LIGO-T020072-00-D

LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
Technical Note

The Signal Injection Handbook

Peter Shawhan, Daniel Sigg

June 3, 2002

Introduction

The LIGO length-control servo, and other "front end" control systems, have the capability to add user-supplied "excitation" waveforms to the regular feedback waveforms at various points in the system. This was originally used only to inject sine waves and other periodic signals (*e.g.* for calibration purposes), but in October 2001 we wrote software to allow simulated waveforms of arbitrary duration to be injected into the interferometer hardware. This document is intended to serve as a User's Guide for this capability.

The basic approach is that a client program streams waveform data in one-second chunks over ethernet to the GDS "arbitrary waveform generator" processor, which sends the waveform to the excitation channels with the proper synchronization. There is a library of client-side routines (called "`SIStr`", for Signal Injection Stream) which provides a very simple external call interface, and internally takes care of all the buffering and timing issues. At present, there are two ways in which this can be used:

1. There is a utility called `awgstream` which reads waveform data (*i.e.* a sequence of real numbers) from a formatted ASCII file and makes the appropriate `SIStr` calls to send the waveform to a specified excitation channel.

2. Users can write their own client programs in C which call functions in the `SIStr` library. This opens up the possibility of injecting very long waveforms which are computed on-the-fly, *e.g.* signals from periodic sources.

Any client program must run on one of the Sun workstations on the CDS network. The waveforms are synchronized to the GPS clock, so it would be possible to inject waveforms simultaneously into multiple interferometers with a known timing relationship.

How to Use the **`awgstream`** Utility

1. Determine what excitation channel you will use

The following channels (all sampled at 16384 Hz) are available for injecting signals at various points in the LSC servo:

At Hanford:	
H1:LSC-ITMX_EXC	and similarly for ITMY, ETMX, ETMY, BS
H1:LSC-DARM_CTRL_EXC	and similarly for CARM, MICH, PRC
H1:LSC-DARM_ERR_EXC	and similarly for CARM, MICH, PRC
H2:LSC-ITMX_EXC	and similarly for ITMY, ETMX, ETMY, BS
H2:LSC-DARM_CTRL_EXC	and similarly for CARM, MICH, PRC
H2:LSC-DARM_ERR_EXC	and similarly for CARM, MICH, PRC
At Livingston:	
L1:LSC-ITMX_EXC	and similarly for ITMY, ETMX, ETMY, BS
L1:LSC-DARM_CTRL_EXC	and similarly for CARM, MICH, PRC
L1:LSC-DARM_ERR_EXC	and similarly for CARM, MICH, PRC

Other excitation channels are available to inject waveforms into other subsystems, such as the alignment servo. There is a trick to get a complete list of excitation channels and their sampling rates: log into any machine on the CDS cluster and type:

```
  setenv GDS_DIR /opt/CDS/d/gds/diag      [/opt/LLO/c/gds/diag at LLO]
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$GDS_DIR/lib
  ~controls/pshawhan/awgstream -d -d blah 1 null.dat
```

2. Prepare a waveform file

This should be a formatted ASCII file containing a list of real numbers separated by newlines or spaces. For example:

```
  -4.903158e-04
  -2.675681e-04
  -4.475717e-05
  1.780641e-04
  ...
```

Note that any parsing error will cause the input file to be close and the waveform, up to the point of the parsing error, to be flushed to the front end.

The sampling rate must match the intrinsic rate of the excitation channel onto which this waveform is to be injected. The normalization is nominally in "counts", but you will be able to scale the entire waveform by a constant factor when you inject it. The one tricky part about generating the waveform is that you must account for the transfer function between the point in the servo system where you inject it, and the actual mirror motion.

For example, if you inject a waveform onto the LSC-ETMX_EXC excitation channel, this causes current to flow proportionally in the coils, but the actual motion of the mirror is subject to the "pendulum" response function, which goes as $-1/f^2$ at frequencies far above the pendulum frequency. Thus, the waveform injected into the system should be the gravitational-wave strain waveform, weighted by a factor proportional to f^2 in the frequency domain. For simulated inspiral waveforms, I made this adjustment by reading the waveform into a Matlab array, doing a fast Fourier transform, weighting the frequency-domain data, then doing an inverse FFT to go back to the time domain.

3. Copy the waveform file to the CDS cluster

The CDS unix cluster is on a private network and is not visible to the outside world, except for a single gateway machine (`red.ligo-wa.caltech.edu` at Hanford, and `london.ligo-la.caltech.edu` at Livingston). The standard way to copy files to the CDS cluster is:

1. Log into the gateway machine as '`ops`' . You will need to know the password to do this, of course.
2. Make a subdirectory below `~ops` for your personal use. The custom is to use your usual unix username for the subdirectory name, e.g. '`mkdir pshawhan`'
3. Use `scp` to copy the remote waveform file to your personal subdirectory.

4. Log into any machine on the CDS cluster as 'ops', and get set up

After logging in, you need to set your `LD_LIBRARY_PATH` to include the directory with the libraries for communicating with the servo system. At Hanford, do:

```
  setenv GDS_DIR /opt/CDS/d/gds/diag
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$GDS_DIR/lib
```

At Livingston, do:

```
  setenv GDS_DIR /opt/LLO/c/gds/diag
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$GDS_DIR/lib
```

Then change directories to your personal subdirectory.

5. Remind yourself of the syntax for the `awgstream` utility

This is the utility program which reads a waveform out of an ASCII file and injects it onto the excitation channel you specify. It is located in the `~controls/pshawhan` directory. To get a reminder of the syntax, just run it without any arguments:

```
  ~controls/pshawhan/awgstream
```

This will print out:

```
Usage: awgstream <channel> <rate> <file> [<scale> [<gpstime>]] [-d]
  <channel> is case-sensitive and must be a real excitation channel
  <rate> is in Hz and must match the excitation channel's true rate
  <file> is the file of waveform data, which must be a FORMATTED ascii file
      with the values separated by whitespace (e.g. newlines and/or spaces).
      Any conversion error causes the rest of the file to be skipped.
      If <file> is '-' (i.e. a dash), data is read from standard input.
  <scale> is an optional scale factor to apply to the data in the file.
  <gpstime> is the time to start injecting the waveform.  It can take any
      real value, not just an integer.  It must occur during the 24-hour
      period following the execution of the awgstream command.
      If omitted, the waveform injection will start in about 10 seconds.
  The '-d' option causes debugging information to be printed to the screen.
      Specify '-d' twice for extra information.
```

6. Determine the scale factor to use when injecting the waveform

Each sample in the waveform file is multiplied by the scale factor argument to `awgstream`, if present. The resulting value is in units of "counts" within the LSC servo, which ultimately gets converted to a current in the coil drivers. The exact conversion factor may be obscure, but in practice net amplitudes of a few counts to a few hundred counts are probably appropriate. Injecting signals of more than a few thousand counts will knock the interferometer out of lock and/or make people nervous about over-driving the coil drivers. Choose a scale factor accordingly.

7. Choose a start time

You can specify a GPS time that is at least several seconds in the future, and not more than 24 hours in the future. An error will occur if you specify a time which does not fall within this window. With sufficient tunneling through gateway machines, you should be able to inject waveforms synchronously at both observatories.

If you do not specify a start time, the software picks a start time several seconds in the future, aligned to an exact integer number of GPS seconds. However, by default it does not tell you what that time is. Use the `'-d'` flag to cause this and other "debugging" information to be printed to the screen.

8. Prepare to monitor the injected waveform

Before injecting the waveform, set up Data Viewer to look at the excitation channel you will be using. Since excitation channels are disabled except when explicitly used, you may see a repeated pattern of garbage data on the screen until the injection actually starts. You should make sure the vertical scale is auto-ranging, and de-select the "Units" checkbox to disable any (probably meaningless) conversion from "counts" to some other units.

9. Inject the waveform

Now you can run the `awgstream` utility! For example:

```
  ~controls/pshawhan/awgstream H2:LSC-ETMX_EXC 16384 inspiral16384.dat 0.12 -d
```

Caveats

If the waveform does not seem to be showing up on the interferometer output (and you think the amplitude is large enough that you should be able to see it), make sure the LSC servo is not disabling the signal that you inject into it. For instance, I tried injecting inspiral chirps into ITMX, only to discover that ITMX was disabled on the servo control screens, so that the waveform I injected was being ignored.

Miscellaneous

There is a command-line interface to the GDS arbitrary waveform generator. To invoke it, type:

```
    diag –l –c
```

Here are a few example commands:

```
    awg show 1
    awg free 2007
```

The arbitrary waveform generator also keeps track of cumulative statistics. To view them, do:

```
    awg stat 1
```

To clear the statistics registers:

```
    awg stat 1 clear
```

Note that the `awgstream` utility will read the waveform from standard input if you specify a dash for the input filename.

How to Use the C Interface

The Signal Injection Stream library provides a simple interface so that users can write their own client programs to send waveform data to the front end. The source code resides in the files `SIStr.h` and `SIStr.c`, and there are only three or four functions which are commonly called by the user's code.

Basic skeleton of a user program

```
#include <stdlib.h>
#include <stdio.h>
#include "SIStr.h"
SIStream sis;

status = SIStrOpen( &sis, channel, samprate, starttime );
if ( status != SIStr_OK ) {
  fprintf( stderr, "Error opening stream: %s\n", SIStrErrorMsg(status) );
  return 2;
}

while ( there is waveform data to send ) {
  status = SIStrAppend( &sis, data_array, ndata, scale );
  if ( status != SIStr_OK ) {
    fprintf( stderr, "Error streaming data: %s\n", SIStrErrorMsg(status) );
    break;
  }
}

status = SIStrClose( &sis );
if ( status != SIStr_OK ) {
  fprintf( stderr, "Error closing stream: %s\n", SIStrErrorMsg(status) );
  return 2;
}
```

Notes on calling `SIStr` functions

Note how we check for an error after each call to a `SIStr` function, but if `SIStrAppend` returns an error, then we do **not** exit out of the program; we just break out of the loop so we can still call `SIStrClose` to clean up.

In the call to `SIStrOpen`, you must pass the name of a valid excitation channel, and a sampling rate that matches the actual sampling rate for that channel. You can pass an explicit start time (in GPS seconds, as a double-precision value), which must be between the current time, and the current time plus one day. Or you can pass `starttime=0.0`, in which case the waveform injection will start "right away" (actually, after a delay of several seconds to permit adequate buffering).

In the call to `SIStrAppend`, `data_array` is a pointer to an array of single-precision floating-point values, and ndata is the number of values. You may append any number of values at a time, and the number may change from one call to the next. Feel free to append a single value at a time if you want (*i.e.* `ndata=1`), but note that if you have the value in a scalar variable, then `data_array` must be a **pointer** to that variable; you can't pass the value directly.

Waveform data passed to `SIStrAppend` is assembled into fixed-size buffers which, once filled, are sent to the front-end awg server at appropriate times using the remote procedure call (RPC) mechanism. All the buffering and timing is taken care of internally by the `SIStr` library functions, so you do not have to worry about this as long as you can calculate the waveform and pass it to `SIStrAppend` at a rate faster than real-time.

Other SIStr functions

`SIStrBlank( &sis, duration )`
Appends zeros for specified number of seconds, up to one day. (`duration` is a double.)

`SIStrFlush( &sis )`
Fills rest of current buffer with zeros, sends all local buffers to front end, and sleeps until after the last part of the waveform has actually been injected by the front end. `SIStrClose` calls this function, so normally you do not have to call it explicitly.

`SIStrAbort( &sis )`
Clears all local buffers (but cannot clear buffers which have already been sent to the front end). Also marks the stream as "aborted" so that any attempt to append more waveform data to it will fail. You must still be sure to call `SIStrClose` on this stream to clean up properly.

How to compile and link

Before you compile your program, you must be sure that the GDS shared-object libraries can be found by the compiler. At Hanford, do:

```
  setenv GDS_DIR /opt/CDS/d/gds/diag
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$GDS_DIR/lib
```

At Livingston, do:

```
  setenv GDS_DIR /opt/LLO/c/gds/diag
  setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$GDS_DIR/lib
```

To compile your program and link it to the SIStr library, do:

```
  cc myprogram.c SIStr.o -I$GDS_DIR/src/util -I$GDS_DIR/src/awg \
            -L$GDS_DIR/lib -lawg -ltestpoint -lrt -o myprogram
```

Important note: whenever you want to run your program, the `LD_LIBRARY_PATH` environment variable must be set to include `$GDS_DIR/lib` as described above.

Appendix: **SIStr.h**

```
#ifndef _SISTR_H
#define _SISTR_H

/*===========================================================================
SIStr.h - Header file for client API to stream a waveform to the awg front end
Written 28 Sep - 4 Oct 2001 by Peter Shawhan
See comments about compiling and linking at the beginning of the file SIStr.c
===========================================================================*/

#ifdef  __cplusplus
extern "C" {
#endif

/*------ Global variable ---------------------------------------------------*/
#ifdef _SISTR_LIBRARY
int SIStr_debug = 0;
#else
extern int SIStr_debug;
#endif

/*------ Compile-time parameters -------------------------------------------*/

#define SIStr_MAGICVAL 12345678
#define SIStr_MAXCHANNAMELENGTH 64
#define SIStr_MAXCHANLISTSIZE 32768

/* Target "lead time" for sending waveform data, in NANOseconds */
#define SIStr_LEADTIME 6000000000LL

/* Block size in "epoch" units (1/16 sec).  Allowed values are 1 through 16 */
#define SIStr_BLOCKSIZE 16

#define SIStr_MAXBUFSIZE 16384
#define SIStr_MAXBUFS 8

/* Max number of times to try sending same data */
#define SIStr_MAXTRIES 5

/*------ Status codes ------------------------------------------------------*/

#define SIStr_OK        0

/* Status codes returned from RPC calls */
#define SIStr_WFULL     1  /* Data accepted but front-end buffer is now full */
#define SIStr_WDUP      2  /* Data accepted but time was duplicated */
#define SIStr_WGAP      3  /* Data accepted but was not contiguous with prev */
#define SIStr_EBADSLOT -1  /* This awg slot is not set up for stream data */
#define SIStr_EBADDATA -2  /* Invalid data block or size */
#define SIStr_EPAST    -3  /* Time is already past */
#define SIStr_EFUTURE  -4  /* Time is unreasonably far in the future */
#define SIStr_ECONN    -5  /* RPC connection failed */

/* Error codes internal to SIStr */
#define SIStr_EBADARG   -101  /* Bad function argument */
#define SIStr_EBADSTR   -102  /* Stream is not correctly open for writing */
#define SIStr_EBADRATE  -103  /* Invalid sampling rate */
#define SIStr_EGAP      -104  /* Gap detected in stream data */
#define SIStr_EUNINIT   -105  /* Stream was not properly initialized */
```

```
#define SIStr_EMALLOC   -107  /* Error allocating memory */
#define SIStr_EOTHER    -108  /* Other error */
#define SIStr_EABORTED  -110  /* Attempted to append data to a stream which had
                            been aborted */
#define SIStr_EBADSTART -111  /* Attempted to set start time to an unreasonable
                            value */
#define SIStr_EINTERNAL -112  /* Unexpected internal error */
#define SIStr_EBUFSIZE  -113  /* Tried to create too large a data buffer */
#define SIStr_ETIMEOUT  -114  /* Timeout while trying to send data */
#define SIStr_ELISTERR  -115  /* Error retrieving channel list */
#define SIStr_ELISTNONE -116  /* Channel list is empty */
#define SIStr_ELISTSIZE -117  /* Channel list is too large */
#define SIStr_EBADCHAN  -118  /* Not a valid excitation channel */
#define SIStr_EDIFFRATE -119  /* Specified rate differs from actual rate */
#define SIStr_ESETSLOT  -120  /* Error setting up an awg slot for channel */
#define SIStr_ESETTP    -121  /* Error setting up test point */
#define SIStr_ESETCOMP  -122  /* Error setting up awgStream component */
#define SIStr_ECLRCOMP  -123  /* Error clearing awgStream component */
#define SIStr_ECLRTP    -124  /* Error clearing test point */
#define SIStr_ECLRSLOT  -125  /* Error freeing awg slot */
#define SIStr_ECLRBOTH  -126  /* Errors clearing test point AND freeing slot */

/*------ Structure definitions ----------------------------------------------*/

typedef struct tagSIStrBuf
{
  int gpstime;  /* Start time of this block (integer number of GPS seconds) */
  int epoch;    /* Start time of this block (epoch counter) */
  int iblock;   /* Block number in sequence, starting with 1 */
  int size;     /* Size of the data array */
  int ndata;    /* Number of values added to the data array so far */
  struct tagSIStrBuf *next;  /* Pointer to next buffer in linked list */
  float *data;
} SIStrBuf;

typedef struct tagSIStream
{
  int magic;    /* "Magic number" to allow sanity checks */
  char channel[SIStr_MAXCHANNAMELENGTH];   /* Channel name */
  int samprate; /* Sampling rate in Hz */
  double starttime; /* Start time of waveform (double-precision GPS seconds) */
  int slot;      /* awg slot number */
  int tp;        /* Flag to indicate whether test point has been set up */
  int comp;      /* Flag to indicate whether awgStream component is set up */
  int blocksize; /* Block size in "epoch" units (1/16 second time intervals)
                  e.g. a block 0.25 seconds long has a blocksize of 4.
                  Allowed values are 1 through 16. */
  int nblocks;  /* Total number of blocks buffered and/or sent so far */
  int curgps;   /* GPS time (integer number of seconds) of current buffer
                 (or next buffer to be created) */
  int curepoch; /* Epoch counter of current/next buffer */

  int sentgps;    /* GPS time (integer seconds) and epoch of last buffer */
  int sentepoch;  /*  sent to the front end */

  int nbufs;    /* Current number of buffers resident in memory */
  SIStrBuf *curbuf;   /* Pointer to current buffer.  A buffer is not created
                     until there is some data to put into it; therefore, it
                     is possible for a stream to have NO current buffer
                     (in which case curbuf==NULL) at various times. */
  SIStrBuf *firstbuf; /* Pointer to first buffer in linked list */
  SIStrBuf *lastbuf;  /* Pointer to last buffer in linked list.  This is the
                     current buffer if there is one; if not, it is the last
                     buffer which was filled. */
  long long lastsend; /* Time that last data was sent to front end, in
                     GPS nanoseconds */
  long long minwait;  /* Enforced minimum on the time interval between RPC
                     calls to transfer data to the front end, in
                      nanoseconds */
  int aborted;        /* Flag to indicate if stream has been aborted */
} SIStream;

/*------ Function prototypes -------------------------------------------*/

int SIStrOpen( SIStream *sis, char *channel, int samprate, double starttime );
int SIStrAppend( SIStream *sis, float newdata[], int ndata, float scale );
int SIStrBlank( SIStream *sis, double duration );
int SIStrFlush( SIStream *sis );
int SIStrClose( SIStream *sis );
int SIStrAbort( SIStream *sis );
char * SIStrErrorMsg( int status );

#ifdef  __cplusplus
}
#endif

#endif /* _SISTR_H */
```

Appendix: Source Code for the awgstream Utility

```
/*============================================================================
awgstream - Command-line client to stream data from a file to the awg front end
             using the SIStr interface
Written Oct 2001 by Peter Shawhan

To compile:
  cc awgstream.c SIStr.o -I$GDS_DIR/src/util -I$GDS_DIR/src/awg \
            -L$GDS_DIR/lib -lawg -ltestpoint -lrt -o awgstream
where GDS_DIR on red      = /opt/CDS/d/gds/diag

To run: $GDS_DIR/lib must be in your LD_LIBRARY_PATH
==========================================================================*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "SIStr.h"


/*==========================================================================*/
void PrintUsage( void )
{
  fprintf( stderr,
"Usage: awgstream <channel> <rate> <file> [<scale> [<gpstime>]] [-d]\n" );
  fprintf( stderr,
"  <channel> is case-sensitive and must be a real excitation channel\n" );
  fprintf( stderr,
"  <rate> is in Hz and must match the excitation channel's true rate\n" );
  fprintf( stderr,
"<file> is the file of waveform data, which must be a FORMATTED ascii file\n");
  fprintf( stderr,
"   with the values separated by whitespace (e.g. newlines and/or spaces).\n");
  fprintf( stderr,
"      Any conversion error causes the rest of the file to be skipped.\n" );
  fprintf( stderr,
"      If <file> is '-' (i.e. a dash), data is read from standard input.\n");
  fprintf( stderr,
"  <scale> is an optional scale factor to apply to the data in the file.\n");
  fprintf( stderr,
"  <gpstime> is the time to start injecting the waveform.  It can take any\n");
  fprintf( stderr,
"      real value, not just an integer.  It must occur during the 24-hour\n" );
  fprintf( stderr,
"      period following the execution of the awgstream command.\n" );
  fprintf( stderr,
"      If omitted, the waveform injection will start in about %d seconds.\n",
   (int) (SIStr_LEADTIME/1000000000LL) + 4 );
  fprintf( stderr,
"The '-d' option causes debugging information to be printed to the screen.\n");
  fprintf( stderr,
"      Specify '-d' twice for extra information.\n" );
  return;
}
```

```
/*===========================================================================*/
int main( int argc, char **argv )
{
  char *arg;
  int iarg;
  int nposarg = 0;
  char *channel = NULL;
  int samprate = 0;
  char *filename = NULL;
  float scale = 1.0;
  double starttime = 0.0;
  FILE *file;
  SIStream sis;
  int status;
  float val;

  /*------ Beginning of code ------*/

  if ( argc <= 1 ) {
    PrintUsage(); return 0;
  }

  /*-- Parse command-line arguments --*/
  for ( iarg=1; iarg<argc; iarg++ ) {
    arg = argv[iarg];
    if ( strlen(arg) == 0 ) { continue; }

    /*-- See whether this introduces an option --*/
    if ( arg[0] == '-' && arg[1] != '\0' ) {
      /*-- The only valid option is "-d" --*/
      if ( strcmp(arg,"-d") == 0 ) {
       SIStr_debug++;
      } else {
       fprintf( stderr, "Error: Invalid option %s\n", arg );
       PrintUsage(); return 1;
      }

    } else {
      /*-- This is a positional argument --*/

      nposarg++;
      switch (nposarg) {
      case 1:
       channel = arg;
       break;
      case 2:
       samprate = atol( arg );
       if ( samprate < 1 || samprate > 16384 ) {
         fprintf( stderr, "Error: Invalid sampling rate\n" );
         PrintUsage(); return 1;
       }
       break;
      case 3:
       filename = arg;
       break;
      case 4:
       scale = atof( arg );
       /*
       if ( scale == 0.0 || scale > 600000000.0 ) {
         fprintf( stderr, "Error: Invalid scale factor\n" );
         PrintUsage(); return 1;
       }
       */
       break;
      case 5:
       starttime = atof( arg );
       if ( starttime < 600000000.0 || starttime > 1800000000.0 ) {
         fprintf( stderr, "Error: Invalid start time\n" );
         PrintUsage(); return 1;
       }
       break;
      default:
       fprintf( stderr, "Error: Too many arguments\n" );
       PrintUsage(); return 1;
      }

    }

  }  /* End loop over command-line arguments */

  /*-- Make sure the required arguments were present --*/
  if ( channel == NULL ) {
    fprintf( stderr, "Error: Channel was not specified\n" );
  }
  if ( samprate == 0 ) {
    fprintf( stderr, "Error: Sampling rate was not specified\n" );
  }
  if ( filename == NULL ) {
    fprintf( stderr, "Error: File was not specified\n" );
  }
  if ( channel == NULL || samprate == 0 || filename == NULL ) {
    PrintUsage(); return 1;
  }

  /*------------------------------------*/
  /* Open the file for reading (if not reading from stdin) */
  if ( strcmp(filename,"-") != 0 ) {
    file = fopen( filename, "r" );
    if ( file == NULL ) {
      /* An error occurred */
      fprintf( stderr, "Error while opening %s for reading: ", filename );
      perror( NULL );
      return -2;
    }
  } else {
    file = stdin;
  }

  /*------------------------------------*/
  /* Open the Signal Injection Stream */
  status = SIStrOpen( &sis, channel, samprate, starttime );
  if ( SIStr_debug ) { printf( "SIStrOpen returned %d\n", status ); }
  if ( status != SIStr_OK ) {
    fprintf( stderr,
             "Error while opening SIStream: %s\n", SIStrErrorMsg(status) );
    return 2;
  }

  /*------------------------------------*/
  /* Read data from input file and send it */
  while ( fscanf(file,"%f",&val) == 1 ) {
    status = SIStrAppend( &sis, &val, 1, scale );
    if ( SIStr_debug >= 2 ) { printf( "SIStrAppend returned %d\n", status ); }
    if ( status != SIStr_OK ) {
      fprintf( stderr,
              "Error while adding data to stream: %s\n",
              SIStrErrorMsg(status) );
      break;
    }
  }

  /* Close the input file (unless it is stdin) */
  if ( file != stdin ) {
    fclose( file );
  }

  /*-----------------------------------*/
  /* Close the stream */
  status = SIStrClose( &sis );
  if ( SIStr_debug ) { printf( "SIStrClose returned %d\n", status ); }
  if ( status != SIStr_OK ) {
    fprintf( stderr,
            "Error while closing SIStream: %s\n", SIStrErrorMsg(status) );
    return 2;
  }

  return 0;
}
```