SafeZone™
SafeZone Framework v5.2

Porting Guide

Document Revision: B
Document Date: 2013-02-14
Document Number: 007-914520-304
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	2011-09-05	RWI MHO KLA	• Created a new document derived from SafeZone v5.1 Porting Guide, Rev B • Template update and small editorial changes. • Added note on porting SPAL_Mutex_IsLocked() in chapter 6.5. • Added Appendix A • Updated terminology in Chapter 7.
B	All	2013-02-14	FvdM	• Update template

TABLE OF CONTENTS

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

SafeZone Framework implementations are provided for POSIX and Windows. This document describes how to implement the SafeZone Framework for a new environment, which can be done by porting one of the existing implementations.

1.2 Scope

This document covers the SafeZone Framework that is included in, and used by, many SafeZone products. The details on porting additional components in each SafeZone product are provided in specific *Porting Guide Addendum* documents for those products.

1.3 Related Documents

The following documents are part of the documentation set.

Ref.	Document Name	Document Number
[1]	SafeZone Framework Porting Guide (this document)	007-914520-304
[2]	SafeZone Framework Reference Manual	007-914520-305
[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 Preparing and planning the porting activities

Porting SafeZone software to new targets is usually a relatively easy task. However, it is still a task that you should not attempt without proper preparation.

Before planning the porting, familiarize yourself with the SafeZone software by following the examples in the Getting Started guide, on one of the supported environments such as Linux/x86. Try to make a simple application using the SafeZone libraries.

Also, read this manual through, and take a look at the existing implementation of the Framework.

Also, familiarize yourself with the target system and its operation (unless you are already expert on that system).

If there is a *Porting Guide Addendum* document for the subset of SafeZone you intent to port, it may provide additional information on porting, such as additional issues you need to consider.

Now, you are more prepared to start planning.

Below follows a typical porting plan, including milestones:

1. Port the Public and Implementation definitions.
2. Port the SFZUTF Unit Testing Framework and provide stubs of CLIB, SPAL and EE functions.
3. Check that you are able to execute the Framework test suite (even if not all tests work).
   ⇨ Milestone M1.
4. Port CLIB and EE.
5. Port SPAL dynamic memory allocation support, if needed.
   ⇨ Milestone M2.
6. Port the SPAL mutex, threading and semaphore support, by using the test suite as aid.
   ⇨ Milestone M3.
7. The Framework Porting task is now complete and other SafeZone products can be added on top. Perform the porting from the bottom of the software stack and up. The correct order is:
   - CAL
     ⇨ Milestone M4.
   - Secure Storage (SST)
     ⇨ Milestone M5.
   - PKCS#11
     ⇨ Milestone M6.
   - Certificate Library
     ⇨ Milestone M7.

   Details for porting each product can be found in the *Porting Guide Addendum* documents for each of the SafeZone products. Use the Test Suite provided for each API layer to check the functionality.

   Remember: the next software layer can only work correctly when the underlying layers are working correctly too.

3 C dialects

SafeZone is technically using ISO C99 (see A.3.1). However, it requires only a small subset of the features above ANSI C (also known as ISO C89 or ISO C90). Therefore, SafeZone may also be compiled with most ISO C89 compilers, with only a small amount of configuration required, such as providing the `uint32_t` type.

The reason for using ISO C99 is that it has features such as `inline` that make it easy to inline small portions of commonly used code. Alternatives available in earlier C standards (static functions and macros) both have their problems. Static functions cause function invocation overhead (=> efficiency problem, occasionally also code growth). Macros produce less type-safe code and because they are based on textual expansion, their usage is more likely to cause unexpected consequences than inline functions. Nevertheless, the SafeZone usage of inline functions shall automatically degrade into usage of static functions if the C compiler does not support the ISO C99 `inline` keyword.

ISO C99 defines a very comprehensive standard library that requires a platform with features like input/output capabilities, file system, etc. Also, it has support for wide (international) characters.

SafeZone, on the other hand is aimed at embedded devices and they may not have all those features. Therefore, SafeZone does not use ISO C99 standard libraries. Instead, it uses its own Framework, that is, for the biggest part, a subset of ISO C99 standard libraries, containing features that are suitable for embedded systems. It is important to notice that few embedded environments provide a full ISO C99 standard library, and therefore this approach improves the portability.

The most notable ISO C99 feature required by SafeZone is an ISO C99 compatible preprocessor. If the desired compiler does not provide such a preprocessor, it might be required to use a build process where files are separately preprocessed with an ISO C99-conformant C preprocessor before compiling with the desired target compiler.

3.1 Support for C features beyond ISO C99

When SafeZone software is compiled, it can detect which software is used. If this is the GNU C compiler, SafeZone Framework enables some extra diagnostics supported by that compiler, such as ensuring that the arguments to L_DEBUG (output logging macro) seem to be correct. These options in no way effect the operation of the software, but allow detecting some of the potential problems sooner.

3.2 Runtime requirements beyond ISO C99

It is assumed, for most parts of the software, that the system is capable of multithreading, and contains some kind of scheduler. The ISO C99 standard does not describe the operation of multithreaded programs.

Instead, we resort to multithreading APIs such as POSIX.1: POSIX threads.

4 Building SafeZone software

The build system shipped with SafeZone is based on the GNU *Autotools*, see A.2.1, and GNU `make`. It uses `automake` (recommended version 1.10 or later), `autoconf` (recommended version 2.61 or later) and `make` (recommended version 3.81 or later). *Autotools* produces make files that are compatible with GNU `make` and GNU compiler tools, but some other tool chains and make implementations may also be able to use them.

One important reason for using `automake` and `autoconf` tools is that they make it relatively easy to support cross compilation, which is commonly required by embedded targets. If the target is supported by some GNU-based tool chain, such as *Code Sourcery GNU G++* tool chains, and the tool chain runs under Linux/x86, it is likely that the build system does not require any porting at all.

Also, the C libraries as provided with those tool chains implement most of the functions that are required by the provided reference implementations of the SafeZone Framework.

If the compiler requires another kind of build system than make-based (e.g. it has its own project files based build process), the build instructions need to be implemented for that environment. Please note that a mixed environment with more than one compiler can cause difficulties. Whenever possible, use a single compiler for all the code that must be linked together.

If a GNU-based tool chain is available for the target platform, it can be useful to try it prior to porting SafeZone Framework. The benefits of this approach are:

- Build instructions are unlikely to need significant changes;
- The libraries provided with the tool chains (GNU C library, uClibc, newlib etc.) are a fairly good starting points for porting;
- Once porting with the GNU tool chain is done, there are only a few changes required in the headers or code to make it work with the final tool chain.
- If the final compiler cannot be used with *make*, new build files need to be implemented at this point. It is significantly easier to create build files once it is known that the software to compile is (mostly) complete.

5 SafeZone Framework

SafeZone Framework consists of the following APIs:

- Public definitions
  Implementation definitions (including 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
- Configuration files

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

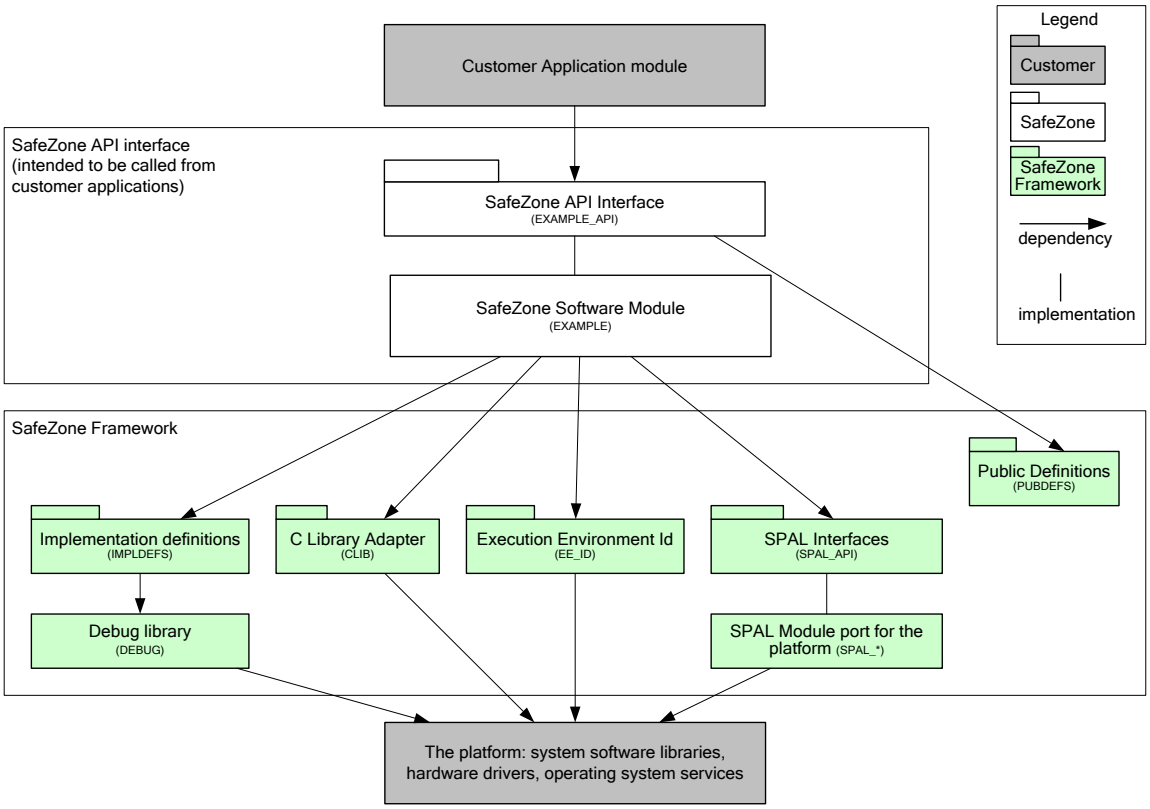

Figure 1 Relationship between SafeZone Framework and the rest of the system.

5.1 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.

The reference implementation is in `Framework/PUBDEFS/incl/public_defs.h`.

5.2 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.

The reference implementation can be found in `Framework/IMPLDEFS/incl/implementation_defs.h`.

A lot of compilers offer a feature that tells if there are identifiers defined within a function, which are actually not being used at all. Due to various reasons, it is actually common to have some identifiers that are not used, for example within the parameter list of a function. Implementation definitions provides the macros `PARAMETER_NOT_USED()` and `IDENTIFIER_NOT_USED()` that can be used to tell that the variables are indeed unused, and the compiler should not complain about them.

Because implementation definitions make it easier to monitor software quality, and in a few cases, write clearer and more maintainable code, they are used a lot in the SafeZone source code. In addition, for ease of use, implementation definitions have convenient and short macro names. It is possible that these definitions conflict with other definitions that the customer has, aimed at similar purposes. This is the reason that they are only included in implementation files, but not in header files.

5.2.1 DEBUG library

The DEBUG library is a simple helper library for `implementation_defs.h`. When logging is enabled in the configuration file `cf_impldefs.h`, the use of `L_DEBUG` and `L_TRACE` result in a call to the `DEBUG_printf()` function.

The default implementation just outputs the logs to `stderr`, but it is easy to customize logging to use a target platform specific method.

The DEBUG library also implements the function `DEBUG_abort()` that is called in fatal error situations where the code has no means of recovering. Examples are when `ASSERT` or `PANIC` are used. By default, the implementation calls `abort()`.

5.3 CLIB

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

The reference implementation of CLIB just calls the corresponding functions in the C library provided by the compiler. In systems without such a C library, it is possible that some of these functions have different names or do not exist at all. It is also very easy to write substitutes for these functions, as the functions themselves have no special dependencies.

5.4 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 uniquely identifies the Execution Environment within the system. It depends on the system how many different execution environments exist.

The Application Identifier is a unique identifier for the application. By default, the implementation supports a Universally Unique Identifiers (UUID), which makes sure that there are no accidental conflicts in identifiers used by applications.

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.

Secure Storage (SST) associates objects with the credentials (Global Application Id) of the application that stores the object. It also includes application credentials (Global Application Id) in the Access Policy that control object access and manipulation.

5.5 SPAL

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

SPAL dynamic memory management consists of functions for memory allocation that are comparable to the standard C library functions `malloc()`, `calloc()`, `realloc()` and `free()`.

SPAL supports the creation of new threads, as well as waiting for child threads to finish. The library attempts to be easy to interface to the most common thread libraries, and to minimize functionality to ensure easy porting.

SPAL contains two libraries for basic mutual exclusion handling: mutexes and semaphores. The mutex API is intentionally very limited, to make it easy to port. For cases where more functionality is required, semaphores are provided. The most of SafeZone attempts to use mutexes only, to make the libraries more portable, however, semaphores are used where it is absolutely required.

The differences between these two are minimal and it is foreseen that a few implementations actually uses the same underlying primitives. However, roughly the differences are following:

- SPAL “Mutex” implementation may opt to use critical sections instead of actual mutexes if the platform does not provide actual mutexes. SPAL “Mutex” is associated with the thread that locked (acquired) the mutex, where as in Semaphores it is allowed that some other thread releases it, and therefore Semaphores must co-operate with the scheduler.
- SafeZone libraries typically use mutexes to protect the integrity of data used or accessed by the SafeZone library when the API is allowed to be called from multiple threads.
- SafeZone software typically uses semaphores for synchronization between threads.

5.6 Configuration files

Some of the SafeZone Framework components allow configuration via configuration files. There are two kinds of configuration files:

- `cf_*` files allow defining and un-defining preprocessor definitions: they are declared or not.
- `cfg_*` files contain preprocessor definitions that are variable values, such as the size of a buffer or the name of a device.

Currently, the Framework components that require configuration are Implementation definitions and SPAL.

The configuration files of the Implementation definitions allow enabling and disabling various logging options for checking function inputs in more or less detail.

The SPAL configuration files allow you to declare the size of semaphore and mutex structures as required by the current semaphore and mutex implementations. The rationale behind these files is that software using mutexes and/or semaphores does not need to include the header files from the underlying implementation of mutual exclusion/semaphores (such as `pthreads.h`) but instead only needs to include SPAL headers.

6 Reference Implementations

SafeZone Framework implementations are provided to cover some of the most common environments. These implementations allow the SafeZone software to build and run on these environments.

The following two SafeZone Framework implementations are provided:

- For 32-bit POSIX like systems, like Linux 2.6.
- For 32-bit Windows platforms, like Windows XP.

Supporting different environments and compilers has been essential in ensuring portability of the SafeZone Framework. The following compilers are supported by SafeZone Framework sources:

- The GNU C compiler;
- The ARM C compiler;
- The Microsoft C compiler

For other environments, this manual and the existing Framework implementations will aid the porting activities. It is very likely that parts of the existing implementations can be reused for other environments.

6.1 Public definitions

Public definitions are compatible with ISO C99-compatible systems.

For Windows systems without ISO C99 support, definitions of the types are also provided.

6.2 Implementation definitions and DEBUG library

The implementation definitions and the DEBUG library require an ISO C99 compatible C preprocessor. Except for the preprocessor, they are compatible with ANSI C89 and above (including fully ISO C99 compliant platforms).

The implementation definitions have two configuration files that can be found in the `Config` directory. The default settings are for a ‘debug’ configuration where log lines are generated and assertion checks are active.

To create a ‘heavy debug’ build, the preprocessor define `CFG_IMPLDEFS_ENABLE_TRACE` must be set. This is typically done on a single source file, or on a software module of interest.

To create a ‘release’ build the preprocessor define `CFG_IMPLDEFS_NO_DEBUG` must be set on the command line or at the top of `cf_impldefs.h`.

6.3 CLIB

The default implementation as provided with SafeZone is a wrapper for the standard C library.

If your environment does not offer these standard C library functions, it is possible to find replacements for example from the newlib C library.

These functions use the standard C library for performance reasons, in current C libraries a few of these functions make use of larger units to speed up operations.

6.4 Execution Environment Identifier Library

The current implementation of the Execution Environment Identifier Library offers facilities to operate with Application and EE identifiers. However, the library accepts the supplied identifiers as they are; it does not attempt to enforce their correctness or prevent applications from spoofing their identity.

This means that the Execution Environment Identifier Library is very architecture and operating system independent. However, as a side-effect the current system does not provide strong application separation.

If strong application separation is desired, it is required that system hardware, a kernel or hypervisor enforces application identities, based on information from application signatures or some other mechanism that creates a verifiable tie between application code and its identity. Also,

the conforming system needs to ensure that identities remain the same during execution of the applications and they need to provide inter-application communication facilities that allow ensuring the identity of the peer application.

6.5 SPAL

SPAL dynamic memory management has a single implementation that uses the standard C library to implement SPAL memory allocation functions.

For mutexes, threads and semaphores there are two implementations: implementations that are based on POSIX Threads and implementation based on the Win32 API.

The sizes and alignment requirements of mutex and semaphore structures are provided within the SPAL configuration file. The default values are current for Linux on most 32-bit targets. On targets with longer word length (like 64-bit) or with some other operating system, it is likely you need to specify new values.

Note on porting `SPAL_Mutex_IsLocked`

The objective of `SPAL_Mutex_IsLocked()` is checking that a mutex is held by the currently executing thread. Various ways exist to implement this function:

- "is_owned" check that the mutex is held by this thread if such mechanism is available.
- "is_locked" examine the mutex itself to see if its locked
- Trial locking "trylock" + "unlock". Notice this is only possible if the mutex implementation does not support lock nesting.

If the mutex implementation does not support inspection of the lock status of the mutex, this function must return `true`.

7 Testing the port

7.1 Software Unit Testing Framework (SFZUTF)

Once public definitions and implementation definitions have been ported to the target platform, it may be possible (often even easy) to get the Software Unit Testing Framework (SFZUTF) ported for the target platform.

When the Framework and SFZUTF have been ported, the Framework Test Suite can be used to verify operation of the SafeZone Framework. However, public definitions and implementation definitions are prerequisites of SFZUTF itself, and therefore these need to be ported before SFZUTF can be ported.

To get familiar with the Software Unit Testing Framework, please read the *Software Unit Testing Framework User Guide* [3]. A major feature of the SFZUTF is the fact that it consists of a *TestRunner*, i.e. a perl script running on a host connected to the target system (or on the target system itself), and a *TestBinary*, i.e. the actual code under test that runs on the target system. Also, *TestRunner* may run the *TestBinary* via a specified interpreter program.

The items that need consideration when porting SFZUTF are:

- Prerequisites of SFZUTF:
  - PUBDEFS and IMPLDEFS ported (See sections 6.1 and 6.2 for details on porting these.)
  - `malloc()`, `calloc()` and `free()` standard C functions (or SPAL memory allocation API)
  - `memcmp()` and `memcpy()` or equivalent CLIB functions (`c_memcmp()` and `c_memcpy()`)
- How to get arguments from the *TestRunner* to the *TestBinary*.
- How to get execution logs from the *TestBinary* back to the *TestRunner*.

7.1.1 Getting arguments to the TestBinary

SFZUTF provides two conventions for passing arguments to the *TestBinary*:

- Command line arguments
- File system

Both of these are implemented within SFZUTF in `SFZUTF/impl/sfzutf_main_stdio.c`.

For smaller embedded systems, a typical solution constists of: running serial/parallel communication software to load the *TestBinary* into the target environment (along with any arguments) and performing a reset of the target system.

7.1.2 Getting execution logs from the TestBinary

The default mechanism uses IMPLDEFS and its debug logging, which in turn uses `fprintf()`. This solution works for generic operating systems. In embedded systems, it is likely that logging needs to be sent somewhere else, although the basic principle remains: logs need to be transferred to *TestRunner*.

For example, if a target system supports serial communications, the tests can write to a serial communication device and a serial communication application could be used as the interpreter executed by *TestRunner*.

One pitfall in logging is real time operation. The available logs need to be provided to *TestRunner* immediately. Failure to do so may cause *TestRunner* to report timeouts wrongly.

A solution to this problem is provided in the distribution: `SFZUTF/impl/nobuffer.c` that compiles into `nobuffer.so`. This binary must be loaded before starting the *TestBinary* by using the *TestRunner* argument `-ld-preload=nobuffer.so`. This program forces the (default) interpreter application to flush the `stdout/stderr` after each log it generates. Practice has shown that many interpreters on UNIX and Linux need this.

7.1.3 Memory management

SFZUTF internal memory management is either based on the standard C memory management, or SPAL memory management. For typical environments, it is recommended to use ANSI C memory management functions (default) as this avoids "polluting" the SPAL memory allocation statistics with SFZUTF (test) memory allocations.

However, it is also possible to force SFZUTF to use SPAL (add the compilation parameter `-DSFZUTF_CONFIG_USE_SPAL` to `CFLAGS`). This allows SFZUTF to use memory management on very bare bone systems where standard C memory management functions are not available.

7.2 Using the Framework Test Suite

The SafeZone Framework has a test suite that covers all the Framework APIs. By default, the test suite executes each of the tests and outputs a lot of information about the progress and test results. A single failure will cause the test suite to abort.

The SFZUTF `testrunner.pl` script interprets this output and automatically continues testing after any failed test. Examples are shown below.

Raw output from the SafeZone Framework Test Suite:

```
Build/build_linux # ./framework_testsuite
LL_TESTLOG, LF_SUITE_BEGIN, ./../../SFZUTF/impl/sfzutf.c:441: sfzutf_run _suite:
Framework_Test Suite
LL_TESTLOG, LF_TESTCASE_START, ./../../SFZUTF/impl/sfzutf.c:390: sfzutf_
run_tcases: PUBDEFS_Tests
LL_TESTLOG, LF_TEST_START, ./../../SFZUTF/impl/sfzutf.c:401: sfzutf_run_ tcases:
test_ints
LL_TESTLOG, LF_TESTFUNC_INVOKED, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:31: test_ints: test_ints:0
LL_TESTLOG, LF_TESTFUNC_SUCCESS, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:73: test_ints: test_ints:0
LL_TESTLOG, LF_TEST_END, ./../../SFZUTF/impl/sfzutf.c:404: sfzutf_run_tc ases:
test_ints
LL_TESTLOG, LF_TEST_START, ./../../SFZUTF/impl/sfzutf.c:401: sfzutf_run_ tcases:
test_uints
LL_TESTLOG, LF_TESTFUNC_INVOKED, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:75: test_uints: test_uints:0
LL_TESTLOG, LF_TESTFUNC_SUCCESS, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:117: test_uints: test_uints:0
LL_TESTLOG, LF_TEST_END, ./../../SFZUTF/impl/sfzutf.c:404: sfzutf_run_tc ases:
test_uints
LL_TESTLOG, LF_TEST_START, ./../../SFZUTF/impl/sfzutf.c:401: sfzutf_run_ tcases:
test_bool
LL_TESTLOG, LF_TESTFUNC_INVOKED, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:120: test_bool: test_bool:0
LL_TESTLOG, LF_TESTFUNC_SUCCESS, ./../../Framework/FRAMEWORK_TESTSUITE/s
rc/framework_testsuite.c:138: test_bool: test_bool:0
LL_TESTLOG, LF_TEST_END, ./../../SFZUTF/impl/sfzutf.c:404: sfzutf_run_tc ases:
test_bool
LL_TESTLOG, LF_TEST_START, ./../../SFZUTF/impl/sfzutf.c:401: sfzutf_run_ tcases:
test_ptr_size
...
```

The results when using the SFZUTF `testrunner.pl` script:

```
Build/build_linux# ../../SFZUTF/scripts/testrunner.pl ./framework_testsuite
Running test suite Framework_Test Suite:
  Running testcase PUBDEFS_Tests:
    Test test_ints                                              SUCCESS
    Test test_uints                                             SUCCESS
    Test test_bool                                              SUCCESS
    Test test_ptr_size                                          SUCCESS
  Running testcase IMPLDEFS_Tests:
    Test test_min                                               SUCCESS
    Test test_max                                               SUCCESS
    Test test_bit                                               SUCCESS
    Test test_bit_ops                                           SUCCESS
    Test test_aligned_to                                        SUCCESS
    Test test_null                                              SUCCESS
    Test test_offsetof                                          SUCCESS
    Test test_alignmentof                                       SUCCESS
    Test test_compile_static_assert                             SUCCESS
    Test test_precondition:0                                    SUCCESS
    Test test_precondition:1                                    SUCCESS
...
```

A Conventions, References and Compliances

A.1 *Conventions Used in this Manual*

A.1.1 Acronyms

API	Application Program Interface
CAL	Cryptographic Abstraction Layer
CLIB	C library abstraction
EE	Execution Environment
EE_ID	Execution Environment IDentifier
PKCS	Public Key Cryptography Standard
SFZUTF	Software Unit Test Framework
SPAL	SafeZone Platform Abstraction Layer
SST	Secure STorage

A.2 *References*

A.2.1 Autotools

The term *autotools* is often used to refer to the *GNU build system*, a suite of programming tools designed to assist in making source-code packages portable between Unix-like systems. The main tools are *autoconf*, *automake* and *libtool*.

For detailed information on these tools, refer to the following sites:

http://www.gnu.org/software/autoconf/

http://www.gnu.org/software/automake/

http://www.gnu.org/software/libtool/

A.3 *Compliance*

A.3.1 ISO C99 Standard

The final version of the C99 standard can be found at:

http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

(End of Document)