SSM (Safe Strings and Memory)

Library of Safe Strings and Memory buffers for the C language
Version 1.2, 20 November 2015

Thierry Lelégard

This manual is for SSM (version 1.2, 20 November 2015), a Safe Strings and Memory buffers library for C.

Table of Contents

1 Overview

The SSM library provides safe functions for handling strings and memory buffers in C.

This software is distributed under the terms of the GNU Lesser General Public License (LGPL) version 2.1.

Its source code and documentation are available online:

- Home page: http://ssmlibrary.sourceforge.net/
- Project page: http://sourceforge.net/projects/ssmlibrary/

1.1 The classical C strings library

The problems of the unsafe C string functions from the C-library are well-known. The null-terminated C string representation and the infamous `strcpy()` and `strcat()` functions date back from the 70's, in the early days of the UNIX operating system and the original "Kernighan & Ritchie" C language.

Several functions write a null-terminated variable-length string into a destination buffer which is known by its base address only. Since the function has no clue about the buffer size, it writes past the end of the output buffer if the input data are larger than the buffer size; overwriting memory areas which may contain valuable data. In this context, the concept of *buffer* as a bounded memory area is purely in the developer's mind since the language and its library work at a very low abstraction level, similar to assembly code.

The impacts of such buffer overflows include:

- **Reliability**: Data corruption.
- **Robustness**: Application erratic behaviour or crash (when the overwritten data contained pointers).
- **Security**: Malicious code injection or *ret-to-libc* attack (when the overwritten data include the stack frame and return pointer).

Using the old unsafe C strings library should be banned from all modern code, especially when security matters.

To mitigate the intrinsic buffer overflow issue with the old C library functions, variants were introduced with an additional parameter indicating the maximum size of the buffer. By convention, most of these functions have an `'n'` in the middle of their name such as `strncpy()` or `strncat()`. If they are used correctly, these functions never write past the end of the user-supplied buffer. However, using them correctly is quite hard since these functions are inconsistent with each other:

- The semantic of the size parameter differs:
  - Sometimes it includes the final null character and sometimes it does not. In the later case, the size parameter must be the buffer size minus one.
- When the data to write are larger than the buffer, no extra data is overwritten. But the exact behavior depends on the function:
  - Overflow detection: Can we detect that a larger buffer would have been required? Some functions return a specific value indicating that the buffer was too short. With some other functions, the returned data is silently truncated, generating an incorrect result without any way to detect it.
  - Truncated or undefined content: Is the string in the returned buffer truncated or undefined (ie. maybe not written at all)?
  - Null-terminated or not: When the string is too large, some functions write `n` characters without a null terminator while other functions write `n-1` characters followed by a null terminator.

As a consequence, the `'n'` functions are quite hard to use correctly. Each function has a specific behavior and must be surrounded by distinct user-written checks every time. In other words, using the `'n'` functions is very error-prone and they should be banned as well.

With the C11 language standard, an optional set of `'_s'` functions was introduced. Their API is slightly better defined and more consistent than for the `'n'` functions. See Section 8.1 [C11 Annex K Overview], page 41 for more details.

More generally, the `'n'` and `'_s'` functions have the same fundamental flaw as the classical string functions, their abstraction level is very low. The concepts of *string* or *buffer* do not exist. These functions manipulate addresses and sizes independently. Providing the correct size at the right time is entirely left to the application developer whose work is more complex and consequently more error-prone.

Other alternative C strings libraries exist such as the Better String Library[1], the Safe C Library[2] or the Managed String Library[3]. An online article[4] lists a comparative description of a large number of string libraries, although not limited to the C language.

Some of these libraries provide a higher level of abstraction to manipulate strings. But no general consensus exists on a safe alternative to the old C string library.

1.2 The SSM library

The SSM library is a safe and reliable alternative the old C string library.

As mentioned in the previous section, several alternatives already exist for the unsafe C string library. So, why creating a new library?

The SSM library presents a unique combination of characteristics that should be considered as mandatory for environments requiring both safety and security. To our knownledge, no other library meets all these requirements altogether.

These requirements are:

- Handle *strings* and raw *memory buffers*.
- For both types of object, provide a *static* alternative (fixed size) and a *dynamic* alternative (unbounded size using dynamic memory allocation).
- No dependency on other external libraries.
- Portable and light-weight. Can be used in various environments:
  - Application code.
  - Linux kernel modules.
  - Embedded systems.
- Proven and robust code for safety and security:
  - Simple and straightforward code.
  - Complete unitary tests based on `CUnit`.
  - Automatic non-regression tests based on unitary tests.
  - Automatic static code analysis using `cppcheck` and `flawfinder` (other analyzers may be added).
  - Automatic code coverage analysis using `gcov`.
  - Achieve 100% code coverage in analysis.
- Complete reference documentation.

[1] http://bstring.sourceforge.net
[2] http://sourceforge.net/projects/safeclib/
[3] https://www.securecoding.cert.org/confluence/display/seccode/Managed+String+Library
[4] http://www.and.org/vstr/comparison

Note that using this library in C++ is discouraged since the C++ Standard Template Library (STL) contains much better classes, namely `std::string` and `std::vector`. This library is only a pitiful attempt to fix one of the worst achievements in software engineering, the C string library...

1.3 SSM objects

The SSM library defines two types of objects: *strings* and *memory buffers*. Each type exists in two flavors: *static* and *dynamic*.

A memory buffer contains raw binary data of any type. Its size is expressed in bytes.

A string is supposed to contains characters only (type `char`). For compatibility with classical C-strings, the internal representation of a string is always followed by a zero byte. This terminating zero byte, however, is not considered as a part of the string (it is not included in the string *length* for instance).

A static string or memory buffer is declared with a static maximum size, just like a regular C array. It can contain up to that number of characters or bytes. A static string or memory object is a fast low-level type without any sort of memory allocation. The storage for the characters or bytes is reserved within the variable, like C arrays.

Whenever an operation requires more memory than reserved in a static string or memory buffer, the result is safely truncated. In the case of a static string, a truncated result is always null-terminated. The truncation is always reported in a returned status.

A dynamic string or memory buffer uses dynamic memory allocation to store the content. One advantage is that truncation never occurs. One drawback is the overhead of memory allocation.

See Section 2.4 [Dynamic memory management], page 10 for more details on the implementation of memory allocation.

But the worst drawback of dynamic objects is the risk of memory leak. All dynamic objects must be explicitely freed by the user. Failing to do so results in a memory leak. This is particularly dangerous when an *early return path* is taken, bypassing the cleanup code at the end of a block. Note that the main reason for this risk is that the C language has no concept of *destructor* as in C++.

Another risk of dynamic objects is the memory allocation failure. Whenever a memory allocation failed within an SSM function, the function returns the status value `SSM_NOMEMORY`. In that case, the dynamic object which could not be reallocated is not modified. So, the result is still a *safe* object. But the application is *reliable* only if the user code properly checks the returned status value.

1.4 Using the SSM library

All declaractions are contained in the single header files `ssm.h`.

All public identifiers which are exported by the SSM library start with the prefix `ssm_` (functions) or `SSM_` (constants).

All code is contained into one single static library. To allow fine-grained selective linking in constrained environments, each function is implemented in a separate object file.

There is also a shared library version of the SSM library. Due to the usage restrictions of the GNU Lesser General Public License, proprietary applications must link against the shared library version. They are not allowed to link against the static library, unless the application is provided in an object form which can be relinked using another version of the SSM library.

On UNIX, Linux and Cygwin environments, the static library is named `libssm.a` and the shared library is named `libssm.so`.

On Windows systems with Microsoft Visual C++, the static library is named `ssmlib.lib`.

On Windows systems with Microsoft Visual C++, the dynamic library is named `ssmdll.dll` and the corresponding symbol library is named `ssmdll.lib` (this is the file which is used by the linker).

Important: When linking against the DLL `ssmdll.dll`, a special symbol must be defined during the compilation. Define the symbol `SSM_USE_DLL` before including `ssm.h`, either as a project option (preferred solution) or in the source file as follow:

```
#define SSM_USE_DLL
#include "ssm.h"
```

1.5 Thread-safety

Unless explicitely specified otherwise, all functions in the SSM library are thread-safe as long as they work on distinct objects.

If the same object (static string, dynamic string, static buffer, dynamic buffer) is concurrently accessed by multiple threads, some explicit exclusive access mechanism must be implemented at the application code level before invoking the SSM library on this object.

1.6 Dynamic memory allocation

For dynamic strings and dynamic memory buffers, the default memory allocation and deallocation functions are the standard `malloc()` and `free()`. However, they may not be ideal in all environments, especially in constrained systems. To cope with that, the user application may specify alternative memory management functions using `ssm_set_memory_management()`. This function shall be called before any usage of dynamic strings or buffers.

Another usage of the replacement of the memory allocation functions is the handling of memory allocation failures. In many applications, a memory allocation failure is fatal and cannot be recovered. It could be useful to provide an application-specific memory allocation routine which reports the allocation failure and aborts the application.

Note that there is no replacement for `realloc()`. No such function is used. Although `realloc()` could bring some performance improvement over a sequence of `malloc()`, copy and `free()`, the result is unpredictable in case of memory allocation failure; there is no guarantee that the previous memory area was preserved. To keep the library safe and predictable, it does not use `realloc()`.

In environments where the standard `malloc()` and `free()` are not available, such as the Linux kernel, there is no default memory management functions. The user shall invoke `ssm_set_memory_management()` before using dynamic strings or buffers. Otherwise, all memory allocations will fail.

1.7 Using "canary" runtime checks

All functions in this library are *safe by design*, meaning that no memory corruption can occur using the library. However, there is always a risk that some user code corrupts the memory areas which are used to store the safe strings and memory buffers.

All functions in this library exist in two flavors. The default form of a function assumes that no external cause of memory corruption exists and is typically used for production code. The second form of a function uses "*canary*" runtime checks. In this approach, all data structures are protected using "*canary*" values at the start and end of all data structures. If a memory corruption occurs, it is likely that these canary values are modified. The canary form of each function performs runtime checks to detect memory corruptions.

To enable the canary runtime checks, define the symbol `SSM_USE_CANARY` before including `ssm.h` as follow:

```
#define SSM_USE_CANARY
#include "ssm.h"
```

See the function `ssm_set_canary_corrupted_handler()` for a description of the handling of memory corruptions when they are detected.

1.8 Supported platforms

The SSM library has been tested on the following platforms in user-mode applications. When a Linux kernel version is specified, the SSM library has also been tested in Loadable Kernel Modules (LKM) on the platform.

Operating System	**Architecture**	**Compiler**	**Linux Kernel**
Ubuntu 12.04 LTS	Intel x86-64	gcc 4.6.3	3.2.0
Ubuntu 14.04 LTS	Intel x86-64	gcc 4.8.2	3.13.0
Fedora 20	Intel x86-64	gcc 4.8.2	3.14.8
Red Hat Enterprise Linux 6.1	Intel x86-64	gcc 4.4.5	2.6.32
Linaro 12.05	ARM v7	gcc 4.6.3	
Microsoft Windows 7	Intel x86	Visual C++ 2010 Express	
Microsoft Windows 7	Intel x86	Visual C++ 2013 Express	
Microsoft Windows 7	Intel x86	Visual C++ 2015 Express	
Microsoft Windows 7	Intel x86-64	Visual C++ 2015 Express	
Microsoft Windows 7	Intel x86	gcc 4.8.2 (Cygwin)	
Microsoft Windows 7	Intel x86-64	gcc 4.8.2 (Cygwin)	

1.9 Code footprint

In embedded systems, there is a usual requirement for a small code footprint. The following table summarizes the code footprint in bytes of various usages of the SSM library on some standard platforms for the version 1.1 of the SSM library.

Feature	**ARM v7**	**Intel 32**	**Intel 64**
Total code size	6363	14948	13092
Production only (non-canary)	3711	8356	7314
Static strings only	1201	2912	2506
Dynamic strings only	1422	3768	3185
Static buffers only	1113	2600	2249
Dynamic buffers only	1334	3456	2928
C11K only, without strerror	744	1980	1657

Please note the following points:

- Each value represents the total size of the code, data and BSS segments in bytes.
- The code was compiled with size optimization in mind. With the default optimizations, the code is likely to be larger.
- All results are produced by GCC. MSVC code size was not checked.
- Slightly different values may be found with different versions of GCC.
- The reported values depend on the version of the library and need to be updated with the code. Make sure the documentation was correctly updated.
- The code is compiled as position-independent (`-fPIC`) by default to allow the creation of shared libraries. It has been observed that the code footprint is slightly smaller without this option.

- The total library code size includes the versions with and without "canary checks", which is not used in practice. The first line is consequently not meaningful in production environments. All subsequent results include only the "production" code (without "canary" checks).
- The sections about static or dynamic strings or memory buffers list the total code size for a given feature. But note that:
  - An application embeds only the code it requires, not all modules.
  - Some features use common code. Using two features usually requires less than the sum of the code for the two features.

2 Common mechanisms

2.1 SSM library identification

⋆ SSM_MAJOR_VERSION

```
#define SSM_MAJOR_VERSION 1
```

This macro defines the major version number of the SSM library, 1 in the example.

⋆ SSM_MINOR_VERSION

```
#define SSM_MINOR_VERSION 2
```

This macro defines the minor version number of the SSM library, 2 in the example.

⋆ SSM_VERSION

```
#define SSM_VERSION 102
```

This macro defines the version number of the SSM library as one single integer value equal to $100 * major + minor$. So version 1.2 gives 102 in the example.

⋆ SSM_VERSION_STRING

```
#define SSM_VERSION_STRING "1.2"
```

This macro defines the version of the SSM library as a string, "1.2" in the example.

2.2 Error reporting

⋆ ssm_status_t

Status values, as returned by all SSM functions.

```
typedef enum {...} ssm_status_t;
```

This enumeration type defines the status values, as returned by all SSM functions.

As a general rule, if the returned value is not zero (SSM_OK), the result is not the expected one. But, in all cases, the output buffers are *safe values*, meaning that truncated strings are still correctly null-terminated and the size of a truncated memory buffer indicates the useable (truncated) part of the buffer.

Status values can be categorized in three classes: success, non-fatal errors and fatal errors. The application has the choice to either check for individual status values or use one of the status checking macros.

Status value	Description	Severity
SSM_OK	The function executed successfully.	Success
SSM_TRUNCATED	The result is truncated but safe.	Success

SSM_EQUAL	Objects are equal after comparison.	Success
SSM_LOWER	Object 1 is lower than object 2 after comparison.	Success
SSM_GREATER	Object 1 is greater than object 2 after comparison.	Success
SSM_NULLOUT	A NULL pointer was provided as output parameter.	Non-fatal
SSM_SIZETOOLARGE	Some size is larger than SSM_SIZE_MAX.	Non-fatal
SSM_INDEXRANGE	An index or size parameter is out of range.	Non-fatal
SSM_SIZEZERO	Some size is zero.	Non-fatal
SSM_NULLIN	A NULL pointer was provided as input parameter.	Non-fatal
SSM_NOMEMORY	Memory allocation failure, result is unchanged.	Fatal
SSM_CORRUPTED	Memory was previously corrupted, result is undefined but safe.	Fatal
SSM_BUG	Internal inconsistency, there is a bug in the SSM library.	Fatal

The status values SSM_SIZEZERO and SSM_NULLIN are usually returned by the C11K functions only. The SSM functions always accept an empty destination buffer or a null pointer as input (equivalent to an empty string or buffer).

The following functions (they are actually macros) check the severity of a ssm_status_t value.

⋆ ssm_success

Check if a status indicates a success.

```
int ssm_success (ssm_status_t status);
```

Parameter	Mode	Description
status	in	A status to check.
	return	A non-zero value (true) if *status* indicates a success and zero (false) if it indicates an error.

A success means that the operation completed successfully. A success status may also indicate a truncation but, in the general case, this is not an error, the supplied inputs were too large for the application buffer but the result is safe.

As a general rule, if you want to test that an operation gave the *exact* result, compare it against SSM_OK. If you simply want to check if the application processing may reasonably continue, use ssm_success().

⋆ ssm_error

Check if a status indicates an error.

```
int ssm_error (ssm_status_t status);
```

Parameter	Mode	Description
status	in	A status to check.
	return	A non-zero value (true) if *status* indicates an error (fatal or non-fatal) and zero (false) if it indicates a success.

⋆ ssm_fatal

Check if a status indicates a fatal error.

```
int ssm_fatal (ssm_status_t status);
```

Parameter	Mode	Description
status	in	A status to check.
	return	A non-zero value (true) if *status* indicates a fatal error and zero (false) if it indicates a success or a non-fatal error.

A fatal error indicates that something went badly wrong in the application such as a memory corruption or no more available memory. Trying to continue after a fatal error may be dangerous since the application environment may be unstable. The usual response to a fatal error is terminating the application after the minimal cleanup of the resources.

Conversely, a non-fatal error indicates that the SSM function could not perform the requested operation because, for instance, a parameter was incorrect. In that case, the output objects (if any) are not modified and the application may continue after performing the adequate error processing.

⋆ ssm_status_string

Get a null-terminated string describing a status value.

```
const char* ssm_status_string (ssm_status_t status);
```

Parameter	Mode	Description
status	in	A status to get the description of.
	return	Address of a static constant null-terminated string describing *status*.

All calls with the same *status* return the same address. The pointed string shall not be modified by the application. Any *status* which does not correspond to a known value will return the same `"Unknown"` string.

2.3 Addresses and sizes

⋆ SSM_SIZE_MAX

The value of this macro specifies the maximum size of a used-defined memory area.

Sometimes, the application has to pass the size of a memory area to the SSM library. Such a size uses the predefined type `size_t`. The C standard defines this type as unsigned. Computing a `size_t` value may result in an underflow, giving a very large unsigned value. To detect these wrong values, the SSM library does not accept `size_t` values above some arbitrary large but reasonable value named `SSM_SIZE_MAX`.

This value is hard-coded to $2^{31}-1$ (approximately 2 giga-bytes). In practice, `size_t` is either 32 or 64 bits wide. But, even on 64-bit platforms, having a user-defined memory area larger that 2 GB is suspect and consequently rejected.

⋆ SSM_ADDRESS_MAX

The value of this macro specifies the maximum representable address in the system as a value of type `void*`.

⋆ ssm_addr_size

Compute a safe address plus an offset.

```
const void* ssm_addr_size (const void* addr, size_t size);
```

Parameter	Mode	Description
addr	in	Base address.
size	in	Offset to add to *addr*.
	return	Return *addr* plus *size* without overflow. The result is always greater than *addr*, returning `SSM_ADDRESS_MAX` in case of overflow.

When very large (and usually incorrect) sizes are added to an address, there is a risk of arithmetic overflow. The resulting address is lower than the base address, which usually produces nasty side effects in address or pointer comparisons. This function (which is inlined for performance) solves this problem.

2.4 Dynamic memory management

This section describes how the application can override the default memory management functions for the dynamic strings and dynamic memory buffers.

⋆ ssm_malloc_t

Memory allocation function type.

```
typedef void* (*ssm_malloc_t) (size_t size);
```

Parameter	Mode	Description
size	in	Size in bytes of the area to allocate.
	return	The base address of the allocated area or `NULL` if no memory is available.

This function profile is used to specify an alternative function to `malloc()`.

The SSM library never allocates memory areas larger that `SSM_SIZE_MAX`.

⋆ ssm_free_t

Memory deallocation function type.

```
typedef void (*ssm_free_t) (void* ptr);
```

Parameter	Mode	Description
ptr	in	Address of a previously allocated area.

This function profile is used to specify an alternative function to `free()`.

The SSM library never invokes the free function with a `NULL` pointer.

⋆ ssm_set_memory_management

Specify alternate functions for memory management.

```
void ssm_set_memory_management (ssm_malloc_t newMalloc, ssm_free_t newFree);
```

Parameter	Mode	Description
newMalloc	in	Alternate function for `malloc()`.
newFree	in	Alternate function for `free()`.

These functions are used by dynamic strings and dynamic memory buffers.

For user-mode programs, the default functions are the standard `malloc()` and `free()`. On some specific platforms, such as the Linux kernel, there is no default memory allocation functions and `ssm_set_memory_management()` must be invoked before using dynamic strings or dynamic memory buffers.

If any of the parameters *newMalloc* and *newFree* is `NULL`, then the library reverts to the corresponding default memory management function.

This function is not thread-safe and shall be invoked before the first usage of dynamic strings or dynamic memory buffers.

3 Static strings

3.1 Static strings overview

A static string is declared with a static maximum size, just like a regular C array. It can contain up to that number of characters. A static string is a fast low-level type without any sort of memory allocation. The storage for the string characters is reserved within the variable, like C arrays.

A static string is always null-terminated, even if an operation results in a truncation.

A static string is a polymorphic object which is formally defined by the abstract type `ssm_sstring_t`. In that case, *abstract* means that pointers to existing objects of this type can be used but no object shall be defined using this type name.

This is why a static string should be declared by the macro `ssm_sstring_declare()`. This macro ensures that the correct amount of storage is reserved and the variable is properly initialized to an empty string.

The following example defines a static string named `foo` with a maximum capacity of 50 characters, not including the trailing null character:

```
ssm_sstring_declare (foo, 50);
```

In practice, a variable which is defined by the macro `ssm_sstring_declare()` has no named type. It is only specified that it contains a field named `str` which is of type `ssm_sstring_t`. Thus, the actual `ssm_sstring_t` is in fact `foo.str`. Example:

```
ssm_sstring_declare (foo, 10);
ssm_sstring_import (&foo.str, "foo bar");
size_t len = ssm_sstring_length (&foo.str);
```

Note that, unlike C++, the C language does not forbid the definition of objects of an abstract type (the concept of *abstract type* does not even exist in C). So the following code compiles but it is incorrect: the variable is uninitialized and no storage is reserved for the string characters.

SO, DO NOT USE THIS:

```
ssm_sstring_t s;
```

3.2 Creating static strings

⋆ ssm_sstring_t

Abstract definition of a static string.

```
typedef ... ssm_sstring_t;
```

No object of this type shall be defined. To define an actual static string, use the macro `ssm_sstring_declare()`. See the section “Static strings overview” above for more details on the usage of static strings.

Correct example:

```
ssm_sstring_declare (s, 10);

ssm_sstring_import (&s.str, "foo bar");
size_t len = ssm_sstring_length (&s.str);
myFunc (&s.str);
```

Incorrect example, DO NOT USE:

```
ssm_sstring_t s;
```

Pointer to the type `ssm_sstring_t` may be used as function parameters. Example:

```
void myFunc (const ssm_sstring_t* s) {
    size_t len = ssm_sstring_length (s);
    ...
}
```

⋆ ssm_sstring_declare

Declare a `ssm_sstring_t` variable.

```
#define ssm_sstring_declare(variable,size) \
    ssm_sstring_struct(size) variable = ssm_sstring_init(size)
```

Parameter	**Description**
variable	Name of the variable to declare. The variable is safely initialized to the empty string.
size	Maximum number of characters in the string, not including the trailing null character. Shall be a compile-time constant, unless the compiler accepts variable-length arrays. If the compiler accepts such runtime sizes, *size* shall be an idempotent expression (ie. its result shall yield the same value for each evaluation and it shall have no side effect).

This macro declares a `ssm_sstring_t` variable.

In practice, *variable* has no named type. It is only specified that it contains a field named `str` which is of type `ssm_sstring_t`. Thus, the actual `ssm_sstring_t` is in fact `variable.str`.

For syntactic reasons, this macro cannot be used when the `ssm_sstring_t` is not a single variable but part of a structure for instance. In that case, use the macros `ssm_sstring_struct()` and `ssm_sstring_init()`.

⋆ ssm_sstring_struct

Declare an uninitialized `ssm_sstring_t` field.

```
#define ssm_sstring_struct(size) ...
```

Parameter	**Description**
size	Maximum number of characters in the string, not including the trailing null character. Shall be a compile-time constant, unless the compiler accepts variable-length arrays.

This macro declares an uninitialized `ssm_sstring_t` field.

When declaring a simple `ssm_sstring_t` variable, use the macro `ssm_sstring_declare()` instead. The macro `ssm_sstring_struct()` shall be reserved for contexts where it is not possible to initialize the data, such as in the case of a field within a structure. Be sure to initialize the corresponding data using the macro `ssm_sstring_init()` with **exactly the same size value**.

Example:

```
typedef struct {
    int before;
    ssm_sstring_struct(50) str;
```

```
    int after;
} my_struct_t;

my_struct_t a = {
    .before = 0x1234567,
    .str = ssm_sstring_init(50),
    .after = 0x1ABCDEF
};
```

⋆ `ssm_sstring_init`

Initializer for a `ssm_sstring_t` field.

```
#define ssm_sstring_init(size) ...
```

Parameter	Description
size	Maximum number of characters in the string, not including the trailing null character. Shall be a compile-time constant, unless the compiler accepts variable-length arrays. Shall be exactly the same value as used in the corresponding `ssm_sstring_struct()` macro.

This macro returns the initializer for a `ssm_sstring_t` field.

See the macro `ssm_sstring_struct()` for an example.

When declaring a simple `ssm_sstring_t` variable, use the macro `ssm_sstring_declare()` instead.

3.3 Manipulating static strings

⋆ `ssm_sstring_import`

Copy ("import") a null-terminated C-string into a `ssm_sstring_t`.

```
ssm_status_t ssm_sstring_import (ssm_sstring_t* dest, const char* src);
```

Parameter	Mode	Description
dest	out	Receive the C-string.
src	in	Address of a null-terminated C-string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ `ssm_sstring_import_size`

Copy ("import") an optionally null-terminated C-string into a `ssm_sstring_t`.

```
ssm_status_t ssm_sstring_import_size (ssm_sstring_t* dest,
                                      const char* src,
                                      size_t maxSize);
```

Parameter	Mode	Description
dest	out	Receive the C-string.

src	in	Address of a possibly null-terminated C-string. Can be `NULL` (same as empty string).
maxSize	in	Maximum number of characters to copy from *src*.
	return	A status value.

⋆ ssm_sstring_chars

Read-only access to a `ssm_sstring_t` as a null-terminated C-string.

```
const char* ssm_sstring_chars (const ssm_sstring_t* str);
```

Parameter	Mode	Description
str	in	The static string to read.
	return	The address of a read-only null-terminated C-string. The application is not allowed to modify this string. In case of detected memory corruption, return the address of an empty C-string.

Can be used to invoke legacy functions requiring a C-string.

⋆ ssm_sstring_length

Get the length of a `ssm_sstring_t`.

```
size_t ssm_sstring_length (const ssm_sstring_t* str);
```

Parameter	Mode	Description
str	in	The static string to read.
	return	The length of the string. In case of null *src* or detected memory corruption, return zero.

This function executes in constant time (the length value is stored, unlike `strlen()` there is no need to read the string up to the end to find the length).

⋆ ssm_sstring_max_length

Get the maximum string length that can be held in a `ssm_sstring_t`.

```
size_t ssm_sstring_max_length (const ssm_sstring_t* str);
```

Parameter	Mode	Description
str	in	The static string to read.
	return	The maximum number of characters that can be held in the string object, not including the null terminator. In case of null *src* or detected memory corruption, return zero.

⋆ ssm_sstring_set

Set all characters in a `ssm_sstring_t` to a given value.

```
ssm_status_t ssm_sstring_set (ssm_sstring_t* str, char value);
```

Parameter	Mode	Description
str	in,out	The static string to update.
value	in	Value to set in all characters in the string.
	return	A status value.

All characters inside the string are updated with the same common value. The size of the string is unchanged.

⋆ ssm_sstring_set_range

Set a range of characters in a `ssm_sstring_t` to a given value.

```
ssm_status_t ssm_sstring_set_range (ssm_sstring_t* str,
                                    size_t start,
                                    size_t length,
                                    char value);
```

Parameter	Mode	Description
str	in,out	The static string to update.
start	in	Starting index in the string of the area to modify.
length	in	Length in bytes of the area to modify.
value	in	Value to set in all characters in the area to modify.
	return	A status value.

All characters inside a specified range in the string are updated with the same common value. The size of the string is unchanged.

If the specified range is partially or entirely outside the current string size, the part of the range which is inside the string is updated and `SSM_TRUNCATED` is returned.

⋆ ssm_sstring_copy

Copy the content of a `ssm_sstring_t` into another one.

```
ssm_status_t ssm_sstring_copy (ssm_sstring_t* dest, const ssm_sstring_t* src);
```

Parameter	Mode	Description
dest	out	Receive a copy of the static string *src*.
src	in	Source static string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ ssm_sstring_concat

Append the content of a `ssm_sstring_t` at the end of another one.

```
ssm_status_t ssm_sstring_concat (ssm_sstring_t* dest, const ssm_sstring_t* src);
```

Parameter	Mode	Description
dest	in,out	Receive a copy of the static string *src* at end of previous value.
src	in	Source static string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ ssm_sstring_compare

Compare the content of two `ssm_sstring_t`.

```
ssm_status_t ssm_sstring_compare (const ssm_sstring_t* buf1,
                                  const ssm_sstring_t* buf2);
```

Parameter	Mode	Description
buf1	in	First string to compare. Can be `NULL` (same as empty string).
buf2	in	Second string to compare. Can be `NULL` (same as empty string).
	return	A status value, `SSM_GREATER`, `SSM_EQUAL` or `SSM_LOWER` according to whether the first string is greater than, equal to or less than the second string. Can also be an error code.

⋆ ssm_sstring_status_string

Get the description of a status value in a `ssm_sstring_t`.

```
ssm_status_t ssm_sstring_status_string (ssm_sstring_t* dest,
                                        ssm_status_t status);
```

Parameter	Mode	Description
dest	out	Receive the description string.
status	in	A status to get the description of.
	return	A status value.

All calls with the same *status* return the same description string. Any *status* which does not correspond to a known value will return the same `"Unknown"` string.

4 Dynamic strings

4.1 Dynamic strings overview

A dynamic string uses dynamic memory allocation to store the content.

See Section 2.4 [Dynamic memory management], page 10 for more details on the implementation of memory allocation.

A dynamic string is defined by the type `ssm_dstring_t`. Since a dynamic string points to allocated memory, it is essential that any such object is properly initialized. This is why a dynamic string should be declared by the macro `ssm_dstring_declare()`. This macro ensures that the variable is properly initialized to an empty string.

The following example defines a dynamic string named `foo`:

```
ssm_dstring_declare (foo);

ssm_dstring_import (&foo, "foo bar");
size_t len = ssm_dstring_length (&foo);
```

Note that, unlike C++, the C language does not have any *constructor* mechanism providing a guaranteed initial content to an object. Using an uninitialized dynamic string object may lead to crash or other unexpected behavior. So the following code compiles but it is incorrect: the variable is uninitialized and may point to invalid data.

SO, DO NOT USE THIS:

```
ssm_dstring_t s;
```

To avoid memory leaks, it is essential that any dynamic string object is freeed before the object goes out of scope. Use the function `ssm_dstring_free()` as illustrated below:

```
void f(void)
{
    ssm_dstring_declare (foo);
    ...
    ssm_dstring_free (&foo);
}
```

Pay attention to *early return paths* which could lead to memory leaks as in the following example:

```
void f(void)
{
    ssm_dstring_declare (foo);
    ...
    if (some_condition) {
        /* MEMORY LEAK HERE */
        return;
    }
    ...
    ssm_dstring_free (&foo);
}
```

4.2 Creating and destroying dynamic strings

⋆ ssm_dstring_t

Definition of a dynamic string.

```
typedef ... ssm_dstring_t;
```

To define an actual dynamic string, use the macro `ssm_dstring_declare()`. See the section "Dynamic strings overview" above for more details on the usage of dynamic strings.

Correct example:

```
ssm_dstring_declare (s);

ssm_dstring_import (&s, "foo bar");
size_t len = ssm_dstring_length (&s);
```

Incorrect example, the object is not properly initialized. DO NOT USE:

```
ssm_dstring_t s;
```

⋆ ssm_dstring_declare

Declare a `ssm_dstring_t` variable.

```
#define ssm_dstring_declare(variable) \
    ssm_dstring_t variable = ssm_dstring_init
```

Parameter	**Description**
variable	Name of the variable to declare. The variable is safely initialized to the empty string.

This macro declares a `ssm_dstring_t` variable.

For syntactic reasons, this macro cannot be used when the `ssm_dstring_t` is not a single variable but part of a structure for instance. In that case, use the macro `ssm_dstring_init()` to initialize the `ssm_dstring_t` field.

⋆ ssm_dstring_init

Initializer for a `ssm_dstring_t` field.

```
#define ssm_dstring_init(size) ...
```

This macro returns the initializer for a `ssm_dstring_t` field.

When declaring a simple `ssm_dstring_t` variable, use the macro `ssm_dstring_declare()` instead. The macro `ssm_dstring_init` shall be reserved for contexts where it is not possible to initialize the data, such as in the case of a field within a structure. Be sure to initialize the corresponding data using the macro `ssm_dstring_init`.

Example:

```
typedef struct {
    int before;
    ssm_dstring_t str;
    int after;
} my_struct_t;

my_struct_t a = {
```

```
        .before = 0x1234567,
        .str = ssm_dstring_init,
        .after = 0x1ABCDEF
    };
```

⋆ `ssm_dstring_free`

Free a `ssm_dstring_t` dynamic string.

```
ssm_status_t ssm_dstring_free (ssm_dstring_t* str);
```

Parameter	Mode	Description
buf	in,out	Dynamic string to free.
	return	A status code.

This function frees a `ssm_dstring_t`.

Upon return, the object is still valid but has the semantic of an empty string and has no longer any dynamic storage associated with it.

4.3 Manipulating dynamic strings

⋆ `ssm_dstring_import`

Copy ("import") a null-terminated C-string into a `ssm_dstring_t`.

```
ssm_status_t ssm_dstring_import (ssm_dstring_t* dest, const char* src);
```

Parameter	Mode	Description
dest	out	String to fill.
src	in	Address of a null-terminated C-string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ `ssm_dstring_import_size`

Copy ("import") an optionally null-terminated C-string into a `ssm_dstring_t`.

```
ssm_status_t ssm_dstring_import_size (ssm_dstring_t* dest,
                                      const char* src,
                                      size_t maxSize);
```

Parameter	Mode	Description
dest	out	String to fill.
src	in	Address of a possibly null-terminated C-string. Can be `NULL` (same as empty string).
maxSize	in	Maximum number of characters to copy from *src*.
	return	A status value.

⋆ `ssm_dstring_chars`

Read-only access to a `ssm_dstring_t` as a null-terminated C-string.

```
const char* ssm_dstring_chars (const ssm_dstring_t* str);
```

Parameter	Mode	Description
str	in	The dynamic string to read.
	return	The current address of a read-only null-terminated C-string. The application is not allowed to modify this string. In case of detected memory corruption, return the address of an empty C-string.

Warning: The returned address is valid only as long as `str` is not modified. Since this function is quite fast, it is recommended to **not** store the returned value and invoke the function each time the address is needed.

Good example:

```
dump (ssm_dstring_data (&str));
ssm_dstring_import (&str, "foo");
dump (ssm_dstring_data (&str));
```

Bad example, DO NOT DO THIS:

```
const void* data = ssm_dstring_data (&str);
dump (data);
ssm_dstring_import (&str, "foo");
dump (data);  /* WRONG, 'data' may no longer point to 'str' characters */
```

⋆ ssm_dstring_length

Get the length of a `ssm_dstring_t` content.

```
size_t ssm_dstring_length (const ssm_dstring_t* str);
```

Parameter	Mode	Description
str	in	The dynamic string to read.
	return	The length of the string. In case of null *src* or detected memory corruption, return zero.

This function executes in constant time (the length value is stored, unlike `strlen()` there is no need to read the string up to the end to find the length).

⋆ ssm_dstring_set

Set all characters in a `ssm_dstring_t` to a given value.

```
ssm_status_t ssm_dstring_set (ssm_dstring_t* str, char value);
```

Parameter	Mode	Description
str	in,out	The dynamic string to update.
value	in	Value to set in all characters in the string.
	return	A status value.

All characters inside the string are updated with the same common value. The size of the string is unchanged.

⋆ ssm_dstring_set_range

Set a range of characters in a `ssm_dstring_t` to a given value.

```
ssm_status_t ssm_dstring_set_range (ssm_dstring_t* str,
                                    size_t start,
                                    size_t length,
                                    char value);
```

Parameter	Mode	Description
str	in,out	The dynamic string to update.
start	in	Starting index in the string of the area to modify.
length	in	Length in characters of the area to modify.
value	in	Value to set in all characters in the area to modify.
	return	A status value.

All characters inside a specified range in the string are updated with the same common value. The size of the string is unchanged.

If the specified range is partially or entirely outside the current string size, the part of the range which is inside the string is updated and `SSM_TRUNCATED` is returned.

⋆ ssm_dstring_copy

Copy the content of a `ssm_dstring_t`.

```
ssm_status_t ssm_dstring_copy (ssm_dstring_t* dest, const ssm_dstring_t* src);
```

Parameter	Mode	Description
dest	out	Receive a copy of the dynamic string *src*.
src	in	Source dynamic string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ ssm_dstring_concat

Append the content of a ssm_dstring_t at the end of another one.

```
ssm_status_t ssm_dstring_concat (ssm_dstring_t* dest, const ssm_dstring_t* src);
```

Parameter	Mode	Description
dest	in,out	Receive a copy of the dynamic string *src* at end of previous value.
src	in	Source dynamic string. Can be `NULL` (same as empty string).
	return	A status value.

⋆ ssm_dstring_compare

Compare the content of two `ssm_dstring_t`.

```
ssm_status_t ssm_dstring_compare (const ssm_dstring_t* buf1,
                                  const ssm_dstring_t* buf2);
```

Parameter	Mode	Description
buf1	in	First string to compare. Can be `NULL` (same as empty string).
buf2	in	Second string to compare. Can be `NULL` (same as empty string).

	return	A status value, `SSM_GREATER`, `SSM_EQUAL` or `SSM_LOWER` according to whether the first string is greater than, equal to or less than the second string. Can also be an error code.

⋆ ssm_dstring_status_string

Get the description of a status value in a `ssm_dstring_t`.

```
ssm_status_t ssm_dstring_status_string (ssm_dstring_t* dest,
                                        ssm_status_t status);
```

Parameter	Mode	Description
dest	out	Receive the description string.
status	in	A status to get the description of.
	return	A status value.

All calls with the same *status* return the same description string. Any *status* which does not correspond to a known value will return the same `"Unknown"` string.

5 Static memory buffers

5.1 Static memory buffers overview

A static memory buffer is declared with a static maximum size, just like a regular C array. It can contain up to that number of bytes. A static buffer is a fast low-level type without any sort of memory allocation. The storage for the buffer data is reserved within the variable, like C arrays.

A static memory buffer also has a *current size* (or *length*) which indicates the number of usable bytes in the buffer, meaning the number of bytes which were actually written into it by user code.

A static buffer is a polymorphic object which is formally defined by the abstract type `ssm_sbuffer_t`. In that case, *abstract* means that pointers to existing objects of this type can be used but no object shall be defined using this type name.

This is why a static memory buffer should be declared by the macro `ssm_sbuffer_declare()`. This macro ensures that the correct amount of storage is reserved and the variable is properly initialized to an empty buffer.

The following example defines a static memory buffer named `foo` with a maximum capacity of 50 bytes:

```
ssm_sbuffer_declare (foo, 50);
```

In practice, a variable which is defined by the macro `ssm_sbuffer_declare()` has no named type. It is only specified that it contains a field named `buf` which is of type `ssm_sbuffer_t`. Thus, the actual `ssm_sbuffer_t` is in fact `foo.buf`. Example:

```
ssm_sbuffer_declare (foo, 10);
ssm_sbuffer_import (&foo.buf, &someData, sizeof(someData));
size_t len = ssm_sbuffer_length (&foo.buf);
```

Note that, unlike C++, the C language does not forbid the definition of objects of an abstract type (the concept of *abstract type* does not even exist in C). So the following code compiles but it is incorrect: the variable is uninitialized and no storage is reserved for the buffer data.

SO, DO NOT USE THIS:

```
ssm_sbuffer_t foo;
```

5.2 Creating static buffers

⋆ `ssm_sbuffer_t`

Abstract definition of a static buffer.

```
typedef ... ssm_sbuffer_t;
```

No object of this type shall be defined. To define an actual static buffer, use the macro `ssm_sbuffer_declare()`. See the section “Static buffers overview” above for more details on the usage of static buffers.

Correct example:

```
ssm_sbuffer_declare (b, 10);

ssm_sbuffer_import (&b.buf, &someData, sizeof(someData));
size_t len = ssm_sbuffer_length (&b.buf);
myFunc (&b.buf);
```

Incorrect example, DO NOT USE:

```
ssm_sbuffer_t b;
```

Pointer to the type `ssm_sbuffer_t` may be used as function parameters. Example:

```
void myFunc (const ssm_sbuffer_t* b) {
    size_t len = ssm_sbuffer_length (b);
    ...
}
```

⋆ ssm_sbuffer_declare

Declare a `ssm_sbuffer_t` variable.

```
#define ssm_sbuffer_declare(variable,size) \
    ssm_sbuffer_struct(size) variable = ssm_sbuffer_init(size)
```

Parameter	**Description**
variable	Name of the variable to declare. The variable is safely initialized to an empty buffer.
size	Maximum number of bytes in the buffer. Shall be a compile-time constant, unless the compiler accepts variable-length arrays. If the compiler accepts such runtime sizes, *size* shall be an idempotent expression (ie. its result shall yield the same value for each evaluation and it shall have no side effect).

This macro declares a `ssm_sbuffer_t` variable.

In practice, *variable* has no named type. It is only specified that it contains a field named `buf` which is of type `ssm_sbuffer_t`. Thus, the actual `ssm_sbuffer_t` is in fact `variable.buf`.

For syntactic reasons, this macro cannot be used when the `ssm_sbuffer_t` is not a single variable but part of a structure for instance. In that case, use the macros `ssm_sbuffer_struct()` and `ssm_sbuffer_init()`.

⋆ ssm_sbuffer_struct

Declare an uninitialized `ssm_sbuffer_t` field.

```
#define ssm_sbuffer_struct(size) ...
```

Parameter	**Description**
size	Maximum number of bytes in the buffer. Shall be a compile-time constant, unless the compiler accepts variable-length arrays.

This macro declares an uninitialized `ssm_sbuffer_t` field.

When declaring a simple `ssm_sbuffer_t` variable, use the macro `ssm_sbuffer_declare()` instead. The macro `ssm_sbuffer_struct()` shall be reserved for contexts where it is not possible to initialize the data, such as in the case of a field within a structure. Be sure to initialize the corresponding data using the macro `ssm_sbuffer_init()` with **exactly the same size value**.

Example:

```
typedef struct {
    int before;
    ssm_sbuffer_struct(50) buf;
    int after;
} my_struct_t;
```

```
my_struct_t a = {
    .before = 0x1234567,
    .buf = ssm_sbuffer_init(50),
    .after = 0x1ABCDEF
};
```

⋆ `ssm_sbuffer_init`

Initializer for a `ssm_sbuffer_t` field.

```
#define ssm_sbuffer_init(size) ...
```

Parameter	Description
size	Maximum number of bytes in the buffer. Shall be a compile-time constant, unless the compiler accepts variable-length arrays. Shall be exactly the same value as used in the corresponding `ssm_sbuffer_struct()` macro.

This macro returns the initializer for a `ssm_sbuffer_t` field.

See the macro `ssm_sbuffer_struct()` for an example.

When declaring a simple `ssm_sbuffer_t` variable, use the macro `ssm_sbuffer_declare()` instead.

5.3 Manipulating static buffers

⋆ `ssm_sbuffer_import`

Copy ("import") a memory area into a `ssm_sbuffer_t`.

```
ssm_status_t ssm_sbuffer_import (ssm_sbuffer_t* dest,
                                 const void* src,
                                 size_t maxSize);
```

Parameter	Mode	Description
dest	out	Buffer to fill.
src	in	Address of a memory buffer. Can be `NULL` (same as empty buffer).
maxSize	in	Maximum number of bytes to copy from *src*.
	return	A status value.

⋆ `ssm_sbuffer_data`

Read-only access to a `ssm_sbuffer_t` binary content.

```
const void* ssm_sbuffer_data (const ssm_sbuffer_t* buf);
```

Parameter	Mode	Description
buf	in	The static buffer to read.
	return	The address of the buffer binary content. The application is not allowed to modify this buffer.

⋆ ssm_sbuffer_length

Get the length of a `ssm_sbuffer_t` content.

```
size_t ssm_sbuffer_length (const ssm_sbuffer_t* buf);
```

Parameter	Mode	Description
buf	in	The static buffer to read.
	return	The used length of the buffer. In case of null *src* or detected memory corruption, return zero.

⋆ ssm_sbuffer_max_length

Get the maximum data size that can be held in a `ssm_sbuffer_t`.

```
size_t ssm_sbuffer_max_length (const ssm_sbuffer_t* buf);
```

Parameter	Mode	Description
buf	in	The static buffer to read.
	return	The maximum number of bytes that can be held in the buffer object. In case of null *src* or detected memory corruption, return zero.

⋆ ssm_sbuffer_resize

Resize a `ssm_sbuffer_t`.

```
ssm_status_t ssm_sbuffer_resize (ssm_sbuffer_t* buf, size_t length);
```

Parameter	Mode	Description
buf	in,out	The static buffer to resize.
length	in	New length of the static buffer. If larger than the maximum size of the buffer, the buffer is resized to its maximum length and `SSM_TRUNCATED` is returned.
	return	A status value.

The content of the buffer is resized to the specified length. If the new length is longer than the previous length, the binary content of the buffer after the previous content is undefined. If the new length is shorter than the previous length, the buffer content is truncated and the returned status is `SSM_OK`.

⋆ ssm_sbuffer_set

Set all bytes in a `ssm_sbuffer_t` to a given value.

```
ssm_status_t ssm_sbuffer_set (ssm_sbuffer_t* buf, uint8_t value);
```

Parameter	Mode	Description
buf	in,out	The static buffer to update.
value	in	Value to set in all bytes in the buffer.
	return	A status value.

All bytes inside the buffer are updated with the same common value. The size of the buffer is unchanged.

⋆ ssm_sbuffer_set_range

Set a range of bytes in a `ssm_sbuffer_t` to a given value.

```
ssm_status_t ssm_sbuffer_set_range (ssm_sbuffer_t* buf,
                                    size_t start,
                                    size_t length,
                                    uint8_t value);
```

Parameter	Mode	Description
buf	in,out	The static buffer to update.
start	in	Starting index in the buffer of the area to modify.
length	in	Length in bytes of the area to modify.
value	in	Value to set in all bytes in the area to modify.
	return	A status value.

All bytes inside a specified range in the buffer are updated with the same common value. The size of the buffer is unchanged.

If the specified range is partially or entirely outside the current buffer size, the part of the range which is inside the buffer is updated and `SSM_TRUNCATED` is returned.

⋆ ssm_sbuffer_copy

Copy the content of a `ssm_sbuffer_t` into another one.

```
ssm_status_t ssm_sbuffer_copy (ssm_sbuffer_t* dest, const ssm_sbuffer_t* src);
```

Parameter	Mode	Description
dest	out	Receive a copy of the static buffer *src*.
src	in	Source static buffer. Can be `NULL` (same as empty buffer).
	return	A status value.

⋆ ssm_sbuffer_concat

Append the content of a `ssm_sbuffer_t` at the end of another one.

```
ssm_status_t ssm_sbuffer_concat (ssm_sbuffer_t* dest, const ssm_sbuffer_t* src);
```

Parameter	Mode	Description
dest	in,out	Receive a copy of the static buffer *src* at end of previous value.
src	in	Source static buffer. Can be `NULL` (same as empty buffer).
	return	A status value.

⋆ ssm_sbuffer_compare

Compare the content of two `ssm_sbuffer_t`.

```
ssm_status_t ssm_sbuffer_compare (const ssm_sbuffer_t* buf1,
                                  const ssm_sbuffer_t* buf2);
```

Parameter	Mode	Description
buf1	in	First buffer to compare. Can be `NULL` (same as empty buffer).
buf2	in	Second buffer to compare. Can be `NULL` (same as empty buffer).
	return	A status value, `SSM_GREATER`, `SSM_EQUAL` or `SSM_LOWER` according to whether the first buffer is greater than, equal to or less than the second buffer. Can also be an error code.

```
ssm_status_t ssm_sbuffer_insert (ssm_sbuffer_t* buf,
                                 size_t start,
                                 size_t length);
```

Parameter	Mode	Description
buf	in,out	The buffer to modify.
start	in	Index in the buffer where to insert a "hole".
length	in	Size of the "hole".
	return	A status value.

The content size of the buffer is increased by *length* bytes. All bytes in the buffer starting at index *start* are moved *length* bytes upward. The previous content of the buffer in the created "hole" is left unmodified. If the new size of the buffer would exceed its maximum size, it truncated to its maximum size and `SSM_TRUNCATED` is returned. If *start* is greater than the current buffer size, the buffer is unmodified and `SSM_INDEXRANGE` is returned.

6 Dynamic memory buffers

6.1 Dynamic memory buffers overview

A dynamic memory buffer uses dynamic memory allocation to store the content.

See Section 2.4 [Dynamic memory management], page 10 for more details on the implementation of memory allocation.

A dynamic memory buffer is defined by the type `ssm_dbuffer_t`. Since a dynamic memory buffer points to allocated memory, it is essential that any such object is properly initialized. This is why a dynamic memory buffer should be declared by the macro `ssm_dbuffer_declare()`. This macro ensures that the variable is properly initialized to an empty buffer.

The following example defines a dynamic memory buffer named `foo`:

```
ssm_dbuffer_declare (foo);

ssm_dbuffer_import (&foo, &someData, sizeof(someData));
size_t len = ssm_dbuffer_length (&foo);
```

Note that, unlike C++, the C language does not have any *constructor* mechanism providing a guaranteed initial content to an object. Using a uninitialized dynamic buffer object may lead to crash or other unexpected behavior. So the following code compiles but it is incorrect: the variable is uninitialized and may point to invalid data.

SO, DO NOT USE THIS:

```
ssm_dbuffer_t foo;
```

To avoid memory leaks, it is essential that any dynamic buffer object is freeed before the object goes out of scope. Use the function `ssm_dbuffer_free()` as illustrated below:

```
void f(void)
{
    ssm_dbuffer_declare (foo);
    ...
    ssm_dbuffer_free (&foo);
}
```

Pay attention to *early return paths* which could lead to memory leaks as in the following example:

```
void f(void)
{
    ssm_dbuffer_declare (foo);
    ...
    if (some_condition) {
        /* MEMORY LEAK HERE */
        return;
    }
    ...
    ssm_dbuffer_free (&foo);
}
```

6.2 Creating and destroying dynamic buffers

⋆ ssm_dbuffer_t

Definition of a dynamic buffer.

```
typedef ... ssm_dbuffer_t;
```

To define an actual dynamic buffer, use the macro `ssm_dbuffer_declare()`. See the section "Dynamic buffers overview" above for more details on the usage of dynamic buffers.

Correct example:

```
ssm_dbuffer_declare (b);

ssm_dbuffer_import (&b, &someData, sizeof(someData));
size_t len = ssm_dbuffer_length (&b);
```

Incorrect example, the object is not properly initialized. DO NOT USE:

```
ssm_dbuffer_t b;
```

⋆ ssm_dbuffer_declare

Declare a `ssm_dbuffer_t` variable.

```
#define ssm_dbuffer_declare(variable) \
    ssm_dbuffer_t variable = ssm_dbuffer_init
```

Parameter	**Description**
variable	Name of the variable to declare. The variable is safely initialized to an empty buffer.

This macro declares a `ssm_dbuffer_t` variable.

For syntactic reasons, this macro cannot be used when the `ssm_dbuffer_t` is not a single variable but part of a structure for instance. In that case, use the macro `ssm_dbuffer_init()` to initialize the `ssm_dbuffer_t` field.

⋆ ssm_dbuffer_init

Initializer for a `ssm_dbuffer_t` field.

```
#define ssm_dbuffer_init(size) ...
```

This macro returns the initializer for a `ssm_dbuffer_t` field.

When declaring a simple `ssm_dbuffer_t` variable, use the macro `ssm_dbuffer_declare()` instead. The macro `ssm_dbuffer_init` shall be reserved for contexts where it is not possible to initialize the data, such as in the case of a field within a structure. Be sure to initialize the corresponding data using the macro `ssm_dbuffer_init`.

Example:

```
typedef struct {
    int before;
    ssm_dbuffer_t str;
    int after;
} my_struct_t;

my_struct_t a = {
```

```
        .before = 0x1234567,
        .str = ssm_dbuffer_init,
        .after = 0x1ABCDEF
    };
```

⋆ ssm_dbuffer_free

Free a ssm_dbuffer_t dynamic buffer.

```
ssm_status_t ssm_dbuffer_free (ssm_dbuffer_t* buf);
```

Parameter	Mode	Description
buf	in,out	Dynamic buffer to free.
	return	A status code.

This function frees a ssm_dbuffer_t.

Upon return, the object is still valid but has the semantic of an empty buffer and has no longer any dynamic storage associated with it.

6.3 Manipulating dynamic buffers

⋆ ssm_dbuffer_import

Copy ("import") a memory area into a ssm_dbuffer_t.

```
ssm_status_t ssm_dbuffer_import (ssm_dbuffer_t* dest,
                                 const void* src,
                                 size_t maxSize);
```

Parameter	Mode	Description
dest	out	Buffer to fill.
src	in	Address of a memory buffer. Can be NULL (same as empty buffer).
maxSize	in	Maximum number of bytes to copy from *src*.
	return	A status value.

⋆ ssm_dbuffer_data

Read-only access to a ssm_dbuffer_t binary content.

```
const void* ssm_dbuffer_data (const ssm_dbuffer_t* buf);
```

Parameter	Mode	Description
buf	in	The dynamic buffer to read.
	return	The current address of the buffer binary content. The application is not allowed to modify this buffer.

Warning: The returned address is valid only as long as buf is not modified. Since this function is quite fast, it is recommended to **not** store the returned value and invoke the function each time the address is needed.

Good example:

```
dump (ssm_dbuffer_data (&buf));
ssm_dbuffer_import (&buf, &someData, sizeof(someData));
dump (ssm_dbuffer_data (&buf));
```

Bad example, DO NOT DO THIS:

```
const void* data = ssm_dbuffer_data (&buf);
dump (data);
ssm_dbuffer_import (&buf, &someData, sizeof(someData));
dump (data);  /* WRONG, 'data' may no longer point to 'buf' content */
```

⋆ ssm_dbuffer_length

Get the length of a `ssm_dbuffer_t` content.

```
size_t ssm_dbuffer_length (const ssm_dbuffer_t* buf);
```

Parameter	Mode	Description
buf	in	The dynamic buffer to read.
	return	The used length of the buffer. In case of null *src* or detected memory corruption, return zero.

⋆ ssm_dbuffer_resize

Resize a `ssm_dbuffer_t`.

```
ssm_status_t ssm_dbuffer_resize (ssm_dbuffer_t* buf, size_t length);
```

Parameter	Mode	Description
buf	in,out	The dynamic buffer to resize.
length	in	New length of the dynamic buffer.
	return	A status value.

The content of the buffer is resized to the specified length. If the new length is longer than the previous length, the binary content of the buffer after the previous content is undefined. If the new length is shorter than the previous length, the buffer content is truncated and the returned status is `SSM_OK`.

⋆ ssm_dbuffer_set

Set all bytes in a `ssm_dbuffer_t` to a given value.

```
ssm_status_t ssm_dbuffer_set (ssm_dbuffer_t* buf, uint8_t value);
```

Parameter	Mode	Description
buf	in,out	The dynamic buffer to update.
value	in	Value to set in all bytes in the buffer.
	return	A status value.

All bytes inside the buffer are updated with the same common value. The size of the buffer is unchanged.

⋆ ssm_dbuffer_set_range

Set a range of bytes in a `ssm_dbuffer_t` to a given value.

```
ssm_status_t ssm_dbuffer_set_range (ssm_dbuffer_t* buf,
                                    size_t start,
                                    size_t length,
                                    uint8_t value);
```

Parameter	Mode	Description
buf	in,out	The dynamic buffer to update.
start	in	Starting index in the buffer of the area to modify.
length	in	Length in bytes of the area to modify.
value	in	Value to set in all bytes in the area to modify.
	return	A status value.

All bytes inside a specified range in the buffer are updated with the same common value. The size of the buffer is unchanged.

If the specified range is partially or entirely outside the current buffer size, the part of the range which is inside the buffer is updated and `SSM_TRUNCATED` is returned.

⋆ ssm_dbuffer_copy

Copy the content of a `ssm_dbuffer_t`.

```
ssm_status_t ssm_dbuffer_copy (ssm_dbuffer_t* dest, const ssm_dbuffer_t* src);
```

Parameter	Mode	Description
dest	out	Receive a copy of the dynamic buffer *src*.
src	in	Source dynamic buffer. Can be `NULL` (same as empty buffer).
	return	A status value.

⋆ ssm_dbuffer_concat

Append the content of a ssm_dbuffer_t at the end of another one.

```
ssm_status_t ssm_dbuffer_concat (ssm_dbuffer_t* dest, const ssm_dbuffer_t* src);
```

Parameter	Mode	Description
dest	in,out	Receive a copy of the dynamic buffer *src* at end of previous value.
src	in	Source dynamic buffer. Can be `NULL` (same as empty buffer).
	return	A status value.

⋆ ssm_dbuffer_compare

Compare the content of two `ssm_dbuffer_t`.

```
ssm_status_t ssm_dbuffer_compare (const ssm_dbuffer_t* buf1,
                                  const ssm_dbuffer_t* buf2);
```

Parameter	Mode	Description
buf1	in	First buffer to compare. Can be `NULL` (same as empty buffer).
buf2	in	Second buffer to compare. Can be `NULL` (same as empty buffer).
	return	A status value, `SSM_GREATER`, `SSM_EQUAL` or `SSM_LOWER` according to whether the first buffer is greater than, equal to or less than the second buffer. Can also be an error code.

⋆ ssm_dbuffer_insert

Insert a "hole" in a `ssm_dbuffer_t`, shifting the rest of the buffer upward.

```
ssm_status_t ssm_dbuffer_insert (ssm_dbuffer_t* buf,
                                 size_t start,
                                 size_t length);
```

Parameter	Mode	Description
buf	in,out	The buffer to modify.
start	in	Index in the buffer where to insert a "hole".
length	in	Size of the "hole".
	return	A status value.

The content size of the buffer is increased by *length* bytes. All bytes in the buffer starting at index *start* are moved *length* bytes upward. The previous content of the buffer in the created "hole" is left unmodified. If *start* is greater than the current buffer size, the buffer is unmodified and `SSM_INDEXRANGE` is returned.

7 Low-level mechanisms

7.1 Memory corruption detection

As explained in Section 1.7 [Using "canary" runtime checks], page 4, the SSM library can be used with "canary checks", a debug mode which detects memory corruptions.

When a memory corruption is detected in an application, the default behavior is to report an error message and abort the application. Within the Linux kernel, a kernel message of severity *alert* is logged.

This section explains how to override this default behavior.

⋆ ssm_canary_corrupted_handler_t

Memory corruption handler profile.

```
typedef void (*ssm_canary_corrupted_handler_t) (const char* file, size_t line);
```

Parameter	Mode	Description
file	in	Application source file name.
line	in	Line number in the source file.

A function of this type is invoked when "canary" runtime checks are enabled and a memory corruption is detected.

The parameters of the function indicate the location of the failure, ie. the user code which invoked an SSM function which detected the memory corruption.

⋆ ssm_set_canary_corrupted_handler

Establish a user-defined memory corruption handler.

```
ssm_canary_corrupted_handler_t
    ssm_set_canary_corrupted_handler (ssm_canary_corrupted_handler_t handler);
```

Parameter	Mode	Description
handler	in	New handler to set. If NULL, revert to the default handler.
	return	The previous handler or NULL if the default handler was active.

This function specifies which function should be used when a memory corruption is detected. This function does nothing if canary runtime checks are disabled.

In user-mode programs, the default handler displays an error message on the standard error and aborts the application.

This function is not thread-safe.

7.2 Manipulating raw memory

The functions in this section manipulate raw memory areas. They should not be used in normal operations and should be reserved to extreme situations.

⋆ ssm_copy

General-purpose memory copy.

```
ssm_status_t ssm_copy (void* dest,
                       size_t destSize,
                       const void* src,
                       size_t srcSize,
                       size_t* copiedSize);
```

Parameter	Mode	Description
dest	out	Base address of destination area.
destSize	in	Maximum size in bytes of destination area.
src	in	Base address of source area. Can be `NULL`, in which case *srcSize* is implicitely zero.
srcSize	in	Size in bytes of source area. Ignored if *src* is `NULL`.
copiedSize	out	Receive the actual number of copied bytes. Can be `NULL` (value not returned).
	return	A status value.

This function is an alternate version of the standard `memcpy()` with bounded sizes. Overlapping memory areas between *src* and *dest* are allowed.

If not `NULL`, *copiedSize* receives the actual number of copied bytes. If the returned status is `SSM_OK`, it gets the same value as *srcSize*. If the returned status is `SSM_TRUNCATED`, it gets the same value as *destSize*. For all other returned status, the value pointed by *copiedSize* is not modified. It is safe to point directly to a `size_t` variable containing the used size of *dest*. In case of error, neither *dest* nor ∗*copiedSize* are modified).

⋆ ssm_compare

General-purpose memory comparison.

```
ssm_status_t ssm_compare (const void* addr1,
                          size_t size1,
                          const void* addr2,
                          size_t size2);
```

Parameter	Mode	Description
addr1	in	Base address of first memory area.
size1	in	Size in bytes of first memory area.
addr2	in	Base address of second memory area.
size2	in	Size in bytes of second memory area.
	return	The result of the comparison between the two memory areas, either `SSM_EQUAL`, `SSM_LOWER`, `SSM_GREATER` or an error code.

This function is an alternate version of the standard `memcmp()` with bounded sizes.

The two memory areas are compared byte by byte. The function returns `SSM_GREATER`, `SSM_EQUAL` or `SSM_LOWER` according to whether the first memory area is greater than, equal to or less than the second memory area.

Two memory areas are equal if they have the same size and same content. If the two areas have different sizes, the comparison is made on the smallest of the two sizes and, if both contents are identical, the area with the smallest size is logically less than the other area.

If any of *addr1* or *addr2* is `NULL`, it is equivalent to an empty area the size of which is zero. All empty areas, whether their address is `NULL` or their size is explicitly zero, are considered as equal.

⋆ ssm_set

General-purpose memory initialization.

```
ssm_status_t ssm_set (void* dest, size_t destSize, uint8_t value);
```

Parameter	Mode	Description
dest	out	Base address of destination area.
destSize	in	Size in bytes of destination area.
value	in	The value to set in each byte of the destination area.
	return	A status value.

This function is an alternate version of the standard `memset()` with bounded sizes.

⋆ ssm_cstring_length

Compute the length of a C-string.

```
size_t ssm_cstring_length(const char* str, size_t maxSize);
```

Parameter	Mode	Description
str	in	Address of a null-terminated C-string.
maxSize	in	Maximum size of the C-string. Specify `SSM_SIZE_MAX` if unbounded.
	return	The length of the C-string or *maxSize* if no null character was found in the first *maxSize* characters.

This function is an alternate version of the standard `strnlen()` with bounded size. If *maxSize* is greater than `SSM_SIZE_MAX`, the size is assumed to be incorrect and zero is returned.

8 Subset of C11 Annex K (Bound-Checking Interfaces)

8.1 C11 Annex K Overview

The problems of the unsafe C string functions from the C-library have been known for a long time in the C community. Amongst the various *safer-but-not-so-safe* alternatives to these functions, the Annex K of the C11 standard defines a list of string handling functions with "bound-checking interfaces".

In the rest of this chapter, we will use the notation "C11K" to designate this annex. Since the names of the C11K functions end with `_s`, they are also often called the "`_s` functions".

Why using these `_s` functions when we have the SSM library? It is often necessary to interface with legacy code which manipulates strings and memory buffers using *address* and *size*. While new code can use high-level objects from the SSM library, these high-level objects must be imported or exported as raw *address* and *size* pairs to interface with legacy code. When there is some need to manipulate memory using *address* and *size* pairs in the "glue" code between the new and legacy code, the C11K functions are a reasonable alternative to the old unsafe functions.

However, the C11 standard states that the Annex K is optional. In fact, most C11 implementations, including the GNU C library, do not include the `_s` functions[1]. As a workaround, the SSM library implements a subset of the C11K functions.

8.2 C11 Annex K Support

The subset of the C11K which is implemented in the SSM library provides the basic services to manipulate strings and memory buffers. There is no support for higher-level services which are defined in C11K, namely file handing services, multi-bytes character strings or sorting functions.

The following table lists the C11K subset which is supported in the SSM library.

Declaration	**C11 Reference**	**Declaration**	**C11 Reference**
`errno_t`	K.3.2	`strcat_s`	K.3.7.2.1
`rsize_t`	K.3.3	`strncat_s`	K.3.7.2.2
`RSIZE_MAX`	K.3.4	`memset_s`	K.3.7.4.1
`memcpy_s`	K.3.7.1.1	`strerror_s`	K.3.7.4.2
`memmove_s`	K.3.7.1.2	`strerrorlen_s`	K.3.7.4.3
`strcpy_s`	K.3.7.1.3	`strlen_s`	K.3.7.4.4
`strncpy_s`	K.3.7.1.4		

Please also note the following precisions or restrictions concerning the C11K subset in the SSM library, compared to the C11 standard:

- C11K specifies that the `_s` functions are defined in the various standard headers. In the SSM library, they are defined in `ssm.h`.
- C11K specifies that the `_s` functions may optionally set `errno` on error. The SSM library does not set `errno`.
- C11K specifies that the `_s` functions return `errno_t` values but no precise value is defined for any error case. The only requirement is that zero is returned on success. The `_s` functions from the SSM library return `errno_t` values which are numerically identical to

[1] The fact that the `_s` functions were originally defined by Microsoft as a proprietary extension is a common explanation.

the `ssm_status_t` enumeration values. This is compatible whith the C11K standard since `SSM_OK` is zero.

- There is no support for "runtime-constraint handlers". The function `set_constraint_handler_s` is not provided.

⋆ `SSM_C11K`

To be defined in the application to enable the standard naming from C11K.

```
#define SSM_C11K
#include "ssm.h"
```

To avoid any interference with another implementation of C11K which could be used in other modules of the application, all defined symbols follow the SSM naming rules: they start with the prefix `ssm_` (functions) or `SSM_` (constants). For instance, the function `strcpy_s` is in fact `ssm_strcpy_s`.

To write "standard" C11 code, define the symbol `SSM_C11K` before including `ssm.h`, either as a project option (when possible by the development tools) or in the source file as illustrated above.

When `SSM_C11K` is defined, the standard C11K names are defined as macros. Thus, the function `strcpy_s` becomes available.

8.3 C11 Annex K Reference

This chapter describes the `_s` functions which are implemented in the SSM library. This is mainly a copy from the C11 standard (ISO/IEC 9899:2011) Annex K.

⋆ `ssm_errno_t` **and** `errno_t`

A integer type for error codes.

```
typedef int ssm_errno_t;
typedef int     errno_t;  /* with SSM_C11K */
```

C11K adds this type for clarity and specifies it as `int`. The usual error codes in the C library use `int` directly. It is redefined in SSM since it is part of most C11K function profiles.

⋆ `ssm_rsize_t` **and** `rsize_t`

An integer type for data sizes.

```
typedef size_t ssm_rsize_t;
typedef size_t     rsize_t;  /* with SSM_C11K */
```

C11K specifies this type as `size_t`. It is unclear why this was necessary. It is redefined in SSM since it is part of most C11K function profiles.

⋆ `SSM_RSIZE_MAX` **and** `RSIZE_MAX`

Maximum size of strings and memory buffers.

```
#define SSM_RSIZE_MAX  ((size_t)SSM_SIZE_MAX)
#define     RSIZE_MAX  ((size_t)SSM_SIZE_MAX)  /* with SSM_C11K */
```

C11K specifies `RSIZE_MAX` with the same semantics as `SSM_SIZE_MAX`, a simple and pragmatic way of detecting incorrectly computed sizes. The only difference is that `RSIZE_MAX` must expand to a value of type `size_t`. It is unclear why C11K specifies it as `size_t` and not `rsize_t`, both types being equivalent in fact.

⋆ `ssm_memmove_s` and `memmove_s`

Copy a memory area.

```
ssm_errno_t ssm_memmove_s (void* s1, ssm_rsize_t s1max,
                           const void* s2, ssm_rsize_t n);

errno_t memmove_s (void* s1, rsize_t s1max, const void* s2, rsize_t n);
```

Parameter	Mode	Description
s1	out	Destination buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination buffer. Cannot be greater than `RSIZE_MAX`.
s2	in	Source buffer. Cannot be `NULL`.
n	in	Number of bytes to copy. Cannot be greater than *s1max*.
	return	A status value.

The `memmove_s` function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. This copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* or *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*.

If there is a runtime-constraint violation, the `memmove_s` function stores zeros in the first *s1max* characters of the object pointed to by *s1* if *s1* is not a null pointer and *s1max* is not greater than `RSIZE_MAX`.

⋆ `ssm_memcpy_s` and `memcpy_s`

Copy a memory area.

```
ssm_errno_t ssm_memcpy_s (void* s1, ssm_rsize_t s1max,
                          const void* s2, ssm_rsize_t n);

errno_t memcpy_s (void* s1, rsize_t s1max, const void* s2, rsize_t n);
```

Parameter	Mode	Description
s1	out	Destination buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination buffer. Cannot be greater than `RSIZE_MAX`.
s2	in	Source buffer. Cannot be `NULL`.
n	in	Number of bytes to copy. Cannot be greater than *s1max*.
	return	A status value.

In C11K, the `memcpy_s` function has the same specification as the `memmove_s` function, except that the objects pointed to by *s1* or *s2* cannot overlap. In the SSM library, the `memcpy_s` function allows overlapping areas. So, in practice, the `memcpy_s` and `memmove_s` functions are identical.

⋆ ssm_memset_s and memset_s

Initialize a memory area with a given value.

```
ssm_errno_t ssm_memset_s (void* s, ssm_rsize_t smax, int c, ssm_rsize_t n);

errno_t memset_s (void* s, rsize_t smax, int c, rsize_t n);
```

Parameter	Mode	Description
s	out	Destination buffer. Cannot be `NULL`.
smax	in	Maximum size of the destination buffer. Cannot be greater than `RSIZE_MAX`.
c	in	The value to set in each byte of the destination buffer.
n	in	Number of bytes to initialize. Cannot be greater than *smax*.
	return	A status value.

The `memset_s` function copies the value of *c* (converted to an `unsigned char`) into each of the first *n* characters of the object pointed to by *s*. Unlike `memset`, any call to the `memset_s` function shall be evaluated strictly according to the rules of the abstract machine as described in C11 5.1.2.3. That is, any call to the `memset_s` function shall assume that the memory indicated by *s* and *n* may be accessible in the future and thus must contain the values indicated by *c*.

If there is a runtime-constraint violation, then if *s* is not a null pointer and *smax* is not greater than `RSIZE_MAX`, the `memset_s` function stores the value of *c* (converted to an `unsigned char`) into each of the first *smax* characters of the object pointed to by *s*.

⋆ ssm_strcpy_s and strcpy_s

Copy a null-terminated string.

```
ssm_errno_t ssm_strcpy_s (char* s1, ssm_rsize_t s1max, const char* s2);

errno_t strcpy_s (char* s1, rsize_t s1max, const char* s2);
```

Parameter	Mode	Description
s1	out	Destination string buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination string buffer. Cannot be greater than `RSIZE_MAX`. Shall be greater than `strnlen_s(`*s2*`, `*s1max*`)`.
s2	in	Source string. Cannot be `NULL`.
	return	A status value.

The `strcpy_s` function copies the string pointed to by *s2* (including the terminating null character) into the array pointed to by *s1*.

All elements following the terminating null character (if any) written by `strcpy_s` in the array of *s1max* characters pointed to by *s1* take unspecified values when `strcpy_s` returns.

C11K specifies that copying shall not take place between objects that overlap. In the SSM library, the `strcpy_s` function allows overlapping areas.

If there is a runtime-constraint violation, then if *s1* is not a null pointer and *s1max* is greater than zero and not greater than `RSIZE_MAX`, then `strcpy_s` sets *s1*`[0]` to the null character.

⋆ ssm_strncpy_s and strncpy_s

Copy a null-terminated string with bounded size.

```
ssm_errno_t ssm_strncpy_s (char* s1, ssm_rsize_t s1max,
                           const char* s2, ssm_rsize_t n);

errno_t strncpy_s (char* s1, rsize_t s1max, const char* s2, rsize_t n);
```

Parameter	Mode	Description
s1	out	Destination string buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination string buffer. Cannot be greater than `RSIZE_MAX`. Shall be greater than `strnlen_s(s2, s1max)`.
s2	in	Source string. Cannot be `NULL`.
n	in	Maximum number of characters to copy, not including the trailing null-character. If *n* is not less than *s1max*, then *s1max* shall be greater than `strnlen_s(s2, s1max)`.
	return	A status value.

The `strncpy_s` function copies not more than *n* successive characters (characters that follow a null character are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If no null character was copied from *s2*, then *s1*`[n]` is set to a null character.

All elements following the terminating null character (if any) written by `strncpy_s` in the array of *s1max* characters pointed to by *s1* take unspecified values when `strncpy_s` returns.

The `strncpy_s` function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array

C11K specifies that copying shall not take place between objects that overlap. In the SSM library, the `strncpy_s` function allows overlapping areas.

If there is a runtime-constraint violation, then if *s1* is not a null pointer and *s1max* is greater than zero and not greater than `RSIZE_MAX`, then `strncpy_s` sets *s1*`[0]` to the null character.

⋆ `ssm_strcat_s` and `strcat_s`

Concatenate two null-terminated strings.

```
ssm_errno_t ssm_strcat_s (char* s1, ssm_rsize_t s1max, const char* s2);

errno_t strcat_s (char* s1, rsize_t s1max, const char* s2);
```

Parameter	Mode	Description
s1	in,out	Destination string buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination string buffer. Cannot be greater than `RSIZE_MAX`. Cannot be zero.
s2	in	Source string. Cannot be `NULL`.
	return	A status value.

The `strcat_s` function appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The initial character from *s2* overwrites the null character at the end of *s1*.

All elements following the terminating null character (if any) written by `strcat_s` in the array of *s1max* characters pointed to by *s1* take unspecified values when `strcat_s` returns.

Let *m* denote the value *s1max* - `strnlen_s(s1, s1max)` upon entry to `strcat_s`. *m* shall not equal zero. *m* shall be greater than `strnlen_s(s2, m)`.

C11K specifies that copying shall not take place between objects that overlap. In the SSM library, the `strcat_s` function allows overlapping areas.

If there is a runtime-constraint violation, then if *s1* is not a null pointer and *s1max* is greater than zero and not greater than `RSIZE_MAX`, then `strcat_s` sets *s1*`[0]` to the null character.

⋆ ssm_strncat_s and strncat_s

Concatenate two null-terminated strings with bounded size.

```
ssm_errno_t ssm_strncat_s (char* s1, ssm_rsize_t s1max,
                           const char* s2, ssm_rsize_t n);

errno_t strncat_s (char* s1 rsize_t s1max, const char* s2, rsize_t n);
```

Parameter	Mode	Description
s1	in,out	Destination string buffer. Cannot be `NULL`.
s1max	in	Maximum size of the destination string buffer. Cannot be greater than `RSIZE_MAX`. Cannot be zero.
s2	in	Source string. Cannot be `NULL`.
n	in	Maximum number of characters to copy, not including the trailing null-character. Cannot be greater than `RSIZE_MAX`.
	return	A status value.

Let *m* denote the value *s1max* - `strnlen_s(`*s1*, *s1max*`)` upon entry to `strcat_s`. *m* shall not equal zero. If *n* is not less than *m*, then *m* shall be greater than `strnlen_s(`*s2*, *m*`)`.

The `strncat_s` function appends not more than *n* successive characters (characters that follow a null character are not copied) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial character from *s2* overwrites the null character at the end of *s1*. If no null character was copied from *s2*, then *s1*`[`*s1max*-*m*+`n]` is set to a null character.

All elements following the terminating null character (if any) written by `strncat_s` in the array of *s1max* characters pointed to by *s1* take unspecified values when `strncat_s` returns.

C11K specifies that copying shall not take place between objects that overlap. In the SSM library, the `strcat_s` function allows overlapping areas.

If there is a runtime-constraint violation, then if *s1* is not a null pointer and *s1max* is greater than zero and not greater than `RSIZE_MAX`, then `strncat_s` sets *s1*`[0]` to the null character.

⋆ ssm_strnlen_s and strnlen_s

Get the length of a null-terminated string with bounded size.

```
size_t ssm_strnlen_s (const char* s, size_t maxsize);

size_t strnlen_s (const char* s, size_t maxsize);
```

Parameter	Mode	Description
s	in	String to get the length of.
maxsize	in	Maximum number of characters to check.
	return	Length of the string or zero if *s1* is `NULL`.

The `strnlen_s` function computes the length of the string pointed to by *s*.

The `strnlen_s` function returns the number of characters that precede the terminating null character. If there is no null character in the first *maxsize* characters of *s* then `strnlen_s` returns *maxsize*. At most the first *maxsize* characters of *s* shall be accessed by `strnlen_s`.

⋆ ssm_strerror_s and strerror_s

Map an error code to a string value.

```
ssm_errno_t ssm_strerror_s (char* s, ssm_rsize_t maxsize, ssm_errno_t errnum);

errno_t strerror_s (char* s, rsize_t maxsize, errno_t errnum);
```

Parameter	Mode	Description
s	out	The string buffer to receive the message string. Cannot be `NULL`.
maxsize	in	Maximum size of the string buffer. Cannot be zero, cannot be greater than `RSIZE_MAX`.
errnum	in	The error code to get a description of.
	return	A status value.

The `strerror_s` function maps the number in *errnum* to a locale-specific message string. Typically, the values for *errnum* come from `errno`, but `strerror_s` shall map any value of type `int` to a message.

Restriction: The SSM library implementation of `strerror_s` does not return locale-specific messages. All returned messages are in English. Furthermore, only the `errno_t` values returned by the C11K functions of the SSM library are recognized. General system error code from `errno` are not valid values for *errnum*. Any *errnum* which does not correspond to a known `ssm_status_t` value will return the same `"Unknown"` string.

If the length of the desired string is less than *maxsize*, then the string is copied to the array pointed to by *s*.

Otherwise, if *maxsize* is greater than zero, then *maxsize*`-1` characters are copied from the string to the array pointed to by *s* and then *s*`[`*maxsize*`-1]` is set to the null character. Then, if *maxsize* is greater than 3, then *s*`[`*maxsize*`-2]`, *s*`[`*maxsize*`-3]` and *s*`[`*maxsize*`-4]` are set to the character period (`.`).

If there is a runtime-constraint violation, then the array (if any) pointed to by *s* is not modified.

⋆ ssm_strerrorlen_s and strerrorlen_s

Get the length of the string that maps an error code.

```
size_t ssm_strerrorlen_s (ssm_errno_t errnum);

size_t strerrorlen_s (errno_t errnum);
```

Parameter	Mode	Description
errnum	in	The error code to get a description of.
	return	Number of characters (not including the null character) in the full message string.

The `strerrorlen_s` function calculates the length of the (untruncated) locale-specific message string that the `strerror_s` function maps to *errnum*.

Restriction: The SSM library implementation of `strerrorlen_s` has the same restrictions as `strerror_s`.

Index

E

M

R

S