SafeZone™
SafeZone Framework v5.2

Reference Manual

Document Revision: B
Document Date: 2013-02-14
Document Number: 007-914520-305
Document Status: Accepted

INSIDE Secure B.V.
Boxtelseweg 26A
5261 NE Vught
The Netherlands
Phone: +31-73-6581900
Fax: +31-73-6581999
http://www.insidesecure.com/

For further information contact: ESSEmbeddedHW-Support@insidesecure.com

Revision History

Doc Rev	Page(s) Section(s)	Date (Y-M-D)	Author	Purpose of Revision
A	All	2011-08-31	RWI MHO	• Created this document from SafeZone Framework Reference Manual v5.1 Rev A. • Template update and small editorial changes. • Removed the TODO macro. • Added the UNREACHABLE and PARAMETER_CHECK macros. • Added chapter describing C Lib Abstraction API.
B	All	2013-02-14	FvdM	• Update template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 *Purpose*

SafeZone software implements system security middleware components for embedded systems. For portability and small footprint, SafeZone software is implemented in the C language. Most of the SafeZone software is implemented on top of the SafeZone Framework described in this document.

1.2 *Scope*

This document covers the SafeZone Framework, which is shared between all SafeZone Products. This manual describes the interfaces that are provided by the SafeZone Framework implementation.

1.3 *Related Documents*

The following documents are part of the documentation set.

Ref.	Document Name	Document Number
[1]	SafeZone Framework Reference Manual (this document)	007-914520-305
[2]	SafeZone Framework Porting Guide	007-914520-304
[3]	Software Unit Testing Framework User Guide	007-918110-309

This information is correct at the time of document release. INSIDE Secure reserves the right to update the related documents without updating this document. Please contact INSIDE Secure for the latest document revisions.

For more information or support, please go to https://essoemsupport.insidesecure.com/ for our online support system. In case you do not have an account for this system, please ask one of your colleagues who already has an account to create one for you or send an e-mail to ESSEmbeddedHW-Support@insidesecure.com.

2 SafeZone Framework

SafeZone Framework consists of the following sub-modules that each provides an interface:

- Public definitions
- Implementation definitions
- Debug support
- C library abstraction
- Execution Environment Identifier Library (EE_ID)
- SafeZone Platform Abstraction Layer (SPAL)
    - Dynamic memory management library
    - Mutex support
    - Threading support library
    - Semaphores
    - Time Management

The relationships between these sub-modules, the operating system and applications are illustrated in Figure 1. A closer look at each of these modules is available in the following sections.

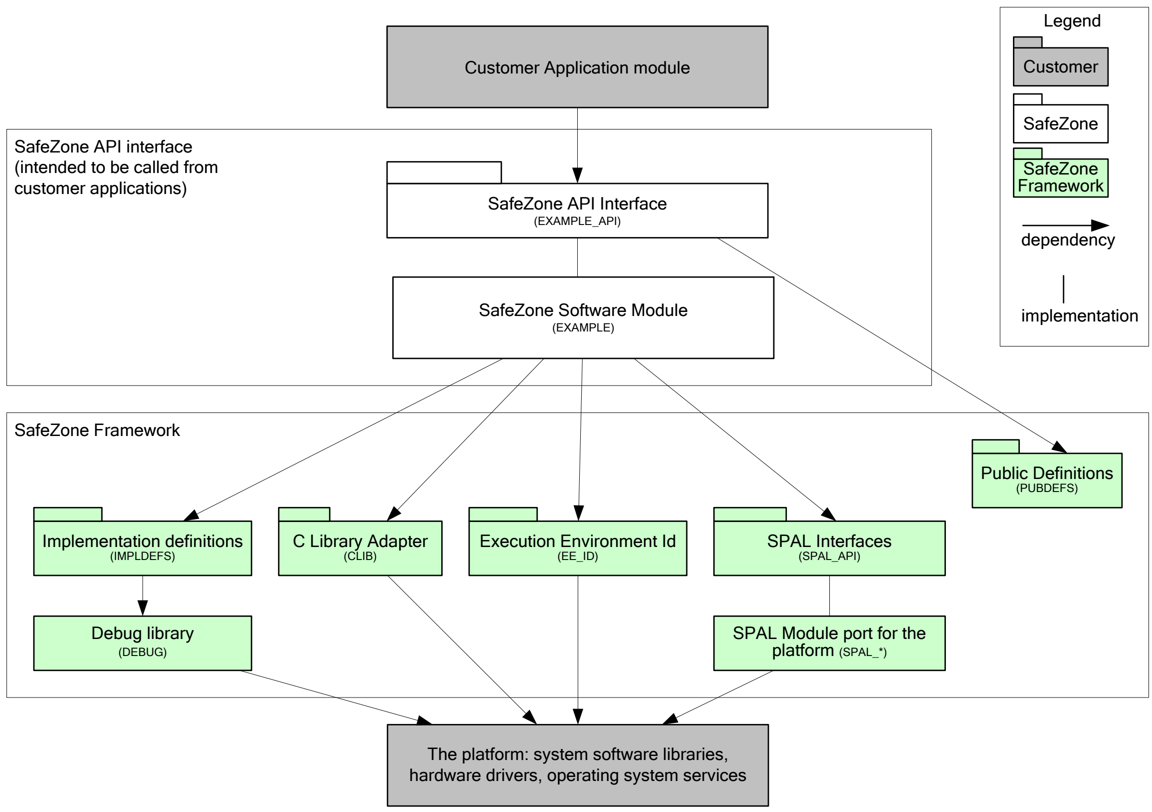

Figure 1 Relationship between SafeZone Framework and the rest of the system.

2.1 *Public definitions*

SafeZone software uses data types that have specific storage sizes. This is primarily to make it easy to calculate the resource usage of certain structures. Added benefit is the fact that it is harder to make programming mistakes related to different data type sizes on different platforms.

Because the sizes are required by SafeZone APIs and related data structures, those definitions need to be available for API definition headers.

2.2 Implementation definitions

Implementation definitions are the common framework for the SafeZone library code. It provides macros that allow using certain idioms (such as `PARAMETER_NOT_USED`), but also to allow the SafeZone software, in co-operation with the compiler, to do quality checking that goes beyond ISO C99.

2.3 DEBUG library

The DEBUG library is a simple helper library for Implementation Definitions. The use of `L_DEBUG()`, `L_TRACE()`, `ASSERT()` or `PANIC()` can result in a calls to this sub-module.

2.4 CLIB

CLIB, the C library adapter, contains a selection of functions from the standard C header files `string.h` and `ctype.h`. Most of SafeZone software avoids including the standard header files directly, and uses CLIB instead.

2.5 Execution Environment Identifier (EE_ID)

The execution environment identifier library deals with three objects:

- Execution Environment Identifier (Execution Environment Id or EE Id).
- Application Identifier (Application Id).
- Global Application Identifier (Global Application Id) that combines both preceding identifiers.

The Execution Environment Identifier library takes care of providing basic operations for dealing with these identifiers, including querying the identifier for the current application and comparing identifiers. The sizes of these identifiers are also provided via the API, to allow allocating enough storage space where these identifiers are used.

2.6 SPAL

SPAL contains libraries to deal with dynamic memory management, multithreading, mutual exclusion and time management.

3 The Framework APIs

3.1 How to read this manual

This section contains API references for the SafeZone Framework.

3.1.1 Definitions

Caller	The program code that is using the specified API.
valid pointer	In this document a "valid pointer" is a pointer that is not `NULL`, and points to a memory region of at least the size of the type the pointer points to. For void pointers the size is 1 (one) in this specification. If the pointer is a pointer to const the memory area must be readable, otherwise it must be both readable and writable.

3.1.2 C Language definitions

This section describes how to interpret the API definitions in this document. The C Language definitions show the exact definitions of the item in C. The same definition shall be used in the C headers of the API.

3.1.3 Function definitions

This document specifies each function in its own subsection. Each subsection defining a function may define:

- C Language function prototype of the function
  This shows the declaration of the functions as it shall be after inclusion of the defining header file. The exact types of return values and parameters are also clearly visible.
- Pre-conditions of the function
  This defines the set of conditions that must be true when the function is called. The implementation of the function may not check for these conditions when compiled for production use. An implementation should implement a compile time configuration option to enable checks for the preconditions when possible (some preconditions cannot be tested in a reliable fashion and therefore their checking might not be possible). If preconditions are not true when function is called the result of the call is undefined.
- Post-conditions of the function
  This defines the set of conditions that apply after the function has successfully returned (or they contain remark that the function is never supposed to return). A function that does not have a return value always returns successfully (when preconditions apply). A function with a return value returns successfully when the return value so indicates.
- Exception of the function
  This lists the set of unsuccessful results that a function can return. For each result the value and conditions are given. The defined list is exhaustive, that is, a function implementation shall not return any other result than listed in this specification.
- Rationale for the function
  This is an optional part of the function definition. It attempts to rationalize some part of the function behavior where, for example multiple different behaviors would have been acceptable.

3.2 Public Definitions

The public definitions are provided by header `public_defs.h`. Inclusion of the header to a source file must provide definitions for:

- ISO C99 exact-width integer types and their limits,
- ISO C99 `boolean` concept,
- `restrict` keyword.

3.2.1 Exact-width integers types

The public definitions shall include following exact width integer types:

- `int64_t`
- `int32_t`
- `int16_t`
- `int8_t`
- `uint64_t`
- `uint32_t`
- `uint16_t`
- `uint8_t`
- `uintptr_t`

Also, corresponding limit macros `INT`***N***`_MIN`, `INT`***N***`_MAX`, and `UINT`***N***`_MAX`, where ***N*** is one of 8, 16, 32 and `UINTPTR_MAX` must be present. For parts of the software, also 64-bit type is required.

On platforms providing ISO C99 headers this can be achieved by including `stdint.h` from the public definitions header, assuming the platform has suitable integer types.

3.2.2 Boolean

The public definitions shall include definition of ISO C99 boolean macros: `bool`, `true`, and `false`. If the `stdbool.h` header is not present, `public_defs.h` must define these values itself.

3.2.3 Restrict keyword

ISO C99 `restrict` keyword is used in SafeZone interfaces to give the compiler hints that a specific pointer must be unique. This feature can be used by the compiler for more efficient code as well as for better compile time analysis.

Some compilers do not provide standard restrict keyword, but instead `_restrict` or `__restrict`. Code resembling:

```
#define __restrict restrict
```

Must be used for compilers without support for the `restrict` keyword. `restrict` shall be defined as blank on environments with no support for the keyword.

Note: *This definition of the `restrict` keyword is not needed when the SafeZone Build System is used because the build system detects automatically if the compiler supports the restrict variant.*

3.3 Implementation definitions

The implementation definitions are provided by the header file `implementation_defs.h`. It must provide definitions for:

- Possibility to define functions as `static inline`
- `PARAMETER_NOT_USED` macro,
- `PARAMETER_CHECK` macro,
- `PRECONDITION` macro,
- `POSTCONDITION` macro,
- `PANIC` macro,
- `COMPILE_GLOBAL_ASSERT` macro,
- `COMPILE_STATIC_ASSERT` macro,
- `UNREACHABLE macro.`

The following subsections describe the provided definitions.

3.3.1 static inline

The implementation files including module internal headers may define functions as static inline for optimization purposes.

The implementation definitions header should define the `inline` keyword in a way that functions defined "`static inline`":

- produce *inline functions* if compiler *optimizations* allow this,
- do not produce compiler warnings,
- do not produce compiler errors,
- produce static functions, i.e. "`static inline`" functions can be defined in header files that are included in multiple compilation units.

3.3.2 PARAMETER_NOT_USED

The `PARAMETER_NOT_USED` macro is used to mark parameters of functions that are not used by the implementation of the functions. This serves two purposes:

- prevention of compiler warnings for such parameters, and
- documenting that the parameter is ignored on purpose.

The `PARAMETER_NOT_USED` macro shall take one macro argument that is the name of the parameter that is not used by the implementation. A definition that works on some compilers is:

```
#define PARAMETER_NOT_USED(__p) if (__p) {}
```

This definition makes the compiler think as the parameter actually was used without affecting the behavior of the function and producing code that the compiler can optimize out.

3.3.3 PARAMETER_CHECK

This macro can be used to insert optional code that checks a function parameter and returns with a specified value when the parameter does not fulfill the condition. When activated, the macro implementation looks like this:

```
#define PARAMETER_CHECK(__condition, __ret_val)  \
    if ((__condition)) return (__ret_val)
```

The macro can be used like this:

```
int foo(void * bar_p)
{
    PARAMETER_CHECK(bar_p == NULL, -1);
}
```

3.3.4 PRECONDITION

The `PRECONDITION` macro is used to write assertions on conditions that should hold when the function is called. The macro takes one macro argument: an expression that evaluates to true when a required condition holds. For a debug build this should expand to a run-time check that aborts the execution of the software when the expression evaluates to false.

The macro can be used multiple times. All macro calls should appear in the function body before any functional code.

Example usage:

```
int foo(void * bar_p)
{
    PRECONDITION(bar_p != NULL);
}
```

3.3.5 POSTCONDITION

The `POSTCONDITION` macro is used to write assertions on conditions that should hold after the function. The macro takes one macro argument: an expression that evaluates to true when a required condition holds. For a debug build this should expand to a run-time check that aborts the execution of the software when the expression evaluates to false.

The macro shall be used just before return statements in the code. Each return statement should have its own post-conditions specified. Each return statement may have one or more `POSTCONDITION` macro calls before it.

3.3.6 PANIC

The `PANIC` macro is used on code branches that should never be executed. A call to the `PANIC` macro shall abort the execution of the software. The `PANIC` macro receives formatting arguments in `printf` compatible format. The parameters are provided for convenience and it may be useful for debugging to print them out. However, in the final system it might even be better to not print `PANIC` messages, in an unfortunate event they may reveal something useful to an attacker. Also, the `PANIC` macro may be called in a context where printing the message is impossible, therefore the machine may halt instead of producing the message.

3.3.7 COMPILE_GLOBAL_ASSERT

The `COMPILE_GLOBAL_ASSERT` macro is used for testing conditions at compile time. The macro is to be used only in the global scope and not within function bodies, see section 3.3.8. The macro takes two arguments, a description variable name and an expression that should evaluate to true when a required condition holds and to false otherwise. The description should be a valid C identifier to enable using it as a part of variable name in the implementation of the macro. When the specified expression evaluates to false the macro shall result in a compilation error. The compilation error might not be very descriptive, though. One possible implementation of `COMPILE_GLOBAL_ASSERT` is:

```
#define COMPILE_GLOBAL_ASSERT(description, condition) \
    extern int global_assert_##description[1 - 2*(!(condition))]
```

This implementation causes a declaration of an integer array of size -1 when the condition evaluates to false. For conditions that evaluate to true the macro declares an external array of size 1.

3.3.8 COMPILE_STATIC_ASSERT

The `COMPILE_STATIC_ASSERT` macro is used for testing conditions on compile time. The macro is to be used only within function bodies and not in the global scope. The macro takes two arguments, a description variable name and an expression that should evaluate to true when a required condition holds and to false otherwise. The description should be a valid C identifier to enable using it as a part of variable name in the implementation of the macro. The macro should produce a single C statement with no effect on the execution of the code. When the specified expression evaluates to false the macro shall result in a compilation error. The compilation error might not be very descriptive, though. One possible implementation of `COMPILE_STATIC_ASSERT` is:

```
#define COMPILE_STATIC_ASSERT(description, condition) \
do { \
int static_assert_##description_var[1-2*(!(condition))]; \
} while (0)
```

This implementation causes a declaration of an integer table of size -1 when the condition evaluates to false. For conditions that evaluate to true the macro declares an external table of size 1.

3.3.9 UNREACHABLE

The UNREACHABLE macro can be used in the code to indicate that certain paths are intentionally not reachable. This can be used to avoid compiler warning.

3.4 C Lib Abstraction

The C library abstraction API avoids direct dependencies on C Run-time Library functions in the code, thereby simplifying porting. This API also limits the number of C runtime library functions used in the code.

Table 1 shows the C Lib Abstraction API function name and the corresponding C Run-time Library function name with compatible prototype.

Table 1: C Lib API functions overview

Function	CRT original function name
c_memcpy	memcpy
c_memmove	memmove
c_memset	memset
c_memcmp	memcmp
c_strcmp	strcmp
c_strcpy	strcpy
c_strcat	strcat
c_strncpy	strncpy
c_strncmp	strncmp
c_strlen	strlen
c_strstr	strstr
c_strtol	strtol
c_strchr	strchr
c_tolower	tolower
c_toupper	toupper
c_memchr	memchr

3.5 EE_ID API Reference

Definitions for Execution Environment (EE) identification and EE Application Id objects.

Each application has an Application Id, which distinguishes applications within the same Execution Environment from each other. The suitable length for the Application Id is dictated by its use. The current SafeZone software uses a Universally Unique Identifier (UUID) of 16 bytes, which makes it very unlikely that collisions happen unknowingly. The EE_ID API by itself does not prevent collisions from happening, instead that is the responsibility of the Execution Environment if it desires to provide enforced application separation. To configure the Application Id to be smaller, e.g. to conserve a tiny amount of memory space, you need to modify the definitions in this header file.

`EE_Id` and `EE_ApplicationId` are considered plain-byte arrays from the perspective of implementing `EE_Id` and `EE_ApplicationId`, i.e. you may initialize them as you like.

This library contains facilities to deal with these identifiers.

3.5.1 Data Structures

- struct ***EE_Id_t***
  Structure to store a (Task) Execution Environment Id.
- struct ***EE_ApplicationId_t***
  Structure to store an Application Id.
- struct ***EE_GlobalApplicationId_t***
  Structure to store the Global Application Id (the Execution Environment Id and Application Id pair).

3.5.2 Defines

- `#define EE_ID_SIZE 1`
  The Execution Environment Id size.
- `#define EE_APPLICATION_ID_SIZE 16`
  The Execution Environment Application Id size.
- ```
  #define GLOBAL_APPLICATION_ID_ENCODED_SIZE \
      ((EE_ID_SIZE) + (EE_APPLICATION_ID_SIZE))
  ```
  Define a constant that tells how much space is needed by the Global Execution Environment Application Id in encoded format.

3.5.3 Details of defines

3.5.3.1 EE_APPLICATION_ID_SIZE

```
#define EE_APPLICATION_ID_SIZE 16
```

The Execution Environment Application Id size.

Identifier for the application within an Execution Environment.

16-byte long identifiers allow UUIDs to be used as application identifiers and therefore prevent accidental identifier collisions.

3.5.3.2 EE_ID_SIZE

```
#define EE_ID_SIZE 1
```

The Execution Environment Id size.

For typical embedded devices a single byte is enough (up to 256 distinct execution environments).
```

3.5.3.3 *GLOBAL_APPLICATION_ID_ENCODED_SIZE*

```
#define GLOBAL_APPLICATION_ID_ENCODED_SIZE \
    ((EE_ID_SIZE) + (EE_APPLICATION_ID_SIZE))
```

Defines a constant that tells how much space is needed by the Global Execution Environment Application Id in encoded format.

This is the sum of the Execution Environment and Application Id sizes.

3.5.4 Functions

EE_ID provides following functions:

- `EE_GetId`
- `EE_GetApplicationId`
- `EE_GetGlobalApplicationId`
- `EE_GlobalApplicationId_Build`
- `EE_GlobalApplicationId_Encode`
- `EE_GlobalApplicationId_Decode`
- `EE_SetGlobalApplicationId`
- `EE_SetApplicationId`
- `EE_Id_Cmp`
- `EE_ApplicationId_Cmp`
- `EE_GlobalApplicationId_Cmp`
- `EE_Id_Eq`
- `EE_ApplicationId_Eq`
- `EE_GlobalApplicationId_Eq`

3.5.4.1 *EE_ApplicationId_Cmp*

```
int
EE_ApplicationId_Cmp(
    const EE_ApplicationId_t *const ApplicationId_1_p,
    const EE_ApplicationId_t *const ApplicationId_2_p)
```

Compare two Execution Environment Application Ids.

The definition of the return values of this function is equivalent to `memcmp()`. However, as this function may not perform direct memory comparison, the results may be something different than a straight `memcmp()` over the structures. At the very least this function will ignore any padding within the structures.

Note: *In the future, this function is likely to be inlined and/or macroized for smaller code size. Please take suitable precautions for macro-like side-effects for any parameters you pass to this function.*

Parameters:

ApplicationId_1_p	[in]	First Application Id.
ApplicationId_2_p	[in]	Second Application Id.

Returns:

The function returns zero if the identifiers are the same. If the returned value is smaller than zero, then the first identifier can be considered to be of smaller order than the second one. The reverse is true when the return value is positive.

3.5.4.2 EE_ApplicationId_Eq

```
bool
EE_ApplicationId_Eq(
    const EE_ApplicationId_t *const ApplicationId_1_p,
    const EE_ApplicationId_t *const ApplicationId_2_p)
```

Test if two Execution Environment Application Ids are equivalent.

Parameters:

ApplicationId_1_p	[in]	First Application Id.
ApplicationId_2_p	[in]	Second Application Id.

Returns:

This function returns `true` if the first identifier is equivalent to the second identifier, meaning that this function returns `true` only if the equivalent `EE_*Id_Cmp` function returns zero.

3.5.4.3 EE_GetApplicationId

```
const EE_ApplicationId_t *
EE_GetApplicationId(void)
```

Retrieve the pointer to the current Application Id.

Note: *This function always succeeds.*

Returns:

The pointer to the current Application Id.

3.5.4.4 EE_GetGlobalApplicationId

```
const EE_GlobalApplicationId_t *
EE_GetGlobalApplicationId(void)
```

Retrieve the pointer to the current Global Application Id.

The result is the combination of current Execution Environment and Application Ids.

Note: *This function always succeeds.*

Returns:

The pointer to the current Global Application Id.

3.5.4.5 EE_GetId

```
const EE_Id_t *
EE_GetId(void)
```

Retrieve the pointer to the current Execution Environment Id.

Note: *This function always succeeds.*

Returns:

The pointer to the current Execution Environment Id.

3.5.4.6 EE_GlobalApplicationId_Build

```
void
EE_GlobalApplicationId_Build(
    EE_GlobalApplicationId_t *const GlobalApplicationId_p,
    const EE_Id_t *const            EEId_p,
    const EE_ApplicationId_t *const ApplicationId_p)
```

Build the Global Application Id out of Execution Environment Id and Application Id.

Parameters:

GlobalApplicationId_p	[out]	The Global Application Id shall be built here.
EEId_p	[in]	The Pointer to Execution Environment Id.
ApplicationId_p	[in]	The Pointer to the Application Id.

Returns:

N/A

3.5.4.7 EE_GlobalApplicationId_Cmp

```
int
EE_GlobalApplicationId_Cmp(
    const EE_GlobalApplicationId_t *const GlobalApplicationId_1_p,
    const EE_GlobalApplicationId_t *const GlobalApplicationId_2_p)
```

Compare two Global Application Ids.

The definition of the return values of this function is equivalent to `memcmp()`. However, as this function may not perform direct memory comparison, the results may be something different than a straight `memcmp()` over the structures. At the very least this function will ignore any padding within the structures.

Note: *In the future, this function is likely to be inlined and/or macroized for smaller code size. Please take suitable precautions for macro-like side-effects for any parameters you pass to this function.*

Parameters:

GlobalApplicationId_1_p	[in]	First Global Application Id.
GlobalApplicationId_2_p	[in]	Second Global Application Id.

Returns:

The function returns zero if the identifiers are the same. If the returned value is smaller than zero, then the first identifier can be considered to be of smaller order than the second one. The reverse is true when the return value is positive.

3.5.4.8 *GlobalApplicationId_Decode*

```
bool
EE_GlobalApplicationId_Decode(
    EE_GlobalApplicationId_t *const GlobalApplicationId_p,
    const uint8_t *const            EncodedGlobalApplicationId_p,
    const uint32_t                  EncodedGlobalApplicationIdLen)
```

Decode Global Application Id from opaque data.

Reverse effect of the `EE_GlobalApplicationId_Encode()` function.

Parameters:

GlobalApplicationId_p	[out]	`EE_GlobalApplicationId_t` pointer to receive the decoded Global Application Id.
EncodedGlobalApplicationId_p	[in]	Pointer to byte array that holds the encoded Global Application Id.
EncodedGlobalApplicationIdLen	[in]	Size of data provided in `EncodedGlobalApplicationId_p`.

Returns:

`true` if decoding was successful (the correct length was provided).

3.5.4.9 *EE_GlobalApplicationId_Encode*

```
bool
EE_GlobalApplicationId_Encode(
    const EE_GlobalApplicationId_t *const GlobalApplicationId_p,
    uint8_t *const                        EncodedGlobalApplicationId_p,
    uint32_t *const                       EncodedGlobalApplicationIdLen_p)
```

Express Global Application Id as opaque data.

The Global Application Id is typically expressed as a structure when it is kept in memory. Encoding it transforms it into a byte array that is at most as large as the typical memory presentation of `EE_GlobalApplicationId_t` (the exact size required is provided by the constant `GLOBAL_APPLICATION_ID_ENCODED_SIZE`). The byte array can be written to long-term storage and later transformed back into `EE_GlobalApplicationId`.

Parameters:

GlobalApplicationId_p	[in]	The global application identifier to be encoded shall be provided in this parameter.
EncodedGlobalApplicationId_p	[out]	The pointer to the byte array to receive Encoded Global Application Id.
EncodedGlobalApplicationIdLen_p	[in, out]	In goes the amount of memory space available where the parameter `EncodedGlobalApplicationId_p` points to and out comes the amount of space used or needed.

Returns:

`true` if encoding was successful (enough space was provided).

3.5.4.10 EE_GlobalApplicationId_Eq

```
bool
EE_GlobalApplicationId_Eq(
    const EE_GlobalApplicationId_t *const GlobalApplicationId_1_p,
    const EE_GlobalApplicationId_t *const GlobalApplicationId_2_p)
```

Test if two EE Global Application identifiers are equivalent.

Parameters:

GlobalApplicationId_1_p	[in]	The first global application identifier.
GlobalApplicationId_2_p	[in]	The second global application identifier.

Returns:

This function returns true if the first identifier is equivalent to the second identifier, i.e. this function returns true only if the equivalent `EE_*Id_Cmp()` function returns zero.

3.5.4.11 EE_Id_Cmp

```
int
EE_Id_Cmp(
    const EE_Id_t *const EEId_1_p,
    const EE_Id_t *const EEId_2_p)
```

Compare two EE identifiers.

The definition of the return values of this function is equivalent to `memcmp()`. However, as this function may not perform direct memory comparison, the results may be something different than a straight `memcmp()` over the structures. At the very least this function will ignore any padding within the structures.

Note: *In the future, this function is likely to be inlined and/or macroized for smaller code size. Take suitable precautions for macro-like side-effects for any parameters you pass to this function.*

Parameters:

EEId_1_p	[in]	The first execution environment identifier.
EEId_2_p	[in]	The second execution environment identifier.

Returns:

The function returns zero if the identifiers are the same. If the returned value is smaller than zero, then the first identifier can be considered to be of smaller order than the second one. The reverse is true when the return value is positive.

3.5.4.12 EE_Id_Eq

```
bool
EE_Id_Eq(
    const EE_Id_t *const EEId_1_p,
    const EE_Id_t *const EEId_2_p)
```

Test if two EE identifiers are equivalent.

Parameters:

EEId_1_p	[in]	The first execution environment identifier.
EEId_2_p	[in]	The second execution environment identifier.

Returns:

This function returns `true` if the first identifier is equivalent to the second identifier, i.e. this function returns `true` only if the equivalent `EE_*Id_Cmp()` function returns zero.

3.5.4.13 EE_SetGlobalApplicationId

```
void
EE_SetGlobalApplicationId(
    const EE_GlobalApplicationId_t *const GlobalApplicationId_p)
```

Set Global Application Id information.

In trusted environments, this Id/ApplicationId information is typically retrieved from application metadata that is protected with a signature, and therefore the data is immutable.

Note: *This call should only be used when the software execution environment is not able to enforce the link between software and its identity.*

Parameters:

GlobalApplicationId_p	[in]	The desired `EE_Id_t` and `Application_Id_t` pair.

Precondition:

1. `EE_SetGlobalApplicationId` has not been called previously.
2. `EE_Id_t` within provided `GlobalApplicationId_t` matches the identifier of the current execution environment.
3. `Application_Id_t` provided within `GlobalApplicationId_t` matches the current application. (As far as the current execution environment is able to check it.)

Returns:

N/A

3.6 SPAL General definitions

3.6.1 Types

```
typedef enum SPAL_ResultCodes SPAL_Result_t;
```

The type `SPAL_Result_t` is the common return value type for the functions of all SPAL Interfaces that have a return value.

3.6.2 SPAL Result Codes

The result codes that can be returned by SPAL functions are defined as enumerated values. The identifier of the enumeration is `enum SPAL_ResultCodes`. The result codes shall always be referred to with defined identifiers not assuming a specific value matching a specific result. As an exception, successful return denoted by `SPAL_SUCCESS` always has value 0 (zero). Table 2 lists the result code identifiers and descriptions of the results.

Table 2 SPAL Result Codes.

Identifier	Description
SPAL_SUCCESS	Function has returned successfully. Post conditions for function apply.
SPAL_RESULT_NOMEM	Function has failed due to failure to allocate dynamic memory.
SPAL_RESULT_NORESOURCE	Function has failed due to failure to allocate a system resource.
SPAL_RESULT_LOCKED	Function has failed due to an already locked resource.
SPAL_RESULT_INVALID	Function has failed due to invalidity of one or more input parameters.
SPAL_RESULT_CANCELED	Function has failed due to a referenced thread has been canceled.
SPAL_RESULT_TIMEOUT	Function has failed due to a timeout.

3.6.3 Type Customization

The SPAL API provides a mechanism for customization of the types utilized by the interfaces. This is important for efficiency since many types have exact equivalents provided by the platform SPAL is implemented on. The SPAL headers all include a customization header called `cfg_spal.h`. This header shall provide size and alignment configuration for the types. Table 3 lists the customizable types and their configuration parameters.

Table 3 SPAL customizable types.

Configuration parameter	Type	Interface	Description
SPAL_CFG_MUTEX_SIZE	SPAL_Mutex_t	Mutual Exclusion	Size of the mutex type
SPAL_CFG_MUTEX_ALIGN_TYPE	SPAL_Mutex_t	Mutual Exclusion	A type with required alignment
SPAL_CFG_SEMAPHORE_SIZE	SPAL_Semaphore_t	Semaphore	Size of the semaphore type
SPAL_CFG_SEMAPHORE_ALIGN_TYPE	SPAL_Semaphore_t	Semaphore	A type with required alignment

3.7 SPAL Mutex Interface

SPAL Mutex Interface is defined in header file `spal_mutex.h`. The prefix for SPAL Mutex Interface is `SPAL_Mutex`.

3.7.1 Types

The SPAL Mutex Interface defines following types:

```
struct SPAL_Mutex {
    union {
#ifdef SPAL_CFG_MUTEX_ALIGN_TYPE
        SPAL_CFG_MUTEX_ALIGN_TYPE Alignment;
#endif
        uint8_t Size[SPAL_CFG_MUTEX_SIZE];
    } Union;
};
typedef struct SPAL_Mutex SPAL_Mutex_t;
```

The type `SPAL_Mutex_t` is a handle to a SPAL Mutex. The value of a variable of type `SPAL_Mutex_t` shall not be interpreted by the Caller.

3.7.2 Functions

The SPAL Mutex Interface declares the following functions:

- `SPAL_Mutex_Init`
- `SPAL_Mutex_Lock`
- `SPAL_Mutex_TryLock`
- `SPAL_Mutex_UnLock`
- `SPAL_Mutex_IsLocked`
- `SPAL_Mutex_Destroy`

The purpose, pre- and post-conditions for each function are defined in following subsections.

3.7.2.1 SPAL_Mutex_Init

```
SPAL_Result_t
SPAL_Mutex_Init(
    SPAL_Mutex_t * const Mutex_p);
```

The function initializes a thread mutex handle pointed to by `Mutex_p`.

Preconditions

1. `Mutex_p` is a valid pointer for read and write access.

Postconditions

1. The value pointed to by `Mutex_p` is initialized as a handle to a thread mutex.
2. `Mutex_p` points to a handle of an unlocked mutex.

Exceptions

1. `SPAL_RESULT_NOMEM`: The implementation has to allocate dynamic memory to initialize a thread mutex and the memory allocation has failed. The mutex handle is not initialized.

Note: *It is strongly recommended that the implementation tries to define `SPAL_Mutex_t` (via `SPAL_CFG_MUTEX_SIZE` and `SPAL_CFG_MUTEX_ALIGN_TYPE`) so that no memory allocation is required in the `SPAL_Mutex_Init` function.*

2. `SPAL_RESULT_NORESOURCE`: The implementation has failed to allocate some system resource. The system might be out of mutex handles, for example. The mutex handle is not initialized.

3.7.2.2 SPAL_Mutex_Lock

```
void
SPAL_Mutex_Lock(
    SPAL_Mutex_t Mutex);
```

The function locks the `Mutex`. If the `Mutex` is already locked by another thread the execution of the Caller is suspended. The execution of the Caller is unsuspended when another thread unlocks the `Mutex`.

Preconditions

1. The `Mutex` is a value initialized by a call to the `SPAL_Mutex_Init()` function.
2. The `Mutex` has not been already locked by the caller.

Postconditions

1. The `Mutex` is locked by the caller.

Exceptions

This function has no exceptions.

3.7.2.3 SPAL_Mutex_TryLock

```
SPAL_Result_t
SPAL_Mutex_TryLock(
    SPAL_Mutex_t Mutex);
```

The function attempts to lock a `Mutex` returning success when locking succeeded.

Preconditions

1. The `Mutex` is a value initialized by a call to the `SPAL_Mutex_Init()` function.
2. The `Mutex` has not been already locked by the caller.

Postconditions

1. The `Mutex` is locked by the caller.

Exceptions

1. `SPAL_RESULT_LOCKED`: The `Mutex` was already locked by another thread.

3.7.2.4 SPAL_Mutex_UnLock

```
void
SPAL_Mutex_UnLock(
    SPAL_Mutex_t Mutex);
```

The function unlocks the `Mutex`.

Preconditions

1. The `Mutex` is a value initialized by a call to the `SPAL_Mutex_Init()` function.
2. The `Mutex` has been locked by the caller.

Postconditions

1. The `Mutex` is unlocked by the caller.
2. If there are other threads suspended on this `Mutex` exactly one of them shall get the `Mutex` locked.

Exceptions

This function has no exceptions.

3.7.2.5 SPAL_Mutex_IsLocked

```
bool
SPAL_Mutex_IsLocked(
    SPAL_Mutex_t Mutex);
```

The function checks whether a `Mutex` is in locked state.

Preconditions

1. The `Mutex` is a value initialized by a call to the `SPAL_Mutex_Init()` function.

Postconditions

1. The function returns ***true*** if the `Mutex` was locked by this thread.
2. The function returns ***false*** if the `Mutex` was not locked by any thread.
3. The function may return either ***true*** or ***false*** if the `Mutex` was locked by some other thread during the `SPAL_Mutex_Init()` function call.

Exceptions

This function has no exceptions.

Rationale

This function deviates from the policy of always returning a return value of type `SPAL_Result_t`, because result code does not contain proper values to describe the status to be returned; result code is either successful or unsuccessful and both return values are successful for this function. This Boolean value could be returned through a pointer to a Boolean parameter, but returning the value was chosen over the parameter to enable easy usage of the function from assertion macros.

3.7.2.6 SPAL_Mutex_Destroy

```
void
SPAL_Mutex_Destroy(
    SPAL_Mutex_t * const Mutex_p);
```

The function destroys a thread mutex handles pointed to by `Mutex_p` and releases all associated resources.

Preconditions

1. The pointer `Mutex_p` points to a mutex handle initialized with the `SPAL_Mutex_Init()` function.
2. The mutex handle is not locked by any thread.

Postconditions

1. The value pointed to by `Mutex_p` becomes an invalid mutex handle.
2. All resources associated with mutex are released.

Exceptions

This function has no exceptions.

3.8 SPAL Thread Management Interface

The SPAL Thread Management Interface is defined in header file `spal_thread.h`. The prefix for the SPAL Thread Management Interface is `SPAL_Thread`.

3.8.1 Types

The SPAL Thread Management Interface defines following types:

```
typedef SPAL_Platform_Thread_t SPAL_Thread_t;
```

The type `SPAL_Thread_t` is a handle to a SPAL Thread that is mapped to a platform thread identifier by the SPAL implementation. The value of a variable of type `SPAL_Thread_t` shall not be interpreted by the caller.

3.8.2 Functions

The SPAL Thread Management Interface declares following functions:

- `SPAL_Thread_GetId`
- `SPAL_Thread_Create`
- `SPAL_Thread_Detach`
- `SPAL_Thread_Join`
- `SPAL_Thread_Exit`

The purpose, pre- and post-conditions for each function are defined in following subsections.

3.8.2.1 SPAL_Thread_GetId

```
SPAL_Thread_t
SPAL_Thread_GetId(void);
```

The function returns the thread identifier of the calling thread.

Postconditions

The function returns the identifier of the calling thread.

Exceptions

This function has no exceptions.

3.8.2.2 SPAL_Thread_Create

```
SPAL_Result_t
SPAL_Thread_Create(
    SPAL_Thread_t * const Thread_p,
    const void * const    Reserved_p,
    void *               (*StartFunction_p)(void * const Param_p),
    void * const          ThreadParam_p);
```

The function creates a new thread. The new thread must start in joinable state, it may be detached or joined later.

Preconditions

1. `Thread_p` is a valid pointer.
2. `Reserved_p` is `NULL`.
3. `StartFunction_p` is a valid function pointer.

Postconditions

1. A new thread is created.
2. Within the context of the new thread `StartFunction_p` is called passing `ThreadParam_p` as the parameter `Param_p`.
3. The identifier of the new thread is passed to the caller through `Thread_p`.

Exceptions

1. `SPAL_RESULT_NOMEM`: The implementation has to allocate dynamic memory to initialize a thread and the memory allocation has failed. The value pointed to by `Thread_p` shall not be modified.
2. `SPAL_RESULT_NORESOURCE`: The implementation has failed to allocate some system resource. The system might be out of thread identifiers, for example. The value pointed to by `Thread_p` shall not be modified.

3.8.2.3 SPAL_Thread_Detach

```
SPAL_Result_t
SPAL_Thread_Detach(
    const SPAL_Thread_t Thread);
```

The function detaches thread identified by `Thread`.

Preconditions

1. The provided thread identifier must have been the result of a successful `SPAL_Thread_Create()` function invocation.
2. The provided thread identifier must have not been previously detached.

Postconditions

1. The identified thread becomes detached.
2. If the thread has not exited, the system resources for the specified thread will be reclaimed when the thread ends.
3. If the thread has already exited, the resources are reclaimed immediately.
4. If the thread is already detached, the function does nothing.

Exceptions

1. `SPAL_RESULT_INVALID`: the thread identifier is not an identifier of any existing thread.

3.8.2.4 SPAL_Thread_Join

```
SPAL_Result_t
SPAL_Thread_Join(
    const SPAL_Thread_t Thread,
    void ** const       Status_p);
```

The function waits for the thread identified by `Thread` to finish and returns its exit status.

Preconditions

1. The provided thread identifier must have been the result of a successful `SPAL_Thread_Create()` function invocation.
2. The identified thread must not have been detached.

Postconditions

1. The resources allocated for the identified thread are released.
2. If `Status_p` is not `NULL` the exit status is returned via `Status_p`.
3. If `Status_p` is `NULL`, no exit status is returned.

Exceptions

1. `SPAL_RESULT_INVALID`: the thread identifier is not an identifier of any existing thread.
2. `SPAL_RESULT_CANCELED`: the thread was canceled, no exit status is available.

3.8.2.5 SPAL_Thread_Exit

```
void
SPAL_Thread_Exit(
    void * const Status);
```

The function terminates the calling thread with the given exit status.

Preconditions

1. The provided thread identifier must have been the result of a successful `SPAL_Thread_Create()` function invocation.

Postconditions

1. The function never returns.
2. If the calling thread is detached its resources are freed immediately.
3. If the calling thread is not detached, its exit status is made available to any waiting thread.

Exceptions

This function has no exceptions.

3.9 SPAL Dynamic Memory Interface

The SPAL Dynamic Memory Interface is defined in header file `spal_memory.h`. The prefix for the SPAL Dynamic Memory Interface is `SPAL_Memory`.

3.9.1 Types

The SPAL Dynamic Memory Interface does not define any types.

3.9.2 Functions

The SPAL Dynamic Memory Interface declares following functions:

- `SPAL_Memory_Alloc`
- `SPAL_Memory_Calloc`
- `SPAL_Memory_ReAlloc`
- `SPAL_Memory_Free`

The purpose, pre- and post-conditions for each function are defined in following subsections.

3.9.2.1 SPAL_Memory_Alloc

```
void *
SPAL_Memory_Alloc(
    const size_t Size);
```

The function allocates a block of memory of size `Size` bytes. The block shall be at least aligned as required for `uint64_t` or the largest native type that fits in `Size` bytes. If `Size` is zero, the function is allowed to either allocate a new block or to return `NULL`.

Preconditions

1. `Size` must be less than 2147483648 (2^31).

Postconditions

1. The function returns a pointer to the allocated memory.
2. The contents of memory returned are undefined.

Exceptions

1. The function returns `NULL` denoting that no memory is available to allocate a block of the requested size.

3.9.2.2 SPAL_Memory_Calloc

```
void *
SPAL_Memory_Calloc(
    const size_t MemberCount,
    const size_t MemberSize);
```

The function allocates a memory array that is initialized to zeros.

Preconditions

1. `MemberCount` must be greater than or equal to 1.
2. `MemberSize` must be greater than or equal to 1.
3. `MemberCount*MemberSize` must be less than 2147483648 (2^31).

Postconditions

1. The function returns a pointer to the allocated array.
2. The array is filled with zeros.

Exceptions

1. The function returns NULL denoting that no memory is available to allocate a block of the requested size.

3.9.2.3 SPAL_Memory_ReAlloc

```
void *
SPAL_Memory_ReAlloc(
    void * const Mem_p,
    const size_t NewSize);
```

The function reallocates a given memory handle with an area of the specified size (in bytes).

Preconditions

1. The provided pointer Mem_p must not be NULL and must have been returned by an earlier call to the functions SPAL_Memory_Alloc(), SPAL_Memory_Calloc() or SPAL_Memory_ReAlloc().
2. NewSize must be greater than or equal to 1.
3. NewSize must be less than 2147483648 (2^31).

Note: *This function cannot be used as substitute for the functions SPAL_Memory_Alloc() and SPAL_Memory_Free() unlike the ANSI C function realloc().*

Postconditions

1. The function returns a pointer to a newly allocated memory or the original value of Mem_p.
2. If the return value of the function is not the original value of Mem_p, then the memory area originally pointed to by Mem_p is free after execution of this function.
3. All bytes up to the smaller one of the size of original memory area and NewSize contain their original values; however, the contents of the memory after that point are undefined.

Exceptions

1. The function returns NULL denoting that no memory is available to resize memory allocation. In this case, the original memory array pointed to by Mem_p remains allocated, but its size is not changed.

3.9.2.4 SPAL_Memory_Free

```
void
SPAL_Memory_Free(
    void * const Pointer);
```

The function frees memory pointed to by Pointer.

Preconditions

1. Pointer must have been returned by the function SPAL_Memory_Alloc() or SPAL_Memory_Calloc().
2. Pointer must not have been freed before calling the function SPAL_Memory_Free().
3. Pointer must not be a NULL pointer.

Postconditions

1. The memory is freed.
2. The memory shall not be accessed anymore. (Unless, it is allocated again.)

Exceptions

This function has no exceptions.

3.10 SPAL Semaphore Interface

The SPAL Semaphore Interface is defined in header file `spal_semaphore.h`. The prefix for the SPAL Semaphore Interface is `SPAL_Semaphore`.

3.10.1 Types

The SPAL Semaphore Interface defines following types:

```
struct SPAL_Semaphore {
    union {
#ifdef SPAL_CFG_SEMAPHORE_ALIGN_TYPE
        SPAL_CFG_SEMAPHORE_ALIGN_TYPE Alignment;
#endif
        uint8_t Size[SPAL_CFG_SEMAPHORE_SIZE];
    } Union;
};
typedef struct SPAL_Semaphore SPAL_Semaphore_t;
```

The type `SPAL_Semaphore_t` is a handle to a SPAL Semaphore. The value of a variable of type `SPAL_Semaphore_t` shall not be interpreted by the caller.

3.10.2 Functions

The SPAL Semaphore Interface declares following functions:

- `SPAL_Semaphore_Init`
- `SPAL_Semaphore_Wait`
- `SPAL_Semaphore_TryWait`
- `SPAL_Semaphore_TimedWait`
- `SPAL_Semaphore_Post`
- `SPAL_Semaphore_Destroy`

The purpose, pre- and post-conditions for each function are defined in following subsections.

3.10.2.1 SPAL_Semaphore_Init

```
SPAL_Result_t
SPAL_Semaphore_Init(
    SPAL_Semaphore_t * const Semaphore_p,
    const unsigned int       InitialCount);
```

The function initializes a semaphore pointed to by `Semaphore_p`.

Preconditions

1. `Semaphore_p` is a valid pointer for read and write access.

Postconditions

1. The memory pointed to by `Semaphore_p` is an initialized semaphore.
2. The count of the semaphore is initialized to `InitialCount`.

Exceptions

1. `SPAL_RESULT_NOMEM`: The implementation has to allocate dynamic memory to initialize a semaphore and the memory allocation has failed. The semaphore is not initialized.
2. `SPAL_RESULT_NORESOURCE`: The implementation has failed to allocate some system resource. The system might be out of semaphore handles, for example. The semaphore is not initialized.

3.10.2.2 SPAL_Semaphore_Wait

```
SPAL_Result_t
SPAL_Semaphore_Wait(
    SPAL_Semaphore_t * const Semaphore_p);
```

The function waits until the count of a semaphore pointed to by `Semaphore_p` is greater than zero. When the count of the semaphore becomes greater than zero the value is decreased by one and the function returns.

Note: *If the semaphore count is never greater than zero, the function never returns.*

Preconditions

1. `Semaphore_p` points to a semaphore initialized by the `SPAL_Semaphore_Init()` function.

Postconditions

1. The count of the semaphore is decreased by one.

3.10.2.3 SPAL_Semaphore_TryWait

```
SPAL_Result_t
SPAL_Semaphore_TryWait(
    SPAL_Semaphore_t * const Semaphore_p);
```

This function provides the same functionality as `SPAL_Semaphore_Wait()`, but instead of actually waiting, this function returns an error code to indicate it would block.

Preconditions

1. `Semaphore_p` points to a semaphore initialized by the `SPAL_Semaphore_Init()` function.

Postconditions

1. The count of the semaphore is decreased by one.

Exceptions

1. `SPAL_RESULT_LOCKED`: The count of the semaphore was not greater than zero. The count is not decreased.

3.10.2.4 SPAL_Semaphore_TimedWait

```
SPAL_Result_t
SPAL_Semaphore_TimedWait(
    SPAL_Semaphore_t * const Semaphore_p,
    const unsigned int       TimeoutMilliSeconds);
```

The function tries to decrease the count of the semaphore pointed to by `Semaphore_p`. The function returns if the timeout runs out prior to decreasing the count of the semaphore being possible. If the count of the semaphore becomes greater than zero the value is decreased by one and the function returns.

Preconditions

1. `Semaphore_p` points to a semaphore initialized by the `SPAL_Semaphore_Init()` function.

Postconditions

1. The count of the semaphore is decreased by one and function returns `SPAL_SUCCESS`.

Exceptions

1. `SPAL_RESULT_TIMEOUT`: The semaphore was not obtained during given timeout. The count is not decreased.

3.10.2.5 SPAL_Semaphore_Post

```
void
SPAL_Semaphore_Post(
    SPAL_Semaphore_t * const Semaphore_p);
```

The function increases the count of semaphore pointed to by `Semaphore_p`. If the count was initially not greater than zero, one of the threads possibly waiting on the semaphore is released.

Preconditions

1. `Semaphore_p` points to a semaphore initialized by the `SPAL_Semaphore_Init()` function.

Postconditions

1. The count of the semaphore is increased by one.

Exceptions

This function has no exceptions.

3.10.2.6 SPAL_Semaphore_Destroy

```
void
SPAL_Semaphore_Destroy(
    SPAL_Semaphore_t * const Semaphore_p);
```

The function destroys a semaphore pointed to by `Semaphore_p` and releases all associated resources.

Preconditions

1. `Semaphore_p` points to a semaphore initialized by the `SPAL_Semaphore_Init()` function.
2. The semaphore is not waited for by any thread.

Postconditions

1. The semaphore becomes invalid.
2. All resources associated with semaphore are released.

Exceptions

This function has no exceptions.

3.11 SPAL Time Management Interface

The SPAL Time Management Interface is defined in header file `spal_sleep.h`. The prefix for the SPAL Time Management Interface is `SPAL_`.

3.11.1 Functions

The SPAL Time Management Interface declares following functions:

- `SPAL_SleepMS`

3.11.1.1 SPAL_SleepMS

```
void
SPAL_SleepMS(
    unsigned int Milliseconds);
```

The function blocks the caller for the specified number of milliseconds.

The typical implementation will sleep the thread allowing other threads or processes to be scheduled. Because the implementation will put the thread to sleep and the thread will be woken up after sleep, depending on system load and the granularity of used clock, the sleep may take somewhat longer than the desired number of milliseconds.

Parameters

Milliseconds	[in]	Number of milliseconds to sleep.

(End of Document)