FSD assumes that access will be sequential through the file. The client FSD ensures that when crossing from one stripe group to the next, the new stripe group is put into the appropriate real time mode before any access occurs. The real time requirements of the previous stripe group will be released.

Structure

```
typedef struct _rtreq {
       union {
              int32_t ru_rtios;      /* ios per sec */
              int32_t ru_rtmb;       /* mb per sec */
       } rq_un;

       #define rq_rtios rq_un.ru_rtios
       #define rq_rtmb rq_un.ru_rtmb

       uint32_t    rq_flags;
              #define RT_IO 0x01     /* r_rtios valid */
              #define RT_MB 0x02     /* r_rtmb valid */
              #define RT_CLEAR 0x04  /* clear RT */
              #define RT_SET 0x08    /* set RT */
              #define RT_MUST 0x10   /* fail if can't satisfy */
```

```
                #define RT_SEQ0x20    /* sequential IO */
                #define RT_GET0x40    /* get rt params */
                #define RT_NOGATE0x80 /* ungated IO */
                #define RT_NOLOAD0x100/* don't load extents */
                #define RT_ABSOLUTE 0x200 /* absolute req not incr
*/


        uint32_t    rq_sgid;  /* stripe group ID */

} RtReq_t;

typedef struct _rtreply {
        union {
                int32_t ru_rtios;      /* ios per sec */
                int32_t ru_rtmb;       /* mb per sec */
        } rr_un;

        #define rr_rtiorr_un.ru_rtios
        #define rr_rtmbrr_un.ru_rtmb

        uint32_t    rr_flags;
                #define RT_IO0x01     /* r_rtios valid */
                #define RT_MB0x02     /* r_rtmb valid */
        uint32_t    rr_pad1;
} RtReply;
```

UNIX ioctl structure

```
typedef union rtreqrep {
        RtReq_t req;
        RtReply_treply;
} RtReqReply_t;
```

reQuest Fields

rq_rtios	Requested real time I/Os per sec. Valid only if RT_IO is set. RT_MB may not also be set.
rq_rtmb	Requested megabytes per sec. Valid only if RT_MB is set. RT_IO may not also be set.
rq_flags	Control flags
RT_IO	rr_rtios is valid.

RT_MB	rr_rtmb is valid.
RT_CLEAR	Clear real time from stripegroup. Either RT_CLEAR or RT_SET must be set.
RT_SET	Set real time parameters for stripegroup.
RT_MUST	Fail the request if the requested amount can't be satisfied. This prevents the FSM from returning a lesser value than what was requested.
RT_SEQ	The application will be performing sequential I/O and the client FSD should handle crossing of stripe groups. (This functionality will be implemented in a future release.)
RT_NOGATE	If RT_SET is specified put the handle into ungated mode. In this mode, the I/O using this handle consumes no RTIO and is not gated. If RT_CLEAR is specified, the handle is removed from this mode.
RT_NOLOAD	Do not pre-load extents for the file.
RT_ABSOLUTE	The amount specified in the rq_rtios field is an *absolute* value for the stripe group specified in rq_sgid. The system adjusts the request depending on the current state of the stripe group and the current amount of RTIO already allocated. This flag is only valid when used on the root directory (that is, not on a regular file).
rq_sgid	Stripe group ordinal, identifying stripe group. See the API for retrieving the stripe group name to match names with ordinals.

Reply Fields

rr_rtios	Allowed real time I/Os per second. Valid only if RT_IO is set. May be less than or greater than the amount requested if RT_MUST flag is clear.
rr_rtmb	Allowed real time MB per second. Valid only if RT_MB is set. May be less than or greater than the amount requested if RT_MUST flag is clear.

Platform Dependencies

If the same file is shared in real time and non-real time mode on a Unix platform (that is, anything other than NT4, Win2k, or XP), the caller must use the fcntl(2) system call to differentiate between the real time and non-real time accesses. This is because UNIX platforms do not typically export file descriptor flags down to the file system. The value to use is different, depending on the platform. On Irix and Solaris, the caller must do a fcntl (fd, F_SETFL, O_SYNC) to identify the real time file descriptor.

Note: The file system cannot distinguish between the use of O_SYNC for identifying real time file descriptors and its use for specifying synchronous writes. Therefore, on Irix and Solaris platforms, when a file is opened O_SYNC (or if the O_SYNC is set on the file via fcntl), all writes will be synchronous and all I/O performed on the file will be non-gated.

On Linux, the caller must do a fcntl (fd, F_SETFL, O_NONBLOCK). All other file descriptors will be gated.

File System Configuration and Location Management APIs

These APIs are used primarily for reporting purposes and provide details on file parameters as well as configuration of the underlying disk volumes that make up a StorNext file system.

CvApi_GetAffinity

This API gets the affinity for a file.

Handle

Target file.

Notes

Affinities can direct allocations to specific stripe groups. This call will return the current affinity for the file.

Structure

No request structure

```
typedef struct getaffinityreply {
       uint64_tar_affinity;
} GetAffinityReply_t;
```

UNIX ioctl structure.

None. Use GetAffinityReply_t directly.

Reply Fields

ar_affinity Current affinity identifier for file.

CvApi_GetExtList

This API gets the list of extents for a file.

Handle

Target file.

Notes

This call does not load the extents in the client file system extent map for a file. It returns as many extents as it can in a single call directly from the FSM.

This is an iterative call. It is the responsibility of the caller to allocate any free buffers. Since there may be many extents for a file, the caller can iterate over the list, specifying a different starting offset in gq_startfrbase. When there are no more extents available the call returns the platform equivalent of ENOENT.

The caller must allocate the buffer for the reply data, and initialize the gq_buf field to point to it. The buffer should be aligned on a minimum of an 8 byte boundary.

Structure

```
typedef sruct _cvexternalextent {
       uint64_tex_frbase;    /* file relative offset */

       uint64_tex_base;      /* fs starting offset */
       uint64_tex_end;       /* fs ending offset, inclusive*/
       uint32_tex_sg;        /* stripe group number */
       uint32_tex_depth;     /* sg depth for this extent */
       uint32_tex_pad1;

} CvExternalExtent_t;

typedef struct _getextlistreq {
       uint64_t gq_startfrbase;
       uint32_t gq_numbufs;
       uint32_t gq_pad1;
       void           *qb_buf;
} GetExtListReq_t;


typedef struct _getextlistreply {
       uint32_t       gr_numreturned;
       uint32_t       gr_pad;
} GetExtListReply_t;
```

UNIX ioctl structure

```
typedef union _getextlistreqrep {
       GetExtListReq_treq;
       GetExtListReply_treply;
} GetExtListReqReply_t;
```

Fields, CvExternalExtent_t

ex_frbase — File relative starting byte offset. This offset can be anywhere in the extent; it does not have to correspond exactly to the starting offset of an extent. Any extent that contains ex_frbase will be returned.

ex_base — File system starting byte offset.

ex_end — File system ending byte, inclusive. Since this byte offset is inclusive and specifies the last valid byte in the extent, this value will not be a multiple of the file system block size. To

	obtain the next extent, add one to e_end for the starting offset of the next extent.
ex_sg	ID of stripe group.
ex_depth	Depth of stripe group for this extent. Stripe groups can grow and shrink dynamically, so the depth of the extent may not match the current depth of the stripe group.

reQuest Fields

gq_startfrbase	Starting file relative byte offset.
gq_numbufs	Number of CvExtent_t sized buffers in response.
gq_buf	User allocated buffer where the call will place an array of CvExternalExtent_t structs.

Reply Fields

gr_numreturned	Number of CvExternalExtent_t elements in gq_buf.

Error Returns

VOP_ENOENT	No more entries available.
VOP_EFAULT	Buffer is invalid.
VOP_EINVAL	Invalid starting offset. The extent list has probably changed.

CvApi_GetPhysLoc

This API determines the physical location of any byte offset in a file.

Handle

Target file.

Notes

This call returns extent information about the target offset, as well as the location in the file system and on the target disk.

All offsets are rounded up to the nearest file system block size. All values are specified in bytes.

Structure

```
typedef struct _physlocreq {
       uint64_tpq_offset;
} PhysLocReq_t;

typedef struct _physlocreply {
       CvExtentpr_extent;

       uint64_tpr_breadth;
       uint64_tpr_depth;

       uint64_tpr_voloffset;/* VolHder sz in bytes    */
       uint64_tpr_blkoffset;/* Block offset in bytes   */

       uint32_tpr_edev;
       uint32_tpr_pad1;

} PhysLocReply_t;

Unix ioctl structure

typedef union _phylocreqreply {
       PhysLocReq_treq;
       PhysLocReply_treply;
} PhysLocReqReply_t;
```

reQuest Fields

pq_offset Desired byte offset.

Reply Fields

pr_extent	Extent information for the target offset.
pr_breadth	Stripe breadth. This is the amount of data written on each disk.
pr_depth	Stripe depth. This is the number of disks in a stripe.

pr_voloffset	Amount to skip in basic blocks from start of volume (size of disk label info).
pr_blkoffset	Device block offset from beginning of disk.
pr_edev	Unit object pointer. Fairly useless outside of the kernel.

CvApi_GetSgInfo

This API gets the parameters associated with a stripe group.

Handle

Any, but usually handle to root directory.

Notes

This call retrieves all the kernel states associated with the specified stripe group, including the affinity identifiers.

This call is local to the client and only queries the information on partition groups that are valid on the client. If a stripe group is not being accessed by the client and is therefore not currently active on the client, then the call cannot return information about the stripe group. It will return ENOENT.

Structure

```
#define SG_NAMELEN 256

typedef struct _sginforeq {
       uint32_t sqi_id;
       uint32_t sqi_pad0;
       CvUserAddr_t sqi_keys;
       CvUserAddr_t sqi_keycnt;
} SgInfoReq_t;

typedef struct _sginforeply {
       uint64_t sri_totblks;
       uint64_t sri_freeblks;

       uint32_t sri_breadth;
       uint32_t sri_depth;
```

```
        uint32_t sri_flags;
                #define SG_PART_VALID 0x1
                #define SG_PART_ONLINE 0x2
                #define SG_PART_METADATA 0x4
                #define SG_PART_JOURNAL 0x8
                #define SG_PART_EXCLUSIVE 0x10
        uint32_t sri_bsize;

        char           sri_name[SG_NAMELEN];

} SgInfoReply_t;
```

UNIX ioctl structure:

```
typedef union _sginfoqrep {
        SgInfoReq_t req;
        SgInfoReply_t reply;
} SgInfoReqReply_t;
```

reQuest Fields

sqi_id	ID of stripe group to search for.
sqi_keys	A user supplied uint64_t array to be filled with affinity identifiers. If sqi_keys is NULL, no identifiers will be returned.
sqi_keycnt	Pointer to a uint32_t. On input, this integer should be the maximum number of affinity identifiers to be returned (i.e. the size of the array sqi_keys). On output, the integer will contain the number of indentifiers actually returned. If sqi_keycnt is NULL, no keys will be returned.

Reply Fields

sri_totblks	Total blocks on stripe group.
sri_freeblks	Free blocks available on stripe group.
sri_breadth	Stripe breadth in file system blocks (number of blocks written to each disk in a single chunk).
sri_depth	Stripe breadth (number of disks in stripe group).

sri_flags	State flags
SG_PART_ONLINE	Stripe group is valid and online.
SG_PART_VALID	Stripe group is valid; may be offline.
SG_PART_METADATA	Stripe group is used for metadata.
SG_PART_JOURNAL	Stripe group is used for journal.
SG_PART_EXCLUSIVE	Stripe group is exclusive.

sri_bsize	Block size for stripe group.
sri_name	Name of stripe group.

Error Returns

VOP_ENOENT	No more entries available.
VOP_EFAULT	Buffer is invalid.
VOP_E2BIG	Stripe group has more affinity IDs than the user-supplied sqi_keycnt.

CvApi_GetSgName

This API gets the ASCII name for a stripe group or the ordinal for a specified ASCII name.

Handle

Any, but usually handle to root directory.

Notes

Many of the requests use an ordinal to identify a stripe group. This API converts an ordinal into its user-visible name. In this manner, a user can iterate through the stripe groups and match up the names in the FSM configuration file with the current ordinal. The names returned will match the [StripeGroup xxx] directive in the FSM config file. The call returns the name of the stripe group matching the ordinal specified in the sq_id field, or the ordinal of the stripe group with the matching name in sq_name.

This call is local to the client, and only queries the information on partition groups that are valid on the client. If a stripe group is not being accessed by the client and is therefore not currently active on the client, the call cannot return information about the stripe group. It will return ENOENT.

Each stripe group name is 256 bytes long, including NULL. This is the minimum buffer size. Buffers that are shorter than the minimum will result in an exception, returning EFAULT.

When there are no more entries available, the call returns ENOENT. If no stripe group corresponds to the specified ordinal, it means that the list changed between calls, and the call returns EINVAL.

Structure

```
#define SG_NAMELEN256

typedef struct _sgnamereq {

        uint32_tsq_flags;
                #define SG_GETNAME1
                #define SG_GETNUM2

        uint32_tsq_pad1;

        union {
                uint32_tsqu_id;
                charsqu_buf[SG_NAMELEN];
        } sq_un;

} SgNameReq_t;
        #define sq_idsq_un.squ_id
        #define sq_namesq_un.squ_buf

typedef struct _sgnamereply {
        union {
                uint32_tsru_id;
                charsru_buf[SG_NAMELEN];
        } sr_un;

} SgNameReply_t;
```

UNIX ioctl structure

```
typedef union _sgnamereqrep {
       SgNameReq_treq;
       SgNameReply_treply
} SgNameReqReply_t;
```

reQuest Fields

sg_flags	If SG_GETNAME is set, sq_id contains the ordinal of the stripe group to search for. If SG_GETNUM is set, then sq_name contains the ASCII name of the stripe group to search for.
sr_un.sru_id	ID of stripe group to search for.
sr_un.sru_buf	Name of stripe group to search for.

Reply Fields

sr_un.sru_buf	Buffer with name of stripe group.
sr_un.sru_id	Ordinal of stripe group.

Error Returns

VOP_ENOENT	No more entries available.
VOP_EFAULT	Buffer is invalid.
VOP_EINVAL	Invalid stripe group ordinal.

CvApi_SetAffinity

This API sets the affinity for a file.

Handle

Target file.

Note: Affinities can direct allocations to specific stripe groups. This call sets the current affinity for the file and affects all future allocations.

Structure

```
typedef struct setaffinityreq {
       uint64_tsq_affinity;
} SetAffinityReq_t;
```

No reply structure

UNIX *ioctl* structure.

None. Use SetAffinityReq_t directly.

Request Fields

sq_affinity Affinity identifier for file>

Access Management APIs

These APIs allow you to control concurrent file operations and quotas, and they provide additional reporting utilities.

CvApi_ClearConcWrite

This API disables concurrent writes to a file.

Handle

Target file.

Notes

This API disables concurrent writes for all users of the file. This works on a file, not handle basis.

Structure

None.

CvApi_ClearRdHoleFail

This API causes reads from a hole in the file to return zero (default behavior).

Handle

Target file.

Notes

This restores the default behavior of returning zero for a read from non-allocated space in a file.

Structure

None.

CvApi_CvFstat

This API gets the UNIX-like stat struct from a file.

Handle

Target file.

Notes

This call performs much the same as the UNIX stat(2) call. See also CvApi_StatPlus.

Structure

No request structure.

```
typedef struct _statreply {
        int32_t sr_dev;
        uint32_tsr_mode;

        uint64_tsr_ino;

        uint64_tsr_size;
```

```
        uint64_tsr_nblocks;

        int32_t sr_nlink;
        uint32_tsr_bsize;

        int32_t sr_uid;
        int32_t sr_gid;

        int32_t sr_rdev;
        int32_t sr_atim;

        int32_t sr_mtim;
        int32_t sr_ctim;
} StatReply_t;
```

UNIX *ioctl* structure:

None. Use StatReply_t directly.

Reply Fields:

sr_dev	Device identifier, unique per mounted file system.
sr_mode	Unix mode.
sr_ino	File handle, unique per file system.
sr_size	Size of file in bytes.
sr_nblocks	Number of “basic” (512 byte) blocks allocated to file.
sr_nlink	Number of links to the file.
sr_bsize	Blocksize of file system.
sr_uid	Unix User Identifier.
sr_gid	Unix Group Identifier.
sr_rdev	Device node for devices. 0 on the file system.
sr_atim	Last access time in seconds since January 1, 1970.
sr_mtim	Last modify time in seconds since January 1, 1970.
sr_cim	Last “change” time in seconds since January 1, 1970.

CvApi_CvOpenStat

This API retrieves open status on the given file.

Handle

Target file.

Notes

This method returns status regarding the open state of a file. These status objects are useful in finding the open state across the cluster. The os_sharedwrite and os_sharedread bits may be used to indicate whether a file is opened on another client. The os_opencount and os_refcount values indicate how the local client is using the given file.

Structure

```
typedef struct _openstatreply {
       uint32_tos_sharedwrite;
       uint32_tos_sharedread;
       uint32_tos_opencount;
       uint32_tos_refcount;
} OpenStatReply_t;
```

CvApi_GetDiskInfo

This API gets disk information for a stripe group.

Handle

Any, but usually handle to root directory.

Notes

The information returned by this call is somewhat limited. More information may be returned in future releases.

Structure

```
#define MAXPATHS 4
typedef struct _cvdiskinfo {
```

```
        char       di_name[256]; /* CVFS disk name */
        uint32_t  di_nameloc;   /* disk byte offset to disk name */
        uint32_t  di_vhsize;    /* Volume header size */
        char      di_serialnum[64]; /* WWN or disk serial number */
        uint32_t  di_sectorsize;    /* disk sector size in bytes */
        uint32_t  di_npaths;      /* number of active paths */
        struct di_dev_info {
               char d_bdev[256];     /* block device name */
               char d_rdev[256];     /* character device name */
        } di_paths[MAXPATHS];
} CvDiskInfo_t;

typedef struct _diskinforeq {
        uint32_t       sq_sg;
        uint32_t       sq_pad0;
        CvUserAddr_tsq_dinfobuf;
        CvUserAddr_tsq_dcnt;
} DiskInfoReq_t;
```

No reply structure

UNIX `ioctl` structure.

None. Use DiskInfoReq_t directly.

Fields, CvDiskInfo_t

di_name	Disk name. For example, “CvfsDisk_0”.
di_nameloc	Disk offset (in bytes) where the disk name resides.
di_vhsize	The size of the disk volume header, in bytes.
di_serialnum	Either the disk’s WWN (for fibre-attached devices) or its serial number.
di_sectorsize	Sector size of the disk, in bytes.
di_npaths	Number of paths to the disk.
di_paths	A structure containing the block and character device names for active paths to the disk. A single disk may have up to MAXPATHS (4) paths. Therefore, there may be up to 4 block and raw device names for a single disk.

reQuest Fields

sq_id	ID of stripe group.
sq_dinfobuf	A user supplied CvDiskInfo_t array to be filled in.
sq_dcnt	Pointer to a uint32_t. On input, this integer should be the maximum number of disk info structures to be returned (that is, the size of the array sq_dinfobuf). On output, the integer will contain the number of disk info structures actually returned.

Error Returns

VOP_ENOENT	sq_id is not a valid stripe group number.
VOP_E2BIG	The stripe group contains more disks than the user supplied sq_dcnt.
VOP_EFAULT	sq_dinfobuf or sq_dcnt is an invalid pointer.

CvApi_GetQuota

This API gets the current quota usage and limits for a given user or group.

Handle

Any, but usually handle to root directory.

Notes

The source handle may refer to any open file or directory that resides on the file system. It is not necessary that the file be owned by the user or group whose quota values are being queried.

Structure

```
#define MAX_QUOTA_NAME_LENGTH 256

typedef struct getquotareq {
       uint32_t  gq_type;
              #define QUOTA_TYPE_USER(uint32_t)’U’
              #define QUOTA_TYPE_GROUP(uint32_t)’G’
```

```
        uint32_t  gq_pad;
        char      gq_quotaname[MAX_QUOTA_NAME_LENGTH];
} GetQuotaReq_t;

typedef struct getquotareply {
        uint64_t  gr_hardlimit;  /* in bytes */
        uint64_t  gr_softlimit;  /* in bytes */
        uint64_t  gr_cursize;    /* in bytes */
        uint32_t  gr_timelimit;  /* in minutes */
        uint32_t  gr_timeout;    /* in seconds since January 1,
1970 */
} GetQuotaReply_t;

typedef union _getquotareqreply {
        GetQuotaReq_t    req;
        GetQuotaReply_t  reply;
} GetQuotaReqReply_t;
```

reQuest Fields

gq_type	Either QUOTA_TYPE_USER or QUOTA_TYPE_GROUP.
gq_quotaname	The null-terminated name of the user or group whose quota is being queried.

Reply Fields

gr_hardlimit	The hard quota limit in bytes, rounded up to the nearest file system block.
gr_softlimit	The soft quota limit in bytes, rounded up.
gr_cursize	The current usage in bytes, rounded up.
gr_timelimit	When the soft limit is exceeded, the amount of time (in minutes) before the soft limit will be treated as a hard limit.
gr_timeout	If the soft quota has been exceeded, gr_timeout will contain the time (in seconds since January 1, 1970) when the soft limit will be treated as a hard limit.

Error Returns

VOP_ENOTSUP	The quota system is not enabled on the FSM and/or client.
VOP_INVAL	gq_type is not QUOTA_TYPE_USER or QUOTA_TYPE_GROUP.\
VOP_ENOENT	gq_quotaname is not a valid user or group name.

CvApi_GetVerInfo

This API retrieves version information.

Handle

Any.

Notes

This returns the version, build, and creation date of the kernel, as well as the version of the external API. Note that since individual calls can be up-revved independently, the version number that is returned may not be the same for all calls.

Structure

No request structure.

```
#define CVVERINFO_MAX   64

typedef struct _verinfo_reply {
        char            vr_version[CVVERINFO_MAX];
        char            vr_build[CVVERINFO_MAX];
        char            vr_creationdate[CVVERINFO_MAX];
        uint32_tvr_apiversion;
        uint32_tvr_pad1;
} VerInfoReply_t;
```

Reply Fields

vr_version	A string containing the release and build number, such as #!@$ CVFS Client Revision 2.1.1 Build 60.

vr_build	A string containing platform information, such as #!@$ Built for Windows 2000 i386.
vr_creationdate	A string containing the creation date, such as #!@$ Created on Thu Dec 19 14:15:53 PST 2010.
vr_apiverion	The current version of the external API.

CvApi_LoadExtents

This API pre-loads a range of extents for a file.

Handle

Target file.

Notes

This call requests that the extent information from the FSM be pre-loaded into the file system. This cuts down on first access (cold-start) time. If there are any holes in the file, they will be preserved. No space is allocated by this call.

Structure

```
typedef struct _LoadExtReq {
       uint64_tlq_size;
       uint64_tlq_offset;
       uint32_tlq_pad1;
       uint32_tlq_pad2;
} LoadExtReq_t;
```

reQuest Fields

lq_size	The number of bytes to load. Specifying zero (0) will cause the extents for the entire file to be loaded. If lq_size + lq_offset extends beyond the end of the file, extents will be loaded up to the end of the file.
lq_offset	Starting offset to begin loading. The offset does not have to be on an existing extent boundary; any offset will suffice.

Error Returns

VOP_EINVAL	Offset is greater than end of file.
Other	Communications failure with the FSM.

CvApi_MoveRange

This API moves a range of extents from a source file to a target file within the same SNFS file system.

Structure

```
int
CvApi_MoveRange(
    int         f_fd,
    uint64_t    f_target_ino,
    uint64_t    f_source_froffset,
    uint64_t    f_target_froffset,
    uint64_t    f_length,
    tmspec_t    f_source_mtime,
    tmspec_t    f_target_mtime,
    uint32_t    f_flags,
    uint64_t    *f_rsvd)
```

reQuest Fields

f_fd: Source file descriptor.

f_target_ino: Target file inode which will receive extents from the source. This inode should be acquired by calling `CVApi_CVFStat` and using the `sr_ino` field in the returned `StatReply_t` object. The file must be kept open during the API invocation.

Note: Do not attempt to use the `st_ino` returned by UNIXstat(2), as this will often cause `CvApi_MoveExtents` to fail. Also note, do not use the inode number returned by `CvApi_ExtendedStat` either as this could cause failures.

f_source_froffset:	Starting byte offset in the source file. This offset can be anywhere in the file; it does not have to correspond exactly to the starting offset of an extent. However, it must correspond to a multiple of the file system block size and the range must be within EOF or EINVAL will be returned. (See special notes below regarding 'Append Mode'.)
f_target_froffset:	Starting byte offset in the target file. This offset can be anywhere in the file; it does not have to correspond exactly to the starting offset of an extent. However, it must correspond to a multiple of the file system block size or EINVAL will be returned. Overlap with an existing extent on the target will result in the target extent being discarded. (See Special Rules for Append Mode.)
f_length:	Number of bytes to move. Must correspond to a multiple of the file system block size or EINVAL will be returned. The range on the source file must be entirely within EOF or EINVAL will be returned. (See Special Rules for Append Mode.)
f_source_mtime:	This parameter is reserved for future use.
f_target_mtime:	This parameter is reserved for future use.
f_flags:	Supported flags only, otherwise EINVAL is returned.

```
#define MOVE_RSVD1 0x000000001 /* Invalid - reserved for future */

#define MOVE_APPEND 0x000000002 /* Append Mode */
```

The Append mode modifier provides a means to coalesce multiple source file fragments to a singe target file. Each call to this API will place the source extents at the current EOF on the target file and then move EOF on the target file to account for the added extent space. Every subsequent call, when using append, mode must start at exactly EOF on the target file (i.e., f_target_froffset).

Also, `f_target_froffset` must be FS block aligned. Therefore, before an append mode invocation you must ensure that the target file EOF is FS block aligned. If it is not, then it is recommended to use the ftruncate(2) system call to extend the target EOF to FS block alignment.

WARNING: Do not use `CvApi_SetFileSize` for this purpose as it will leave uninitialized (ie. 'corrupt') data in this file region.

Also, in order to be robust in the event of a system crash and restart, before invoking the API you should check the target file EOF, perform ftruncate(2) as needed, and adjust `f_target_froffset`, `f_source_froffset`, and `f_length` to ensure that the range begins at exactly EOF on the target file.

Special Rules for Append Mode

1. `f_target_froffset` must exactly equal EOF on the target file or EINVAL is returned.
2. The range spanned by `f_source_froffset` and f_length must be within EOF on the source file or EINVAL is returned.

Reply Fields

f_rsvd: Reserved for future use. Must be NULL.

Error Returns

EPERM	The user does not have permission to use this API. The API can only be used by superuser.
ENOENT	The source file or target file doesn't exist. You may also receive this error if the source and target files do not reside on the same file system.
EBADF	The source or target file is not open with write permission.
EBUSY	Either the source or target file mtime verification failed.
EWOULDBLOCK	Either the source or destination file has DMAPI events present. Files must be present on disk. (On Windows the error status returned is `STATUS_FILE_IS_OFFLINE`.)
EINVAL	See above.

CvApi_SetConcWrite

This API allows concurrent writes to a file.

Handle

Target file.

Notes

This call allows multiple handles to write concurrently to a file without being serialized. It is important to note that this `ioctl` call operates on the file, not the handle. Once a file is in concurrent write mode, all users of the file will be able to write concurrently. The concurrent write feature is reset to the default (non-concurrent) when the last user closes the file, or when it is explicitly disabled.

No data buffering is done. Malformed I/O returns the platform equivalent of EINVAL. It is the responsibility of those using this feature to maintain separate handles and separate offsets. The primary users of this feature are drivers that are layered directly on top of the FSD and use IRPs to communicate.

Structure

None.

CvApi_SetQuota

This API sets quota limits for a given user or group.

Handle

Any, but usually handle to root directory.

Notes

The source handle may refer to any open file or directory that resides on the file system. It is not necessary that the file be owned by the user whose quota values are being queried.

The specified hard and soft limits are automatically rounded up to the nearest file system block by the quota system.

Structure

```
#define MAX_QUOTA_NAME_LENGTH 256

typedef struct _setquotareq {
       char      sq_quotaname[MAX_QUOTA_NAME_LENGTH];
       uint32_t  sq_type;
                     #define QUOTA_TYPE_USER(uint32_t)'U'
              #define QUOTA_TYPE_GROUP(uint32_t)'G'

       uint32_t  sq_timelimit;      /* in minutes */
       uint64_t  sq_hardlimit;      /* in bytes */
       uint64_t  sq_softlimit;      /* in bytes */
} SetQuotaReq_t;
```

No reply structure

UNIX *ioctl* structure.

None. Use SetQuotaReq_t directly.

reQuest Fields

sq_quotaname	The name of the user or group whose quota limits are being set.
sq_type	Either QUOTA_TYPE_USER or QUOTA_TYPE_GROUP.
sq_timelimit	The soft quota grace period, in minutes.
sq_hardlimit	The hard limit, in bytes.
sq_softlimit	The soft limit, in bytes.

Error Returns

VOP_ENOTSUP	The quota system is not enabled on the FSM and/or client.
VOP_INVAL	sq_type is not QUOTA_TYPE_USER or QUOTA_TYPE_GROUP.
VOP_ENOENT	sq_quotaname is not a valid user or group name.

VOP_IO	sq_softlimit is greater than sq_hardlimit (or an internal error occurred).

CvApi_SetRdHoleFail

This API causes reads from a hole in a file to fail.

Handle

Target file.

Notes

During recovery processing after a client or FSM crash, it can be desirable to have reads from non-allocated space return an error rather than the default of zeros. This call affects all handles of a file. If an attempt is made to read from non-allocated space, the platform equivalent of EACCESS will be returned. On Windows, this translates into STATUS_ACCESS_DENIED. This behavior remains in affect until all handles to the file have been closed or the feature has been explicitly cleared.

Structure

None.

CvApi_StatFs

This API gets file system information.

Handle

Any, but usually handle to root directory.

Notes

Gets basic file system information.

Structure

No request structure.

```
typedef struct __statfs {
        uint32_tfr_options
                #define FSOPTION_DMIG(1<<0)
                #define FSOPTION_QUOTAS(1<<1)
                #define FSOPTION_BRLS(1<<2)
                #define FSOPTION_GLOBALSU(1<<3)
                #define FSOPTION_WINSEC(1<<4)

        uint32_tfr_blocksize;

        uint64_tfr_epoch;

        uint64_tfr_total_blocks;

        uint64_tfr_blocks_free;

        uint32_tfr_reserved[8];

} StatFsReply_t;
```

UNIX *ioctl* structure:

None. Use StatFsReply_t directly.

Reply Fields

fr_options	Mask containing enabled file system options.
fr_blocksize	File system basic block size in bytes.
fr_epoch	File system creation date in microseconds since January 1, 1970.
fr_total_blocks	Total capacity of the file system in blocks.
fr_blocks_free	Number of unallocated blocks.
fr_reserved	Fields reserved for future use.

Error Returns

VOP_EFAULT	Bad buffer offset.

VOP_ENOMEM	Out of memory.
Other	Communications failure with the FSM.

CvApi_StatPlus

This API gets the UNIX-like stat struct from a file, plus additional file information

Handle

Target File

Notes

This call performs much the same as the UNIX stat(2) call except that additional information is returned such as the storage state. Also see CVApi_CVFStat.

Structure

No request struct.

```
typedef struct _statplusreply {
        int32_t sr_dev;
        uint32_t sr_mode;

        uint64_t sr_ino;

        uint64_t sr_size;

        uint64_t sr_nblocks;

        int32_t sr_nlink;
        uint32_t sr_bsize;

        int32_t sr_uid;
        int32_t sr_gid;

        int32_t sr_storagestate;
                #define STORESTATE_ON_DISK_ONLY(1)
                #define STORESTATE_ON_TAPE_ONLY(2)
                #define STORESTATE_ON_DISK_AND_TAPE(3)
```

```
        int32_t sr_atim;

        int32_t sr_mtim;
        int32_t sr_ctim;

        uint32_tsr_reserved[8];
} StatPlusReply_t;
```

UNIX *ioctl* structure:

None. Use StatPlusReply_t directly.

Reply Fields

sr_dev	Device identifier, unique per mounted file system.
sr_mode	Unix mode.
sr_ino	File handle, unique per file system.
sr_size	Size of file in bytes.
sr_nblocks	Number of “basic” (512 byte) blocks allocated to file.
sr_nlink	Number of links to the file.
sr_bsize	Blocksize of file system.
sr_uid	Unix User Identifier.
sr_gid	Unix Group Identifier.
sr_storestate	Data migration: current file storage state.
sr_atim	Last access time in seconds since January 1, 1970.
sr_mtim	Last modify time in seconds since January 1, 1970.
sr_cim	Last “change” time in seconds since January 1, 1970.
sr_reserved	Fields reserved for future use.

Error Returns

VOP_EFAULT	Bad buffer offset.
VOP_ENOMEM	Out of memory.
Other	Communications failure with the FSM.

CvApi_SwapExtents

This API swaps all the extents for a file.

Handle

Source handle.

Notes

During defrag, the defrag utility allocates an extent and copy into it the data for a file. When the copy is complete, it attempts to swap the extents for the file, replacing the prior highly fragmented extent list in a file with, hopefully, a much smaller one.

This API swaps all the extents for one file (the file identified by the calling handle) with the file identified in the API structure. After this operation, the caller can unlink the source handle (if desired).

Structure

```
typedef struct _swapextreq {
       uint64_tsq_targhandle;
       uint32_tsq_msec;
       uint32_tsq_pad1;
} SwapExtReq_t;
```

No reply structure

UNIX *ioctl* structure.

None. Use SwapExtReq_t directly.

reQuest Fields

sq_handle — Handle to target file that will have its extents replaced by the source. This handle should be acquired by calling CVApi_CVFStat and using the sr_ino field in the returned StatReply_t object. **NOTE:** Do not attempt to use the st_ino returned by UNIXstat(2), as this will often cause CvApi_SwapExtents to fail.

sq_msec	A date in UNIX time(2) format. If non-zero, the value of sq_msec is checked against the modification time of the source file. If they are not the same, VOP_EBUSY is returned. So, sq_msec can be used as an additional sanity check to prevent attempts to defragment files that are actively being written.

Error Returns

VOP_EBUSY	The file is in use.
VOP_EPERM	The user does not have permission to defrag the file. Under UNIX, the user must be superuser or the owner of the file. Under Windows, the user must have write access to the file.
VOP_ENOENT	The file doesn't exist.
VOP_EINVAL	The file is not a “regular” file

Appendix A
File System API Example

Following is a test API sample program that illustrates how to use many of the StorNext File System APIs described in this document. This example applies only to the File System APIs, not the Storage Manager APIs.

```
/*
Copyright (c) 1997-2011
    All Rights Reserved.
    StorNext File System
Provided AS-IS, with no warranties expressed or implied.
*/
/*
 * tapi.c -- Test API
 *
 * Sample application to illustrate how to use the SNFS external API.
 *
 * This requires a 'getopt' routine that is available on most unix boxes,
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <time.h>           /* localtime() */

#if defined (_WIN32)
  #include <io.h>           /* for posix open(2) */
#endif

/* Local Headers */
#if defined(_WIN32)
  #include <cvinttypes.h>       /* for integer typedefs */
  #include <bsd_getopt.c>       /* need a getopt routine */
```

```
  #include <bsd_strtoll.c>
#endif

extern char *optarg;
extern int optind;

#if defined(__linux__)
  #include <stdint.h>
   #ifndef NULL
      #define NULL 0
   #endif /* NULL */
#endif /* linux */

#include <extapi.h>
#include <cvapi.h>

/* Macros */
#if defined(_WIN32)
  #define ARG64X    "0x%I64x"
  #define ARG64D    "%I64d"
#else
  #define ARG64X    "0x%llx"
  #define ARG64D    "%lld"
#endif /* _WIN32 */

#define EXTMAX24
#define MAXDISKS 32

/* File scope variables */
char   *Progname;
int    Verbose;
int    ErrorFlag;

/* External variables */
extern int optind;

/* External functions */
/* Structures and Unions */

/* Signal Catching Functions */
    /* NONE */
void
Usage()
{


    fprintf(stderr, "Usage: %s <args> filename \n", Progname);
    fprintf(stderr, "\t -A alloc space\n");
    fprintf(stderr, "\t -a affinity\n");
    fprintf(stderr, "\t -B stat fs\n");
    fprintf(stderr, "\t -b broadcast size on alloc\n");
    fprintf(stderr, "\t -C set concurrent write\n");
    fprintf(stderr, "\t -c clear concurrent write\n");
    fprintf(stderr, "\t -D get disk info for stripe group <n>\n");
    fprintf(stderr, "\t -E print extent list\n");
    fprintf(stderr, "\t -F set affinity\n");
```

```
    fprintf(stderr, "\t -f get affinity \n");
    fprintf(stderr, "\t -G get stripe group name from Ordinal\n");
    fprintf(stderr, "\t -g get stripe group ordinal from Name\n");
    fprintf(stderr, "\t -I Info for stripe group <n> \n");
    fprintf(stderr, "\t -L get physical Location for offset in file\n");
    fprintf(stderr, "\t -l (alloc flag) load extent in client fsd\n");
    fprintf(stderr, "\t -N keep size on allocation \n");
    fprintf(stderr, "\t -n nbytes [kmg] \n");
    fprintf(stderr, "\t -O Open stat file\n");
    fprintf(stderr, "\t -o offset [kmg] \n");
    fprintf(stderr, "\t -P punch hole \n");
    fprintf(stderr, "\t -p get PerfectFit status\n");
    fprintf(stderr, "\t -Q (user) set the quota limits for (user)\n");
    fprintf(stderr, "\t -q (user) get the quota usage and limits for (user)\n");
    fprintf(stderr, "\t -R set read hole fail \n");
    fprintf(stderr, "\t -r clear read hole fail \n");
    fprintf(stderr, "\t -S stat file (old version)\n");
    fprintf(stderr, "\t -s (alloc flag) stripe align allocation \n");
    fprintf(stderr, "\t -T stat file (new \"plus\" version)\n");
    fprintf(stderr, "\t -t (alloc flag) set perfect fit status\n");
    fprintf(stderr, "\t -v verbose\n");
    fprintf(stderr, "\t -V get version info\n");
    fprintf(stderr, "\t -w wait before exiting\n");
    fprintf(stderr, "\t -x load extents\n");
    fprintf(stderr, "\t -Y VerifyAlloc allocation\n");
    fprintf(stderr, "\t -y Toggle PerfectFit status\n");
    fprintf(stderr, "\t -Z set file size\n");
    fprintf(stderr, "\t -z size [kmg]\n");

}
/*
 * PURPOSE
 *  Print out the time for an 'ls' style listing
 */
void
PrintTime(
    int32_tf_secs)
{
    struct tm*tp;
    tp = localtime((time_t *)&f_secs);
    switch (tp->tm_mon) {
    case 0:
       printf("Jan");
       break;
           case 1:
       printf("Feb");
       break;
    case 2:
       printf("Mar");
       break;
    case 3:
       printf("Apr");
       break;
    case 4:
```

```
       printf("May");
       break;
    case 5:
       printf("Jun");
       break;
    case 6:
       printf("Jul");
       break;
    case 7:
       printf("Aug");
       break;
    case 8:
       printf("Sep");
       break;
    case 9:
       printf("Oct");
       break;
    case 10:
       printf("Nov");
       break;
    case 11:
       printf("Dec");
       break;
    }
    printf(" %d ", tp->tm_mday);
    printf("%02d:%02d", tp->tm_hour, tp->tm_min);
}
/*
 * PURPOSE
 *  Stat file, old version
 */
int
StatFile(
    int     f_fd,
    char    *f_filename)
{
    int         error = 0;
    StatReply_t     sb;
    /*
     * Get the stats
     */
    error = CvApi_CvFstat(f_fd, &sb);
    if (error) {
      return error;
    }
    /*
     * Print out the stats in no particular order
     */
    printf("File stats for file '%s'\n", f_filename);
    printf("Dev %d rdev %d nlink %d bsize %d\n",
       sb.sr_dev, sb.sr_rdev, sb.sr_nlink, sb.sr_bsize);
    printf("Size " ARG64D " nblocks " ARG64D "\n", sb.sr_size, sb.sr_nblocks);
    printf("Inode " ARG64D " uid %d gid %d mode 0%o \n",
           sb.sr_ino, sb.sr_uid, sb.sr_gid, sb.sr_mode);
    printf("atime: ");
    PrintTime(sb.sr_atim);
    printf(" mtime: ");
    PrintTime(sb.sr_mtim);
```

```
    printf(" ctime: ");
    PrintTime(sb.sr_ctim);
    printf("\n");
    return error;
}
/*
 * PURPOSE
 *  Stat file, new "plus" version
 */
int
StatFilePlus(
    int          f_fd,
    char    *f_filename)
{
    int                     error = 0;
    StatPlusReply_t         sb;
    char            *storestate;
    /*
     * Get the stats
     */
    error = CvApi_StatPlus(f_fd, &sb);
    if (error) {
      return error;
    }
    /*
     * Print out the stats in no particular order
     */
    printf("File stats for file '%s'\n", f_filename);
    printf("Dev %d nlink %d bsize %d\n",
      sb.sr_dev, sb.sr_nlink, sb.sr_bsize);

    switch(sb.sr_storestate) {

    case STORESTATE_ON_DISK_ONLY:
       storestate = "on disk only";
       break;
    case STORESTATE_ON_TAPE_ONLY:
       storestate = "on tape only";
       break;
    case STORESTATE_ON_DISK_AND_TAPE:
       storestate = "on disk and tape";
       break;
    default:
       storestate = "unknown";
       break;
    }
    printf("Store state: %d (%s)\n", sb.sr_storestate, storestate);
    printf("Size " ARG64D " nblocks " ARG64D "\n", sb.sr_size, sb.sr_nblocks);
    printf("Inode " ARG64D " uid %d gid %d mode 0%o \n",
           sb.sr_ino, sb.sr_uid, sb.sr_gid, sb.sr_mode);
    printf("atime: ");
    PrintTime(sb.sr_atim);
    printf(" mtime: ");
    PrintTime(sb.sr_mtim);
    printf(" ctime: ");
    PrintTime(sb.sr_ctim);
    printf("\n");
    return error;
```

```
}
/*
 * PURPOSE
 *  Stat VFS
 */
int
StatFs(
    int           f_fd,
    char    *f_filename)
{
    int                     error = 0;
    StatFsReply_t           sb;
    time_t           esecs;
    /*
     * Get the stats
     */
    error = CvApi_StatFs(f_fd, &sb);
    if (error) {
      return error;
    }
    /*
     * Print out the stats in no particular order
     */
    printf("FS stats for '%s'\n", f_filename);
    printf("options: 0x%x\n", sb.fr_options);
    if (sb.fr_options & FSOPTION_DMIG)
       printf("\tFSOPTION_DMIG\n");
    if (sb.fr_options & FSOPTION_QUOTAS)
       printf("\tFSOPTION_QUOTAS\n");
    if (sb.fr_options & FSOPTION_BRLS)
       printf("\tFSOPTION_BRLS\n");
    if (sb.fr_options & FSOPTION_GLOBALSU)
       printf("\tFSOPTION_GLOBALSU\n");
    if (sb.fr_options & FSOPTION_WINSEC)
       printf("\tFSOPTION_WINSEC\n");
    esecs = (time_t)(sb.fr_epoch / (uint64_t)1000000);
    printf("epoch: " ARG64X " - %s", sb.fr_epoch, ctime(&esecs));
    printf("block size: %d\n", sb.fr_blocksize);
    printf("total blocks: " ARG64D "\n", sb.fr_total_blocks);
    printf("blocks free: " ARG64D "\n", sb.fr_blocks_free);
    printf("inode stripe breadth: " ARG64D "\n", sb.fr_inode_stripe_width);
    printf("\n");
    return error;
}
/*
 * PURPOSE
 *  OpenStat file
 */
int
OpenStatFile(
    int      f_fd,
    char    *f_filename)
{
    int                  error = 0;
    OpenStatReply_t      sb;
    /*
     * Get the stats
     */
```

```
    error = CvApi_CvOpenStat(f_fd, &sb);
    if (error) {
        return error;
    }
    /*
     * Print out the stats in no particular order
     */
    printf("File Open_stats for file '%s'\n", f_filename);
    printf("RefCount: %u\n", sb.os_refcount);
    printf("OpenCount: %u\n", sb.os_opencount);
    if(sb.os_sharedread)
       printf("SharedRead TRUE\n");
    else
       printf("SharedRead FALSE\n");
    if(sb.os_sharedwrite)
        printf("SharedWrite TRUE\n");
    else
        printf("SharedWrite FALSE\n");
    printf("\n");
    return error;
}
/*
 * PURPOSE
 *  Get the physical location for a file given an offset
 */
int
GetPhysLoc(
    int               f_fd,
    char       *f_filename,
    uint64_t    f_offset)
{
    int                      error = 0;
    CvExternalExtent_t      *exp;
    PhysLocReply_t loc;
    /*
     * Get the location
     */
    error = CvApi_GetPhysLoc(f_fd, f_offset, &loc);
    if (error) {
       return error;
    }
    printf("Physical location for offset " ARG64X " in file '%s' \n",
           f_offset, f_filename);
    exp = &loc.pr_extent;
    printf("Extent: sg %d file relative base " ARG64X "\n" , exp->ex_sg,
           exp->ex_frbase);
    printf("        filesystem base " ARG64X, exp->ex_base);
    printf(" filesystem end " ARG64X , exp->ex_end);
    printf("\n");
    printf("SG breadth " ARG64X " depth " ARG64X "\n", loc.pr_breadth,
           loc.pr_depth);
    printf("volume offset " ARG64X " device relative blkoffset " ARG64X "\n",
           loc.pr_voloffset, loc.pr_blkoffset);
    printf("Device pseduo id %x\n", loc.pr_edev);
    return error;
}
/*
 * PURPOSE
```

```
 *  Dump out the extents for a file
 */
int
GetExtList(
    intf_fd)
{
    CvExternalExtent_t     *buf = NULL, *junkbuf = NULL, *exp;
    int                     allocsize;
    uint32_t         cnt, i, numbufs;
    int                     error = 0;
    uint64_t         offset;
    StatReply_t             sb;
    /*
     * Get the stats
     */
    error = CvApi_CvFstat(f_fd, &sb);
    if (error) {
       fprintf(stderr, "Can not stat file, error %d\n", error);
       return error;
    }
    /*
     * Alloc space for the extent buffer. We swag
     * and get 24 extents at a time.
     */
    numbufs = EXTMAX;
    allocsize = sizeof(CvExternalExtent_t) * numbufs;
    buf = malloc(allocsize);
    if (buf == NULL) {
       fprintf(stderr, "Can not alloc space for extent buffers\n");
       return 1;
    }
    memset(buf, 0, allocsize);
    if (ErrorFlag == 3) {   /* EINVAL */
       offset = 99999;
    }
    else  {
       offset = 0;
    }
    cnt = 0;
    do {
        if (ErrorFlag == 2)   /* EFAULT */
        {
            error = CvApi_GetExtList(f_fd, offset, &numbufs, junkbuf);
        }
        else {
            error = CvApi_GetExtList(f_fd, offset, &numbufs, buf);
        }
    if (error) {
        if (error != ENOENT) {
          fprintf(stderr,
               "Can not get extent list, offset " ARG64X " error %d\n",
               offset, error);
    }
    /*
     * ENOENT means no more extents
     */
           if ((error == ENOENT) && (ErrorFlag != 1))
           {
```

```
                error = 0;
                break;
        }
    }
    exp = buf;
    for (i=0; i<numbufs; i++) {

            printf("Extent %d frbase " ARG64X " sg 0x%x fsbase " ARG64X
                   " fsend " ARG64X " depth %x\n",
              cnt++, exp->ex_frbase, exp->ex_sg, exp->ex_base, exp->ex_end,
              exp->ex_depth);
             /* Look for the next extent. */
            offset = exp->ex_frbase + ((exp->ex_end + 1) - exp->ex_base);
            exp++;
        }
    } while (error == 0);
    printf("%d total extents\n", cnt);
    return error;
}
/*
 * PURPOSE
 *  Punch a hole in a file
 */
int
PunchHole(
    int     f_fd,
    char    *f_filename,
    uint64_t f_offset,
    uint64_t f_nbytes)
{
    uint64_toffset, end, nblks, freed;
    int    error = 0;
    offset = f_offset;
    end = f_offset + f_nbytes;
    end--;            /* last byte, inclusive */
    if (Verbose) {
       printf("Punching a hole in '%s' from " ARG64X " to " ARG64X "\n",
           f_filename, f_offset, end);
    }
    error = CvApi_PunchHole(f_fd, &offset, &end, &nblks, &freed);
    if (error) {
       return error;
    }
    if (Verbose) {
       printf("Hole punched in '%s' from " ARG64X " to " ARG64X
             "blks freed " ARG64X " , blocks now in file " ARG64X "\n",
             f_filename, offset, end, freed, nblks);
    }
    return error;
}
/*
 * PURPOSE
 *  Load extents space
 */
int
LoadExtents(
    int     f_fd,
    char    *f_filename,
```

```
    uint64_t f_offset,
    uint64_t f_nbytes)
{
    uint64_toffset, nbytes;
    int    error = 0;
    offset = f_offset;
    nbytes = f_nbytes;
    if (Verbose) {
       printf("Loading extents in '%s' " ARG64D " bytes at offset "
              ARG64D "\n", f_filename, nbytes, offset);
    }
    error = CvApi_LoadExtents(f_fd, nbytes, offset);
    if (error) {
           return error;
    }
    if (Verbose) {
       printf("Loaded " ARG64D " bytes starting at offset " ARG64D "\n",
              nbytes, offset);
    }
    return error;
}
/*
 * PURPOSE
 *  Alloc space
 */
int
AllocSpace(
    int     f_fd,
    char    *f_filename,
    uint64_t f_offset,
    uint64_t f_nbytes,
    uint64_t f_affinity,
    uint32_t f_flags)
{
    uint64_toffset, nbytes;
    int    error = 0;
    offset = f_offset;
    nbytes = f_nbytes;
    if (Verbose) {
       printf("Allocating space in '%s' " ARG64D " bytes at offset "
              ARG64D "\n", f_filename, nbytes, offset);
    }
    error = CvApi_AllocSpace(f_fd, &nbytes, &offset, f_affinity,  f_flags);
    if (error) {
           return error;
    }
    if (Verbose) {
       printf("Allocated " ARG64D " bytes starting at offset " ARG64D "\n",
              nbytes, offset);
    }
    return error;
}
/*
 * PURPOSE
 *  Get PerfectFit status
 */
int
PerfectFitStatus(
```

```
    int     f_fd,
    char    *f_filename)
{
    int isperfectfit;
    int error;
    error = CvApi_GetPerfectFitStatus(f_fd, &isperfectfit);
    if (error) {
          return error;
    }
    if (Verbose) {
       printf("File %s does%s have the PerfectFit bit set.\n",
             f_filename, isperfectfit ? "" : " not");
    }
    return error;
}
/*
 * PURPOSE
 *  Get PerfectFit status
 */
int
TogglePerfectFitStatus(
    int          f_fd,
    char        *f_filename)
{
    int isperfectfit;
    int setperfectfit;
    int error;
    error = CvApi_GetPerfectFitStatus(f_fd, &isperfectfit);
    if (error) {
          return error;
    }
    if (Verbose) {
        printf("%s PerfectFit bit on file %s.\n",
              isperfectfit ? "Clearing" : "Setting", f_filename);
    }
    if(isperfectfit)
       setperfectfit = 0;
    else
       setperfectfit = 1;
    error = CvApi_SetPerfectFitStatus(f_fd, setperfectfit);
    return error;
}
/*
 * PURPOSE
 *  Get the affinity for a file
 */
int
GetAffinity(
    int     f_fd,
    char    *f_filename)
{
    uint8_t *cp;
    int     error = 0;
    uint64_t affinity;
    int     i;
      error = CvApi_GetAffinity(f_fd, &affinity);
    if (error == 0) {
       printf("Affinity for file '%s' is : ", f_filename);
```

```
       cp = (uint8_t *)&affinity;
       for (i=0; i<8; i++) {
           printf("%c", *cp++);
       }
       printf("\n");
    }
    return error;
}
/*
 * PURPOSE
 *  Set a file into concurrent write mode
 */
int
ConcWrite(
    int     f_fd,
    char    *f_filename)
{
    int   error = 0;
    error = CvApi_SetConcWrite(f_fd);
    if (error == 0) {
       printf("File '%s' is now in concurrent write mode. ", f_filename);
    }
    return error;
}
/*
 * PURPOSE
 *  Unset a file from concurrent write mode
 */
int
NoConcWrite(
    int      f_fd,
    char    *f_filename)
{
    int   error = 0;
    error = CvApi_ClearConcWrite(f_fd);
    if (error == 0) {
        printf("File '%s' is now out of concurrent write mode. ", f_filename);
    }
    return error;
}
/*
 * PURPOSE
 *   Print out info about a stripe group
 */
int
SgInfo(
    int     f_fd,
    int      f_sg)
{
    uint64_t    totblks;
    uint64_t    freeblks;
    uint32_t    breadth;
    uint32_t    depth;
    uint32_t    flags;
    uint32_t    bsize;
    char        *junkbuf = NULL;
    char        sgbuf[SG_NAMELEN];
    uint64_t    nativekeys[32];
```

```
    uint32_tkeycnt;
    int         error = 0;
    uint32_ti;

    if (ErrorFlag == 1) { /* ENOENT */
        f_sg = -1;
    }
    keycnt = 32;

    if (ErrorFlag == 2) { /* EFAULT */
         error = CvApi_GetSgInfo(f_fd, f_sg,
                        &totblks, &freeblks, &breadth,
                        &depth, &flags, &bsize, junkbuf,
                        nativekeys, &keycnt);
                            }
    else {
        error = CvApi_GetSgInfo(f_fd, f_sg,
                        &totblks, &freeblks, &breadth,
                        &depth, &flags, &bsize, sgbuf,
                        nativekeys, &keycnt);
    }
    if (error)
       return error;
    printf("Stripe group info for '%s' <%d>\n", sgbuf, f_sg);
    printf("Total blocks " ARG64D " (" ARG64X ")\n", totblks, totblks);
    printf("free blocks " ARG64D " (" ARG64X ")\n", freeblks, freeblks);
    printf("Breadth %x depth %x bsize %d (0x%x)\n",
       breadth, depth, bsize, bsize);
    printf("Flags (0x%x) ", flags);
    if (flags & SG_PART_VALID)
       printf(" valid " );
    if (flags & SG_PART_ONLINE)
       printf(" online " );
    if (flags & SG_PART_METADATA)
       printf(" metadata " );
    if (flags & SG_PART_JOURNAL)
       printf(" journal " );
    if (flags & SG_PART_EXCLUSIVE)
       printf(" exclusive " );
    printf("\n");
    for(i = 0; i < keycnt; i++) {
       char buf[9];
       memcpy(&buf, &nativekeys[i], sizeof(nativekeys[i]));
       buf[8] = '\0';
       printf("NativeKey[%d] = %s (%llx)\n", i, buf, nativekeys[i]);
    }
    return error;
}
/*
 * PURPOSE
 *   Print out disk info given a stripe group ordinal
 */
int
DiskInfo(
    int     f_fd,
    int      f_sg)
{
    CvDiskInfo_t dinfo[MAXDISKS];
```

```
    uint32_t ndisks;
    int error;
    ndisks = MAXDISKS;
    error = CvApi_GetDiskInfo(f_fd, f_sg, dinfo, (int *)&ndisks);
    if (!error) {
       int i, j;
       printf("Disk Info for Stripe group #%d:\n", f_sg);
       for(i = 0; i < (int)ndisks; i++) {
           printf("\tDisk #%d: name=\"%s\" vhsize=%u nameloc=%u secsize=%u
serialnum=%s\n",
              i, dinfo[i].di_name, dinfo[i].di_vhsize,
              dinfo[i].di_nameloc, dinfo[i].di_sectorsize,
              dinfo[i].di_serialnum);
          for(j = 0; j < dinfo[i].di_npaths; j++) {
             printf("\t\tpath[%d]: blkdev=%s, rawdev=%s\n",
                 j,
                 dinfo[i].di_paths[j].d_bdev,
                 dinfo[i].di_paths[j].d_rdev);
          }
       }
    }
    return error;
}
/*
 * PURPOSE
 *   Print out ordinal of stripe group
 */
int
SgOrdinal(
    int  fd,
    char *sgname)
{
    uint32_t    sg;
    char        sgbuf[SG_NAMELEN];
    char        *junkbuf = NULL;
    int         error = 0;
    if ((ErrorFlag == 1) || (ErrorFlag == 3)) { /* ENOENT */
       sgbuf[0] = '\0';
    } else {
       strncpy(sgbuf, sgname, SG_NAMELEN);
       sgbuf[SG_NAMELEN - 1] = '\0';
    }
    if (ErrorFlag == 2) { /* EFAULT */
        error = CvApi_GetSgName(fd, SG_GETNUM, &sg, junkbuf);
    }
    else {
        error = CvApi_GetSgName(fd, SG_GETNUM, &sg, sgbuf);
    }
    if (error)
        return error;
    printf("Stripe group ordinal for '%s' : <%d>\n", sgname, sg);
    return error;
}
/*
 * PURPOSE
 *   Print out name of stripe group
 */
int
```

```
SgName(
    int       fd,
    uint32_t  sg)
{
    char        sgbuf[SG_NAMELEN];
    char        *junkbuf = NULL;
    int         error = 0;

    if (ErrorFlag == 1) { /* ENOENT */
       sgbuf[0] = '\0';
    }
    if ((ErrorFlag == 1) || (ErrorFlag == 3)) { /* ENOENT */
       sg = -1;                                 /*   or   */
    }                                           /* EINVAL */
    if (ErrorFlag == 2) { /* EFAULT */
         error = CvApi_GetSgName(fd, SG_GETNAME, &sg, junkbuf);
    }
    else {
         error = CvApi_GetSgName(fd, SG_GETNAME, &sg, sgbuf);
    }
    if (error)
        return error;
    printf("Stripe group name for sg '%d' : <%s>\n", sg, sgbuf);
    return error;
}
/*
 * PURPOSE
 *   Set the read hole fail option for a file
 */
int
SetRDHoleFail(int  f_fd)
{
    int error = 0;
    error = CvApi_SetRdHoleFail(f_fd);
    if (error) {
        return error;
    }
    printf("File now in read hole fail mode\n");
    return error;
}
/*
 * PURPOSE
 *   Clear the read hole fail option for a file
 */
int
ClearRDHoleFail(int  f_fd)
{
    int error = 0;
    error = CvApi_ClearRdHoleFail(f_fd);
    if (error) {
        return error;
    }
    printf("File cleared from read hole fail mode\n");
    return error;
}
 /*
 * PURPOSE
 *   Retrieve the version info
```

```
 */
int
GetVersionInfo(int f_fd)
{
    int error = 0;
    VerInfoReply_tverinfo;
    error = CvApi_GetVerInfo(f_fd, &verinfo);
    if (error) {
        return error;
    }
    printf("Version string:\n %s \n", verinfo.vr_version);
    printf("Build string:\n %s \n", verinfo.vr_build);
    printf("Date string:\n %s \n", verinfo.vr_creationdate);
    printf("External API version: %d\n", verinfo.vr_apiversion);
    return error;
}
int
SetQuota(int f_fd, char *f_quotaname)
{
    int error;
    error = CvApi_SetQuota(f_fd, QUOTA_TYPE_USER, f_quotaname,
       (uint64_t)12000000, (uint64_t)10000000, 60);
    return(error);
}
int
GetQuota(int f_fd, char *f_quotaname)
{
    int error;
    GetQuotaReply_t qrep;
    error = CvApi_GetQuota(f_fd, QUOTA_TYPE_USER, f_quotaname, &qrep);
    if (!error) {
       printf("GetQuota results for %s:\n", f_quotaname);
       printf("hardlimit = %lld\n", qrep.gr_hardlimit);
       printf("softlimit = %lld\n", qrep.gr_softlimit);
       printf("cursize   = %lld\n", qrep.gr_cursize);
       printf("timelimit = %u\n", qrep.gr_timelimit);
       printf("timeout   = %u\n", qrep.gr_timeout);
    }
    return(error);
}
int
SetFileSize(int f_fd, uint64_t f_len)
{
    int error;
    error = CvApi_SetFileSize(f_fd, f_len);
    return(error);
}
/*
 * PURPOSE
 *  New Alloc space call
 */
int
VerifyAlloc(
    int          f_fd,
    char        *f_filename,
    uint64_t     f_offset,
    uint64_t     f_nbytes,
    uint32_t     f_flags)
```

```
{
    uint64_t    offset, nbytes;
    int         error = 0;
    offset = f_offset;
    nbytes = f_nbytes;
    if (Verbose) {
        printf("Allocating space in '%s' " ARG64D " bytes at offset "
                ARG64D " (flags 0x%X)\n", f_filename, nbytes, offset, f_flags);
    }
    error = CvApi_VerifyAlloc(f_fd, offset, nbytes, f_flags);
    if (error) {
            return error;
    }
    if (Verbose) {
        printf( ARG64D " bytes starting at offset " ARG64D " are allocated\n",
                nbytes, offset);
    }
    return error;
}

/*
 * PURPOSE
 *  Extract a value from the command line
 */
uint64_t
GetVal(
    char    *f_arg)
{
    uint64_tval = 0;

    val = strtoll(f_arg, NULL, 0);

    if (strrchr(f_arg, 'k'))
       val *= 1024;
    else if (strrchr(f_arg, 'm'))
       val *= (1024 * 1024);
    else if (strrchr(f_arg, 'g'))
       val *= (1024 * 1024 * 1024);
    return val;
}
int
main(argc, argv)
int    argc;
char   *argv[];
{
    char     *filename, *cp;
    char        *sgname = NULL;
    int     c, error = 0;
    int     fd;
    uint32_t flags = 0;
    uint32_t     sg = 0;
    uint64_t offset, nbytes, affinity, size;
    int     AllocFlag, ExtentFlag, PunchFlag, StatFlag, InfoFlag;
    int     GetAffFlag, SetAffFlag, PhysFlag, ConcFlag, NoConcFlag;
    int     SgNameFlag, SgOrdinalFlag, SetRDHoleFlag, ClearRDHoleFlag;
    int     GetQuotaFlag, SetQuotaFlag, DiskFlag, PerfectFlag;
    int     StatPlusFlag, SetFileSizeFlag, StatFsFlag;
    int     VerifyAllocFlag, SetPerfectFitFlag;
```

```
    char    *quotauser = NULL;
    int     offset_set, nbytes_set, size_set;
    int     openflags, VersFlag, LoadExtFlag, WaitFlag, OpenStatFlag;
    char    waitbuf[4];

    Progname = argv[0];
    if ((cp = strrchr(Progname, '/')) != NULL)
Progname = cp + 1;

    AllocFlag =  ExtentFlag = PunchFlag = StatFlag = InfoFlag = 0;
    PhysFlag = GetAffFlag = SetAffFlag = ConcFlag = NoConcFlag = 0;
    SgNameFlag =  SgOrdinalFlag = SetRDHoleFlag = ClearRDHoleFlag = 0;
    GetQuotaFlag = SetQuotaFlag = DiskFlag = PerfectFlag = SetFileSizeFlag = 0;
    VersFlag = LoadExtFlag = WaitFlag = OpenStatFlag = StatPlusFlag = 0;
    StatFsFlag = VerifyAllocFlag = SetPerfectFitFlag = 0;
    size_set = offset_set = nbytes_set = 0;
    affinity = offset = nbytes = 0;

    while ((c = getopt(argc, argv, "ABbCcD:EFfG:g:I:LOPpQ:q:RrSTa:lN:n:o:svVwxYyZz:"))
           != EOF)
    {
       switch (c) {
       case 'A':
           AllocFlag = 1;
           break;
       case 'B':
           StatFsFlag = 1;
           break;
       case 'b':
           flags |= ALLOC_SETSIZE;
           break;
       case 'C':
           ConcFlag = 1;
           break;
       case 'c':
           NoConcFlag = 1;
           break;
       case 'D':
           DiskFlag = 1;
           sg = strtoul(optarg, NULL, 0);
           break;
       case 'E':
           ExtentFlag = 1;
           break;
       case 'F':
           SetAffFlag = 1;
           break;
       case 'f':
           GetAffFlag = 1;
           break;
       case 'G':
           SgOrdinalFlag = 1;
           sg = strtoul(optarg, NULL, 0);
           break;
       case 'g':
           SgNameFlag = 1;
           sgname = optarg;
           break;
```

```
case 'I':
    InfoFlag = 1;
    sg = strtoul(optarg, NULL, 0);
    break;
case 'L':
    PhysFlag = 1;
    break;
case 'O':
    OpenStatFlag = 1;
    break;
case 'P':
    PunchFlag = 1;
    break;
case 'p':
    PerfectFlag = 1;
    break;
case 'R':
    SetRDHoleFlag = 1;
    break;
case 'r':
    ClearRDHoleFlag = 1;
    break;
case 'S':
    StatFlag = 1;
    break;
case 'T':
    StatPlusFlag = 1;
    break;
case 'a':
    /*
     * It is important to remember that affinities are
     * ASCII strings, so we have to special case '0'
     */
        if (strlen(optarg) > 8) {

            fprintf(stderr, "The affinity type (-a) argument must be "
            "eight characters or less.");
        Usage();
exit(2);
    }
    if ((strlen(optarg) == 1) && (optarg[0] == '0')) {
affinity = 0;
    } else {
            strncpy((char*)&affinity, optarg, strlen(optarg));
    }
    flags |= ALLOC_AFFINITY;
    break;
case 'l':
    flags |= ALLOC_LOAD_EXT;
    break;
case 'n':
        nbytes = GetVal(optarg);
    nbytes_set = 1;
    break;
case 'N':
    flags |= ALLOC_KEEPSIZE;
    nbytes = GetVal(optarg);
    nbytes_set = 1;
```

```
            break;
        case 'o':
                offset = GetVal(optarg);
            offset_set = 1;
            flags |= ALLOC_OFFSET;
            break;
        case 'Q':
            quotauser = optarg;
            SetQuotaFlag++;
            break;
        case 'q':
            quotauser = optarg;
            GetQuotaFlag++;
            break;
        case 's':
            flags |= ALLOC_STRIPE_ALIGN;
            break;
        case 't':
            flags |= ALLOC_PERFECTFIT;
            break;
        case 'v':
            Verbose = 1;
            break;
        case 'V':
            VersFlag = 1;
            break;
        case 'w':
                    WaitFlag = 1;
                    break;
                case 'x':
                    LoadExtFlag = 1;
                    break;
        case 'Y':
            VerifyAllocFlag = 1;
            break;
                case 'y':
                    SetPerfectFitFlag = 1;
                    break;
        case 'z':
                size = GetVal(optarg);
            size_set = 1;
            break;
        case 'Z':
            SetFileSizeFlag = 1;
            break;
        case '?':
        default:
            Usage();
            exit(2);
        }
    }

    if ((argc - optind) < 1) {
        fprintf(stderr, "Must supply filename\n");
        Usage();
        exit(3);
    }
    filename = argv[optind];
```

```
if (AllocFlag || VerifyAllocFlag ||
    PunchFlag || SetAffFlag || ConcFlag) {
    openflags = O_RDWR | O_CREAT;
} else {
    openflags = O_RDONLY;
}
if ((fd = open(filename, openflags, 0777)) < 0) {
    error = errno;
    fprintf(stderr, "Can not open filename %s, error '%s' (%d)\n",
         filename, strerror(error), error);
    exit( error);
}
if (AllocFlag) {
   /*
    * We used to check for 0 offset and size, but
    * now we default to 0 for the offset, and
    * if the user says alloc zero bytes, we need
    * to verify we return an error
    */
   error = AllocSpace(fd, filename, offset, nbytes, affinity,  flags);
   if (error) {
      fprintf(stderr, "Can not alloc " ARG64D " bytes in filename %s, "
             " error '%s' (%d)\n", nbytes, filename,
             strerror(error), error);
     exit(error);
}
}
if (VerifyAllocFlag) {
    error = VerifyAlloc(fd, filename, offset, nbytes, flags);
    if (error) {
        fprintf(stderr, "Can not alloc " ARG64D " bytes in filename %s, "
                " error '%s' (%d)\n", nbytes, filename,
                strerror(error), error);
        exit(error);
    }
}
if (SetPerfectFitFlag) {
    error = TogglePerfectFitStatus(fd, filename);
    if (error) {
        fprintf(stderr, "Can not get PerfectFit status for filename %s, "
                " error '%s' (%d)\n", filename,
                strerror(error), error);
        exit(error);
    }
}
if (PerfectFlag) {
   error = PerfectFitStatus(fd, filename);
   if (error) {
       fprintf(stderr, "Can not get PerfectFit status for filename %s, "
            " error '%s' (%d)\n", filename,
            strerror(error), error);
       exit(error);
   }
}
if (PunchFlag) {
   if ((offset_set == 0) || (nbytes_set == 0)) {
       fprintf(stderr, "Must supply offset and number of bytes\n");
```

```
            Usage();
            exit(3);
        }
        error = PunchHole(fd, filename, offset, nbytes);
    }
    if (ExtentFlag) {
        error = GetExtList(fd);
        if (error) {
          fprintf(stderr, "Can not get extent list for file '%s'"
                " error '%s' (%d)\n", filename,
                strerror(error), error);
          exit(error);
        }
    }
    if (SetFileSizeFlag) {
        if (size_set == 0) {
            fprintf(stderr, "Must supply size\n");
            Usage();
            exit(3);
        }
        error = SetFileSize(fd, size);
        if (error) {
            fprintf(stderr, "Can not set size of "ARG64D" for file '%s'"
                  " error '%s' (%d)\n", size, filename,
                     strerror(error), error);
            exit(error);
        }
    }
    if (StatFlag) {
        error = StatFile(fd, filename);
        if (error) {
            fprintf(stderr, "Can not stat file '%s'"
                 " error '%s' (%d)\n", filename,
                    strerror(error), error);
            exit(error);
       }
    }
    if (StatPlusFlag) {
        error = StatFilePlus(fd, filename);
        if (error) {
            fprintf(stderr, "Can not do \"stat plus\" on file '%s'"
                  " error '%s' (%d)\n", filename,
                  strerror(error), error);
            exit(error);
        }
    }
    if (StatFsFlag) {
        error = StatFs(fd, filename);
        if (error) {
            fprintf(stderr, "Can not do StatFs on file '%s'"
                  " error '%s' (%d)\n", filename,
                  strerror(error), error);
            exit(error);
        }
    }
    if (OpenStatFlag) {
        error = OpenStatFile(fd, filename);
        if (error) {
```

```
        fprintf(stderr, "Can not open_stat file '%s'"
                " error '%s' (%d)\n", filename,
                strerror(error), error);
        exit(error);
    }
}
if (SetAffFlag) {
   if (Verbose) {
       printf("Setting affinity in '%s' to " ARG64X "\n",
          filename, affinity);
   }
   error = CvApi_SetAffinity(fd, affinity);
   if (error) {
       fprintf(stderr, "Can not set affinity for file '%s'"
             " error '%s' (%d)\n", filename,
             strerror(error), error);
       exit(error);
  }
}
if (GetAffFlag) {
   error = GetAffinity(fd, filename);
   if (error) {
       fprintf(stderr, "Can not get affinity for file '%s'"
            " error '%s' (%d)\n", filename,
            strerror(error), error);
       exit(error);
   }
}
if (PhysFlag) {
   if (offset_set == 0) {
       fprintf(stderr, "Must supply offset \n");
       Usage();
       exit(3);
   }
   error = GetPhysLoc(fd, filename, offset);
   if (error) {
       fprintf(stderr, "Can not get physical location for file '%s'"
            " error '%s' (%d)\n", filename,
            strerror(error), error);
       exit(error);
   }
}
if (ConcFlag) {
   error = ConcWrite(fd, filename);
   if (error) {
       fprintf(stderr, "Can not perform concurrent writes for file '%s'"
            " error '%s' (%d)\n", filename,
            strerror(error), error);
       exit(error);
   }
}
if (NoConcFlag) {
    error = NoConcWrite(fd, filename);
    if (error) {
        fprintf(stderr, "Can not clear concurrent writes for file '%s'"
                " error '%s' (%d)\n", filename,
                strerror(error), error);
        exit(error);
```

```
    }
}
if (SetQuotaFlag) {
   error = SetQuota(fd, quotauser);
   if (error) {
       fprintf(stderr, "Cannot set quota for user %s, error = %d\n",
            quotauser, error);
       exit(error);
   } else {
       printf("Quota for %s successfully set\n", quotauser);
   }
}
if (GetQuotaFlag) {
   error = GetQuota(fd, quotauser);
   if (error) {
       fprintf(stderr, "Cannot get quota for user %s, error = %d\n",
            quotauser, error);
       exit(error);
   }
}
if (InfoFlag) {
   error = SgInfo(fd, sg);
   if (error) {
       fprintf(stderr, "Can not get stripe group info for %d"
            " error '%s' (%d)\n", sg,
            strerror(error), error);
       exit(error);
   }
}
if (DiskFlag) {
   error = DiskInfo(fd, sg);
   if (error) {
       fprintf(stderr, "Can not get disk info for stripe group %d"
            " error '%s' (%d)\n", sg,
            strerror(error), error);
       exit(error);
   }
}
if (SgNameFlag) {
   error = SgOrdinal(fd, sgname);
   if (error) {
       fprintf(stderr, "Can not get stripe group ordinal for %s"
            " error '%s' (%d)\n", sgname,
            strerror(error), error);
       exit(error);
   }
}
if (SgOrdinalFlag) {
   error = SgName(fd, sg);
   if (error) {
       fprintf(stderr, "Can not get stripe group name for %d"
            " error '%s' (%d)\n", sg,
            strerror(error), error);
       exit(error);
   }
}
if (SetRDHoleFlag) {
    error = SetRDHoleFail(fd);
```

```
        if (error) {
            fprintf(stderr, "Can not set RD Hole Fail for file '%s'"
                    " error '%s' (%d)\n", filename,
                    strerror(error), error);
            exit(error);
        }
    }
    if (ClearRDHoleFlag) {
        error = ClearRDHoleFail(fd);
        if (error) {
            fprintf(stderr, "Can not clear RD Hole fail for file '%s'"
                    " error '%s' (%d)\n", filename,
                    strerror(error), error);
            exit(error);
        }
    }
    if (VersFlag) {
        error = GetVersionInfo(fd);
        if (error) {
            fprintf(stderr, "Can not retrieve version info "
                    " error '%s' (%d)\n",
                    strerror(error), error);
            exit(error);
        }
    }
    if (LoadExtFlag) {
       error = LoadExtents(fd, filename, offset, nbytes);
       if (error) {
           fprintf(stderr, "Can not load " ARG64D " bytes in filename %s, "
                " error '%s' (%d)\n", nbytes, filename,
               strerror(error), error);
           exit(error);
       }
    }
    if (WaitFlag) {
       printf("Waiting, press return to continue...\n");
       fflush(stdout);
       (void)fgets(waitbuf, sizeof(waitbuf), stdin);
    }
    return 0 ;
}
```