SCIC

Release 1.10.4

Ulises Jeremias Cornejo Fandos

Aug 27, 2018

Contents

CHAPTER 1

Introduction

The SciC is a collection of routines and apps written from scratch in C, and present a modern Applications Programming Interface (API) for C programmers, allowing wrappers to be written for very high level languages. The source code is distributed under the MIT License.

1.1 Routines available in SCIC

The libraries cover a wide range of topics in scientific computing. Routines are available for the following areas,

Error Handling	Numerical Computation	Data Structures

1.2 Conventions used in this manual

This manual contains many examples which can be typed at the keyboard. A command entered at the terminal is shown like this:

```
$ command
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign $ is used as the standard prompt in this manual, although some systems may use a different character.

The examples assume the use of the GNU operating system. There may be minor differences in the output on other systems. The commands for setting environment variables use the Bourne shell syntax of the standard GNU shell (`bash`).

CHAPTER 2

Using the Libraries

This chapter describes how to compile programs that use SCIC libraries, and introduces its conventions.

2.1 Compiling and Linking

The library header files are installed in their own `scic` directory. You should write any preprocessor include statements with a `scic/` directory prefix thus:

```
#include <scic/errno.h>
```

If the directory is not installed on the standard search path of your compiler you will also need to provide its location to the preprocessor as a command line flag. The default location of the `scic` directory is `/usr/local/include`.

A typical compilation command for a source file `example.c` with the GNU C compiler `gcc` is:

```
$ gcc -Wall -I/usr/local/include -c example.c
```

This results in an object file `example.o`. The default include path for `gcc` searches `/usr/local/include` automatically so the `-I` option can actually be omitted when SCIC is installed in its default location.

2.1.1 Linking programs with the library

The library is installed as a single file, e.g. `libscic-errno.a` for the `scic/errno` library. A shared version of the library `libscic-errno.so` is also installed on systems that support shared libraries. The default location of these files is `/usr/local/lib`. If this directory is not on the standard search path of your linker you will also need to provide its location as a command line flag. The following example shows how to link an application with the library:

```
$ gcc -L/usr/local/lib example.o -lscic-errno
```

The default library path for `gcc` searches `/usr/local/lib` automatically so the `-L` option can be omitted when SCIC is installed in its default location.

For a tutorial introduction to the GNU C Compiler and related programs, see "An Introduction to GCC" (ISBN 0954161793).[1]

2.2 Shared Libraries

To run a program linked with the shared version of the library the operating system must be able to locate the corresponding `.so` file at runtime. If the library cannot be found, the following error will occur:

```
$ ./a.out
./a.out: error while loading shared libraries:
libscic-errno.so.0: cannot open shared object file: No such file or directory
```

To avoid this error, either modify the system dynamic linker configuration[2] or define the shell variable `LD_LIBRARY_PATH` to include the directory where the library is installed.

For example, in the Bourne shell (`/bin/sh` or `/bin/bash`), the library search path can be set with the following commands:

```
$ LD_LIBRARY_PATH=/usr/local/lib
$ export LD_LIBRARY_PATH
$ ./example
```

In the C-shell (`/bin/csh` or `/bin/tcsh`) the equivalent command is:

```
% setenv LD_LIBRARY_PATH /usr/local/lib
```

The standard prompt for the C-shell in the example above is the percent character %, and should not be typed as part of the command.

To save retyping these commands each session they can be placed in an individual or system-wide login file.

To compile a statically linked version of the program, use the `-static` flag in `gcc`:

```
$ gcc -static example.o -lscic-errno
```

2.3 ANSI C Compliance

The library is written in ANSI C and is intended to conform to the ANSI C standard (C89). It should be portable to any system with a working ANSI C compiler.

The library does not rely on any non-ANSI extensions in the interface it exports to the user. Programs you write using SCIC can be ANSI compliant. Extensions which can be used in a way compatible with pure ANSI C are supported, however, via conditional compilation. This allows the library to take advantage of compiler extensions on those platforms which support them.

When an ANSI C feature is known to be broken on a particular system the library will exclude any related functions at compile-time. This should make it impossible to link a program that would use these functions and give incorrect results.

To avoid namespace conflicts all exported function names and variables have the prefix `scic_`, while exported macros have the prefix `SCIC_`.

[1] http://www.network-theory.co.uk/gcc/intro/

[2] `/etc/ld.so.conf` on GNU/Linux systems

2.4 Inline functions

The `inline` keyword is not part of the original ANSI C standard (C89) so the library does not export any inline function definitions by default. Inline functions were introduced officially in the newer C99 standard but most C89 compilers have also included `inline` as an extension for a long time.

To allow the use of inline functions, the library provides optional inline versions of performance-critical routines by conditional compilation in the exported header files.

By default, the actual form of the inline keyword is `extern inline`, which is a `gcc` extension that eliminates unnecessary function definitions.

When compiling with gcc in C99 mode (*gcc -std=c99*) the header files automatically switch to C99-compatible inline function declarations instead of extern inline.

2.5 Long double

In general, the algorithms in the library are written for double precision only. The `long double` type is not supported for every computation.

One reason for this choice is that the precision of `long double` is platform dependent. The IEEE standard only specifies the minimum precision of extended precision numbers, while the precision of `double` is the same on all platforms.

However, it is sometimes necessary to interact with external data in long-double format, so the structures datatypes include long-double versions.

It should be noted that in some system libraries the `stdio.h` formatted input/output functions `printf` and `scanf` are not implemented correctly for `long double`. Undefined or incorrect results are avoided by testing these functions during the `configure` stage of library compilation and eliminating certain SCIC functions which depend on them if necessary. The corresponding line in the `configure` output looks like this:

```
checking whether printf works with long double... no
```

Consequently when `long double` formatted input/output does not work on a given system it should be impossible to link a program which uses SCIC functions dependent on this.

If it is necessary to work on a system which does not support formatted `long double` input/output then the options are to use binary formats or to convert `long double` results into `double` for reading and writing.

2.6 Compatibility with C++

The library header files automatically define functions to have `extern "C"` linkage when included in C++ programs. This allows the functions to be called directly from C++.

2.7 Thread-safety

The library can be used in multi-threaded programs. All the functions are thread-safe, in the sense that they do not use static variables. Memory is always associated with objects and not with functions. For functions which use *workspace* objects as temporary storage the workspaces should be allocated on a per-thread basis. For functions which use *table* objects as read-only memory the tables can be used by multiple threads simultaneously.

CHAPTER 3

Error Handling

This chapter describes the way that SciC functions report and handle errors. By examining the status information returned by every function you can determine whether it succeeded or failed, and if it failed you can find out what the precise cause of failure was. You can also define your own error handling functions to modify the default behavior of the library.

The functions described in this chapter are declared in the header file `scic/errno.h`.

3.1 Error Reporting

The library follows the thread-safe error reporting conventions of the POSIX Threads library. Functions return a non-zero error code to indicate an error and `0` to indicate success:

```
int status = scic_function (...)

if (status) { /* an error occurred */
  .....
  /* status value specifies the type of error */
}
```

The routines report an error whenever they cannot perform the task requested of them. For example, a root-finding function would return a non-zero error code if could not converge to the requested accuracy, or exceeded a limit on the number of iterations. Situations like this are a normal occurrence when using any mathematical library and you should check the return status of the functions that you call.

Whenever a routine reports an error the return value specifies the type of error. The return value is analogous to the value of the variable `errno` in the C library. The caller can examine the return code and decide what action to take, including ignoring the error if it is not considered serious.

In addition to reporting errors by return codes the library also has an error handler function `scic_error()`. This function is called by other library functions when they report an error, just before they return to the caller. The default behavior of the error handler is to print a message and abort the program:

```
scic: file.c:67: ERROR: invalid argument supplied by user
Default SCIC error handler invoked.
Aborted
```

The purpose of the `scic_error()` handler is to provide a function where a breakpoint can be set that will catch library errors when running under the debugger. It is not intended for use in production programs, which should handle any errors using the return codes.

3.2 Error Codes

The error code numbers returned by library functions are defined in the file `scic/errno.h`. They all have the prefix `SCIC_` and expand to non-zero constant integer values. Error codes above 1024 are reserved for applications, and are not used by the library. Many of the error codes use the same base name as the corresponding error code in the C library. Here are some of the most common error codes,

int **SCIC_EDOM**
: Domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined (like `EDOM` in the C library)

int **SCIC_ERANGE**
: Range error; used by mathematical functions when the result value is not representable because of overflow or underflow (like `ERANGE` in the C library)

int **SCIC_ENOMEM**
: No memory available. The system cannot allocate more virtual memory because its capacity is full (like `ENOMEM` in the C library). This error is reported when a SCIC routine encounters problems when trying to allocate memory with `malloc()`.

int **SCIC_EINVAL**
: Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function (like `EINVAL` in the C library).

The error codes can be converted into an error message using the function *scic_strerror()*.

const char * **scic_strerror** (const int *scic_errno*)
: This function returns a pointer to a char describing the error code `scic_errno`. For example:

```
printf ("error: %s\n", scic_strerror (status));
```

would print an error message like `error: output range error` for a status value of *SCIC_ERANGE*.

3.3 Error Handlers

The default behavior of the SCIC error handler is to print a short message and call `abort()`. When this default is in use programs will stop with a core-dump whenever a library routine reports an error. This is intended as a fail-safe default for programs which do not check the return status of library routines (we don't encourage you to write programs this way).

If you turn off the default error handler it is your responsibility to check the return values of routines and handle them yourself. You can also customize the error behavior by providing a new error handler. For example, an alternative error handler could log all errors to a file, ignore certain error conditions (such as underflows), or start the debugger and attach it to the current process when an error occurs.

All SCIC error handlers have the type `scic_error_handler_t`, which is defined in `scic_errno.h`,

scic_error_handler_t

This is the type of SCIC error handler functions. An error handler will be passed four arguments which specify the reason for the error (a string), the name of the source file in which it occurred (also a string), the line number in that file (an integer) and the error number (an integer). The source file and line number are set at compile time using the __FILE__ and __LINE__ directives in the preprocessor. An error handler function returns type void. Error handler functions should be defined like this:

```
void handler (const char * reason,
              const char * file,
              int line,
              int scic_errno)
```

To request the use of your own error handler you need to call the function *scic_set_error_handler()* which is also declared in scic_errno.h,

scic_error_handler_t * **scic_set_error_handler** (*scic_error_handler_t* * *new_handler*)

This function sets a new error handler, new_handler, for the SCIC library routines. The previous handler is returned (so that you can restore it later). Note that the pointer to a user defined error handler function is stored in a static variable, so there can be only one error handler per program. This function should be not be used in multi-threaded programs except to set up a program-wide error handler from a master thread. The following example shows how to set and restore a new error handler:

```
/* save original handler, install new handler */
old_handler = scic_set_error_handler (&my_handler);

/* code uses new handler */
.....

/* restore original handler */
scic_set_error_handler (old_handler);
```

To use the default behavior (abort() on error) set the error handler to NULL:

```
old_handler = scic_set_error_handler (NULL);
```

scic_error_handler_t * **scic_set_error_handler_off** ()

This function turns off the error handler by defining an error handler which does nothing. This will cause the program to continue after any error, so the return values from any library routines must be checked. This is the recommended behavior for production programs. The previous handler is returned (so that you can restore it later).

The error behavior can be changed for specific applications by recompiling the library with a customized definition of the SCIC_ERROR macro in the file scic_errno.h.

3.4 Using SCIC error reporting in your own functions

If you are writing numerical functions in a program which also uses SCIC code you may find it convenient to adopt the same error reporting conventions as in the library.

To report an error you need to call the function scic_error() with a string describing the error and then return an appropriate error code from scic_errno.h, or a special value, such as NaN. For convenience the file scic_errno.h defines two macros which carry out these steps:

SCIC_ERROR (reason, scic_errno)

This macro reports an error using the SCIC conventions and returns a status value of scic_errno. It expands to the following code fragment:

```
scic_error (reason, __FILE__, __LINE__, scic_errno);
return scic_errno;
```

The macro definition in `scic_errno.h` actually wraps the code in a `do { ... } while (0)` block to prevent possible parsing problems.

Here is an example of how the macro could be used to report that a routine did not achieve a requested tolerance. To report the error the routine needs to return the error code `SCIC_ETOL`:

```
if (residual > tolerance)
  {
    SCIC_ERROR("residual exceeds tolerance", SCIC_ETOL);
  }
```

SCIC_ERROR_VAL (reason, scic_errno, value)

This macro is the same as `SCIC_ERROR` but returns a user-defined value of `value` instead of an error code. It can be used for mathematical functions that return a floating point value.

The following example shows how to return a `NaN` at a mathematical singularity using the `SCIC_ERROR_VAL` macro:

```
if (x == 0)
  {
    SCIC_ERROR_VAL("argument lies on singularity", SCIC_ERANGE, SCIC_SCIC_NAN);
  }
```

CHAPTER 4

Indices and tables

- genindex

Index