

Revision History

v1.0	December 2008 - Initial, read only release of the SRA toolkit library
v2.0	March 17, 2009 - Write support added to the SRA Toolkit. Tables, rows and column content now be added using the APIs.
v2.01	March 25, 2009 - Minor updates to sample program and removal of '-lkapp' in link instructions.
v2.02	April 29, 2009 - Added read/write schema and updated list of required libraries

Author

Ty Roach (contractor) roachtg@ncbi.nlm.nih.gov

Contents

1. [Revision History](#)
2. [Author](#)
3. [Contents](#)
4. [System Requirements](#)
5. [Overview](#)
6. [Building the ToolKit](#)
7. [Toolkit Content](#)
8. [Data Structures](#)
9. [Using the ToolKit](#)
10. [APIs](#)

System Requirements

Operating System: Linux (tested on SUSE Enterprise Edition 9 SP 3)

Architecture: x86 (32 or 64 bit)

Software: make (version 3.80 or later)
gcc (version 4.1.2 for 64 bit, version 3.4.2 for 32 bit)
icc (version 9.0 or 9.1)

Libraries: libz (version 1)
libbz2 (version 1)
libxml2 (tested with version 2.6.7)

Overview

This document describes the National Center for Biotechnology Information's (NCBI) Short Read Archive (SRA) toolkit library. For more information, please visit the [SRA website](#).

The toolkit library provides the mechanism for inserting and search for information in the SRA database. The current version of the toolkit assumes the archive is on local platform (versus on a remote platform).

SRA Background

Relational databases are good for recording and manipulating related data, indexing, making arbitrarily complex joins, processing complex queries but they are expensive to use for tera and peta-byte long term storage. They are inflexible, tend to be slightly wasteful, purposely avoid using the file system, and encapsulate all data behind their servers. In summary, they are bulky, require significant management, inflexible and costly both in terms of license and use of storage.

Simple file-based repositories are lightweight, make use of the file system that we already have, and are stored according to some object model (i.e. a run is in a single file that can be accessed, shipped, modified, removed, etc.). This approach suffers from a lack of support for indexed queries, relations where necessary. Archiving often means tar and compression often means gzip or bzip2, making the data difficult (or impossible) to access in a repository setting.

We needed something that has much of the power of a relational database while being lightweight, transportable and flexible like a flat-file storage.

The SRA uses a column based approach for managing its data vs. the traditional relational, row-based, databases.

Row vs. Column Based Databases

Row-based databases store data as a series of row structures, where each structure contains one or more fields. Column based databases turn the structure on its side, so to speak, and store data as a series of columns where each field is stored in its own column. Rows are then assembled from multiple columns.

Row-based databases take advantage of having all fields of data in a single structure for efficiency of retrieving the entire row with a single read. Column-based databases take advantage of a single data type per series to achieve better storage utilization (packing and/or compression) and are good at delivering many result set rows of only some of the row's fields (that is, they only retrieve what was requested).

Row-based databases have to work hard to add or remove a row column, and must rewrite entire tables to do so. Column based databases (if properly designed) can leave existing/remaining table data intact when adding/removing a column.

As mentioned previously, the SRA uses a column based approach for managing its data.

The column-based approach allows the flexibility to modify the schema as short read technology matures. Also, the amount of data being stored enormous, requiring effective yet efficient compression. The SRA takes advantage of the ability design both the schema and compressors to achieve custom results.

The needs of a massive repository like the SRA also impact the way data is stored, shared, replicated, and backed up. The SRA column-based design makes use of the file system to keep columns physically separate. This makes it possible for to store the most frequently accessed series (the "fastq" data) in fast, near storage, while signal data can be located in slower, bulk storage. Columns can also be separated into read-only and modifiable storage.

Finally, the column approach allows high efficiency for serving data, using a design that both implements and relies upon the read-ahead caching schemes of modern operating systems.

Building the ToolKit

These are the steps for building the SRA toolkit library:

1. Unpack the tar-ball
2. Change to the root directory after unpacking the toolkit tar-ball
3. "make OUTDIR=\$SRA_HOME out." to build the target.
4. "make" to build the toolkit.

This will result in the following structure being created (note - in this example, the platform is a 64 bit machine):

```
$SRA_HOME/bin64  
$SRA_HOME/lib64
```

When compiling and linking with the SRA toolkit, you will need to set the library and include search paths appropriately. As a reference, using the above example, the include paths would be set as follows:

```
-I<toolkit-root-install-dir>/inc/gcc  
-I<toolkit-root-install-dir>/itf
```

Furthermore, be sure to set the LD_LIBRARY_PATH environment variable to point to the location of the toolkit shared libraries. For example:

```
export LD_LIBRARY_PATH=$SRA_HOME/lib64
```

SDK Content Description

The SRA SDK is organized into binary (bin64 or bin32) and library (lib64 or lib32, ilib64 or ilib3) sub-directories. The binary directory contains a number of tools and scripts, some of which are used for processing of data submitted to NCBI, and others are simply useful tools for analyzing or testing data. The libraries provide access to SRA SDK APIs in the event that custom applications are needed.

Format	Loading Tool (to SRA Format)		Dumping Tool (from SRA Format)	
	Tool Name	Availability	Tool Name	Availability
FASTQ	fastq-loader	April 2009	fastq-dump	December 2008
454(SFF)	sff-loader	June 2009	sff-dump	June 2009
Illumina (Native)	illumina-loader	June 2009	illumina-dump	May 2009
Illumina (SRF)	srf-loader	April 2009	N/A	N/A
AB SOLiD (Native)	abi-loader	April 2009	abi-dump	April 2009
AB SOLiD (SRF)	srf-loader	April 2009	N/A	N/A

Data Structures

SRAError

All messages that can fail are designed to return a status code. They are typed as int to conform to traditional return register convention. Messages that cannot fail will generally have a return type of void or may return a value.

The status/error codes returned are defined as:

sraNoErr	no error
sraUnknownErr	an unanticipated error occurred
sraUnsupported	
sraInvalid	an invalid parameter or state was encountered
sraPermErr	the operation failed due to lack of permission
sraBusyErr	the operation failed
sraBadPath	unable to form a proper path
sraNotFound	a requested or needed object was not found
sraExists	
sraConflict	a needed object is in conflicting state

sraBadRange	an invalid id range was specified
sraMemErr	a memory allocation failed
sraIOErr	an I/O error occurred
sraEndMedia	the target volume is full
sraIncompleteErr	an I/O operation did not complete
sraTooBigErr	
sraCorruptErr	a corrupt object was detected
sraBadArchErr	the current architecture has unsupported byte order
sraBadVersErr	attempt to operate on an unsupported db version
sraFileLimErr	have reached the limit on the number of open files
sraBadKey	bad submission, run or spot key
sraBuffErr	insufficient buffer space was provided
sraBadRange	
sraTimeoutErr	

To generate an English translation for the SRRError value, call the SRRAErrToEnglish() API.

SRAPlatforms

Categorizes platforms into types as follows:

Enumeration	Value
SRA_PLATFORM_UNDEFINED	0
SRA_PLATFORM_454	1
SRA_PLATFORM_ILLUMINA	2
SRA_PLATFORM_ABSOLID	3

SRAReadTypes

Enumeration	Value
SRA_READ_TYPE_TECHNICAL	1
SRA_READ_TYPE_BIOLOGICAL	2

SRA Read Schema

A column is configured as a sequence of blobs, and each blob is a sequence of records, indexed by spot id.

Column	Label	Description
PLATFORM	NCBI:SRA:platform_id	U8 size, SRAPatforms enum value
NAME	ascii	NUL-terminated spot name
X	U16	X coordinate for spot
Y	U16	Y coordinate for spot
SPOT_DESC	NCBI:SRA:SpotDesc	synthesized SRASpotDesc structure
SPOT_LEN	U16	spot length for this row
SIGNAL_LEN	U16	signal length for this row
NREADS	U8	number of reads in this row
CLIP_QUALITY_RIGHT	U16	right-side quality clip for this row
READ_DESC	NCBI:SRA:ReadDesc	synthesized array of SRAReadDesc [nreads]
READ_SEG	NCBI:SRA:Segment	array of SRASegment [nreads] - preferred over READ_LEN
READ_LEN	U16	array of lengths U16 [nreads]
READ_TYPE	NCBI:SRA:read_type	array of U8 [nreads], SRAReadTypes values
READ_FILTER	NCBI:SRA:read_filter	array of U8 [nreads], SRAReadFilter values
CS_KEY	INSDC:fasta	array of char [nreads]
READ <i>default</i>	INSDC:fasta	DNA sequence in IUPAC, char [spot_len]
READ	NCBI:2na	DNA sequence in NCBI 2na format - no 'N' value
READ	NCBI:4na	DNA sequence in NCBI 4na format
CSREAD <i>default</i>	INSDC:csfasta	Color space sequence, char [spot_len]
CSREAD	NCBI:2cs	Color space sequence in 2cs format - no '!' value
QUALITY <i>default</i>	NCBI:qual1	Phred-like quality scores, U8 [spot_len]
QUALITY	NCBI:qual4	Illumina 4-channel qualities I8[4][spot_len]
SIGNAL	NCBI:isamp1	454 signal I16 [sig_len]
SIGNAL	NCBI:isamp4	Illumina/AB signal F32[4][sig_len]
INTENSITY	NCBI:isamp4	Illumina intensities F32[4][sig_len]
NOISE	NCBI:isamp4	Illumina noise F32[4][sig_len]
POSITION	U16	454 base to signal position index

		U16[spot_len]
--	--	---------------

SRA Write Schema

Column	Label	Description
PLATFORM	NCBI:SRA:platform_id	U8 size, SRAPatforms enum value
NAME_FMT	ascii	U8 size, SRAPatforms enum value
NREADS	U8	U8 number of reads per spot
READ_SEG	NCBI:SRA:Segment	SRASegment [nreads] describing read layout
READ_TYPE	NCBI:SRA:read_type	U8 [nreads] with values SRASegmentTypes
READ_FILTER	NCBI:SRA:read_filter	U8 [nreads] with SRASegmentFilter values
LABEL_SEG	NCBI:SRA:Segment	SRASegment [nreads] describing label text
LABEL	ascii	non-delimited string of label characters
CS_KEY	INSDC:fasta	char [nreads] with CS keys
READ	INSDC:fasta	DNA read sequence in IUPAC chars
CSREAD	INSDC:csfasta	Color space read sequence
QUALITY	NCBI:qual1	Phred-like quality scores
CLIP_QUALITY_RIGHT	U16	454 base to signal position index
FTC	NCBI:fsamp1	individual color space signal channels
FAM	NCBI:fsamp1	
CY3	NCBI:fsamp1	
TXR	NCBI:fsamp1	
CY5	NCBI:fsamp1	

Using the ToolKit

Once the toolkit has been installed and configured, you may build applications using the APIs in the toolkit libraries by supplying the following compiler and linker flags:

```

-I<toolkit-root-install-dir>/inc/gcc
-I<toolkit-root-install-dir>/itf
-L$SRA_HOME/lib64
[READ] -lsradb -lvdb -lklib -lkascii -lm -lz -lbz2 -lpthread
or
[WRITE] -lwsradb -lwvdb -lklib -lkascii -lm -lz -lbz2 -lpthread

```

where <sra-toolkit-root> is the full path to the root directory where the installation/configuration process deposited the toolkit.

Using the example from above, it would be \$SRA_HOME/lib64.

NOTE - the "-lsradb" and "-lwsradb" libraries are mutually exclusive, that is, they should *not* be used at the same time. When building applications that read from the archive, you should add "#include <sra/sradb.h>" to your source code and add the "-lsradb" link option.

When building applications that write to the archive, you should add "#include <sra/wsradb.h>" to your source code and add the "-lwsradb" link option.

Be sure to set the LD_LIBRARY_PATH environment variable to point to the location of the toolkit shared libraries. For example:

```
export LD_LIBRARY_PATH=$SRA_HOME/lib64
```

See the example below for a reference application that reads data from an archive using the SRA toolkit.

Read Example

```
#include <stdio.h>
#include <stdlib.h>
#include <klib/defs.h>
#include <vdb/types.h>
#include <klib/rc.h>
#include <sra/sradb.h>

int main ( int argc, char *argv [] ) {
    rc_t          rc;
    SRAMgr const  *sra;
    SRATable const *tbl;
    spotid_t      max;
    const char    *table_to_read = "myTable";
    SRAColumn const *read;
    SRAColumn const *qual;
    const void    *col_data;
    bitsz_t       off, sz;
    spotid_t      id;

    if (argc == 2)
        table_to_read = argv[1];

    printf("initializing manager\n");

    rc = SRAMgrMakeRead(&sra);

    if (rc != 0) {
        fprintf(stderr, "failed initializing sra mgr
(%s)\n", SRAErrToEnglish(SRAErrMake(rc), NULL));
```



```

        return SRAErrMake(rc);
    }

    printf("opening table\n");

    rc = SRAMgrOpenTableRead (sra, &tbl, table_to_read);

    if (rc != 0) {
        fprintf(stderr, "failed opening table
(%s)\n", SRAErrToEnglish(SRAErrMake(rc), NULL));
        SRAMgrRelease(sra);

        return SRAErrMake(rc);
    }

    printf("getting max spot id\n");

    rc = SRATableMaxSpotId(tbl, &max);

    if (rc != 0) {
        fprintf(stderr, "failed getting max spot id\n");
        SRATableRelease(tbl);
        SRAMgrRelease(sra);

        return SRAErrMake(rc);
    }

    printf("opening READ column\n");

    rc = SRATableOpenColumnRead(tbl, &read, "READ", insdc_fasta_t);

    if (rc != 0) {
        fprintf(stderr, "failed opening READ column
(%s)", SRAErrToEnglish(SRAErrMake(rc), NULL));
        SRATableRelease(tbl);
        SRAMgrRelease(sra);

        return SRAErrMake(rc);
    }

    printf("opening QUALITY column\n");

    rc = SRATableOpenColumnRead(tbl, &qual, "QUALITY", ncbi_qual1_t);

    if (rc != 0) {
        fprintf(stderr, "failed opening QUALITY column
(%s)", SRAErrToEnglish(SRAErrMake(rc), NULL));
        SRAColumnRelease(read);
        SRATableRelease(tbl);
        SRAMgrRelease(sra);

        return SRAErrMake(rc);
    }

    for (id = 1; id <= max; ++id) {

```

```

        printf("reading READ column\n");
        rc = SRAColumnRead(read, id, &col_data, &off, &sz);

        if (rc) {
            fprintf(stderr,"failed reading READ column
(%s)", SRAErrToEnglish(SRAErrMake(rc),NULL));
            continue;
        }

        printf("off: %lu, sz: %lu, bases: %lu\n", off, sz, sz / 8);

        printf("reading QUALITY column\n");

        rc = SRAColumnRead(qual, id, &col_data, &off, &sz);

        if (rc) {
            fprintf(stderr,"failed reading QUALITY column
(%s)", SRAErrToEnglish(SRAErrMake(rc),NULL));
            continue;
        }

        printf("off: %lu, sz: %lu, bases: %lu\n", off, sz, sz / 8 );
    } /* end for-loop on spot-id's */

    /*
     * NOTE - It is VERY important to release the toolkit objects once
     their use is no longer required.
     */
    SRAColumnRelease(qual);
    SRAColumnRelease(read);
    SRATableRelease(tbl);
    SRAMgrRelease(sra);

    return SRAErrMake(rc);
}

```

To build this program:

```
gcc -o SimpleRead -I $$SRA_INSTALL_DIR/inc/gcc -I $$SRA_INSTALL_DIR/itf -L
$$SRA_HOME/lib64 SimpleRead.c -lsradb -lvdb -lklib -lkascii -lm -lz -lbz2 -lpthread
```

where `$$SRA_INSTALL_DIR` = <toolkit-root-install-dir>

NOTE - Assumes that `LD_LIBRARY_PATH` has been set as described above and that the above code is in a file called "SimpleRead.c".

APIs

The `sradb` interface uses an object-oriented paradigm with opaque pointers to C structures representing objects, and C functions defining messages upon those objects. All messages return an architecture native integer status code with a value defined the `SRAError` enumeration.

Object ownership is strictly defined, with all externalized objects belonging to the application, and being destroyed by the appropriate destructor message. Requests for destruction are not guaranteed to be honored: Specifically, some requests may be rejected if an object (reference) is being used internally by another object. There is therefore an order to object-ref destruction (releasing) that may be enforced.

All objects are reference counted. To use an object, you may pass in a loaned reference in your messages. This is to say, if you send a message to some object-ref that takes another object-ref as a parameter, you don't have to pass in a new reference to that object, because it can borrow yours.

If a method chooses to hold on to an object-ref it received as a parameter, i.e. a loaned reference, it can send an `*AddRef` message (eg one of `SRANamelistAddRef()`, `SRAMgrAddRef()`, `SRAMgrAddRef()`) to that object-ref in order to attach its own reference to it. In that way, it becomes co-owner.

Every owner of a reference is required to release their reference when the object is no longer needed. The object is actually garbage collected when the last reference is released, which is why one would want to check the status code: an object can refuse to be collected.

Strings are universally represented in UTF-8 UNICODE by a data pointer and size, keeping in mind that with UTF-8, size and length are not synonymous/interchangeable. Strings are required to be zero-terminated.

Paths are represented as standard Unix paths, where the delimiter is a forward slash `"/`. Backward slashes are not permitted. A database is given a file path and must not contain a trailing slash. Paths may be relative, as indicated by an initial character other than `"/`, including the dot `"."` for current directory or dot-dot `".."` for parent directory. The shell meta-character `"~"` is not interpreted.

Key strings may be translated into paths by the implementation, meaning that the string may not contain embedded slashes (`"/` or `"\"`), neither are leading periods `"."` permitted for the same reason. Key strings are recommended to be ASCII-7, a proper subset of UTF-8 UNICODE.

SRAErrMake

Description

convert `rc_t` to simple integer

Prototype



```
int SRAErrMake ( rc_t rc );
```

Parameters:

Parameter	Input Output	Description
rc	in	SRA return code type (rc_t) that is to be converted into an integer.

SRAErrToEnglish

Description

Returns an ASCII NUL-terminated string with an English explanation of error code

Prototype

```
const char *SRAErrToEnglish ( int32_t sraErr, size_t *bytes );
```

Parameters:

Parameter	Input Output	Description
sraErr	in	An SRAErr status code requiring explanation
bytes	out, NULL ok	Optional return parameter for size of return string. also indicates the length, since this interface specifies ASCII-7.

Read APIs

SRANamelistAddRef

Description

Add a Namelist reference

Prototype

```
rc_t SRANamelistAddRef ( const SRANamelist *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRANamelist value for which to add a reference.

SRANamelistRelease

Description

Release a SRANamelist referenced object.

Prototype

```
rc_t SRANamelistRelease ( const SRANamelist *self );
```

Parameters:

Parameter	Input Output	Description
self	in	The SRANamelist pointer to the object to be released.

SRANamelistCount

Description

Get the number of names

Prototype

```
rc_t SRANamelistCount ( const SRANamelist *self, uint32_t *count );
```

Parameters:

Parameter	Input Output	Description
self	in	The SRANamelist pointer.
count	out	Return value.

SRANamelistGet

Description

Get the list of names.

Prototype

```
rc_t SRANamelistGet ( const SRANamelist *self, uint32_t idx, const char **name );
```

Parameters:

Parameter	Input Output	Description
self	in	The SRANamelist pointer.
idx	in	zero-based name index
name	out	Return parameter for zero-terminated name.

SRAMgrAddRef

Description

Add a reference to the SRAMgr handle

Prototype

```
rc_t SRAMgrAddRef ( const SRAMgr *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRAMgr value for which to add a reference.

SRAMgrRelease

Description

Release the reference to the SRAMgr handle.

Prototype

```
rc_t SRAMgrRelease ( const SRAMgr *self );
```

Parameters:

Parameter	Input Output	Description
self	in	The pointer to the SRAMgr object to be released.

SRAMgrMakeRead

Description

Create library handle for read-only access. *NOTE - not implemented in update library and the read-only library may not be mixed with read/write.* In order to read from a Short Read Archive, you must first get a library handle (done via this API call).

Prototype

```
rc_t SRAMgrMakeRead ( const SRAMgr **mgr );
```

Parameters:

Parameter	Input Output	Description
mgr	out	Returns the SRAMgr object that will be used to reference the

		Archive.
--	--	----------

SRAMgrOpenDatatypesRead

Description

Open datatype registry object for requested access.

Prototype

```
rc_t SRAMgrOpenDatatypesRead ( const SRAMgr *self, struct VDatatypes const **dt );
```

Parameters:

Parameter	Input Output	Description
mgr	in	SRAMgr handle.
dt	out	Return parameter for datatypes object

SRAMgrOpenDatatypesUpdate

Description

Open datatype registry object for requested access

Prototype

```
rc_t SRAMgrOpenDatatypesUpdate ( const SRAMgr *self, struct VDatatypes **dt );
```

Parameters:

Parameter	Input Output	Description
mgr	in	SRAMgr handle.
dt	out	Return parameter for datatypes object

SRATableAddRef

Description

Add a reference to a SRATable object

Prototype

```
rc_t SRATableAddRef ( const SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable value for which to add a reference.

SRATableRelease**Description**

Release the reference to the SRATable object.

Prototype

```
rc_t SRATableRelease ( const SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	The pointer to the SRATable object to be released.

SRAMgrOpenTableRead**Description**

Open an existing table.

Prototype

```
rc_t SRAMgrOpenTableRead ( const SRAMgr *self, const SRATable **tbl, const char *path, ... );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
tbl	out	return parameter for the table
path	in	zero-terminated string referencing the table in the filesystem.

SRAMgrVOpenTableRead**Description**

Open an existing table.

Prototype


```
rc_t SRAMgrVOpenTableRead ( const SRAMgr *self, const SRATable **tbl, const char *path, va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
tbl	out	return parameter for the table
path	in	zero-terminated string referencing the table in the filesystem.

SRATableBaseCount

Description

Get the number of stored bases

Prototype

```
rc_t SRATableBaseCount ( const SRATable *self, uint64_t *num_bases );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
num_bases	out	return parameter for the base count.

SRATableSpotCount

Description

Get the number of stored spots

Prototype

```
rc_t SRATableSpotCount ( const SRATable *self, uint64_t *spot_count );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
num_spots	out	return parameter for the spot count.

SRATableMaxSpotId

Description

Returns the maximum spot id. A table will contain a collection of spots with ids from 1 to max (spot_id) unless empty.

Prototype

```
rc_t SRATableMaxSpotId ( const SRATable *self, spotid_t *id );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
id	out	return parameter of last spot id or zero if the table is empty.

SRATableGetSpotId

Description

Convert spot name to spot id

Prototype

```
rc_t SRATableGetSpotId ( const SRATable *self, spotid_t *id, const char *spot_name );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
id	out	return parameter for 1-based spot id.
spot_name	in	External spot name string in platform canonical format.

SRATableListCol

Description

Returns a list of simple column names. Each name represents at least one typed column.

Prototype

```
rc_t SRATableListCol ( const SRATable *self, SRANamelist **names );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
names	out	Return parameter for names list

SRATableColDatatypes

Description

Returns list of type declarations for the named column.

Prototype

```
rc_t SRATableColDatatypes ( const SRATable *self, const char *col, uint32_t *dflt_idx, SRANamelist **typedecls );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
col	in	column name
dflt_idx	out, NULL ok	Returns the zero-based index into "typedecls" of the default datatype for the named column
typedecls	out	List of datatypes available for named column

SRATableMetaRevision

Description

Returns current revision number where zero means tip.

Prototype

```
rc_t SRATableMetaRevision ( const SRATable *self, uint32_t *revision );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
revision	out	Return parameter holding the revision number.

SRATableMaxMetaRevision

Description

Returns the maximum revision available for the given table.

Prototype

```
rc_t SRATableMaxMetaRevision ( const SRATable *self, uint32_t *revision );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
revision	in	Return parameter holding the revision number

SRATableUseMetaRevision

Description

Opens the indicated revision of metadata. All non-zero revisions are read-only.

Prototype

```
rc_t SRATableUseMetaRevision ( const SRATable *self, uint32_t revision );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
revision	in	The metadata table revision to open

SRATableOpenMDataNodeRead

Description

Open a metadata node.

Prototype

```
rc_t SRATableOpenMDataNodeRead ( const SRATable *self, struct KMDDataNode  
const **node, const char *path, ... );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
node	out	return parameter for metadata node
path	in	Simple or hierarchical zero-terminated path to the node.

SRATableVOpenMDataNodeRead

Description

Open a metadata node.

Prototype

```
rc_t SRATableVOpenMDataNodeRead ( const SRATable *self, struct KMDataNode
const **node, const char *path, va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
node	out	return parameter for metadata node
path	in	Simple or hierarchical zero-terminated path to the node.

SRATableListMetaChild

Description

Returns a list of simple child node names

Prototype

```
rc_t SRATableListMetaChild ( const SRATable *self, SRANamelist **names, const
char *node, ... );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.
names	out	return parameter for names list
path	out	simple or hierarchical NULL terminated path to the node.

SRATableVListMetaChild

Description

Returns a list of simple child node names.

Prototype

```
rc_t SRATableVListMetaChild ( const SRATable *self, SRANamelist **names,
const char *node, va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRATable object.

names	out	return parameter for names list
path	out	simple or hierarchical NULL terminated path to the node.

SRAColumnAddRef

Description

Add a reference to a column object.

Prototype

```
rc_t SRAColumnAddRef ( const SRAColumn *self );
```

Parameters:

Parameter	Input Output	Description
self	int	Pointer to the SRAColumn that is to be referenced.

SRAColumnRelease

Description

Release the reference to the column object.

Prototype

```
rc_t SRAColumnRelease ( const SRAColumn *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRAColumn that is to be de-referenced.

SRATableOpenColumnRead

Description

Open a column for reading

Prototype

```
rc_t SRATableOpenColumnRead ( const SRATable *self, const SRAColumn **col,
const char *name, const char *datatype );
```

Parameters:

Parameter	Input	Description
-----------	-------	-------------

	Output	
col	out	return parameter for newly opened column
name	in	zero-terminated string in UTF-8 giving column name
datatype	in, NULL ok	Optional zero-terminated typedef string describing fully qualified column data type, or if NULL the default type for column.

SRAColumnDatatype

Description

Access the data type

Prototype

```
rc_t SRAColumnDatatype ( const SRAColumn *self, struct VTypedecl *type,
struct VTypeDef *def );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRAColumn being accessed.
type	out, NULL ok	returns the column type declaration
def	out, NULL ok	Returns the definition of the type returned in "type_decl"

SRAColumnGetRange

Description

Get a contiguous range around a spot id, e.g. tile for Illumina

Prototype

```
rc_t SRAColumnGetRange ( const SRAColumn *self, spotid_t id, spotid_t *first,
spotid_t *last );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRAColumn being accessed.
id	in	return parameter for 1-based spot id
first	out	First id in range is returned, where at least ONE (first or last) must be NOT-NULL
last	out	Last id in range is returned, where at least ONE (first or last) must be

		NOT-NULL
--	--	----------

SRAColumnRead

Description

Read row data

Prototype

```
rc_t SRAColumnRead ( const SRAColumn *self, spotid_t id, const void **base,
bitsz_t *offset, bitsz_t *size );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to the SRAColumn being accessed.
id	in	spot row id between 1 and max (spot id)
base	out	Pointer to the start of the spot row data (used with offset to get location).
offset	out	bit offset to the start of the spot row data (used with base to get location).
size	out	size in bits of row data.

Write APIs

SRAMgrMakeUpdate

Description

Create library handle for read/write access.*NOTE - not implemented in read-only library, and the read-only library may not be mixed with read/write*

Prototype

```
rc_t SRAMgrMakeUpdate ( SRAMgr **mgr, struct KDirectory *wd );
```

Parameters:

Parameter	Input Output	Description
self	out	return reference to the SRAMgr object.

wd	in, NULL ok	Optional working directory for accessing the file system. mgr will attach its own reference.
-----------	-------------	--

SRAMgrCreateTable

Description

creates a new table

Prototype

```
rc_t SRAMgrCreateTable ( SRAMgr *self, SRATable **tbl, const char *path, ...
);
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to SRAMgr handle
tbl	out	Return parameter for table
path	in	zero-terminated string referencing the table in the filesystem.

SRAMgrVCreateTable

Description

Creates a new table.

Prototype

```
rc_t SRAMgrVCreateTable ( SRAMgr *self, SRATable **tbl, const char *path,
va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to SRAMgr handle
tbl	out	Return parameter for table
path	in	zero-terminated string referencing the table in the filesystem.

SRAMgrOpenTableUpdate

Description

Open an existing table.

Prototype

```
rc_t SRAMgrOpenTableUpdate ( SRAMgr *self, SRATable **tbl, const char *path,
... );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to SRAMgr handle
tbl	out	Return parameter for table
path	in	zero-terminated string referencing the table in the filesystem.

SRAMgrVOpenTableUpdate

Description

Open an existing table.

Prototype

```
rc_t SRAMgrVOpenTableUpdate ( SRAMgr *self, SRATable **tbl, const char *path,
va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Pointer to SRAMgr handle
tbl	out	Return parameter for table
path	in	zero-terminated string referencing the table in the filesystem

SRATableLocked

Description

Check to see if a table is locked. Used in conjunction with updating a table. *Note - you cannot open a locked table for an update.* If the table is locked, you may attempt to unlock the table.

Prototype

```
bool SRATableLocked ( const SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Table handle (pointer to SRATable) to check whether lock exists.

SRATableLock

Description

Lock a table. Used in conjunction with updating a table. *Note - you cannot open a locked table for an update.* Once a table is locked, it can be safely modified until the table is unlocked.

Prototype

```
rc_t SRATableLock ( SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Table handle (pointer to SRATable) to apply lock to.

SRATableUnlock

Description

Unlock a table. Used in conjunction with updating a table. *Note - you cannot open a locked table for an update.*

Prototype

```
rc_t SRATableUnlock ( const SRATable *self, SRATable **unlocked );
```

Parameters:

Parameter	Input Output	Description
self	in	Table handle (pointer to SRATable) to attempt to unlock.
unlocked	out	return pointer to Table handle (pointer to SRATable) of the unlocked table.

SRATableNewSpot

Description

Creates a new spot record, returning spot id.

Prototype

```
rc_t SRATableNewSpot ( SRATable *self, spotid_t *id, uint8_t dim, const uint16_t *coord );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
id	in	return parameter for id of newly created spot
dim	in	spot coordinate, where "dim" >= 3. The coordinate vector is reversed such that: y = coord[0], x = coord[1], <next> = coord[2], ...
coord	in	spot coordinate, where "dim" >= 3. The coordinate vector is reversed such that: y = coord[0], x = coord[1], <next> = coord[2], ...

SRATableOpenSpot

Description

Opens an existing spot record from id

Prototype

```
rc_t SRATableOpenSpot ( SRATable *self, spotid_t id );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
id	in	1-based spot id.

SRATableCloseSpot

Description

Closes a spot opened with either NewSpot or OpenSpot

Prototype

```
rc_t SRATableOpenSpot ( SRATable *self, spotid_t id );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
id	in	1-based spot id.

SRATableCommit

Description

Commit all changes

Prototype

```
rc_t SRATableCommit ( SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.

SRATableOpenColumnWrite

Description

Open a column for write

Prototype

```
rc_t SRATableOpenColumnWrite ( SRATable *self, uint32_t *idx, SRAColumn  
**col, const char *name, const char *datatype );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
idx	out	return parameter for 1-based column index.
col	out, NULL ok	optional return parameter for newly opened column.
name	in	zero-terminated string in UTF-8 giving column name
datatype	in	zero-terminated string in ASCII describing fully qualified column data type

SRATableSetIdxColumnDefault

Description

Give a default value for column. If no value gets written to a column within an open spot, this value is substituted.

Prototype

```
rc_t SRATableSetIdxColumnDefault ( SRATable *self, uint32_t idx, const void *base, bitsz_t offset, bitsz_t size );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
idx	in	1-based column index
base	in	Pointer and to start of row data (used in conjunction with "offset")
offset	in	Offset to start of row data (used in conjunction with "base")
size	in	size in bits of row data

SRATableWriteIdxColumn

Description

Prototype

```
rc_t SRATableWriteIdxColumn ( SRATable *self, uint32_t idx, const void *base, bitsz_t offset, bitsz_t size );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
idx	in	1-based column index
base	in	Pointer and to start of row data (used in conjunction with "offset")
offset	in	Offset to start of row data (used in conjunction with "base")
size	in	size in bits of row data

SRATableMetaFreeze

Description

Freezes current metadata revision. Further modification will begin on a copy.

Prototype

```
rc_t SRATableMetaFreeze ( SRATable *self );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.

SRATableOpenMDataNodeUpdate

Description

Open a metadata node.

Prototype

```
rc_t SRATableOpenMDataNodeUpdate ( SRATable *self, struct KMDataNode **node,  
const char *path, ... );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
node	out	return value to the meta data node (KMDataNode) where the metadata is kept.
path	in	zero-terminated string referencing the table in the filesystem.

SRATableVOpenMDataNodeUpdate

Description

Prototype

```
rc_t SRATableVOpenMDataNodeUpdate ( SRATable *self, struct KMDataNode **node,  
const char *path, va_list args );
```

Parameters:

Parameter	Input Output	Description
self	in	Handle of table (SRATable) that is being referenced.
node	out	return value to the meta data node (KMDataNode) where the metadata is kept.
path	in	zero-terminated string referencing the table in the filesystem