PSDK Programmers Ref.

Products:

-SG100
-SG100 EVO-USB
-SG100 EVO-USB CERT

WINDOWS VERSION

PRE-BUILT DRIVERS

Contents

General About the SG100 Security Generator

The SG100 is designed to be a reliable and versatile device for cryptographic and statistical applications. Care and effort have gone into making the SG100 performing well under all possible operating conditions. In the unlikely situation that the device fails, thorough statistical and electrical testing will detect the situation, and report an error code.

This manual covers the general properties of the SG100 Security Generator from the system programmer's point of view. Most of the automatic checks that have been built into the SG100 drivers is presented and explained. Note that the cryptologic and statistical technology necessary for writing a device driver for a different platform is not included in this manual.

The current version of the SG100 driver support multiple SG100 generators. You will have to provide a separate serial port for each SG100 generator.

The SG100 Hardware

The SG100 hardware is basically a noise generating process and an amplifier. The noise originates from a Zeener diode. After amplification a stream of noise is sent to the UART of the computer. The UART samples the noise stream into 8-bit bytes which is put into a buffer.

More information about the SG100 hardware can be found on Protego Information's Webb-server, including all kinds of measurements and tests.

See **`http://www.protego.se/`**

The Noise Driver Thread

The calling application must provide an execution thread for the SG100 hardware. Single thread applications must have this thread externally provided. If two applications are using the SG100 at the same time only one of the applications may provide a thread for the hardware, or the thread must be externally provided.

The noise driver thread opens the SG100 serial port and read the noise stream from the hardware. You can have only one noise driver thread for each serial port. If you try to open a second driver thread for an already open serial port the SG100 API will report an error-code.

You can not open the serial port directly. The SG100 driver is an integrated part of the SG100.

Inside the driver the noise is tested statistically. As the input is completely unprocessed there is no difficulty to compute a number representing input quality. We have chosen to compute the information speed of the input, calculated on 8-bit-bytes, with a sample size of 32,000 bytes. Other tests are possible.

```
Totalbytes = 32,000;
P[i] = Frequency[ i]/Totalbytes;
Inf_ = ∑( -P[i]log(P[i])) for all 256 bytes i;
Inf_Rate = 100.00%*Inf_ / 8.00 / log( 2.00);
```

If we have an input with low quality noise the situation is handled by reading noise twice. The two strings are mixed together using a function independent to other processing in the system. As the noise stream can have a maximum of 100% information rate this trick works for information rates down to 50%.

More specifically is the noise read once if the input information rate 96%≤Inf_Rate≤100%, two times if 93%≤Inf_Rate≤96%, and tree times if 70%≤Inf_Rate≤93%. For an information rate less than some limit, 70%, the device driver sets an error condition.

The driver also checks for low input voltage, by applying a special test. The SG100 is powered from two signal pins of the serial port, and there can be a situation where insufficient power is available.

The simplest error is that the SG100 has been disconnected. In that case the driver releases a watchdog semaphore, but takes no error action. If the application checks when the watchdog semaphore is released and notifies the user, the user may connect the SG100 again. Processing is then resumed. No message of any kind is forwarded to processes reading noise.

Example code for starting the noise driver thread is given in Appendix A. This code is also provided in compiled form (console application):

```
DRIVER COM1 baudrate
(Note: This program do not print progress information. In the case of an
error, the error-code will be printed.)
```

There is also example code on the source distribution media.

You should note that, in the current version of the software, the library ISAF_N1.DLL is loaded using run-time dynamic linking by the driver DLL and that this affect applications were load-time dynamic linking is asked for. If you wish to load the SG100 libraries into your EXE, using load-time dynamic linking, contact us and we will fix it.

Buffers and Noise Processing

After the noise processing, the noise is put in an output buffer. In the event that the SG100 driver is idle, you may read a continuous string of a length of up to 64,000 bytes without waiting for the SG100 hardware to produce the noise. This buffer has been installed as some applications may need noise very quickly.

We have seen that action is taken to guarantee a high input information rate.

In the ISAF_N1 driver the noise is processed to simulate 100% noise quality. Different types of cryptographic functions may be considered for this processing, but as we know that the input has a high information rate, a reasonable level of computing has been selected. Note that the minimum input information rate is 96%, so simulating the last few percentages is not very difficult.

The streamcipher of the SG100 contains shift registers with maximum length feedback. The length of the longest register is long enough to make exhaustion of the period computationally infeasible. As maximum length shift registers have good (mathematically proven) statistical properties, it is not possible to find any simple (non-cryptographic) statistical property in the SG100 output. The output from the shift registers is concealed by a non-linear function.

A part of the key to the streamcipher is reseeded approximately every three minutes (every ten minutes for an idle device). The noise necessary for this reseeding is consumed, and is not reused by the API. By updating the seed of the streamcipher at enough short intervals we can enforce that even the most complex (cryptographic) attacks also will fail.

It is reasonably to conclude that the (maximum) 4% deficiency of input information rate can not be detected or exploited by any means.

SG100 Timers

When you first start up the SG100 generator & drivers you will find that the output from the ISAF_N1 driver is blocked for several seconds. Under this period of time no noise can be read form the SG100 system (the ISAF_N1.DLL).

When the SG100 system first starts up the noise driver thread it is reading noise from the hardware and are updating the noise buffer. The noise quality improves until it reach a maximum. This may occur in only 2-3 iterations, maybe it takes a little longer.

It is essential that the output buffer is thoroughly randomized before any noise is read from the ISAF_N1 driver. Some applications, such as cryptographic applications, need very high quality noise. Generating a cryptographic key too quickly is a mistake that could have devastating security effects. The default start-up delay is minimum 25 seconds or minimum eight iterations. For demanding applications it is recommended to signal the driver by issuing a call to ”Release_UpdateLock” that sets the update time to 70 seconds and releases a possibly pending driver thread, refreshing the buffer.

The noise reading/testing is restarted every ten minutes even if no calls are made to the SG100 noise system. This assures that the SG100 driver always returns fresh random bits.

The noise driver makes use of timers to control the way the noise is processed. In an initial period (25 minutes) the noise is read more than once as an extra precaution. This time period is restarted if the SG100 is reported to be disconnected.

The timers also controls when the test for low supply power should be applied. The driver also checks for low power supply immediate before the SG100 is shut down to the low-power state.

When the SG100 driver runs out of random bits, and the driver need more random input from the SG100 hardware, the driver returns noise three times following each other. Some applications maybe needs only little noise. This method lets the SG100 driver to return a completely random string (exact 100% information rate) with light load.

If the application is in need of very much noise we will have to depend on computations in the SG100 system for hiding any statistical deficiency.

API Call for Reading Data

To read the noise from the SG100 Security Generator you open ISAF_N1.DLL and initializes the driver. This works if the noise reading thread is properly connected to ISAF_N1.DLL already. You are recommended to use the same DLL file, as ISAF_N1 has some shared memory.

If you want more noise than available in the noise buffer the call will terminate when the number of bytes requested has been read from the hardware.

The call (* getnoise)(...) and (* updatelock)(...) is intended for multi-threaded applications, and can be called simultaneously from several applications or threads. Example code for connecting to the ISAF_N1.DLL may be found in Appendix B. This code is included in the compiled program (console application).

```
NOISE  Output_file_name   Number_of_bytes
```

and

```
LOTTO Total_balls  Balls_to_draw  Extraballs_to_draw  Iterations
```

Integrating SG100 in Applications

Some applications may need large amounts of noise. As the SG100 is too slow, except for lottery and cryptographic applications, a much faster way of generating noise is asked for. We must then use a pseudo random number generator (P-RNG).

You integrate a P-RNG with the SG100 by generating the ”initial seed” for the P-RNG with the SG100 and then take the noise from the P-RNG.

Demanding applications could use a combination of two different P-RNGs to enhance the output. You should reseed your P-RNG at regular and irregular intervals, i.e. when noise is available from the SG100.

Some demanding applications can detect and be influenced by deficiencies in a P-RNG. By reseeding the P-RNG at sufficiently short intervals, any deficiencies in your P-RNG will ”move around” and the influence on your application will diminish in the long run.

We do not include any API for this functionality, such as accessibility to the stream-cipher, as it could circumvent our communication with our customers, maybe leading some customers to believe that the SG100 is a pseudo random number generator.

SG100 Test Programs

The test programs are console applications used during testing and evaluation of previous (beta) versions of the SG100 Security Generator hardware. On the SG100 distribution media there is a file \TEST\TESTING.TXT that has example output from all test programs.

It is possible, with the test programs, to read the hardware directly, write the output to a file, and then apply customer-written hardware tests. The DOWNLOAD program is intended **for test purposes only**. Do not use DOWNLOAD in your application!

```
DOWNLOAD COM1 Baudrate Filename Number_of_bytes
```

The information rate, a simple and useful measure of output quality, can be measured with the

```
BYTESTAT COM1 Baudrate
```

program. A more accurate value (16 bit statistics) is obtained with

```
WORDSTAT COM1 Baudrate
```

Please note that this test must be run for a long time—about 50 000 000 words...

If you wish to investigate which baudrates that can be used call

```
BAUD COM1
```

that do a simple test for several baudrates.

The simplest test is to look at the random output string. Run

```
BITPRINT COM1 Baudrate
```

and take a guess of the output!

Sg100 API Calls

ulong Test_Port(char *Port, DWORD Baudrate, long *Testresult);

Returns a code if requested baudrate is accepted by the SG100 system. Also calculates information speed of the output for the baudrate. Intended use: To optimise the SG100 baudrate to end-customer's computer/COM-port.

```
void Execute_Noise_Loop( char *Port, DWORD Baudrate);
```

Executes the noise DLL, and connects to ISAF_N1.DLL.

```
void End_Noise_Loop();
```

A call here shuts the SG100 system off, and sets an error condition to all pending calls.

```
void DLL_Setup( ulong *Error_Code, ulong *Exception);
```

Call here to open ISAF_N1.DLL

```
void Release_UpdateLock( ulong *Error_Code, ulong *Exception);
```

A call here tells the SG100 system to refresh buffers. The refreshing takes place asyncronuously, at some later time.

```
void Get_Noise( uchar *Byte_PTR, ulong Bytes, ulong *Error_Code, ulong
*Exception);
```

The call returns a string, of specified length, of random bytes, to memory buffer at address Byte_PTR.

```
ulong SG100_Random( ulong Select_Range, ulong *Error_Code, ulong
*Exception);
```

This call returns a random integer in the range [0..(Select_Range-1)] (inclusive)for a specified range Select_Range. The distribution is flat, and the conversion probabilities is exact. Select_Range may be any ulong unsigned number, except zero.

List of Errors

If a call succeeds it returns with Error_Code of 0 and Exception 0. If this is not the case any code is to be expected as an error code.

If some problem has been detected inside one of the SG100 drivers it will return with a non-zero Error_Code. The Exception code may then be decoded:

```
#define REG_EXCEPTION       0x09000000
ulong Major_err_code = 0;
ulong Minor_err_code = 0;
if ( Exception >= REG_EXCEPTION && Exception < REG_EXCEPTION + 0x80000L)
{
   Major_err_code  = Exception - REG_EXCEPTION;
   Minor_err_code  = Major_err_code & 255L;
   Major_err_code  = ( Major_err_code >> 8 ) & 511;
}
```

The value Major_err_code is the type of error that has occurred and Minor_err_code is a serial count within each error group. Together the numbers specify exactly what is wrong.

List of major error codes that apply to the SG100 drivers.

```
#define XMSG_MEMORY_MANAGEMENT_ERROR     0x0000007DL
```

Serious memory problem. Reboot your computer.

```
#define XMSG_CANNOT_ALLOCATE_MEMORY      0x0000007EL
```

Out of memory. This is serious, reboot your computer.

```
#define XMSG_CANNOT_ALLOCATE_INTERNAL_MEMORY  0x0000007FL
```

Out of memory. This is serious, reboot your computer.

```
#define XMSG_ERROR_IN_INTERNAL_MEMORY    0x00000080L
```

Serious memory problem. Reboot your computer.

```
#define XMSG_CPU_MEMORY_TRAP             0x00000081L
```

Something unexpected has occurred.

```
#define XMSG_SYNC_SYSTEM_NOT_OPERATING   0x00000082L
```

The synchronisation system is not working. You must close the SG100 system.

```
#define XMSG_INSERT_NOISEBLOCK_TOO_SMALL 0x00000092L
```

Driver should not insert a noise block less then five words. (This occurs only if the DRV_DLL.DLL is in error.)

```
#define XMSG_INSERT_CIPHERTEXTBLOCK_TOO_SMALL 0x00000093L
```

The ciphertext block is to small. Call with a larger ciphertext block. (The call generating this error is not documented here.)

```
#define XMSG_CANNOT_OPEN_IO_PORT         0x00000094L
```

Port is most probably in use. The port could also be non-existing. Check baudrate.

```
#define XMSG_CANNOT_READ_IO_PORT_STATUS  0x00000095L
```

Get status call failed. Most probably an invalid port specification.

```
#define XMSG_CANNOT_SET_IO_PORT_STATUS   0x00000096L
```

Can not alter the port settings for the port. Check if the port has the proper driver.

```
#define XMSG_IO_PORT_ERROR               0x00000097L
```

General problems with the serial port.

```
#define XMSG_BAD_NOISE_QUALITY           0x00000098L
```

There is a problem with the SG100 hardware. Please erase any strings of noise generated immediately before this error. Check the SG100 hardware on a different computer, using the test programs. Check the serial port hardware using a back-loop connector and service software.

```
#define XMSG_NOISE_BUFFER_NOT_INITIALIZED 0x00000099L
```

The buffer is not initialised. The calls are coming in the wrong order. Check that the driver thread is running before making calls to ISAF_N1.DLL

```
#define XMSG_PROGRAM_BUG                 0x0000009AL
```

Intentional error reported when an unforeseen condition occurs. Report to us in e-mail.

```
#define XMSG_CANNOT_OPEN_DLL            0x0000009EL
```

The driver thread cannot open the ISAF_N1.DLL using the LoadLibrary API call. Check to see if the ISAF_N1.DLL file is present in the DRV_DLL.DLL path search.

```
#define XMSG_CANNOT_GET_DLL_PROCESS_ADRESS 0x0000009FL
```

Can not get address of API call.

```
#define XMSG_GENERAL_NOISE_ERROR        0x000000A1L
```

Returned when some non-specific problem has occurred as calling API SG100_Random with a Select range of zero.

```
#define XMSG_CPU_ERROR_TRAP             0x000000A3L
```

Some serious problem that cannot be identified.

```
#define XMSG_CPU_STACK_OVERFLOW         0x000000A4L
```

Some serious problem that cannot be identified.

If the small program, above, fails, check if Exception matches any of the following system errors:

STATUS_WAIT_0	0x00000000
STATUS_ABANDONED_WAIT_0	0x00000080
STATUS_USER_APC	0x000000C0
STATUS_TIMEOUT	0x00000102
STATUS_PENDING	0x00000103
STATUS_GUARD_PAGE_VIOLATION	0x80000001
STATUS_DATATYPE_MISALIGNMENT	0x80000002
STATUS_BREAKPOINT	0x80000003
STATUS_SINGLE_STEP	0x80000004
STATUS_ACCESS_VIOLATION	0xC0000005
STATUS_IN_PAGE_ERROR	0xC0000006
STATUS_NO_MEMORY	0xC0000017
STATUS_ILLEGAL_INSTRUCTION	0xC000001D
STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
STATUS_INVALID_DISPOSITION	0xC0000026
STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
STATUS_FLOAT_INEXACT_RESULT	0xC000008F
STATUS_FLOAT_INVALID_OPERATION	0xC0000090
STATUS_FLOAT_OVERFLOW	0xC0000091
STATUS_FLOAT_STACK_CHECK	0xC0000092
STATUS_FLOAT_UNDERFLOW	0xC0000093
STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094

STATUS_INTEGER_OVERFLOW	0xC0000095
STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
STATUS_STACK_OVERFLOW	0xC00000FD
STATUS_CONTROL_C_EXIT	0xC000013A

Appendix A

A short description of program code to start a noise reading thread follows. Declare variables to open the SG100 port and the SG100 DLL_DRV library.

```
#include "DLL_DRV.H"
ulong Exception   = 0;
HINSTANCE hLibrary = NULL;    // handle to the library
Execute_Noise_Loop_P    Loop= NULL;
```

The `"Execute_Noise_Loop_P"` is a type for a pointer to a function taking an appropriate set of parameters. This enables compile-time typechecking for C++ compilation. (It is, of course, of no use if you don't use a C++ compiler.) We now open the driver DLL and checks for a valid handle:

```
__try
{
   hLibrary = LoadLibrary( "DLL_DRV" );
   if ( hLibrary != NULL)
   {
      Loop = ( Execute_Noise_Loop_P )GetProcAddress( hLibrary,
                                                "Execute_Noise_Loop");
      char *Portname = "COM1";   // provide name in char * for
      DWORD Baudrate = 57600;    // acceptable rate <= 100,000
      if ( Loop != NULL)
      {
         (* Loop)( Portname, Baudrate);
      }
      FreeLibrary( hLibrary );
      hLibrary = NULL;
   }
}
__except( EXCEPTION_EXECUTE_HANDLER  )
{
   Exception    = GetExceptionCode();
}
```

Note that the call `(* Loop)( Portname, Baudrate)` do not terminate until the host application close or an explicit call to ”End_Noise_Loop” is executed.

To open the watchdog semaphore:

```
#include "DRV_DLL.H"
HANDLE WatchDog = NULL;
DWORD Retval;
/********
SEMAPHORE_ALL_ACCESS is defined in a system *.h file:
#define STANDARD_RIGHTS_REQUIRED         (0x000F0000L)
#define SYNCHRONIZE                      (0x00100000L)
#define SEMAPHORE_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED|SYNCHRONIZE|0x3)
*********
WATCHDOG is defined in Isaf_N1.h:
#define WATCHDOG          __TEXT("WatchDog_Isaf001")
*******/
WatchDog = OpenSemaphore( SEMAPHORE_ALL_ACCESS, TRUE, WATCHDOG);

if ( WatchDog == NULL )
{
   SECURITY_ATTRIBUTES sa;      // security privileges for SEMAPHORES

   // fill out a SECURITY_ATTRIBUTES structure so handles can be inherited

   sa. nLength = sizeof( SECURITY_ATTRIBUTES);   // structure size
   sa. lpSecurityDescriptor = NULL;              // default descriptor
   sa. bInheritHandle = TRUE;                    // inheritable

   /* create the Semaphore */
   WatchDog = CreateSemaphore( &sa, 0, 100, WATCHDOG) ;
}
if ( WatchDog == NULL )
{
   // Error condition.
}
Retval = WaitForSingleObject( WatchDog, 0);
Retval = WaitForSingleObject( WatchDog, 0);
```

The application periodically checks the WatchDog or a separate thread is spawn to check the SG100:

```
// wait until disconnected
Retval = WaitForSingleObject( WatchDog, INFINITE);
// when returns the SG100 is disconnected

// check if disconnected.
Retval = WaitForSingleObject( WatchDog, 0);
if ( Retval == WAIT_OBJECT_0)
{
   // when Retval == WAIT_OBJECT_0 the SG100 is disconnected
}
```

Additional API calls and information may be available in the header files.

Appendix B

Example code, for an application that want to read noise, to connect to ISAF_N1:

```
#include "ISAF_N1.H"
HINSTANCE MMF_Library = NULL;      // handle to the library
ulong Error_Code;
ulong Exception;
Get_Noise_P            getnoise;
DLL_Setup_P            setup;
Release_UpdateLock_P   updatelock;

MMF_Library = LoadLibrary( "ISAF_N1");

if ( MMF_Library == NULL)
{
   // Error, abort
}
getnoise   = ( Get_Noise_P )GetProcAddress( MMF_Library, "Get_Noise" );
setup      = ( DLL_Setup_P )GetProcAddress( MMF_Library, "DLL_Setup" );
updatelock = ( Release_UpdateLock_P )GetProcAddress( MMF_Library,
                                                "Release_UpdateLock" );

if ( getnoise == NULL || setup == NULL || updatelock == NULL )
{
   // Error
}
// Setup the DLL
(* setup)( &Error_Code, &Exception);

if ( Exception != 0 || Error_Code != 0 )
{
   // Error
}
// For demanding applications release pending driver thread
// and read in fresh noise:
(* updatelock)( &Error_Code, &Exception);
if ( Exception != 0 || Error_Code != 0 )
{
   // Error
}

uchar Noise_Buffer[ size];
ulong Bytes_to_read = number;

(* getnoise)(  &Noise_Buffer, Bytes_to_read, &Error_Code, &Exception);
if ( Exception != 0 || Error_Code != 0 )
{
   // Error
}
else
{
   // Bytes_to_read bytes of noise written to
   // memory address ( unsigned char *)( &Noise_Buffer)
}
FreeLibrary( MMF_Library );
MMF_Library = NULL;
```