NAME

sfio – safe/fast string/file input/output

SYNOPSIS

```
#include   <sfio.h>
```

DATA TYPES

```
Void_t;
Sfoff_t;
Sflong_t;
Sfulong_t;
Sfdouble_t;

Sfio_t;

Sfdisc_t;
ssize_t   (*Sfread_f)(Sfio_t*, Void_t*, size_t, Sfdisc_t*);
ssize_t   (*Sfwrite_f)(Sfio_t*, const Void_t*, size_t, Sfdisc_t*);
Sfoff_t   (*Sfseek_f)(Sfio_t*, Sfoff_t, int, Sfdisc_t*);
int       (*Sfexcept_f)(Sfio_t*, int, Void_t*, Sfdisc_t*);

Sffmt_t;
int       (*Sffmtext_f)(Sfio_t*, Void_t*, Sffmt_t*);
int       (*Sffmtevent_f)(Sfio_t*, int, Void_t*, Sffmt_t*);

SFIO_VERSION
```

BIT FLAGS

```
SF_STRING
SF_READ
SF_WRITE
SF_APPEND
SF_LINE
SF_SHARE
SF_PUBLIC
SF_MALLOC
SF_STATIC
SF_IOCHECK
SF_BUFCONST
SF_WHOLE
```

OPENING/CLOSING STREAMS

```
Sfio_t*   sfnew(Sfio_t* f, Void_t* buf, size_t size, int fd, int flags);
Sfio_t*   sfopen(Sfio_t* f, const char* string, const char* mode);
Sfio_t*   sfpopen(Sfio_t* f, const char* cmd, const char* mode);
Sfio_t*   sftmp(size_t size);
int       sfclose(Sfio_t* f);
```

INPUT/OUTPUT OPERATIONS

```
int       sfgetc(Sfio_t* f);
int       sfputc(Sfio_t* f, int c);
int       sfnputc(Sfio_t* f, int c, int n);
int       sfungetc(Sfio_t* f, int c);

Sfulong_t  sfgetu(Sfio_t* f);
int       sfputu(Sfio_t* f, Sfulong_t v);
Sflong_t   sfgetl(Sfio_t* f);
```

```
int        sfputl(Sfio_t* f, Sflong_t v);
Sfdouble_t sfgetd(Sfio_t* f);
int        sfputd(Sfio_t* f, Sfdouble_t v);

char*      sfgetr(Sfio_t* f, int rsc, int type);
ssize_t    sfputr(Sfio_t* f, const char* s, int rsc);
Sfoff_t    sfmove(Sfio_t* fr, Sfio_t* fw, Sfoff_t n, int rsc);

ssize_t    sfread(Sfio_t* f, Void_t* buf, size_t n);
ssize_t    sfwrite(Sfio_t* f, const Void_t* buf, size_t n);
Sfoff_t    sfseek(Sfio_t* f, Sfoff_t offset, int type);
Void_t*    sfreserve(Sfio_t* f, ssize_t n, int lock);
```

DATA FORMATTING

```
int        sfscanf(Sfio_t* f, const char* format, ...);
int        sfsscanf(const char* s, const char* format, ...);
int        sfvsscanf(const char* s, const char* format, va_list args);
int        sfvscanf(Sfio_t* f, const char* format, va_list args);

int        sfprintf(Sfio_t* f, const char* format, ...);
char*      sfprints(const char* format, ...);
int        sfsprintf(char* s, int n, const char* format, ...);
int        sfvsprintf(char* s, int n, const char* format, va_list args);
int        sfvprintf(Sfio_t* f, const char* format, va_list args);

Sffmt_t;
int        (*Sffmtext_f)(Sfio_t* f, Void_t* v, Sffmt_t* fe);
int        (*Sffmtevent_f)(Sfio_t* f, int type, Void_t* v, Sffmt_t* fe);
void       va_copy(va_list to, va_list fr);
long       sffmtversion(Sffmt_t* fe, type);
```

BUFFERING, SYNCHRONIZATION

```
Void_t*    sfsetbuf(Sfio_t* f, Void_t* buf, size_t size);
int        sfsync(Sfio_t* f);
int        sfpoll(Sfio_t** flist, int n, int timeout);
Sfio_t*    sfpool(Sfio_t* f, Sfio_t* poolf, int mode);
int        sfpurge(Sfio_t* f);
```

DISCIPLINE, EVENT HANDLING

```
Sfdisc_t*  sfdisc(Sfio_t* f, Sfdisc_t* disc);
int        sfraise(Sfio_t* f, int type, Void_t* data);
ssize_t    sfrd(Sfio_t* f, Void_t* buf, size_t n, Sfdisc_t* disc);
ssize_t    sfwr(Sfio_t* f, const Void_t* buf, size_t n, Sfdisc_t* disc);
Sfoff_t    sfsk(Sfio_t* f, Sfoff_t offset, int type, Sfdisc_t* disc);
```

STREAM CONTROL

```
int        sfset(Sfio_t* f, int flags, int i);
int        sfsetfd(Sfio_t* f, int fd);
Sfio_t*    sfstack(Sfio_t* base, Sfio_t* top);
Sfio_t*    sfswap(Sfio_t* f1, Sfio_t* f2);
```

STREAM INFORMATION

```
Sfoff_t    sfsize(Sfio_t* f);
Sfoff_t    sftell(Sfio_t* f);
ssize_t    sfvalue(Sfio_t* f);
int        sffileno(Sfio_t* f);
```

```
int      sfstacked(Sfio_t* f);
int      sfeof(Sfio_t* f);
int      sferror(Sfio_t* f);
int      sfclrerr(Sfio_t* f);
int      sfclrlock(Sfio_t* f);

int      sfnotify(void (*notify)(Sfio_t* f, int type, int fd));
```

MISCELLANEOUS FUNCTIONS

```
ssize_t  sfslen();
int      sfulen(Sfulong_t v);
int      sfllen(Sflong_t v);
int      sfdlen(Sfdouble_t v);
ssize_t  sfpkrd(int fd, Void_t* buf, size_t n,
            int rsc, long tm, int peek);
```

FULL STRUCTURE SFIO_T

```
#include  <sfio_t.h>
#define   SFNEW(buf,size,file,flags,disc)
```

EXAMPLE DISCIPLINES

```
int      sfdcdio(Sfio_t* f, size_t bufsize);
int      sfdcdos(Sfio_t* f);
int      sfdcfilter(Sfio_t* f, const char* cmd);
int      sfdclzw(Sfio_t* f);
int      sfdcseekable(Sfio_t* f);
int      sfdcslow(Sfio_t* f);
int      sfdcsubstream(Sfio_t* f, Sfio_t* parent,
                Sfoff_t offset, Sfoff_t extent);
int      sfdctee(Sfio_t* f, Sfio_t* tee);
int      sfdcunion(Sfio_t* f, Sfio_t** array, int n);
```

STDIO-COMPATIBILITY

```
#include  <stdio.h>
cc ... -lstdio -lsfio
```

DESCRIPTION

Sfio is a library of I/O functions to manage buffered streams. Each Sfio stream is a *file stream*, representing a file (see open(2)), or a *string stream*, representing a memory segment. Beyond the usual I/O operations on streams, Sfio provides I/O disciplines for extended data processing, stream stacks for recursive stream processing, and stream pools for automatic data synchronization. The sfprintf()/sfscanf() functions allow applications to define their own formatting patterns as well as to redefine existing patterns.

A discipline defines analogues of the system calls read(2), write(2) and lseek(2). Such system calls or their discipline replacements are used to process stream data. Henceforth, ‘‘*system call*’’ will mean a system call or its discipline replacement.

A system call is said to cause an exception if its return value is non-positive. Unless overridden by exception handlers (see sfdisc()), an interrupted system call (errno == EINTR on UNIX systems) will be automatically reinvoked to continue the ongoing operation.

The buffer of a stream is typically a memory segment allocated via malloc(3) or supplied by the application. File streams may also use memory mapping (mmap(2)) if that is more efficient. When memory mapping is used, the underlying file should not be truncated while the stream is active. Memory mapping can be turned off using sfsetbuf().

There are three *standard streams*: sfstdin for input (file descriptor 0 on UNIX systems), sfstdout for normal output (file descriptor 1), and sfstderr for error output (file descriptor 2).

DATA TYPES

Void_t*

This defines a type suitable to exchange data of unknown types between application and Sfio. Void_t is a macro defined as void for ANSI-C and C++ and char for other compilation environments.

Sfoff_t

This defines an integral type suitable to address the largest possible file extent.

Sfulong_t, Sflong_t, Sfdouble_t

These are respectively the largest unsigned integer, signed integer, and floating point value types on the local platform.

Sfio_t

This defines the stream type.

Sfdisc_t
ssize_t (*Sfread_f)(Sfio_t*, Void_t*, size_t, Sfdisc_t*)
ssize_t (*Sfwrite_f)(Sfio_t*, const Void_t*, size_t, Sfdisc_t*)
Sfoff_t (*Sfseek_f)(Sfio_t*, Sfoff_t, int, Sfdisc_t*)
int (*Sfexcept_f)(Sfio_t*, int, Void_t*, Sfdisc_t*)

Sfdisc_t defines a stream discipline structure. Sfread_f, Sfwrite_f and Sfseek_f are the types of discipline functions to replace the system calls: read(2), write(2) and lseek(2). Sfexcept_f is the type of an event-handling function. See sfdisc() for more details.

Sffmt_t
int (*Sffmtext_f)(Sfio_t*, Void_t*, Sffmt_t*)
int (*Sffmtevent_f)(Sfio_t*, int, Void_t*, Sffmt_t*)

Sffmt_t defines a formatting environment that can be used to extend scanning and formatting in the sfprint()/sfscanf() functions. Sffmtext_f and Sffmtevent_f define the types of extension functions definable in Sffmt_t. See Sffmt_t below for more details.

SFIO_VERSION

This is a macro value of type long int that defines the current version number of Sfio. For example, the Sfio1998's version number is 19980501L (which also indicates its release date).

BIT FLAGS

A number of bit flags control stream operations. They are set either at stream initialization or by calling sfset(). Following are the flags:

SF_STRING:

The stream is memory-based.

SF_READ, SF_WRITE, SF_APPEND:

Flags SF_READ and SF_WRITE indicate readability and writability. Flag SF_APPEND asserts that the stream is a file opened in append mode (see open(2) and fcntl(2)) so that data is always output at the end of file. On systems without direct support for append mode, Sfio uses lseek(2) or its discipline replacement to approximate this behavior.

SF_LINE:

The stream is line-oriented. For a SF_WRITE stream, this means that buffered data is flushed whenever a new-line character, \n, is output. For a SF_READ stream, SF_LINE is only significant during calls to functions in the sfscanf() family. SF_LINE is set on initialization of any stream representing a terminal device.

SF_SHARE, SF_PUBLIC:

Flag SF_SHARE means that the underlying file descriptor is shared by independent entities (for example, multiple processes).

For a seekable file stream, SF_SHARE means that the logical stream and the physical file positions will be made the same before a system call to perform physical I/O. There are different possibilities. If SF_PUBLIC is not set, the physical file position is made equal to the logical stream position. If SF_PUBLIC is set, there are two cases. If the physical file position has changed from its last known position, the logical stream position is made equal to the new physical file position. Finally, if the physical file location remains the same as its last known position, the physical file position is made the same as the logical stream position.

For an unseekable stream (e.g., pipes or terminal devices), if possible, SF_SHARE means that the block and record I/O operations (sfread(), sfwrite(), sfmove(), sfgetr(), sfputr(), sfreserve(), sfscanf() and sfvprintf()) will ensure: (1) after each writing operation, the stream is synchronized and (2) each reading operation only reads the requested amount. Note, however, that (2) is not always possible without proper OS facilities such as recv(2) or streamio(4).

A standard stream that is seekable will be initialized with SF_SHARE|SF_PUBLIC.

SF_MALLOC:
: The stream buffer was obtained via malloc(3) and can be reallocated or freed.

SF_STATIC:
: The stream structure should not be freed when closed (sfclose()). This flag is used by an applications that allocate their own stream structures. Such applications must use the header file sfio_t.h instead of sfio.h.

SF_IOCHECK:
: If the stream has a discipline exception handler, exceptions will be raised in sfsync(), sfpurge() or before a system call read(2) or write(2) (see sfdisc()).

SF_BUFCONST:
: The application guarantees that a stream buffer obtained via sfreserve() or sfgetr() will not be modified. This allows Sfio to tune buffer management and memory maps. For example, a memory-mapped stream will map with MAP_SHARED on and PROT_WRITE off so that the file itself will likely be the backing store for mapped pages.

SF_WHOLE:
: This flag guarantees that data written in any single sfwrite() or sfputr() call will always be output as a whole to the output device. This is useful in certain applications (e.g., networking) where a complex object must be output without being split in different system calls. Note that the respective stream still buffers data as the buffer can accomodate.

OPENING/CLOSING STREAMS

Sfio_t* sfnew(Sfio_t* f, Void_t* buf, size_t size, int fd, int flags)

This function creates or renews a stream. It returns the new stream on success and NULL on error.

f:
: If f is NULL, a new stream is created. Otherwise, f is reused. In this case, if flags does not have SF_EOF, f shall be closed via sfclose() before being reused. During a stream renewal, buffer, pool and discipline stack are preserved. Note that, except for SF_STATIC streams, renewing a stream already closed will result in undefined behavior.

buf, size:
: These determine a buffering scheme. See sfsetbuf() for more details.

fd:
: If SF_STRING is specified in flags, this is ignored. Otherwise, fd is a file descriptor (e.g., from open(2)) to use for raw data I/O. Note that Sfio supports unseekable file descriptors opened for both read and write, e.g., sockets.

flags:
: This is composed from SF_EOF and bit values defined in the **BIT FLAGS** section.

Sfio_t* sfopen(Sfio_t* f, const char* string, const char* mode)

If string is NULL and f is a file stream that has not performed any I/O operation, sfopen() will change the modes of f according to mode. In this case, sfopen() returns f on success and NULL on error. This somewhat unusual usage of sfopen() is good for changing the predefined modes of standard streams.

sfopen() is normally used to create a new stream or renew a stream. In this case, it returns the new stream on success and NULL on error. Below are the meanings of the arguments:

f: This is treated as in sfnew().

string: This is a file name or a string to perform I/O on.

mode: This is composed from the set of letters {s, r, w, +, a, x, b, t}.

s specifies opening a string stream. string can be a null-terminated string or NULL. Specifying s alone is equivalent to specifying sr. If s is not specified, string defines a file name.

r and w specify read and write modes. Write mode creates and/or truncates the given file to make an empty file. The + modifier indicates that the stream is opened for both read and write.

a specifies append mode, i.e., data is always output at end of file.

x specifies exclusive mode, i.e., a file opened for writing should not already exist.

b and t specify binary and text modes.

Sfio_t* sfpopen(Sfio_t* f, const char* cmd, const char* mode)

This function opens a stream that corresponds to the coprocess cmd. The argument mode should be composed from r, w, and +. The argument f, if not NULL, is a stream to be renewed (see sfnew()). sfpopen() returns the new stream or NULL on error.

The standard input/output of cmd is connected to the application via a pipe if the stream is opened for writing/reading. If the stream is opened for both reading and writing, there will be two different associated file descriptors, one for each type of I/O (note the effect on sffileno()).

On opening a coprocess for writing (i.e., mode contains w or +), the signal handler for SIGPIPE in the parent application will be set to SIG_IGN if it is SIG_DFL at that time. This protects the parent application from being accidentally killed on any attempt to write to a coprocess that closes its reading end. Applications that need to detect such write errors should use disciplines and exception handlers (see sfdisc()).

The command cmd is executed by an *interpreter* which is either /bin/sh or an executable command defined by the environment variable SHELL. In either case, the interpreter is invoked with 2 arguments, respectively -c and the given command cmd. When the interpreter is /bin/sh or /bin/ksh, sfpopen() may execute the command cmd itself if there are no shell meta-characters in cmd.

Sfio_t* sftmp(size_t size)

This function creates a stream for temporary data. It returns the new stream or NULL on error.

A stream created by sftmp() can be completely or partially memory-resident. If size is SF_UNBOUND, the stream is a pure string stream. If size is zero, the stream is a pure file stream. Otherwise, the stream is first created as a string stream but when its buffer grows larger than size or on any attempt to change disciplines, a temporary file is created. Two environment variables, TMPPATH and TMPDIR, direct where temporary files are created. TMPPATH, if defined, specifies a colon-separated set of directories to be used in a round-robin fashion to create files. If TMPPATH is undefined, TMPDIR can be used to specify a single directory to create files. If neither of TMPPATH and TMPDIR are defined, /tmp is used.

int sfclose(Sfio_t* f)

This function closes the stream f and frees its resources. SF_STATIC should be used if the stream space is to be preserved. If f is the base of a stream stack (see sfstack()), all streams on the stack are closed. If f is a sfpopen-stream, sfclose() waits until the associated command terminates and returns its exit status. sfclose() returns -1 for failure and 0 for success.

SF_READ|SF_SHARE and SF_WRITE streams are synchronized before closing (see sfsync()). If f has disciplines, their exception handlers will be called twice. The first exception handler call has the type argument as one of SF_CLOSE or SF_NEW (see sfdisc().) The latter, SF_NEW is used when a stream is being closed via sfnew() so that it can be renewed. The second call uses type as SF_FINAL and is done after all closing operations have succeeded but before the stream itself is deallocated. In either case, if the exception handler returns a negative value, sfclose() will immediately return this value. If the exception handler returns a positive value, sfclose() will immediately return a zero value.

INPUT/OUPUT OPERATIONS

int sfgetc(Sfio_t* f)
int sfputc(Sfio_t* f, int c)

These functions read/write a byte from/to stream f. sfgetc() returns the byte read or -1 on error. sfputc() returns c on success and -1 on error.

ssize_t sfnputc(Sfio_t* f, int c, size_t n)

This function attempts to write the byte c to f n times. It returns the number of bytes actually written or -1 on failure.

int sfungetc(Sfio_t* f, int c)

This function pushes the byte c back into f. If c matches the byte immediately before the current position in buffered data, the current position is simply backed up (note the effect on sftell() and sfseek()). There is no theoretical limit on the number of bytes that can be pushed back into a stream. Pushed back bytes not part of buffered data will be discarded on any operation that implies buffer synchronization. sfungetc() returns c on success and -1 on failure.

Sfulong_t sfgetu(Sfio_t* f)
int sfputu(Sfio_t* f, Sfulong_t v)

These functions read and write Sfulong_t values in a compact variable-length portable format. Portability across a write architecture and a read architecture requires that the bit order in a byte is the same on both architectures and the written value is storable in an Sfulong_t on the read architecture. sfgetu() returns the value read or -1 on error. sfputu() returns the number of bytes written or -1 on error. See also sfulen().

Sflong_t sfgetl(Sfio_t* f)
int sfputl(Sfio_t* f, Sflong_t v)

These functions are similar to sfgetu() and sfputu() but for reading and writing (signed) Sflong_t values. See also sfllen().

Sfdouble_t sfgetd(Sfio_t* f)
int sfputd(Sfio_t* f, Sfdouble_t v)

These functions read and write Sfdouble_t values. In this case, portability depends on the input and output architectures having the same floating point value representation. Values are coded and decoded using ldexp(3) and frexp(3) so they are constrained to the sizes supported by these functions. See also sfdlen().

char* sfgetr(Sfio_t* f, int rsc, int type)

This function reads a record of data ending in the record separator rsc. After sfgetr() returns, the length of the record even if it is incomplete can be retrieved with sfvalue(). sfgetr() returns the record on success and NULL on error.

The type argument is composed of some subset of the below bit flags:

SF_STRING:
: A null byte will replace the record separator to make the record into a C string. Otherwise, the record separator is left alone.

SF_LOCKR:
: Upon successfully obtaining a record r, the stream will be locked from further access until it is released with a call sfread(f,r,0).

SF_LASTR:
: This should be used only after a failed sfgetr() to retrieve the last incomplete record. In this case, rsc is ignored.

ssize_t sfputr(Sfio_t* f, const char* s, int rsc)

This function writes the null-terminated string s to f. If rsc is non-negative, (unsigned char)rsc is output after the string. sfputr() returns the number of bytes written or -1 on failure.

Sfoff_t sfmove(Sfio_t* fr, Sfio_t* fw, Sfoff_t n, int rsc)

This function moves objects from input stream fr to output stream fw. sfmove() returns the number of objects moved or -1 on failure.

An object is either a byte or a record. The latter is indicated by a non-negative value for the record separator character rsc. If n is negative, all of fr will be moved. Otherwise, n indicates the number of objects to move. If either fr or fw is NULL, it acts as if it is a stream corresponding to /dev/null, the UNIX device that has no read data and throws away any write data. For example, the call sfmove(f,(Sfio_t*)0,(Sfoff_t)(-1),'\n') simply counts the number of lines in stream f.

ssize_t sfread(Sfio_t* f, Void_t* buf, size_t n)

This function reads up to n bytes from f into buffer buf. It returns the number of bytes actually read or -1 on error.

ssize_t sfwrite(Sfio_t* f, const Void_t* buf, size_t n)

This function writes n bytes from buf to f. If f is SF_STRING, and the buffer is not large enough, an SF_WRITE exception shall be raised. sfwrite() returns the number of bytes written or -1 on failure.

Sfoff_t sfseek(Sfio_t* f, Sfoff_t offset, int type)

This function sets a new I/O position for f. It returns the new position or -1 on failure.

If the stream is a SF_STRING stream and the new address is beyond the current buffer extent, an SF_SEEK exception will be raised (see sfdisc()).

The new position is determined based on offset and type which is composed from the bit flags:

0 or SEEK_SET:
: offset is the desired position.

1 or SEEK_CUR:
: offset is relative to the current position (see SF_PUBLIC below).

2 or SEEK_END:
: offset is relative to the physical end of file.

SF_SHARE:
: The stream is treated as if it has the control bit SF_SHARE on. This implies that a system call seek will be done to ensure that the location seeking to is valid.

SF_PUBLIC:
: The stream is treated as if it has the control bit SF_PUBLIC on. If the physical file position has changed from its last known location, the current position is taken as the new physical position. Otherwise, the current position is the logical stream position.

Void_t* sfreserve(Sfio_t* f, ssize_t n, int lock)

This function reserves a data block from f. For a SF_READ stream, a data block is a segment of data and for a SF_WRITE stream, it is a buffer suitable for writing. For consistency, a stream opened with SF_READ|SF_WRITE will be treated as if it is a SF_READ stream (see sfset() for setting a particular mode.) sfreserve() returns the obtained data block on success and NULL on failure.

After a sfreserve() call, whether or not it succeeds, sfvalue() can be used to obtain the size of the (may-have-been) available data block.

n != 0: f will be filled or flushed as necessary to make available a data block of size at least the absolute value of n. If this is successful and lock is non-positive, the I/O position will advance by the size of the available data block when n is negative or by exactly n when n is positive. For example, sfreserve(f,-1,0) returns a positive size data block and advances the I/O position by its size. On the other hand, sfreserve(f,1,0) returns a positive size data block and advances the I/O position by exactly 1.

n == 0: If lock is zero, f will be filled or flushed as necessary to ensure that a positive-size data block is available. If lock is non-zero, no fill or flush will be performed. In addition, if lock is positive, f will be locked from further access. Therefore, an application can lock f with sfreserve(f,0,1).

lock: When lock is positive, there are restrictions. If f is SF_READ and not using memory-mapping (see sfsetbuf()), reservation must be limited to stream buffer size. If f is SF_READ|SF_SHARE and unseekable, sfreserve() will peek at incoming data using either recv(2) or streamio(4) without reading ahead. In this case, if peeking is not supported by the underlying platform, sfreserve() will fail. Note that SF_SHARE is automatically on for sfstdin so programs (e.g., **/bin/sort**) that will consume all of input anyway should turn this bit off to reduce the number of system calls.

If a reservation successfully results in a data block data, and lock is positive, the stream I/O position does not advance and f will be locked until unlocked with sfread/sfwrite(f,data,size). sfread() should be used on read-only stream and sfwrite() should be used on write-only stream. A stream in both read and write modes can release the lock with either call (with associated operational semantics!)

DATA FORMATTING

Data printing and scanning are done via the sfprintf() and sfscanf() family of functions. These functions are similar to their ANSI-C fprintf() and fscanf() counterparts. However, the Sfio versions have been extended for both portability and generality. In particular, a notion of a formatting environment stack is introduced. Each formatting element on the stack defines a separate *formatting pair* of a format specification string, char* format (the usual second argument in the formatting functions), and an argument list, va_list args (the third argument in functions sfvprintf() and sfvscanf()). A formatting environment element may also specify extension functions to obtain or assign arguments and to provide new semantics for pattern processing. To simplify the description below, whenever we talk about an argument list, unless noted otherwise, it is understood that this means either the true argument list when there is no extension function or the action to be taken by such a function in processing arguments. The manipulation of the formatting environment stack is done via the pattern ! discussed below.

%! and Sffmt_t

The pattern %! manipulates the formatting environment stack to (1) change the top environment to a new environment, (2) stack a new environment on top of the current top, or (3) pop the top environment. The bottom of the environment stack always contains a virtual environment with the original formatting pair and without any extension functions.

The top environment of a stack, say fe, is automatically popped whenever its format string is completely processed. In this case, its event-handling function (if any) is called as (*eventf)(f,SF_FINAL,NIL(Void_t*),fe). The top environment can also be popped by giving an argument NULL to %! or by returning a negative value in an extension function. In these cases, the event-handling function is called as (*eventf)(f,SF_DPOP,form,fe) where form is the remainder of the format string. A negative return value from the event handling function will prevent the environment from being popped.

A formatting environment is a structure of type Sffmt_t which contains the following elements:

```
Sffmtext_f   extf;   /* extension processor       */
Sffmtevent_f eventf; /* event handler             */

char*        form;   /* format string to stack    */
va_list      args;   /* corresponding arg list    */

int          fmt;    /* pattern being processed   */
ssize_t      size;   /* object size               */
int          flags;  /* formatting control flags  */
int          width;  /* width of field            */
int          precis; /* precision required        */
int          base;   /* conversion base           */

char*        t_str;  /* type string in parentheses */
int          n_str;  /* length of t_str           */
```

The first four elements of Sffmt_t must be defined by the application. The two function fields should not be changed during processing. Other elements of Sffmt_t are set on calls to the extension function Sffmt_t.extf and, in turn, can be modified by this function to redirect formatting or scanning. For example, consider a call from a sfprintf() function to process an unknown pattern %t (which we may take to mean ‘‘time’’) based on a formatting environment fe. fe->extf may reset fe->fmt to ‘d’ upon returing to cause sfprintf() to process the value being formatted as an integer.

Below are the fields of Sffmt_t:

extf: extf is a function to extend scanning and formatting patterns. Its usage is discussed below.

eventf: This is a function to process events as discussed earlier.

form and args:

This is the formatting pair of a specification string and corresponding argument list. When an environment fe is being inserted into the stack, if fe->form is NULL, the top environment is changed to fe and its associated extension functions but processing of the current formatting pair continues. On the other hand, if fe->form is not NULL, the new environment is pushed onto the stack so that pattern processing will start with the new formatting pair as well as any associated extension functions. During processing, whenever extf is called, form and args will be set to the current values of the formatting pair in use.

fmt: This is the pattern being processed.

size: This is the size of the object being processed.

flags: This is a collection of bits defining the formatting flags specified for the pattern. The bits are:

SFFMT_LEFT: Flag - in sfprintf().

SFFMT_SIGN: Flag + in sfprintf().

SFFMT_BLANK: Flag *space* in sfprintf().

SFFMT_ZERO: Flag 0 in sfprintf().

SFFMT_THOUSAND: Flag ' in sfprintf().

SFFMT_LONG: Flag l in sfprintf() and sfscanf().

SFFMT_LLONG: Flag ll in sfprintf() and sfscanf().

SFFMT_SHORT: Flag h in sfprintf() and sfscanf().

SFFMT_LDOUBLE: Flag L in sfprintf() and sfscanf().

SFFMT_IFLAG: flag I in sfprintf() and sfscanf().

SFFMT_ALTER: Flag # in sfprintf() and sfscanf().

SFFMT_SKIP: Flag * in sfscanf().

SFFMT_ARGPOS: This indicates argument processing for pos$.

SFFMT_VALUE: This is set by fe->extf to indicate that it is returning a value to be formatted or the address of an object to be assigned.

width: This is the field width.

precis: This is the precision.

base: This is the conversion base.

t_str and n_str:
This is the type string and its size.

int (*Sffmtext_f)(Sfio_t* f, Void_t* v, Sffmt_t* fe)

This is the type of the extension function fe->extf to process patterns and arguments. Arguments are always processed in order and fe->extf is called exactly once per argument. Note that, when pos$ (below) is not used anywhere in a format string, each argument is used exactly once per a corresponding pattern. In that case, fe->extf is called as soon as the pattern is recognized and before any scanning or formatting. On the other hand, when pos$ is used in a format string, an argument may be used multiple times. In this case, all arguments shall be processed in order by calling fe->extf exactly once per argument before any pattern processing. This case is signified by the flag SFFMT_ARGPOS in fe->flags.

In addition to the predefined formatting patterns and other application-defined patterns, fe->extf may be called with fe->fmt being one of '(' (left parenthesis), '.' (dot), and 'I'. The left parenthesis requests a string to be used as the type string discussed below. In this case, upon returning, fe->extf should set the fe->size field to be the length of the string or a negative value to indicate a null-terminated string. The dot requests an integer for width, precision, or base. In this case, the fe->size field will indicate how many dots have appeared in the pattern specification. The 'I' requests an integer to define the object size.

f: This is the input/output stream in the calling formatting function. During a call to fe->extf, the stream shall be unlocked so that fe->extf can read from or write to it as appropriate.

v: For both sfscanf() and sfprintf() functions, v points to a location suitable for storing any primitive data types, i.e., scalars or pointers. On return, fe->extf treats v as discussed below.

fe: This is the current formatting environment.

The return value rv of fe->extf directs further processing. There are two cases. When pos$ is present, a negative return value means to ignore fe in further argument processing while a non-negative return value is treated as the case rv == 0 below. When pos$ is not present, fe->extf is called per argument immediately before pattern processing and its return values are treated as below:

rv < 0: The environment stack is immediately popped.

rv == 0:

The extension function has not consumed (in a scanning case) or output (in a printing case) data out of or into the given stream f. The fields fmt, flags, size, width, precis and base of fe shall direct further processing.

For sfprintf() functions, if fe->flags has the bit SFFMT_VALUE, fe->extf should have set *v to the value to be processed; otherwise, a value should be obtained from the argument list. Likewise, for sfscanf() functions, SFFMT_VALUE means that *v should have a suitable address; otherwise, an address to assign value should be obtained from the argument list.

When pos$ is present, if fe->extf changes fe->fmt, this pattern shall be used regardless of the pattern defined in the format string. On the other hand, if fe->fmt is unchanged by fe->extf, the pattern in the format string is used. In any case, the effective pattern should be one of the standardly defined pattern. Otherwise, it shall be treated as unmatched.

rv > 0: The extension function has accessed the stream f to the extent of rv bytes. Processing of the current pattern ceases except that, for scanning functions, if fe->flags does not contain the bit SFFMT_SKIP, the assignment count shall increase by 1.

void va_copy(va_list to, va_list fr)

This macro function portably copies the argument list fr to the argument list to. It should be used to set the field Sffmt_t.args.

long sffmtversion(Sffmt_t* fe, int type)

This macro function initializes the formatting environment fe with a version number if type is non-zero. Otherwise, it returns the current value of the version number of fe. This is useful for applications to find out when the format of the structure Sffmt_t changes. Note that the version number corresponds to the Sfio version number which is defined in the macro value SFIO_VERSION.

int sfprintf(Sfio_t* f, const char* format, ...)
char* sfprints(const char* format, ...)
int sfsprintf(char* s, int n, const char* format, ...)
int sfvsprintf(char* s, int n, const char* format, va_list args)
int sfvprintf(Sfio_t* f, const char* format, va_list args)

These functions format output data. sfprintf() and sfvprintf() write to output stream f. sfsprintf() and sfvsprintf() write to buffer s which is of size n. sfprints() constructs output in some Sfio-defined buffer. sfvprintf() is the underlying primitive for the other functions. Except for sfprints() which returns a null-terminated string or NULL, other functions return the number of output bytes or -1 on failure.

The length of string constructed by sfprints(), sfsprintf(), or sfvsprintf() can be retrieved by sfslen().

The standard patterns are: n, s, c, %, h, i, d, p, u, o, x, X, g, G, e, E, f and !. Except for ! which shall be described below, see the ANSI-C specification of fprintf(3) for details on the other patterns. Let z be some pattern type. A formatting pattern is defined as below:

```
%[pos$][flag][width][.precision][.base][(type)]z
```

pos$: A pattern can specify which argument in the argument list to use. This is done via pos$ where pos is the argument position. Arguments are numbered so that the first argument after format is at position 1. If pos is not specified, the argument following the most recently used one will be used. The pattern %! (see below) cannot be used subsequent to a usage of pos$. Doing so may cause unexpected behaviors.

flag: The flag characters are h, l, L, I, −, +, *space*, 0, ' and #.

Flag I defines the size or type of the object being formatted. There are two cases: (1) I by itself and (2) I followed by either a decimal number or '*'.

In the first case, for integer and floating point patterns, the object type is taken to be the largest appropriate type (i.e., one of Sflong_t, Sfulong_t or Sfdouble_t). For conversion specifiers s and c, the flag is ignored.

In the second case, a given decimal value would define a size while '*' would cause the size to be obtained from the argument list. Then, if the conversion specifier is s, this size defines the length of the string or strings being formatted (see the discussion of base below). For integer and floating point patterns, the size is used to select a type from one of the below lists as indicated by the conversion specifier:

```
Sflong_t, long, int, short
Sfulong_t, unsigned long, unsigned int, unsigned short
Sfdouble_t, double, float
```

The selection algorithm always matches types from left to right in any given list. Although selection is generally based on sizes in bytes, for compatibility with Microsoft-C, the size 64 is matched with an appropriate type with the same number of bits, if any. If the given size does not match any of the listed types, it shall match one of int, unsigned int, and double as defined by the formatting pattern.

Below are a few examples of using the I flag. The first example prints an Sflong_t integer. This example is actually not portable and only works on platforms where sizeof(Sflong_t) is 8. The second example shows how to that portably. The third example specifies printing a string of length 16. This length shall be used regardless of whether or not the given string is shorter or longer than 16. The last example shows the use of the pattern %n to assign the amount of data already output into a short integer n_output.

```
sfprintf(sfstdout,"%I8d", Sflong_obj);
sfprintf(sfstdout,"%I*d", sizeof(Sflong_obj), Sflong_obj);
sfprintf(sfstdout,"%I*s", 16, s);
sfprintf(sfstdout,"%d%I*n", 1001, sizeof(short), &n_output);
```

Flags h, l, and L are the ANSI-C conventions to select the types of input objects. For example, %hd indicates a short int while %ld indicates a long int. The flags ll and L address respectively the largest integer and floating value types, i.e., Sfulong_t, Sflong_t, and Sfdouble_t.

Flag - left-justifies data within the field (otherwise, right-justification).

Flag + means that a signed conversion will always begin with a plus or minus sign.

Flag *space* is ignored if + is specified; otherwise, it means that if the first character of a signed conversion is not a sign or if the result is empty, a space will be prepended.

Flag 0 means padding with zeros on the left.

Flag ' outputs thousands-separator used by the current locale. setlocale(3) should have been used to set the desired locale.

Flag # indicates an alternative format processing. For %o, the first digit is always a zero. For %x and %X, a non-zero result will have a prefix 0x or 0X. For %e, %E, %f, %g, and %G, the result always contains a decimal point. For %g and %G, trailing zeros will not be removed. For %d, %i and %u, the form is *base#number* where *base* is the conversion base and *number* is represented by digits for this *base*. For example, a base 2 conversion with %#..2d for 10 is 2#1010 instead of 1010 as printed with %..2d.

width: This defines the width of the printing field. A value to be printed will be justified and padded if necessary to fill out the field width.

precis: After a first dot appears, an integral value defines a precision. For floating point value patterns, precision is the number of precision digits. For %c, precision defines the number of times to repeat the character being formatted. For %s, precision defines the maximum number of characters to output.

base: After a second dot appears, an integral value defines a conversion base.

For %i, %d, and %u, a conversion base should be in the inclusive range [2,64]. If base is not in this range, it is defined to be 10. The digits to represent numbers are:

```
01234567890
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
@_
```

For %s, when base is defined (i.e., at least two dots appeared in the pattern specification), the input argument is taken as a NULL-terminated list of strings instead of a single string. Each string is formatted based on the usual width and precision rules. If base is non-zero, it defines an ASCII character used to separate the formatted strings. Below is an example showing both the call and the result of printing a NULL-terminated list of three strings apple, orange, and grape:

```
sfprintf(sfstdout,"%..*s",',',list);
apple,orange,grape
```

Likewise, for %c, when base is defined, the input argument is taken as a null-terminated string instead of a single character. Each character is formatted based on the normal width and precision rules. In addition, if base is non-zero, it defines an ASCII character used to separate the formatted characters if there are more than one.

(str): This defines a string str to be passed to the extension function Sffmt_t.extf. Parentheses shall be balanced. If str is *, the string is obtained from the argument list.

int sfscanf(Sfio_t* f, const char* format, ...)
int sfsscanf(const char* s, const char* format, ...)
int sfvsscanf(const char* s, const char* format, va_list args)
int sfvscanf(Sfio_t* f, const char* format, va_list args)

These functions scan data items. sfscanf() scans from the input stream f while sfsscanf() and sfvsscanf() scan from the null-terminated string s. sfvscanf() is the underlying primitive that performs the actual scanning. Item types are determined from patterns in string format. These functions return the number of items successfully scanned or -1 on error.

A white space character (blank, tab, or new-line) in format normally matches a maximal sequence of input white space characters. However, if the input stream is in SF_LINE mode (see sfset()), a new-line character only matches white spaces up to an input new-line character. This is useful to avoid blocking when scanning typed inputs.

The standard scan patterns are: i, d, u, o, x, X, p, n, f, e, E, g, G, c, %, s, [] and !. Except for ! which shall be described below, see the ANSI-C specification of fscanf(3) for details on other patterns. Let z be some pattern type. A formatting pattern is specified as below:

```
%[*][pos$][width][.width.base][(type)][flag]z
```

pos$: A pattern can specify which argument in the argument list to use. This is done via pos$ where pos is the argument position. Arguments are numbered so that the first argument after format is at position 1. If pos is not specified, the argument following the most recently used one will be used. The pattern %! (see below) cannot be used subsequent to a usage of pos$.

*: This discards the corresponding scanned item.

width and base:
width defines the maximum number of bytes to scan and base defines the base of an integral value being scanned. The '.' (dot) notation also allows specifying a '*' (star) to obtain the value from the argument list. The below example specifies scanning 4 bytes to obtain the value of an integer in base 10. At the end of scanning, the variable v should have the value 1234.

```
sfsscanf("12345678","%.*.*d", 4, 10, &v);
```

(str): This defines a string str to be passed to the extension function Sffmt_t.extf. Parentheses shall be balanced. If str is *, the string is obtained from the argument list.

flag: This is #, I, or some sequence of h, l, and L.

Flag # is significant for pattern %i and %[. For %i, it means that the # symbol does not have its usual meaning in an input sequence base#value. For example, the scanning result of %#i on input 2#1001 is 2 and the next sfgetc() call will return #. For %[, if the next character in the input stream does not match the given scan set of characters, # causes a match to a null string instead of a failure.

Flag I defines the size or type of the object being formatted. There are two cases: (1) I by itself and (2) I followed by either a decimal number or '*'.

In the first case, for integer and floating point patterns, the object type is taken to be the largest appropriate type (i.e., one of Sflong_t, Sfulong_t or Sfdouble_t). For string patterns such as %s, the flag is ignored.

In the second case, a given decimal value would define a size while '*' would cause the size to be obtained from the argument list. For string patterns such as %s or %[, this size defines the length of the buffer to store scanned data. Specifying a buffer size only limits the amount of data copied into the buffer. Scanned data beyond the buffer limit will be discarded. For integer and floating point patterns, the size is used to select a type from one of the below lists as indicated by the conversion specifier:

Sflong_t, long, int, short
Sfulong_t, unsigned long, unsigned int, unsigned short
Sfdouble_t, double, float

The selection algorithm always matches types from left to right in any given list. Although selection is generally based on sizes in bytes, for compatibility with Microsoft-C, the size 64 is matched with an appropriate type with the same number of bits, if any. If the given size does not match any of the listed types, it shall match one of int, unsigned int, and double as indicated by the formatting pattern.

Below are examples of using the I flag. The first example scans a 64-bit integer. The second scans some floating point value whose size is explicitly computed and given. The last example scans a string into a buffer with the given size 128. Note that if the scanned string is longer than 127, only the first 127 bytes shall be copied into the buffer. The rest of the scanned data shall be discarded.

```
sfscanf(sfstdin,"%I64d", &int64_obj);
sfscanf(sfstdin,"%I*f", sizeof(float_obj), &float_obj);
sfscanf(sfstdin,"%I*s", 128, buffer);
```

Flags h, l, and L are the ANSI-C conventions for indicating the type of a scanned element. For example, %hd means scanning a short int. The flags ll and L mean respectively scanning an integer or a floating point value with largest size (i.e, Sflong_t or Sfdouble_t).

The %i, %d and %u patterns scan numbers in bases from 2 to 64. %i scans integral values in self-describing formats. Except for octal, decimal and hexadecimal numbers with the usual formats, numbers in general bases are assumed to be of the form: *base#value* where *base* is a number in base 10 and *value* is a number in the given base. For example, 2#1001 is the binary representation of the decimal value 9. If *base* is 36 or less, the digits for *value* can be any combination of [0-9], [a-z], [A-Z] where upper and lower case digits are not distinguishable. If *base* is larger than 36, the set of digits is:

0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
@_

BUFFERING, SYNCHRONIZATION

Void_t* sfsetbuf(Sfio_t* f, Void_t* buf, size_t size)

This function sets the buffering scheme for the stream f. Except for buffer inquiry (see the case size == 0,) f will be synchronized before any buffer modification. If a new buffer is successfully set and the old buffer has not been deallocated, sfsetbuf() returns the address of the old buffer. Otherwise, it returns NULL.

After a sfsetbuf() call, sfvalue() returns the size of the returned buffer.

size == SF_UNBOUND:
: Sfio will pick a suitable buffer size. If buf is NULL, Sfio will also pick a suitable buffering scheme (such as memory mapping.) If buf is not NULL, its actual value is ignored but the buffer will be allocated via malloc(3). This can be used to avoid memory mapping.

size > 0:
: This is the suggested size to use for buffering or memory mapping. If buf is NULL, Sfio will pick a suitable buffering scheme as discussed above. If buf is not NULL, then buf and size determine a buffer of the given size.

size == 0:

If buf is NULL, the stream will be unbuffered. If buf is not NULL, sfsetbuf() simply returns the stream buffer. In this case, no attempt will be made to synchronize the stream.

int sfsync(Sfio_t* f)

This function synchronizes the logical and physical views of stream f. For a SF_WRITE stream, this means to write out any buffered data. For a seekable SF_READ file stream, the physical file position is aligned with the logical stream position and, if SF_SHARE is on, buffered data is discarded. If f is NULL, all streams are synchronized. If f is the base of a stream stack (see sfstack()), all stacked streams are synchronized. Note that a stacked stream can only be synchronized this way. If f is in a pool (see sfpool()) but not being the head, the pool head is synchronized. After a successful synchronization, if f has flag SF_IOCHECK, a SF_SYNC event is raised. sfsync() returns a negative value for failure and 0 for success.

int sfpoll(Sfio_t flist, int n, int timeout)**

This function polls a set of streams to see if I/O operations can be performed on them without blocking. This is useful for multiplexing I/O over a set of streams. If a stream has a discipline, the exception function may be called before and after the stream is polled (see sfdisc() for details). sfpoll() returns the number of ready streams or -1 on failure.

flist and n:

flist is an array of n streams to be polled. Upon return, ready streams are moved to the front of flist in the same relative order.

timeout:

This defines an ellapse time in milliseconds to wait to see if any stream is ready for I/O. If timeout is negative, sfpoll() will block until some stream become ready. Note that SF_STRING and normal file streams never block and are always ready for I/O. If a stream with discipline is being polled and its readiness is as yet undetermined (e.g., empty buffer,) the discipline exception function will be called with SF_DPOLL before normal actions are taken.

Sfio_t* sfpool(Sfio_t* f, Sfio_t* poolf, int mode)

This function manipulates pools of streams. In a pool, only one stream is at the head and can have buffered data. All other streams in the pool will be synchronized. A stream becomes head when it is used for some I/O operation. sfpool() returns NULL on failure.

f and poolf:

If f is NULL, sfpool() simply returns the head of the pool containing poolf. If f is not NULL and poolf is NULL, f is deleted from its pool. In this case, if no other stream from the same pool can become head, sfpool() will return NULL; otherwise, it returns some stream from the remainder of the pool. If both f and poolf are not NULL, f is moved from its current pool (if any) into the same pool with poolf. In this case, poolf is returned.

mode: If poolf is already in a pool, mode is ignored. Otherwise, mode should be 0 or SF_SHARE. A SF_SHARE pool contains streams with SF_WRITE mode. In addition, on change to a new head stream, buffered write data of the current head is transferred to the new head.

int sfpurge(Sfio_t* f)

This function discards all buffered data unless f is a SF_STRING stream. Note that if f is a SF_READ stream based on an unseekable device, purged data will not be recoverable. If f is a sfpopen-stream opened for both read and write, data of both the read and write pipe ends will be purged (see sfset() to selectively turn off read or write mode if one set of data is to be preserved.) After purging, if f has flag SF_IOCHECK, the event SF_PURGE is raised. sfpurge() returns -1 for failure and 0 for success.

DISCIPLINE, EVENT-HANDLING

A file stream uses the system calls read(2), write(2) and lseek(2) to read, write and position in the underlying file. Disciplines enable application-defined I/O methods including exception handling and data pre/post-processing.

Sfdisc_t* sfdisc(Sfio_t* f, Sfdisc_t* disc)

Each stream has a discipline stack whose bottom is a virtual discipline representing the actual system calls. sfdisc() manipulates the discipline stack of stream f. f will be synchronized before any discipline stack manipulation. After a successful discipline stack manipulation, the stream I/O position (see sfseek() and sftell()) and extent (see sfsize()) are updated to reflect that defined by the top discipline. If disc is SF_POPDISC or (Sfdisc_t*)0, the top element of the stack, if any, is popped and its address is returned. Otherwise, disc is pushed onto the discipline stack. In this case, if successful, sfdisc() returns the discipline that was pushed down. sfdisc() returns NULL on failure.

Note that a discipline can be used only on one stream at a time. An application should take care to allocate different discipline structures for use with different streams. A discipline structure is of the type Sfdisc_t which contains the following public fields:

```
Sfread_f   readf;
Sfwrite_f  writef;
Sfseek_f   seekf;
Sfexcept_f exceptf;
```

The first three fields of Sfdisc_t specify alternative I/O functions. If any of them is NULL, it is inherited from a discipline pushed earlier on the stack. Note that a file stream always has read(2), write(2), lseek(2) and NIL(Sfexcept_f) as the *logical bottom discipline*. Arguments to I/O discipline functions have the same meaning as that of the functions sfrd(), sfwr() and sfsk() described below.

The exception function, (*exceptf)() announces exceptional events during I/O operations. It is called as (*exceptf)(Sfio_t* f, int type, Void_t* value, Sfdisc_t* disc). Unless noted otherwise, the return value of (*exceptf)() is used as follows:

<0: The on-going operation shall terminate.

>0: If the event was raised due to an I/O error, the error has been repaired and the on-going operation shall continue normally.

=0: The on-going operation performs default actions with respect to the raised event. For example, on a reading error or reaching end of file, the top stream of a stack will be popped and closed and the on-going operation continue with the new top stream.

The argument type of (*exceptf)() identifies the particular exceptional event:

SF_LOCKED:
: The stream is in a locked state.

SF_READ, SF_WRITE:
: These events are raised around reading and writing operations.

  If SF_IOCHECK is on, SF_READ and SF_WRITE are raised immediately before read(2) and write(2) calls. In this case, *((ssize_t*)value) is the amount of data to be processed. The return value of (*exceptf)(), if negative, indicates that the stream is not ready for I/O and the calling operation will abort with failure. If it is positive, the stream is ready for I/O but the amount should be restricted to the amount specified by this value. If the return value is zero, the I/O operation is carried out normally.

  SF_READ and SF_WRITE are also raised on operation failures. In such a case, *((ssize_t*)value) is the return value from the failed operation.

SF_SEEK:
This event is raised when a seek operation fails.

SF_NEW, SF_CLOSE, SF_FINAL:
These events are raised during a stream closing. SF_NEW is raised for a stream about to be closed to be renewed (see sfnew()). SF_CLOSE is raised for a stream about to be closed. SF_FINAL is raised after a stream has been closed and before its space is to be destroyed (see sfclose()). For these events, a non-zero return value from (*exceptf)() causes sfclose() to return immediately with the same value.

SF_DPUSH, SF_DPOP, SF_DBUFFER:
Events SF_DPUSH and SF_DPOP are raised when a discipline is about to be pushed or popped. (Sfdisc_t*)value is the to-be top discipline, if any.

A stream buffer is always synchronized before pushing or popping a discipline. If this synchronization fails, SF_DBUFFER will be raised with *((size_t*)value) being the amount of buffered data. If the return value of exceptf is positive, the push or pop operation will continue normally; otherwise, sfdisc() returns failure.

SF_DPOLL:
This event is raised by sfpoll() to see if the stream is ready for I/O. *((int*)value) indicates a timeout interval to wait. A negative return value from the exception function means blocking. A positive return value means non-blocking. A zero return value means that sfpoll() should query the stream file descriptor using default methods.

SF_READY:
This event is raised by sfpoll() for each ready stream after they are determined. A negative return value from the exeption handler causes sfpoll() to return immediately with the same return value. A positive return value causes sfpoll() to retry polling the whole set of streams.

SF_SYNC, SF_PURGE:
If SF_IOCHECK is set, these events are raised immediately after sfsync() or sfpurge() successfully complete their operations and before they return. Note that sfsync() is implied when a SF_WRITE or SF_SHARE|SF_READ stream is closed. Note also that SF_SYNC is not raised for a stream synchronized during a call sfsync((Sfio_t*)0).

SF_ATEXIT:
This event is raised for each open stream before the process exits.

int sfraise(Sfio_t* f, int type, Void_t* data)

This function calls all exception handlers of stream f with the event type and associated data. If an exception handler returns a non-zero value, sfraise() immediate returns the same value. Application-defined events should start from the value SF_EVENT so as to avoid confusion with system-defined events, sfraise() returns 0 on success and -1 on failure.

ssize_t sfrd(Sfio_t* f, Void_t* buf, size_t n, Sfdisc_t* disc)
ssize_t sfwr(Sfio_t* f, const Void_t* buf, size_t n, Sfdisc_t* disc)
Sfoff_t sfsk(Sfio_t* f, Sfoff_t offset, int type, Sfdisc_t* disc)

These functions provides safe methods for a discipline I/O function to invoke earlier discipline I/O functions and to properly handle exceptions. They should not be used in any other context. sfrd() and sfwr() return the number of bytes read or written. sfsk() returns the new seek position. On error, all three functions return a negative value which should be -1 or the value returned by the exception handler.

STREAM CONTROL

int sfset(Sfio_t* f, int flags, int set)

This function sets control flags for the stream f. It returns the previous set of flags or 0 on error.

Settable flags are: SF_READ, SF_WRITE, SF_IOCHECK, SF_LINE, SF_SHARE, SF_PUBLIC, SF_MALLOC, SF_STATIC and SF_BUFCONST. Note that SF_READ and SF_WRITE can be turned on or off only if the stream was opened as SF_READ|SF_WRITE. Turning off one of them means that the stream is to be treated exclusively in the other mode. It is not possible to turn off both. If legal, an attempt to turn on either SF_READ or SF_WRITE will cause the stream to be in the given I/O mode.

set == 0:
: If flags is zero, the current set of flags is simply returned. Note that when a stream is first opened, not all of its flags are initialized yet (more below). If flags is non-zero, an attempt is made to turn off the specified flags.

set != 0:
: If flags is zero, the stream is initialized if not yet done so. Then the current set of flags is returned. If flags is non-zero, an attempt is made to turn on the specified flags.

int sfsetfd(Sfio_t* f, int fd)

This function changes the file descriptor of f. Before a change is realized, (*notify)(f,SF_SETFD,newfd) (see sfnotify()) is called. sfsetfd() returns -1 on failure and the new file descriptor on success.

fd >= 0:
: If the current file descriptor is non-negative, it will be changed using dup(3) to a value larger or equal to fd. Upon a successful change, the previous file descriptor will be closed. If the current file descriptor is negative, it will be set to fd and the stream will be reinitialized.

fd < 0:
: The stream is synchronized (see sfsync()) and its file descriptor will be set to this value. Then, except for sfclose(), the stream will be inaccessible until a future sfsetfd() call resets the file descriptor to a non-negative value. Thus, sfsetfd(f,-1) can be used to avoid closing the file descriptor of f when f is closed.

Sfio_t* sfstack(Sfio_t* base, Sfio_t* top)

This function stacks or unstacks stream. Every stream stack is identified by a base stream via which all I/O operations are performed. However, an I/O operation always takes effect on the top stream. If the top stream reaches the end of file or has an unrecoverable error condition, it is automatically popped and closed (see also sfdisc() for alternative handling of these conditions).

base:
: This is the base stream of the stack. If it is NULL, sfstack() does nothing and returns top.

top:
: If this is SF_POPSTACK or (Sfio_t*)0, the stack is popped and sfstack() returns the popped stream. Otherwise, top is pushed on top of the stack identified by base and sfstack() returns the base stream.

Sfio_t* sfswap(Sfio_t* f1, Sfio_t* f2)

This function swaps contents of f1 and f2. This fails if either stream is in a stream stack but not being a base stream. If f2 is NULL, a new stream is constructed as a duplicate of f1. sfswap() returns f2 or f1 duplicate on success and NULL on failure.

STREAM INFORMATION

Sfoff_t sfsize(Sfio_t* f)

This function returns the size of stream f (see sfnew()). If f is not seekable or if its size is not determinable, sfsize() returns -1.

Sfoff_t sftell(Sfio_t* f)

This function returns the current I/O position in stream f. Note that if f is SF_APPEND and a writing operation was just performed, the current I/O position is at the physical end of file. If f is unseekable, sftell returns the number of bytes read from or written to f. See also sfungetc().

ssize_t sfvalue(Sfio_t* f)

This function returns the string or buffer length for sfreserve(), sfsetbuf(), and sfgetr().

int sffileno(Sfio_t* f)

This function returns the file descriptor of stream f.

int sfstacked(Sfio_t* f)

This function returns a non-zero value if stream f has been stacked.

int sfeof(Sfio_t* f)
int sferror(Sfio_t* f)
int sfclrerr(Sfio_t* f)

sfeof() tells whether or not the stream has an end-of-file condition. sferror() tells whether or not the stream has an error condition. sfclrerr() clears both end-of-file and error conditions. The end-of-file and error conditions are also cleared on an I/O operation.

int sfclrlock(Sfio_t* f)

This function restores the stream back to a normal state. This means clearing locks and possibly throwing away unprocessed data. As such, this operation is unsafe and should be used with care. For example, it may be used before a long jump (longjmp(3)) out of some discipline I/O function to restore the internal stream states. sfclrlock() returns the current set of flags.

int sfnotify((void(*)notify)(Sfio_t*, int, int))

This sets a function (*notify)() to be called as (*notify)(f,type,file) on various stream events. Arguments f and file are stream and related file descriptor. Argument type indicates the reason for the call:

SF_NEW:
: f is being opened and file is the underlying file descriptor.

SF_CLOSE:
: f and file are being closed.

SF_SETFD:
: The file descriptor of f is being changed to file (see sfsetfd().)

SF_READ:
: An attempt to change f to read mode failed.

SF_WRITE:
: An attempt to change f to write mode failed.

MISCELLANEOUS FUNCTIONS

ssize_t sfslen()

This function returns the length of a string just constructed by sfsprintf() or sfprints(). See also sfvalue().

int sfulen(Sfulong_t v)

int sfllen(Sflong_t v)
int sfdlen(Sfdouble_t v)

These functions return respectively the number of bytes required to code the Sfulong_t, Sflong_t or Sfdouble_t value v by sfputu(), sfputl() or sfputd().

ssize_t sfpkrd(int fd, char* buf, size_t n, int rsc, long tm, int action)

This function acts directly on the file descriptor fd. It does a combination of peeking on incoming data and a time-out read. Upon success, it returns the number of bytes received. A return value of 0 means that the end-of-file condition has been detected. A negative value represents an error.

buf, n: These define a buffer and its size to read data into.

rsc: If >=0, this defines a record separator. See action for detail.

tm: If >=0, this defines a time interval in milliseconds to wait for incoming data.

action: When rsc >= 0, the absolute value of action, *r*, determines the number of records to be read. If action > 0, sfpkrd() will peek on incoming data but will not read past it. Therefore, a future sfpkrd() or read(2) will see the same data again. If action == 0, sfpkrd() will not peek. If action < 0, there are two cases: if rsc < 0, sfpkrd() reads n bytes; otherwise, exactly *r* records will be read. Note that, in the last case, reading records from an unseekable device may be slow if the underlying platform does not allow peeking on such a device.

FULL STRUCTURE SFIO_T

#include <sfio_t.h>

Most applications based on Sfio only need to include the header file sfio.h which defines an abbreviated Sfio_t structure without certain fields private to Sfio. However, there are times (e.g., debugging) when an application may require more details about the full Sfio_t structure. In such cases, the header file sfio_t.h can be used in place of sfio.h. Note that an application doing this will become sensitive to changes in the internal architecture of Sfio.

#define SFNEW(buf,size,file,flags,disc)

This macro function is defined in sfio_t.h for use in static initialization of an Sfio_t structure. It requires five arguments:

buf, size:
These define a buffer and its size.

file: This defines the underlying file descriptor if any.

flags: This is composed from bit flags described above.

disc: This defines a discipline if any.

EXAMPLE DISCIPLINES

The below functions create disciplines and insert them into the given streams f. These functions return 0 on success and -1 on failure.

int sfdcdio(Sfio_t* f, size_t bufsize)

This creates a discipline that uses the direct IO feature available on file systems such as SGI's XFS to speed up IO. The argument bufsize suggests a buffer size to use for data transfer.

int sfdcdos(Sfio_t* f)

This creates a discipline to read DOS text files. It basically transforms pairs of \r\n to \n.

int sfdcfilter(Sfio_t* f, const char* cmd)

This creates a discipline that sends data from f to the given command cmd to process, then reads back the processed data.

int sfdclzw(Sfio_t* f)

This creates a discipline that would decompress data in f. The stream f should have data from a source compressed by the Unix **compress** program.

int sfdcseekable(Sfio_t* f)

This creates a discipline that makes an unseekable reading stream seekable.

int sfdcslow(Sfio_t* f)

This creates a discipline that makes all Sfio operations return immediately on interrupts. This is useful for dealing with slow devices.

int sfdcsubstream(Sfio_t* f, Sfio_t* parent, Sfoff_t offset, Sfoff_t extent)

This creates a discipline that makes f acts as if it corresponds exactly to the subsection of parent starting at offset with size extent.

int sfdctee(Sfio_t* f, Sfio_t* tee)

This creates a discipline that copies to the stream tee any data written to f.

int sfdcunion(Sfio_t* f, Sfio_t array, int n)**

This creates a discipline that makes f act as if it is the concatenation of the n streams given in array.

STDIO-COMPATIBILITY

Sfio provides two compatibility packages to Stdio-based applications, a source level interface and a binary level library. These packages provide a union of functions in popular Stdio implementations.

The source Stdio-compatibility interface provides the header file stdio.h that defines a set of macros or inlined functions to map Stdio calls to Sfio ones. This mapping may benignly extend or change the meaning of certain original Stdio operations. For example, the Sfio's version of popen() allows a coprocess to be opened for both reading and writing unlike the original call which only allows a coprocess to be opened for a single mode. Similarly, the Sfio's fopen() call can be used to create string streams in addition to file streams.

The binary Stdio-compatibility library, libstdio.a, provides a complete implementation of Stdio functions suitable for linking applications already compiled with native header stdio.h. Functions in this implementation are also slightly altered or extended as discussed above.

Below are the supported Stdio functions:

```
FILE*  fopen(const char* file, const char* mode);
FILE*  freopen(const char* file, const char* mode, FILE* stream);
FILE*  fdopen(int filedesc, const char* mode);
FILE*  popen(const char* command, const char* mode);
FILE*  tmpfile();
int    fclose(FILE* stream);
int    pclose(FILE* stream);

void   setbuf(FILE* stream, char* buf);
int    setvbuf(FILE* stream, char* buf, int mode, size_t size);
void   setbuffer(FILE* stream, char* buf, size_t size);
int    setlinebuf(FILE* stream);
```

```
int    fflush(FILE* stream);
int    fpurge(FILE* stream);

int    fseek(FILE* stream, long offset, int whence);
void   rewind(FILE* stream);
int    fgetpos(FILE* stream, fpos_t* pos);
int    fsetpos(FILE* stream, fpos_t* pos);
long   ftell(FILE* stream);

int    getc(FILE* stream);
int    fgetc(FILE* stream);
int    getchar(void);
int    ungetc(int c, FILE* stream);
int    getw(FILE* stream);
char*  gets(char* s);
char*  fgets(char* s, int n, FILE* stream);
size_t fread(Void_t* ptr, size_t size, size_t nelt, FILE* stream);

int    putc(int c, FILE* stream);
int    fputc(int c, FILE* stream);
int    putchar(int c);
int    putw(int w, FILE* stream);
int    puts(const char* s, FILE* stream);
int    fputs(const char* s, FILE* stream);
size_t fwrite(const Void_t* ptr, size_t size, size_t nelt, FILE* stream);

int    fscanf(FILE* stream, const char* format, ...);
int    vfscanf(FILE* stream, const char* format, va_list args);
int    _doscan(FILE* stream, const char* format, va_list args);
int    scanf(const char* format, ...);
int    vscanf(const char* format, va_list args);
int    sscanf(const char* s, const char* format, ...);
int    vsscanf(const char* s, const char* format, va_list args);

int    fprintf(FILE* stream, const char* format, ...);
int    vfprintf(FILE* stream, const char* format, va_list args);
int    _doprnt(FILE* stream, const char* format, va_list args);
int    printf(const char* format, ...);
int    vprintf(const char* format, va_list args);
int    sprintf(const char* s, const char* format, ...);
int    snprintf(const char* s, int n, const char* format, ...);
int    vsprintf(const char* s, const char* format, va_list args);
int    vsnprintf(const char* s, int n, const char* format, va_list args);

int    feof(FILE* stream);
int    ferror(FILE* stream);
int    clearerr(FILE* stream);
```

RECENT CHANGES

A few exception types have been added. In particular, exception handlers shall be raised with SF_LOCKED on accessing a locked stream. Before a process exits, the event SF_ATEXIT is raised for each open stream.

A number of disciplines have been added for various processing functions. Of interests are disciplines to use the direct I/O feature on IRIX6.2, read DOS text files, and decompress files compressed by Unix *compress*.

Various new stream and function flags have been added. For example, the third argument of sfgetr() is now a set of bit flags and not just a three-value object. However, the old semantics of this argument of sfgetr() is still supported.

The sfopen() call has been extended so that sfopen(f,NULL,mode) can be used to changed the mode of a file stream before any I/O operations. This is most useful for changing the modes of the standard streams.

The buffering strategy has been significantly enhanced for streams that perform many seek operations. Also, the handling of stream and file positions have been better clarified so that applications that share file descriptors across streams and/or processes can be sure that the file states will be consistent.

AUTHORS

Kiem-Phong Vo, kpv@research.att.com,
David G. Korn, dgk@research.att.com, and
Glenn S. Fowler, gsf@research.att.com.