

emApps

Small applications for
embedded systems

User Guide & Reference Manual

Document: UM16001
Software Version: 1.10.0
Revision: 0
Date: December 19, 2025



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2024-2025 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at <https://www.segger.com/legal/privacy-policy/>.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please report it to us and we will try to assist you as soon as possible.

Contact us for further information on topics or functions that are not yet documented.

Print date: December 19, 2025

Software	Date	By	Description
1.10.0	251120	PC	Initial release.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *C: A Reference Manual* by Harbison and Steele (ISBN 0--13--089592X). This book provides a complete description of the C language, the run-time libraries, and a style of C programming that emphasizes correctness, portability, and maintainability.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
Sample comment	Comments in program examples.
User Input	Text entered at the keyboard by a user in a session transcript.
Secret Input	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures.
Emphasis	Very important sections.
SEgger home page	A hyperlink to an external document or web site.

Table of contents

1	Introducing emApps	11
1.1	Getting started	12
1.1.1	Terminology	12
1.1.2	Where to find things	12
1.1.3	Distributed tools	12
1.2	Hello, world!	13
1.2.1	Get ready	13
1.2.2	Select the application	14
1.2.3	Compile the application	14
1.2.4	Run the application	15
1.2.5	Benchmark your machine	15
1.2.6	Apps on real hardware	16
1.2.7	Summary	17
2	Running applications	18
2.1	Built-in applications	19
2.1.1	Preparing a built-in application	19
2.1.2	The load-and-go code	20
2.1.3	Example run	21
2.2	Using multiple entry points	23
2.2.1	Laying out the application	23
2.2.2	A preview of expected output	23
2.2.3	Executing entry points	24
2.3	Finding and executing services at runtime	26
2.3.1	Preparing for service lookup	26
2.3.2	Finding and executing service functions	26
2.3.3	Bechmarking S32 against native code	27
2.4	Running applications on the desktop	29
2.4.1	AppRun entire listing	29
3	Configuring emApps	39
3.1	The configuration file	40
3.2	Configuration symbols	41
3.2.1	Exact-bit-width types	41
3.2.2	S32_CONFIG_EXECUTOR	42
3.2.3	S32_CONFIG_SANDBOX	43
3.2.4	S32_CONFIG_TRACE_INSN	44
3.2.5	S32_CONFIG_TRACE_RAISE	45
3.2.6	Error status codes	46

3.2.7	Sandbox memory access	47
4	C compiler reference	48
4.1	Introduction	49
4.2	Usage	49
4.3	Language	50
4.3.1	Extensions	50
4.3.2	Restrictions	50
4.3.3	Deviations	50
4.4	Preprocessor	51
4.4.1	Predefined macros	51
4.4.2	Pragmas	51
5	Program API reference	53
5.1	Data types	54
5.1.1	S32_EXEC_GEO	55
5.1.2	S32_EXEC_GEO_REGION	56
5.1.3	S32_EXEC_CONTEXT	57
5.1.4	S32_SERVICE_BINDING	58
5.1.5	S32_SERVICE_FUNC	59
5.1.6	S32_EXPORT_INFO	60
5.1.7	S32_EXPORT_FUNC	61
5.2	General functions	62
5.2.1	S32_GetVersionText()	63
5.2.2	S32_GetCopyrightText()	64
5.2.3	S32_GetErrorText()	65
5.2.4	S32_CheckConfig()	66
5.3	Loading functions	67
5.3.1	S32_InitGeo()	68
5.3.2	S32_LoadHeader()	69
5.3.3	S32_LoadFile()	70
5.4	Image inquiry functions	71
5.4.1	S32_FindImport()	72
5.4.2	S32_FindExport()	73
5.4.3	S32_FindExportEx()	74
5.4.4	S32_IterateImports()	75
5.4.5	S32_IterateExports()	76
5.4.6	S32_NativePtr()	77
5.4.7	S32_SandboxAddr()	78
5.4.8	S32_XDataAddr()	79
5.4.9	S32_XDataLen()	80
5.5	Execution functions	81
5.5.1	S32_Exec()	82
5.5.2	S32_PrepareByName()	83
5.5.3	S32_PrepareByAddr()	84
5.5.4	S32_SetUserContext()	85
5.5.5	S32_GetUserContext()	86
5.6	Service API support functions	87
5.6.1	S32_GetArg()	88
5.6.2	S32_AcceptMem()	89
5.6.3	S32_AcceptStr()	90
5.7	Machine state functions	91
5.7.1	S32_RdPC()	92
5.7.2	S32_RdML()	93
5.7.3	S32_RdXR()	94
5.7.4	S32_RdReg()	95
5.7.5	S32_RdMem_U8()	96
5.7.6	S32_RdMem_U16()	97
5.7.7	S32_RdMem_U32()	98

5.7.8	S32_WrMem_U8()	99
5.7.9	S32_WrMem_U16()	100
5.7.10	S32_WrMem_U32()	101
5.7.11	S32_WrMem_Blк()	102
5.7.12	S32_Push()	103
5.7.13	S32_Raise()	104
5.7.14	S32_Continue()	105
6	Application API reference	106
6.1	Runtime functions	107
6.1.1	S32_API_S32_idiv()	108
6.1.2	S32_API_S32_imod()	109
6.1.3	S32_API_S32_udiv()	110
6.1.4	S32_API_S32_umod()	111
6.2	Introspection functions	112
6.2.1	S32_API_S32_FindService()	113
6.2.2	S32_API_S32_ExecService()	114
6.3	C library functions	115
6.3.1	S32_API_C_memset()	116
6.3.2	S32_API_C_memcpy()	117
6.3.3	S32_API_C_memmove()	118
6.3.4	S32_API_C_memchr()	119
6.3.5	S32_API_C_strcpy()	120
6.3.6	S32_API_C_strcat()	121
6.3.7	S32_API_C_strlen()	122
6.3.8	S32_API_C_strchr()	123
6.3.9	S32_API_C_strcmp()	124
6.3.10	S32_API_C_putchar()	125
6.3.11	S32_API_C_puts()	126
6.3.12	S32_API_C_printf()	127
6.3.13	S32_API_C_sprintf()	128
6.4	General library functions	129
6.4.1	S32_API_General_GetTime_ms()	130
6.4.2	S32_API_General_GetTime_us()	131
6.4.3	S32_API_General_Sleep_ms()	132
6.4.4	S32_API_General_Sleep_ns()	133
6.4.5	S32_API_General_MulDiv()	134
6.4.6	S32_API_General_FOpen()	135
6.4.7	S32_API_General_FClose()	136
6.4.8	S32_API_General_FRead()	137
6.4.9	S32_API_General_FWrite()	138
6.5	ISO 7816 functions	139
6.5.1	S32_API_ISO7816_Init()	141
6.5.2	S32_API_ISO7816_Exit()	142
6.5.3	S32_API_ISO7816_PowerOn()	143
6.5.4	S32_API_ISO7816_PowerOff()	144
6.5.5	S32_API_ISO7816_WarmReset()	145
6.5.6	S32_API_ISO7816_ColdReset()	146
6.5.7	S32_API_ISO7816_GetATR()	147
6.5.8	S32_API_ISO7816_Case1()	148
6.5.9	S32_API_ISO7816_Case2()	149
6.5.10	S32_API_ISO7816_Case3()	150
6.5.11	S32_API_ISO7816_Case4()	151
7	SEGGER formatter reference	152
7.1	Introduction	153
7.2	Format control strings	154
7.3	Using the formatter	155
7.3.1	Formatting contexts	155

7.3.2	Implementing printf()	155
7.4	Preprocessor symbols	159
7.4.1	Formatting flags	159
7.5	Data types	160
7.5.1	SEGGER_FORMAT_VALUE	161
7.5.2	SEGGER_FORMAT_GET_STR_FUNC	162
7.5.3	SEGGER_FORMAT_GET_VAL_FUNC	163
7.5.4	SEGGER_FORMAT_FLUSH_FUNC	164
7.5.5	SEGGER_FORMAT_CONTEXT	165
7.5.6	SEGGER_FORMAT_USER_CONTEXT	166
7.6	Functions	167
7.6.1	SEGGER_FORMAT_Init()	168
7.6.2	SEGGER_FORMAT_Exec()	169

Chapter 1

Introducing emApps

This section presents an overview of emApps, its structure, and its capabilities.

1.1 Getting started

This chapter explains how to get started with emApps by using simple examples that can be run on the desktop.

1.1.1 Terminology

This document uses the following conventions to avoid confusion:

- “on the desktop” means “in a command-line environment on your computer.” For Windows machines, this is the Windows command prompt.
- “application” refers to the source code or compiled code of an emApps application that runs in the emApps virtual machine inside of a sandboxed environment.
- “program” refers to the desktop or embedded firmware that is responsible for launching the execution of an (emApps) application and provides the sandboxed environment.
- “tool” refers to a precompiled program used to prepare, trace, execute, profile, or otherwise manipulate applications on the desktop.

1.1.2 Where to find things

The emApps distribution is divided into a number of folders:

Folder	Content
Doc	emApps documentation including the reference guide and release notes.
Bin	All tools, including the emApps C Compiler and the emApps Desktop Executor.
Apps	Source code of demonstration applications that can be run on the desktop and on a SEGGER Flasher . Files in this folder act as examples and can be customized as required.
Src	Source code of the emApps loader and executor. This is compiled into a program that intends to run emApps applications. Files in this folder <i>must not</i> be modified.
Inc	Public header files that define the API provided by the emApps product to the program. The functions defined in these headers are implemented by the source files in the Src and Etc folders. Files in this folder <i>must not</i> be modified.
Config	Configuration files that customize how the emApps product is built, selecting both features and capabilities. Files in this folder are intended to be edited to configure emApps facilities.
Etc	Additional source code that demonstrates how to provide features to an application by adding code to the program. Files in this folder act as examples and can be customized as required.

1.1.3 Distributed tools

Two tools are provided in the distribution:

- The emApps C compiler.
- The emApps desktop executor.

The C compiler translates source code, written in C, into executable code. The emApps desktop executor takes the executable code, loads it, and then executes it in a sandbox.

The compiler and executor are shipped in the `Bin` folder of the distributions.

1.2 Hello, world!

This section will describe how to compile and run the ubiquitous “Hello, World!” application. It skips irrelevant details and concentrates on a simple application running on the desktop as soon as possible.

1.2.1 Get ready

Open up a command-line environment, such as a Windows Command Prompt:

```
Microsoft Windows [Version 10.0.26200.7171]
(c) Microsoft Corporation. All rights reserved.

C:> _
```

Add the emApps `Bin` folder to the search path. For example, if emApps is installed to `C:\Work\SEGGER\emApps`, use:

```
C:> set PATH=C:\Work\SEGGER\emApps\Bin;%PATH%
C:> _
```

Check that both the emApps C compiler `s32cc` and emApps Desktop Executor `apprun` are found and execute:

```
C:> s32cc

SEGGER S32 C Compiler V4.16.0 compiled Nov 27 2025 03:39:06
Copyright (c) 2023-2025 SEGGER Microcontroller GmbH www.segger.com

Usage:
  s32cc [option...] file

Control:
  -v, --verbose          Run in verbose mode

...and so on...

Notes:
  Use '--extra-help' for more detailed help.

C:> aprun

SEGGER emApps Desktop Executor V1.10.0 compiled Nov 28 2025 14:30:57
Copyright (c) 2008-2025 SEGGER Microcontroller GmbH www.segger.com

Usage:
  AppRun [option...] file

Options:
  -nfunc      Execute 'func' after loading          [default: main]
  -t          Trace instructions executed
  -s          Trace lines executed
  -r          Include register dump (before execution) in instruction trace

...and so on...

Streams:
  stdout - Application output.
  stderr - Trace and message output.

C:> _
```

1.2.2 Select the application

With that done, navigate to the folder containing the source code of the demonstration applications directory, `Apps\Src`:

```
C:> cd \Work\SEGGER\emApps\Apps\Src
C:> _
```

In there will be a few demonstration applications:

```
C:> dir /b
Default.h
DemoAckermann.c
DemoDhrystone_100k.c
DemoDhrystone_2k.c
DemoHelloWorld.c
DemoPrimes.c
DemoSieve.c

C:> _
```

The simplest application is `DemoHelloWorld.c`:

```
C:> more DemoHelloWorld.c
void main(void) {
    printf("Hello, world!\n");
}

C:> _
```

This is the application's source code, is known worldwide to programmers, and can't be any simpler.

1.2.3 Compile the application

The emApps executor does not run this source code directly. The human-readable source code must be translated to *machine code* for execution by *compiling* it.

Compile the application:

```
C:> s32cc DemoHelloWorld.c

SEGGER S32 C Compiler V4.16.0 compiled Nov 28 2025 15:26:55
Copyright (c) 2023-2025 SEGGER Microcontroller GmbH www.segger.com

Files written to disk:
  DemoHelloWorld.pex      120 bytes  App executable
  DemoHelloWorld.lst     6590 bytes  App listing

Exported functions:
  main()                  8 bytes stack

Application size:
  20 bytes code
  25 bytes data
  0 bytes xdata
  3 bytes alignment
  8 bytes stack
-----
 56 bytes total runtime image size

Compilation complete: 0 errors.

C:> _
```

The verbose output from compilation can safely be ignored for now, it's good enough to know that the application compiled without error: `0 errors`.

The compiler takes the source code, checks that it is a valid program, and writes a file containing the machine-executable code and other structural information to a file with a `.pex` file extension, in this case `DemoHelloWorld.pex`.

The application is now ready to be run as an emApp.

1.2.4 Run the application

Running, or *executing*, the application requires an emApps sandbox environment. On the desktop, this is provided by the emApps Desktop Executor, AppRun.

To execute the application and see what it does, invoke AppRun and provide the executable-code file `DemoHelloWorld.pex`:

```
C:> apprun DemoHelloWorld.pex

SEGGER emApps Desktop Executor V1.10.0 compiled Nov 28 2025 14:30:57
Copyright (c) 2008-2025 SEGGER Microcontroller GmbH www.segger.com

Hello, world!
Execution complete:
  9 instructions executed in 0.800 ms.
  main() returned 0.

C:> _
```

That's it. The output is printed along with some additional information. To hide the program's output and show only the application's output, redirect the standard error output to the NUL device:

```
C:> apprun DemoHelloWorld.pex 2>NUL
Hello, world!

C:> _
```

1.2.5 Benchmark your machine

Compiling and running other demonstration applications is just as easy. Try compiling the `Dhrystone_100k` application and run it to compare your machine's performance to a MacBook Pro (M2 Max) running emApps:

```
C:> apprun DemoDhrystone_100k.pex

SEGGER emApps Desktop Executor V1.10.0 compiled Nov 28 2025 16:18:59
Copyright (c) 2008-2025 SEGGER Microcontroller GmbH www.segger.com

204498 Dhrystones/s, 116 DMIPS

Execution complete:
  93300642 instructions executed in 490.430 ms.
  S32 benchmarks at 190.243 MIPS.
  main() returned 0.

C:> _
```

Note: As shipped, the desktop executor supports application tracing and profiling which would not be present in production. If the program is configured without these features, its performance doubles.

The statistic that benchmarks the performance of the S32 in MIPS is only presented when the application takes more than 100 ms to complete.

1.2.6 Apps on real hardware

Running an application on the desktop is convenient, but emApps is designed primarily to execute within an embedded system. Fortunately, SEGGER has been using the technology behind emApps for years and it is deployed in Flashers.

Flashers can run these demonstration applications without modification. To do so, make sure that the [Flasher Software and Documentation Pack](#), version 8.90 or later is installed and that the Flasher's firmware is up to date—use Flasher Configurator to update the Flasher's firmware if required.

The default installation directory for the Flasher software V8.90 on Windows is /Program Files/SEGGER/Flasher_V890. Add this to the path:

```
C:> set PATH=C:\Program Files\SEGGER\Flasher_V890;%PATH%
C:> _
```

Check that the Flasher App Runner utility is found and executes:

```
C:> flasherrun --version
SEGGER Flasher App Runner Utility V8.94 (Compiled Dec 10 2025 15:16:51)
Copyright (c) 2025-2025 SEGGER Microcontroller GmbH www.segger.com
DLL version V8.94, compiled Dec 10 2025 14:54:09

C:> _
```

Check that the correct directory is selected:

```
C:> cd
\Work\SEGGER\emApps\Apps\Src

C:> _
```

Plug in the Flasher and run the “Hello, World!” and Dhrystone applications on the Flasher using FlasherRun, in just the same way as they are run on the desktop using AppRun:

```
C:> flasherrun DemoHelloWorld.pex
SEGGER Flasher App Runner Utility V8.90 (Compiled Nov 26 2025 17:52:51)
Copyright (c) 2025-2025 SEGGER Microcontroller GmbH www.segger.com
DLL version V8.90, compiled Nov 26 2025 17:29:40

Connecting to Flasher via USB...O.K.
Firmware: J-Link / Flasher Compact V7 compiled Nov 26 2025 15:58:58
Hardware version: V7.00
Flasher uptime (since boot): 0d 00h 02m 39s
License(s): JFlash, GDB
USB speed mode: High speed (480 MBit/s)
VTref=0.000V
Successfully loaded '\Work\SEGGER\emApps\Apps\Src\DemoHelloWorld.pex'.
Executing app... (press <Enter> to cancel)
  Hello, world!

App execution finished with return value 0 after 0.031ms.

C:> flasherrun DemoDhrystone_100k.pex
SEGGER Flasher App Runner Utility V8.90 (Compiled Nov 26 2025 17:52:51)
Copyright (c) 2025-2025 SEGGER Microcontroller GmbH www.segger.com
DLL version V8.90, compiled Nov 26 2025 17:29:40

Connecting to Flasher via USB...O.K.
Firmware: J-Link / Flasher Compact V7 compiled Nov 26 2025 15:58:58
Hardware version: V7.00
Flasher uptime (since boot): 0d 00h 00m 49s
License(s): JFlash, GDB
USB speed mode: High speed (480 MBit/s)
```

```
VTref=0.000V
Successfully loaded '\\Work\SEGGER\emApps\Apps\Src\DemoDhrystone_100k.pex'.
Executing app... (press <Enter> to cancel)
  30969 Dhrystones/s, 17 DMIPS

App execution finished with return value 0 after 3229.743ms.

C:> _
```

A brief aside for the technically curious

From using AppRun on the desktop, we know Dhrystone executes 93,300,642 instructions on the S32E virtual CPU. The Flasher accomplishes this in 3,229,743 microseconds. Therefore, the Flasher executes applications at approximately 30 million instructions per second.

1.2.7 Summary

This has been a brief, simplified tour of emApps that demonstrates some of its major features. emApps offers more capabilities than those demonstrated here and the following sections will describe how to extend the software to expose emApps for use by internal and external customers.

Chapter 2

Running applications

This section describes how load and run applications.

2.1 Built-in applications

An application would typically be loaded from disk or from a network connection in order that it can be updated or extended when new features or bug fixes become available. Being able to replace the application clearly means it can't be compiled into the program as that would mean it's almost impossible to replace.

It can be desirable, though, to build an application into the program so that it is always available. In this way, a hardware test application compiled into the program is always ready, as is some system configuration or recovery application, if desired. Delivering prebuilt applications in this manner is just another way of using applications and emApps.

2.1.1 Preparing a built-in application

This section explains how to load and run "Hello, World!" as a built-in application. The program is straightforward, simple to describe, and easy to understand.

The application's executable image is already provided as a built-in application in the file `Apps/Src/DemoHelloWorld_pex.h`. The program to load and execute the "Hello, World!" application is contained in the file `Apps/Prj/RunHelloWorld.c`.

To generate a built-in application with the emApps C compiler, use the `--embed` option. The built-in "Hello, World!" application is created as follows:

```
C:> s32cc --embed DemoHelloWorld.c

SEGGGER S32 C Compiler V4.16.0 compiled Nov 28 2025 15:26:55
Copyright (c) 2023-2025 SEGGGER Microcontroller GmbH www.segger.com

Files written to disk:
  DemoHelloWorld.pex          120 bytes  App executable
  DemoHelloWorld_pex.h       679 bytes  Built-in executable
  DemoHelloWorld.lst         6065 bytes App listing

Exported functions:
  main()                      8 bytes stack

Application size:
  20 bytes code
  25 bytes data
  0 bytes xdata
  3 bytes alignment
  8 bytes stack
-----
 56 bytes total runtime image size

Compilation complete: 0 errors.

C:> _
```

An additional output file that ends `_pex.h` contains a declaration of an array initialized to the binary content of the equivalent emApps executable file:

```
C:> more DemoHelloWorld_pex.h
static const unsigned char _aDemoHelloWorld[] = {
  0x50, 0x43, 0x56, 0x32, 0x08, 0x00, 0x00, 0x00,
  0x2C, 0x01, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x16, 0x00, 0x00, 0x00,
  0x6D, 0x61, 0x69, 0x6E, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x1A, 0x00, 0x00, 0x00,
  0x70, 0x72, 0x69, 0x6E, 0x74, 0x66, 0x00, 0x00,
  0x2C, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x65,
  0x6C, 0x6C, 0x6F, 0x2C, 0x20, 0x77, 0x6F, 0x72,
```

```

0x6C, 0x64, 0x21, 0x0A, 0x00, 0x00, 0xC6, 0x00,
0x07, 0x10, 0x93, 0x00, 0x00, 0x00, 0x06, 0x00,
0x0B, 0x00, 0x00, 0x00,
};
C:> _

```

2.1.2 The load-and-go code

The following lays out the load and execution process:

```

int main(void) {
    S32_EXEC_GEO      Geo;
    S32_EXEC_CONTEXT Ctx;
    S32_U8            * pImage;
    S32_I32           Status;
    //
    if (S32_CheckConfig() != S32_ERROR_NONE) {
        printf("S32 configuration check failed\n");
        exit(EXIT_FAILURE);
    }
    //
    pImage = NULL;
    Status = S32_InitGeo(&Geo, ❶
                        _aServices,
                        sizeof(_aServices)/sizeof(_aServices[0]));
    if (Status >= 0) {
        Status = S32_LoadHeader(&Geo, _aDemoHelloWorld, sizeof(_aDemoHelloWorld)); ❷
        if (Status >= 0) {
            pImage = malloc(sizeof(_aDemoHelloWorld) + Status); ❸
            if (pImage == NULL) {
                Status = -1; // General error, out of memory
            } else {
                memcpy(pImage, _aDemoHelloWorld, sizeof(_aDemoHelloWorld)); ❹
                Status = S32_LoadFile(&Geo, pImage); ❺
                if (Status >= 0) {
                    Status = S32_PrepareByName(&Geo, &Ctx, "main"); ❻
                    if (Status >= 0) {
                        Status = S32_Exec(&Ctx); ❼
                    }
                }
            }
        }
    }
    //
    free(pImage); ❽
    printf("\n*** Application exited with %s (%d)\n", S32_GetErrorText(Status), Status);
    //
    return Status;
}

```

The steps are:

❶ Initialize geometry

To prepare loading an application, the *application geometry* structure must be initialized. Application geometry is used to locate each section in the PEX file and is updated as loading progresses. In addition, an array of *service bindings* are presented that enumerate the services available to an application. This will be discussed in detail later, but for now the table contains a single entry that exposes `printf` to be used by the application, and only `printf`. The geometry is not needed when it comes to execution so can be disposed of after loading.

2 Load the header

After initializing the geometry, the header is presented for verification. Only the first 32 bytes of the header need be presented: if more are presented, the excess will be ignored, and if fewer are presented, the file is invalid and an error is returned.

The value returned from `S32_LoadHeader()` is negative when an error occurs; a return value that is zero or greater indicates the number of additional bytes that must be allocated beyond the application file's image to run the application. The exact details of this will be discussed later, and for the "Hello World!" application this number is zero.

3 Allocate space to run the application

The memory used to execute the image is entirely allocated by the program: the emApps library does not use `malloc()` or `free()` internally. This step allocates the space for the application file, plus any additional requirement reported by `S32_LoadHeader()`. This example uses `malloc()`, but equally some other memory-allocation function could be used, or even a fixed-size static array.

4 Prepare the execution image

Once memory is allocated for the image, the entire PEX file must be copied to the start. `memcpy()` is used here to copy the file stored in a static array into the memory allocated for execution.

5 Validate the image

Now that the image is in read-write memory, it can be fully validated and all references to service functions that it uses can be resolved. The loader is very careful to examine the structure of the application file in memory, ensuring that it is not damaged in an unintentional or malicious manner. To do this, each offset and reference within the application image is validated to ensure it is entirely contained in the image.

The loader processes the list of imports in the application and matches them to the the service bindings given to `S32_InitGeo()`. If an import is required by the application but is not present in the API service table, an error is returned by `S32_LoadFile()` and the application must not be executed.

6 Prepare to execute

The image is loaded and validated and passes all integrity checks and all imports are available. The function to execute, in this case `main()` is prepared to run using `S32_PrepareByName()`. The arguments to this function are the geometry, an execution context, and the name of the function to run. The execution context, which contains the state of the virtual machine, is entirely initialized by calling `S32_PrepareByName()` and is not initialized separately.

If the function cannot be found in the application's export list, an error is returned.

7 Execute the function

If all the previous steps execute correctly, execution can commence. Execution of the function is started by calling `S32_Exec()` with the execution context. The application runs until the top-level function called returns or an exception is raised.

8 Wrap up

Once execution returns to the program, the memory used for the application can be freed and the value returned from executing the application's `main()` is used as an exit code.

2.1.3 Example run

When compiled and run, this output is:

```
C:> RunHelloWorld
Hello, world!
```

```
*** Application exited with no error (0)
```

```
C:> _
```

2.2 Using multiple entry points

All applications presented so far have run to completion and stopped. This section will present another way of calling applications, essentially as a library of functions or as a dynamic-link library, to perform some actions, maintaining state between each call.

2.2.1 Laying out the application

The following is an application that exports three entry points: `Init()`, `Main()`, and `Fini()`:

```
#pragma S32C export("Init");    ❶
#pragma S32C export("Main", Fini);

static int Data = 10;        ❷

void Init(void) {
    printf("In Init(), Data=%d\n", Data);    ❸
    Data += 1;    ❹
}

void Main(void) {
    printf("In Main(), Data=%d\n", Data);    ❸
    Data += 2;    ❹
}

void Fini(void) {
    printf("In Fini(), Data=%d\n", Data);    ❸
    Data += 3;    ❹
}
```

❶ Export entry points

The compiler must be told which functions to export as entry points. The entry points are published using `#pragma S32C export` which takes the name of the function to export. The pragma will accept function names with or without quotation marks according to preference, and will accept a comma-separated list of such names.

❷ Declare static data

Any data declared with static storage class will retain values over the lifetime of the application. In this application, `Data` will be updated by successive entry points and its value displayed prior to update.

❸ Display current data on each entry

Each entry point displays the value of `Data` immediately on entry.

❹ Update the state

Each entry point then alters the value of `Data` as part of its execution.

2.2.2 A preview of expected output

The code to call the three entry points is presented in the following section. Concentrating on the application, if the three functions `Init()`, `Main()`, and `Fini()` are called in that order by C code, the expected output is:

```
In Init(), Data=10
In Main(), Data=11
In Fini(), Data=13
```

Similarly, if those three emApps entry points are executed from a program, the output will be identical: static data retains its state between executions.

2.2.3 Executing entry points

This section demonstrates how to call the three entry points in sequence. The application's executable image is already provided as a built-in application in the file `Apps/Src/DemoMultiEntry_pex.h`. The program to load and execute the application is contained in the file `Apps/Prj/RunMultiEntry.c`.

Loading the application follows the same steps as the "Hello, World!" application. Once loaded, the multi-entry application can be queried to expose one, some, or all of its entry points. The presence of a single named entry point can be queried using `S32_FindExport()` and prepared for execution using `S32_PrepareByAddr()`:

```
int InitAddr;
int FiniAddr;
int MainAddr;
//
InitAddr = S32_FindExport(&Geo, "Init");    ❶
FiniAddr = S32_FindExport(&Geo, "Fini");
MainAddr = S32_FindExport(&Geo, "Main");
//
if (InitAddr >= 0) {    ❷
    if (S32_PrepareByAddr(&Geo, &Ctx, InitAddr) >= 0) {    ❸
        S32_Exec(&Ctx);    ❹
    }
}
//
if (MainAddr >= 0) {    ❺
    S32_PrepareByAddr(&Geo, &Ctx, MainAddr);
    S32_Exec(&Ctx);
}
//
S32_PrepareByAddr(&Geo, &Ctx, FiniAddr);    ❻
S32_Exec(&Ctx);
```

❶ Find entry points by name

Use `S32_FindExport()` to find the entry point, passing the application geometry and the entry point name. The entry point is the name of the exported function without parentheses. In this case, three entry points are looked up. The application will call `Init()` if it exists, followed by `Main()` if it exists, followed by `Fini()` if it exists.

❷ Check if `Init()` exists

If the entry point for `Init()` does not exist or the executable image has been corrupted in a detectable manner, a negative error status is returned. This will be either `S32_ERROR_NOT_FOUND` or `S32_ERROR_BAD_PEX_FILE`.

❸ Prepare for execution

Once the entry point is determined, it can be prepared for execution using `S32_PrepareByAddr()`. In normal operation, the address returned from `S32_FindExport()` will be valid and `S32_PrepareByAddr()` will also succeed with this address, but `S32_PrepareByAddr()` checks the address for validity anyway to ensure that the application is prepared correctly.

❹ Execute `Init()`

After preparation is complete, the application is ready to launch, so it can be started using `S32_Exec()`.

5 Abbreviated execution of Main()

This fragment of code checks that `Main()` exists as an entry point, and then proceeds to prepare for execution and executes it. It does not check that preparation succeeds as, if preparation fails, an exceptional condition is registered in the execution context and any attempt to execute will immediately fail.

5 Minimal execution of Fini()

This fragment of code takes brevity even further: any negative value is never a valid address, so `S32_PrepareByAddr()` will immediately fail, and so will `S32_Exec()`. Of course, it is better programming style to check for errors when they occur, this is simply an example demonstrating that detected errors do not cause erratic execution of random application code.

2.3 Finding and executing services at runtime

Direct references made to functions that the application uses, offered by the program through the populated API service table, are resolved when an application is loaded. If the named service is not present, loading fails with a “not found” error.

It may be convenient to dynamically determine, when the application is running, whether a service is offered by the program or not. This can be used to introduce updated product models that offer better performance by moving compute-heavy code into the program, but otherwise continue to work on older hardware, whilst the application’s executable is the same across all models. Or it might be used to run the same application on base models and more featureful models, adapting behavior as necessary.

2.3.1 Preparing for service lookup

The service lookup feature is exposed to the application developer through `Default.h` as follows:

```

/*****
 *
 *      Introspection functions
 */
__imp int      SYS_FindService(const char *sName);
__imp unsigned SYS_ExecService(int ServiceHandle, ...);

```

To support service lookup in the program, the functions `SYS_FindService()` and `SYS_ExecService()` must be present in the API table, mapping to `S32_API_S32_FindService()` and `S32_API_S32_ExecService()`:

```

static S32_SERVICE_BINDING _aServices[] = {
    { "SYS_FindService", S32_API_S32_FindService },
    { "SYS_ExecService", S32_API_S32_ExecService },
    ...
};

```

2.3.2 Finding and executing service functions

The following program uses the introspection feature to see whether `printf()` is an installed service and, if it is, calls it to print a message. This program is contained in the file `Apps/Src/DemoFindService.c`.

```

int main(void) {
    int PrintfHandle; ❶
    //
    PrintfHandle = SYS_FindService("printf"); ❷
    if (PrintfHandle > 0) {
        SYS_ExecService(PrintfHandle, "Hello, %s!\n", "World"); ❸
        return 0;
    } else {
        return PrintfHandle;
    }
    return 0;
}

```

❶ Declare a handle

Service functions that are looked up at runtime are not invoked by calling through a function pointer. Instead, a *handle* is used that refers to the particular service. This declares such a handle.

❷ Look up the service function

The function `S32_FindService()` takes a service function name, without parentheses, and returns a handle that identifies the service. If the service is not found, the handle is a negative error code.

③ Execute the service

As explained above, the service is not invoked through a function pointer, but by `S32_ExecService()` which calls the particular service, passing through the provided arguments. Note that it is the user's responsibility to provide the correct number of arguments, each of the correct type, to match the prototype of the invoked service function.

2.3.3 Bechmarking S32 against native code

The following application benchmarks `strcpy()` written in S32 instructions against a version provided by the program, if such a service function exists.

This program is contained in the file `Apps/Src/DemoBenchStrcpy.c`.

```
char aDst[8192];
char aSrc[8192];

int main(void) {
    unsigned T;
    int StrcpyHandle;
    int i;
    //
    StrcpyHandle = SYS_FindService("strcpy");
    //
    if (StrcpyHandle > 0) {
        //
        printf("strcpy() is present as a service function\n\n");
        printf("Benchmarking:\n\n");
        //
        memset(aSrc, 'x', 8191);
        //
        T = SYS_GetTime_ms();
        for (i = 0; i < 2000; ++i) {
            strcpy(aDst, aSrc);
        }
        T = SYS_GetTime_ms() - T;
        printf(" strcpy() using S32 code: %5d ms\n", T);
        //
        T = SYS_GetTime_ms();
        for (i = 0; i < 2000; ++i) {
            SYS_ExecService(StrcpyHandle, aDst, aSrc);
        }
        T = SYS_GetTime_ms() - T;
        printf(" strcpy() using service: %5d ms\n\n", T);
        printf("Done\n\n");
        //
    } else {
        printf("strcpy() is not present as a service.");
    }
    return 0;
}
```

Running it shows the relative performance:

```
C:> aprun DemoBenchStrcpy.pex

SEGGER emApps Desktop Executor V1.10.0 compiled Dec 17 2025 22:33:12
Copyright (c) 2008-2025 SEGGER Microcontroller GmbH www.segger.com

strcpy() is present as a service function

Benchmarking:

strcpy() using S32 code:   494 ms
strcpy() using service:    10 ms
```

Done

Execution complete:

98354052 instructions executed in 504.835 ms.

532 benchmarks at 194.824 MIPS.

main() returned 0.

C:> _

2.4 Running applications on the desktop

The desktop application executor that is used throughout this manual is provided in source code in Tools/AppRun/Src/AppRun.cpp. This can be customized, for instance to emulate the system that the application will eventually be installed on, and to start developing applications before even prototype hardware is available.

2.4.1 AppRun entire listing

```

/*****
 *                               *
 *      (c) SEGGER Microcontroller GmbH      *
 *      The Embedded Experts                *
 *      www.segger.com                      *
 *****/

----- END-OF-HEADER -----

Purpose      : S32 sample application executor and profiler.

*/

/*****
 *
 *      #include section
 *
 *****/

#include <string>
#include <list>
#include <set>
#include <map>
#include <format>
#include <print>
#include <chrono>
#include <filesystem>
#include "S32.h"
#include "S32_Lib.h"

/*****
 *
 *      Defines, configurable
 *
 *****/

//
// Coarse-grained inclusion of service functions.
//
#if !defined(CONFIG_API_C)
#define CONFIG_API_C      1
#endif
#if !defined(CONFIG_API_GENERAL)
#define CONFIG_API_GENERAL  1
#endif
#if !defined(CONFIG_API_ISO_7816)
#define CONFIG_API_ISO_7816 0
#endif

/*****
 *
 *      Prototypes
 *
 *****/

[[noreturn]] static void _Die(const std::string &Text);

/*****
 *
 *      Static const data
 *
 *****/

static std::vector<S32_SERVICE_BINDING> _aServices{
  { "__S32_umod",      S32_API_S32_umod      },
  { "__S32_udiv",     S32_API_S32_udiv     },
  { "__S32_imod",     S32_API_S32_imod     },
  { "__S32_idiv",     S32_API_S32_idiv     },
  { "SYS_FindService", S32_API_S32_FindService },
  { "SYS_ExecService", S32_API_S32_ExecService }
};

```

```

#if CONFIG_API_C
{ "memset",          S32_API_C_memset      },
{ "memcpy",         S32_API_C_memcpy     },
{ "memmove",        S32_API_C_memmove    },
{ "memchr",         S32_API_C_memchr     },
{ "strcpy",         S32_API_C_strcpy     },
{ "strchr",         S32_API_C_strchr     },
{ "strncpy",        S32_API_C_strncpy    },
{ "strlen",         S32_API_C_strlen     },
{ "puts",           S32_API_C_puts       },
{ "printf",         S32_API_C_printf     },
{ "sprintf",        S32_API_C_sprintf    },
#endif
#if CONFIG_API_GENERAL
{ "UTIL_MulDiv",    S32_API_General_MulDiv  },
{ "SYS_GetTime_ms", S32_API_General_GetTime_ms },
{ "SYS_GetTime_us", S32_API_General_GetTime_us },
{ "SYS_Sleep_ms",   S32_API_General_Sleep_ms },
{ "SYS_Sleep_ns",   S32_API_General_Sleep_ns },
{ "FOpen",          S32_API_General_FOpen  },
{ "FClose",         S32_API_General_FClose },
{ "FRead",           S32_API_General_FRead  },
{ "FWrite",          S32_API_General_FWrite },
#endif
#if CONFIG_API_ISO_7816
{ "ISO7816_Init",   S32_API_ISO7816_Init    },
{ "ISO7816_Exit",   S32_API_ISO7816_Exit    },
{ "ISO7816_PowerOn", S32_API_ISO7816_PowerOn },
{ "ISO7816_PowerOff", S32_API_ISO7816_PowerOff },
{ "ISO7816_Case1",  S32_API_ISO7816_Case1  },
{ "ISO7816_Case2",  S32_API_ISO7816_Case2  },
{ "ISO7816_Case3",  S32_API_ISO7816_Case3  },
{ "ISO7816_Case4",  S32_API_ISO7816_Case4  },
{ "ISO7816_WarmReset", S32_API_ISO7816_WarmReset },
{ "ISO7816_ColdReset", S32_API_ISO7816_ColdReset },
{ "ISO7816_GetATR", S32_API_ISO7816_GetATR  },
#endif
};

/*****
 *
 *   Static data
 *
 *****/

static std::map<unsigned, std::string> InsnMap;
static std::map<unsigned, std::string> CodeMap;
static std::list<std::string> ListFile;
static std::string LstName;
static std::string PrfName;
static std::string PexName;
static std::map<unsigned, unsigned> InsnCntMap;
static std::set<std::string> Imports;
static std::set<std::string> MissingImports;
static std::string MainName;
static S32_U8 * pPexImage;
static long PexImageLen;
static bool TraceInsns; // Print instruction trace
static bool TraceRegs; // Print registers with trace
static bool TraceSource; // Print source line associated with instruction
static bool Profile; // Write instruction profile file
static bool PrintAPI; // Print application API entries
static bool PrintExports; // Print exported functions, do not run
static bool PrintImports; // Print imported functions, do not run
static bool PrintMissing; // Print imported functions missing from exposed API
static bool Verbose; // Print configuration information
static unsigned long long AppInsnCnt; // Number of instructions executes by the application

/*****
 *
 *   Static code
 *
 *****/

/*****
 *
 *   _Die()
 *
 *   Function description
 *   Die fatally.
 *
 *   Parameters
 *   Text - Descriptive message describing failure.
 *****/
static void _Die(const std::string &Text) {

```

```

std::println(stderr, "fatal: {}", Text);
exit(EXIT_FAILURE);
}

/*****
 *
 *     _IsHex()
 *
 * Function description
 *   String contains all hexadecimal digit?
 *
 * Parameters
 *   Str - String to test.
 *
 * Return value
 *   True if only valid hexadecimal digits.
 */
static bool _IsHex(const std::string &Str) {
    for (auto c : Str) {
        if ('0' <= c && c <= '9') {
            /* Pass */
        } else if ('a' <= c && c <= 'f') {
            /* Pass */
        } else if ('A' <= c && c <= 'F') {
            /* Pass */
        } else {
            return false;
        }
    }
    return true;
}

/*****
 *
 *     _DeHex()
 *
 * Function description
 *   Convert ASCII hexadecimal digit to binary.
 *
 * Parameters
 *   c - Character to convert.
 *
 * Return value
 *   Decoded value.
 */
static int _DeHex(char c) {
    if ('0' <= c && c <= '9') {
        return c - '0';
    } else if ('a' <= c && c <= 'f') {
        return c - 'a' + 10;
    } else if ('A' <= c && c <= 'F') {
        return c - 'A' + 10;
    } else {
        return -1;
    }
}

/*****
 *
 *     _IsExecLine()
 *
 * Function description
 *   Does line correspond to executable content?
 *
 * Parameters
 *   Text - Line from listing.
 *
 * Return value
 *   True if the line corresponds to executable content.
 */
static bool _IsExecLine(const std::string &Text) {
    return Text.length() > 13 &&
        _IsHex(Text.substr(0, 6)) &&
        Text[6] == ' ' &&
        Text[7] == ' ' &&
        _IsHex(Text.substr(8, 4)) &&
        Text[12] == ' ';
}

/*****
 *
 *     _LineLC()
 *
 * Function description
 *   Decode line's location counter.
 *
 * Parameters

```

```

*   Text - Line from listing.
*/
static unsigned _LineLC(const std::string &Text) {
    return _DeHex(Text[0]) * 0x100000 +
           _DeHex(Text[1]) * 0x10000 +
           _DeHex(Text[2]) * 0x1000 +
           _DeHex(Text[3]) * 0x100 +
           _DeHex(Text[4]) * 0x10 +
           _DeHex(Text[5]) * 0x1;
}

/*****
*
*   _DeriveFileNames()
*
*   Function description
*   Construct listing and profile file names.
*
*   Parameters
*   FileName - File name of PEX file.
*/
static void _DeriveFileNames(const std::string &FileName) {
    LstName = std::filesystem::path(FileName).replace_extension(".lst").string();
    PrfName = std::filesystem::path(FileName).replace_extension().string() + "_pro.txt";
}

/*****
*
*   _RdPexFile()
*
*   Function description
*   Read PEX file into memory in binary mode.
*
*   Parameters
*   FileName - File name of PEX file.
*/
static void _RdPexFile(const std::string &FileName) {
    FILE * pPexFile;
    //
    pPexFile = fopen(FileName.c_str(), "rb");
    if (pPexFile == NULL) {
        _Die(std::format("cannot open '{} for reading", FileName));
    }
    //
    fseek(pPexFile, 0, SEEK_END);
    PexImageLen = ftell(pPexFile);
    fseek(pPexFile, 0, SEEK_SET);
    //
    pPexImage = new S32_U8[PexImageLen];
    fread(pPexImage, 1, PexImageLen, pPexFile);
    //
    fclose(pPexFile);
}

/*****
*
*   _RdLstFile()
*
*   Function description
*   Read and parse S32 compiler listing file.
*
*   Parameters
*   FileName - File name of listing file.
*/
static void _RdLstFile(const std::string &FileName) {
    std::string CodeLine;
    //
    FILE *pLstFile = fopen(FileName.c_str(), "r");
    if (pLstFile == NULL) {
        _Die(std::format("cannot open '{} for reading", FileName));
    }
    //
    for (;;) {
        std::string Line;
        int c;
        //
        if (feof(pLstFile)) {
            break;
        }
        for (;;) {
            c = fgetc(pLstFile);
            if (c == '\n' || c == EOF) {
                break;
            }
            Line += c;
        }
        ListFile.push_back(Line);
    }
}

```

```

//
if (Line.length() >= 6 && Line[5] == ':') {
    //
    // C source line.
    //
    CodeLine = Line;
} else if (_IsExecLine(Line)) {
    //
    // Assembly language line.
    //
    InsnMap[_LineLC(Line)] = Line;
    CodeMap[_LineLC(Line)] = CodeLine;
}
}
}

/*****
*
*     _VisitImport()
*
* Function description
* Visitor callback to collect imported function information.
*
* Parameters
* pGeo - Pointer to application geometry.
* sName - Exported function name.
* Index - Imported function index.
* pUserCtx - Pointer to user-supplied context.
*/
static S32_I32 _VisitImport(S32_EXEC_GEO *pGeo,
                           const char *sName,
                           unsigned Index,
                           void *pUserCtx) {
    bool Present = false;
    //
    Imports.insert(sName);
    //
    for (unsigned i = 0; !Present && i < _aServices.size(); ++i) {
        Present = strcmp(_aServices[i].sName, sName) == 0;
    }
    //
    if (!Present) {
        MissingImports.insert(sName);
    }
    //
    return S32_ERROR_NONE;
}

/*****
*
*     _PrMissing()
*
* Function description
* Print imported functions missing from API table.
*
* Parameters
* pGeo - Pointer to application geometry.
*/
static void _PrMissing(S32_EXEC_GEO *pGeo) {
    S32_IterateImports(pGeo, _VisitImport, NULL);
    std::println(stderr, "Functions imported by application but missing from API:");
    if (MissingImports.empty()) {
        std::println(stderr, " None");
    } else {
        for (auto &Name : MissingImports) {
            std::println(stderr, " {}()", Name);
        }
    }
}

/*****
*
*     _VisitExport()
*
* Function description
* Visitor callback to print exported function information.
*
* Parameters
* pGeo - Pointer to application geometry.
* sName - Exported function name.
* Index - Exported function index.
* ParaCnt - Number of function parameters.
* Entry - Entry offset of function relative to execution region.
* pUserCtx - Pointer to user-supplied context.
*/
static S32_I32 _VisitExport(S32_EXEC_GEO *pGeo,
                            const char *sName,

```

```

        unsigned      Index,
        unsigned      ParaCnt,
        S32_U32       Entry,
        void          * pUserCtx) {
    std::println(stderr, " {}(), {} parameters, entry point 0x{:04X}", sName, ParaCnt, Entry);
    return S32_ERROR_NONE;
}

/*****
 *
 *      _PrExports()
 *
 * Function description
 * Print functions exported by the application.
 *
 * Parameters
 * pGeo - Pointer to application geometry.
 */
static void _PrExports(S32_EXEC_GEO *pGeo) {
    std::println(stderr, "Functions exported by application:");
    S32_IterateExports(pGeo, _VisitExport, NULL);
}

/*****
 *
 *      _PrImports()
 *
 * Function description
 * Print functions imported by the application.
 *
 * Parameters
 * pGeo - Pointer to application geometry.
 */
static void _PrImports(S32_EXEC_GEO *pGeo) {
    S32_IterateImports(pGeo, _VisitImport, NULL);
    std::println(stderr, "Functions imported by application:");
    for (auto &Name : Imports) {
        std::println(stderr, " {}()", Name);
    }
}

/*****
 *
 *      _PrAPI()
 *
 * Function description
 * Print functions exported by emApps API.
 */
static void _PrAPI() {
    std::println(stderr, "Functions exported by the API:");
    for (auto API : _aServices) {
        std::println(stderr, " {}()", API.sName);
    }
}

/*****
 *
 *      _WrPrfFile()
 *
 * Function description
 * Write profile to disk.
 */
static void _WrPrfFile() {
    FILE *pPrfFile = fopen(PrfName.c_str(), "w");
    for (auto Line : ListFile) {
        if (_IsExecLine(Line)) {
            Line = std::format("{:7d} ", InsnCntMap[_LineLC(Line)]) + Line;
        } else {
            Line = " " + Line;
        }
        std::println(pPrfFile, "{}", Line);
    }
    fclose(pPrfFile);
}

/*****
 *
 *      _ShowUsage()
 *
 * Function description
 * Print help usage.
 */
static void _ShowUsage() {
    std::println(stderr, "Usage:");
    std::println(stderr, " AppRun [option...] file");
    std::println(stderr);
    std::println(stderr, "Options:");
}

```

```

std::println(stderr, " -nfunc      Execute 'func' after loading          [default: main]");
std::println(stderr, " -t        Trace instructions executed");
std::println(stderr, " -s        Trace lines executed");
std::println(stderr, " -r        Include register dump (before execution) in instruction trace");
std::println(stderr, " -p        Write instruction profile file");
std::println(stderr, " -e        List functions provided by emApps API");
std::println(stderr, " -i        List application's imported functions");
std::println(stderr, " -x        List application's exported functions");
std::println(stderr, " -m        List application's imported functions missing from exposed API");
std::println(stderr, " -a        List application's imported, exported, and missing functions");
std::println(stderr, " -?, --help Print help information");
std::println(stderr);
std::println(stderr, "Streams:");
std::println(stderr, "  stdout - Application output.");
std::println(stderr, "  stderr - Trace and message output.");
exit(EXIT_SUCCESS);
}

/*****
 *
 *      Public code
 *
 *****/

/*****
 *
 *      S32_X_TraceRaise()
 *
 *      Function description
 *      Entry point for exception trace.
 *
 *      Parameters
 *      pCtx - Pointer to execution context.
 */
void S32_X_TraceRaise(S32_EXEC_CONTEXT *pCtx, S32_I32 Exception) {
    //
    // Break raised for function termination?
    //
    if (Exception == S32_ERROR_BRK && S32_RdPC(pCtx) == 4) {
        return;
    }
    //
    std::println(stderr, "Exception {} raised ({})", Exception, S32_GetErrorText(Exception));
    std::println(stderr);
    //
    std::println(stderr, "PC: {:08X}", S32_RdPC(pCtx));
    std::println(stderr, "ML: {:08X}", S32_RdML(pCtx));
    //
    for (unsigned Rn = 0; Rn < 16; ++Rn) {
        if (Rn % 8 == 0) {
            std::print(stderr, "R{}:", Rn);
        }
        std::print(stderr, " {:08X}", S32_RdReg(pCtx, Rn));
        if (Rn % 8 == 7) {
            std::println(stderr);
        }
    }
    std::println(stderr);
}

/*****
 *
 *      S32_X_TraceInsn()
 *
 *      Function description
 *      Entry point for per-instruction trace.
 *
 *      Parameters
 *      pCtx - Pointer to execution context.
 */
void S32_X_TraceInsn(S32_EXEC_CONTEXT *pCtx) {
    static std::string LastCodeLine;
    //
    ++AppInsnCnt;
    //
    if (TraceSource || TraceInsns || Profile) {
        S32_U32 PC = S32_RdPC(pCtx);
        if (PC == 4) {
            // BRK on RET from top-level function
        } else if (InsnMap.find(PC) == InsnMap.end()) {
            S32_Raise(pCtx, S32_ERROR_ACC_VIOLATION);
        } else {
            InsnCntMap[PC] += 1;
            if (TraceSource) {
                if (LastCodeLine != CodeMap[PC]) {
                    LastCodeLine = CodeMap[PC];
                }
            }
        }
    }
}

```

```

        if (!LastCodeLine.empty()) {
            if (TraceInsns) { std::println(stderr); }
            std::println(stderr, "{}", LastCodeLine);
            if (TraceInsns) { std::println(stderr); }
        }
    }
}
if (TraceInsns) {
    std::string Txt = InsnMap[PC];
    if (Txt.find(';') != std::string::npos) {
        Txt = Txt.substr(0, Txt.find(';'));
    }
    if (TraceRegs) {
        while (Txt.size() < 60) {
            Txt += ' ';
        }
        std::print(stderr, "{}", Txt);
        std::print(stderr, " ");
        for (int i = 0; i < 16; ++i) {
            std::print(stderr, "{:08X}", S32_RdReg(pCtx, i));
            if (i % 8 == 3) {
                std::print(stderr, " ");
            } else if (i % 8 == 7) {
                std::print(stderr, " ");
            } else {
                std::println(stderr, " ");
            }
        }
    } else {
        while (Txt.ends_with(" ")) {
            Txt = Txt.substr(0, Txt.length()-1);
        }
        std::println(stderr, "{}", Txt);
    }
    std::println(stderr);
}
}
}
}

/*****
*
*   main()
*
*   Function description
*   Main function.
*
*   Parameters
*   argc - Argument count.
*   argv - Argument vector.
*
*   Return value
*   Exit code.
*/
int main(int argc, const char** argv) {
    S32_EXEC_GEO      Geo;
    S32_EXEC_CONTEXT Ctx;
    S32_EXPORT_INFO  Info;
    bool             Newline;
    S32_US           * pExecImage;
    int              LoadStatus;
    int              ExecStatus;
    std::chrono::high_resolution_clock::rep
                    MicrosecondDuration;

    //
    MainName = "main";
    //
    std::println(stderr);
    std::println(stderr, "SEGGER emApps Desktop Executor V{} compiled " __DATE__ " " __TIME__, S32_GetVersionText());
    std::println(stderr, "{} www.segger.com", S32_GetCopyrightText());
    std::println(stderr);
    //
    for (int i = 1; i < argc; ++i) {
        std::string Arg = argv[i];
        //
        if (Arg == "-t")           { TraceInsns = true; }
        else if (Arg == "-r")      { TraceInsns = true; TraceRegs = true; }
        else if (Arg == "-s")      { TraceSource = true; }
        else if (Arg == "-p")      { Profile = true; }
        else if (Arg == "-e")      { PrintAPI = true; }
        else if (Arg == "-x")      { PrintExports = true; }
        else if (Arg == "-i")      { PrintImports = true; }
        else if (Arg == "-m")      { PrintMissing = true; }
        else if (Arg == "-a")      { PrintImports = true; PrintExports = true; PrintMissing = true; }
        else if (Arg == "-v")      { Verbose = true; }
        else if (Arg == "-n")      { MainName = "main"; }
        else if (Arg == "-?")      { _ShowUsage(); }
    }
}

```

```

else if (Arg == "--help")           { _ShowUsage(); }
else if (Arg.starts_with("-n"))     { MainName = Arg.substr(2); }
else if (Arg.starts_with("-"))     { _Die(std::format("unrecognized option '{}'", Arg)); }
else if (!PexName.empty())         { _Die("cannot execute multiple applications"); }
else                               { PexName = Arg; }
}
//
if (PrintAPI && PexName.empty()) {
    _PrAPI();
    exit(EXIT_SUCCESS);
}
//
if (PexName.empty()) {
    _ShowUsage();
}
//
if (Verbose) {
    std::println(stderr, "Sandbox:          {}", S32_CONFIG_SANDBOX ? "Enabled" : "Disabled");
    std::println(stderr, "Instruction trace: {}", S32_CONFIG_TRACE_INSN ? "Available" : "Not available");
    std::println(stderr, "Exception trace:   {}", S32_CONFIG_TRACE_RAISE ? "Available" : "Not available");
    std::println(stderr);
}
_DeriveFileNames(PexName);
_RdPexFile(PexName);
if (TraceSource || TraceInsns || Profile) {
    _RdLstFile(LstName);
}
//
LoadStatus = S32_InitGeo(&Geo, _aServices.data(), (unsigned)_aServices.size());
if (LoadStatus < 0) {
    return LoadStatus;
}
//
LoadStatus = S32_LoadHeader(&Geo, pPexImage, PexImageLen);
if (LoadStatus >= 0) {
    //
    // Create an execution image for the application.
    //
    pExecImage = new S32_U8[PexImageLen + LoadStatus];
    memset(pExecImage, 0, PexImageLen + LoadStatus); // Fill stack and xdata, not strictly necessary...
    memcpy(pExecImage, pPexImage, PexImageLen);
    //
    LoadStatus = S32_LoadFile(&Geo, pExecImage);
    if (LoadStatus >= 0) {
        Newline = false;
        if (PrintAPI) {
            _PrAPI();
            Newline = true;
        }
        if (PrintImports) {
            if (Newline) { std::println(stderr); }
            _PrImports(&Geo);
            Newline = true;
        }
        if (PrintExports) {
            if (Newline) { std::println(stderr); }
            _PrExports(&Geo);
            Newline = true;
        }
        if (PrintMissing) {
            if (Newline) { std::println(stderr); }
            _PrMissing(&Geo);
            Newline = true;
        }
        if (Newline) {
            exit(EXIT_SUCCESS);
        }
        //
        LoadStatus = S32_FindExportEx(&Geo, MainName.c_str(), &Info);
        if (LoadStatus < 0) {
            std::println(stderr, "fatal: function '{}()' is not exported", MainName.c_str());
            std::println(stderr);
            _PrExports(&Geo);
            exit(EXIT_FAILURE);
        } else {
            LoadStatus = S32_PrepareByName(&Geo, &Ctx, "main");
            if (LoadStatus == 0) {
                auto OriginTime = std::chrono::high_resolution_clock::now();
                ExecStatus = S32_Exec(&Ctx);
                auto T1 = std::chrono::high_resolution_clock::now();
                MicrosecondDuration = std::chrono::duration_cast<std::chrono::microseconds>(T1 - OriginTime).count();
            } else if (LoadStatus > 0) {
                _Die(std::format("unsupported: '{}'" declared with arguments", MainName));
            }
        }
    }
}
}
}
}

```

```

//
if (LoadStatus < 0) {
    std::println(stderr, "Loading failed");
    std::println(stderr, " {} returned - {}.", LoadStatus, S32_GetErrorText(LoadStatus));
    //
    if (LoadStatus == S32_ERROR_NOT_FOUND) {
        std::println(stderr);
        _PrMissing(&Geo);
    }
    return LoadStatus;
} else {
    std::println("Execution complete:");
    if (Profile) {
        _WrPrfFile();
        std::println(stderr, " Instruction profile written to '{}'.", PrfName);
    }
    if (AppInsCnt > 0) {
        std::println(stderr, " {} instructions executed in {:.3f} ms.",
            AppInsCnt,
            MicrosecondDuration / 1.e3);
        if (MicrosecondDuration > 100000) {
            std::println(stderr, " S32 benchmarks at {:.3f} MIPS.",
                (double)AppInsCnt / MicrosecondDuration);
        }
    } else {
        std::println(stderr, " Active for {:.3f} ms.", MicrosecondDuration / 1.e3);
    }
    //
    if (ExecStatus == S32_ERROR_NONE) {
        std::println(stderr, " main() returned {}.", (S32_I32)S32_RdReg(&Ctx, S32_REG_R0));
        return 0;
    } else {
        std::println(stderr, " exception {} raised - {}.", ExecStatus, S32_GetErrorText(ExecStatus));
        return ExecStatus;
    }
}
}
}

/***** End of file *****/

```

Chapter 3

Configuring emApps

This section describes how to configure the emApps module when integrating it into a program.

3.1 The configuration file

emApps is completely configured entirely by editing the file `Config/S32_Conf.h`. It defines some preprocessor macros that configures the executor to use, how memory is accessed, what features the executor provides.

Configuration options have defaults, set using by the file `Inc/S32_ConfDefaults.h`. This file *must not* be edited configure emApps, all configuration must be accomplished by editing `Config/S32_Conf.h` to avoid problems when contacting SEGGER support.

3.2 Configuration symbols

3.2.1 Exact-bit-width types

Description

Types that have an exact number of bits.

Definition

```
#define S32_U8      unsigned char
#define S32_U16    unsigned short
#define S32_U32    unsigned int
#define S32_U64    unsigned long long
#define S32_I8     signed char
#define S32_I16    signed short
#define S32_I32    signed int
```

Symbols

Definition	Description
S32_U8	Consider using <code>uint8_t</code> if available.
S32_U16	Consider using <code>uint16_t</code> if available.
S32_U32	Consider using <code>uint32_t</code> if available.
S32_U64	Consider using <code>uint64_t</code> if available.
S32_I8	Consider using <code>int8_t</code> if available.
S32_I16	Consider using <code>int16_t</code> if available.
S32_I32	Consider using <code>int32_t</code> if available.

Additional information

These must be configured to exactly match the number of bits required. The defaults are generally good for 32-bit architectures. These are not configured to the “`intx_t`” and “`uintx_t`” types from `<stdint.h>` as the ISO standard makes no guarantee that such types exist.

3.2.2 S32_CONFIG_EXECUTOR

Description

Configuration of S32 execution engine implementation.

Definition

```
#define S32_CONFIG_EXECUTOR 0
```

Additional information

If set to zero, the generic C executor is selected to execute applications. If set to one, the Arm executor is selected to execute applications.

3.2.3 S32_CONFIG_SANDBOX

Description

Configuration of environment protected by sandbox.

Definition

```
#define S32_CONFIG_SANDBOX 1
```

Additional information

If set to zero, the generic C executor will not perform software-based access violation trapping. In this configuration, it is expected that the executor is run in its own protected environment provided by a memory protection unit (MPU) or a memory management unit (MMU) under control of an operating system.

If set to nonzero, the generic C executor performs software-based access violation trapping on all accesses: instruction fetch and data read/writes.

3.2.4 S32_CONFIG_TRACE_INSN

Description

Configuration of application execution trace.

Definition

```
#define S32_CONFIG_TRACE_INSN 0
```

Additional information

If set to zero, the generic C executor executes without compiled-in instruction trace support.

If set to nonzero, the generic C executor is compiled with a function call to `S32_X_TraceInsn()` before each instruction is executed.

3.2.5 S32_CONFIG_TRACE_RAISE

Description

Configuration of application exception trace.

Definition

```
#define S32_CONFIG_TRACE_RAISE 0
```

Additional information

If set to zero, the generic C executor executes without compiled-in exception trace support.

If set to nonzero, the generic C executor is compiled with a function call to `S32_X_TraceRaise()` immediately prior to an exception being raised.

3.2.6 Error status codes

Description

Error codes returned by emApps API functions.

Definition

```
#define S32_ERROR_NONE          ( 0)
#define S32_ERROR_BRK          (-11)
#define S32_ERROR_UNDEF_INST   (-12)
#define S32_ERROR_ACC_VIOLATION (-13)
#define S32_ERROR_ZERO_DIVIDE  (-14)
#define S32_ERROR_NOT_FOUND     (-15)
#define S32_ERROR_BAD_PEX_FILE  (-16)
#define S32_ERROR_BAD_VERSION  (-17)
#define S32_ERROR_BAD_CONFIG    (-18)
```

Symbols

Definition	Description
S32_ERROR_NONE	No error.
S32_ERROR_BRK	Executed a BRK instruction.
S32_ERROR_UNDEF_INST	Attempt to execute an undefined instruction.
S32_ERROR_ACC_VIOLATION	Memory access outside of sandbox.
S32_ERROR_ZERO_DIVIDE	Division by zero.
S32_ERROR_NOT_FOUND	File or service function not found.
S32_ERROR_BAD_PEX_FILE	Error in structure of PEX file.
S32_ERROR_BAD_VERSION	PEX file requires more recent executor version.
S32_ERROR_BAD_CONFIG	Incorrect configuration of data types, context offsets, or service table.

Additional information

These error codes, other than [S32_ERROR_NONE](#), are user-configurable in order that they do not conflict with existing codes used by other software (and then combined with emApps). All errors must be assigned negative values.

3.2.7 Sandbox memory access

Description

These macros read and write data to the sandbox. They can be configured for architectures that are big-endian, that fault on misaligned accesses, or both.

Definition

```
#define S32_RDU16LE(ADDR)      (*(S32_U16 *) (ADDR))
#define S32_RDU32LE(ADDR)    (*(S32_U32 *) (ADDR))
#define S32_WRU16LE(ADDR,VAL) (*(S32_U16 *) (ADDR) = (S32_U16)(VAL))
#define S32_WRU32LE(ADDR,VAL) (*(S32_U32 *) (ADDR) = (S32_U32)(VAL))
```

Symbols

Definition	Description
S32_RDU16LE(ADDR)	Read 16-bit unsigned, little endian
S32_RDU32LE(ADDR)	Read 32-bit unsigned, little endian
S32_WRU16LE(ADDR,VAL)	Write 16-bit unsigned, little endian
S32_WRU32LE(ADDR,VAL)	Write 32-bit unsigned, little endian

Chapter 4

C compiler reference

This chapter contains the documentation for the emApps C compiler.

4.1 Introduction

The compiler is located in the `Bin` folder of the emApps product. It is used to compile application source code files into portable executable files that can be loaded and executed in an emApps sandbox environment. A linker is not required.

The compiler adheres to the C90 standard with some C99 extensions (see *Extensions* on page 50). However, there are some restrictions and deviations that are listed in *Restrictions* on page 50 and *Deviations* on page 50.

4.2 Usage

The compiler is called from the command line as follows:

```
S32CC [options] source-file
```

A list with all available options can be shown by invoking the compiler without any arguments.

To compile a source code file, it needs at least the path to the `Inc` directory, which is defined as a *system include*, and the file to compile. The `Inc` directory contains the file `Default.h`, which is automatically included by the compiler and provides the sandbox's C interface to the user.

When the compiler finishes compilation successfully, there are two new files in the same folder as the source code file:

- The portable executable file, with extension `.pex`, containing the code and data for the application.
- A listing file, with extension `.lst`, containing a listing of the executable form of the application.

4.3 Language

4.3.1 Extensions

- C++ single-line comments are supported.
- Data within functions need not immediately following the open brace of a block, but can be defined anywhere within the function.
- Anonymous structures and unions are supported.
- Enumerated types can specify the underlying storage unit. For instance, `enum char { x=1 };` declares an enumeration type with underlying 8-bit storage. By default, enumerations are declared with underlying 32-bit `int` storage.

4.3.2 Restrictions

The supported C programming language is subject to the following restrictions:

- `volatile` is not supported and will be ignored if used. All pointers are automatically considered volatile by the code generator.
- Bitfields are not supported.
- `float` and `double` are not supported.
- `long long` is not supported.
- Passing structures by value to functions and returning structures from functions is not supported.
- Structure assignment is supported (`x = y;` where `x` and `y` are structures), but cascaded structure assignment `x = y = z;` is not.
- The comma operator in expressions is not supported.

4.3.3 Deviations

The supported C programming language is subject to following deviations:

- As there is no linker, the `extern` specifier serves is not required.

4.4 Preprocessor

The compiler has a standard C90 preprocessor with following extensions:

- The preprocessor controls `#warning` and `#remark` function exactly as `#error` but produce diagnostics with warning and remark severity.
- Macros can be defined with variadic arguments lists with the variable arguments collected into the preprocessor symbol `__VA_ARGS__`.

4.4.1 Predefined macros

Following macros are predefined by the compiler:

Macro	Description	Example Output
<code>__DATE__</code>	Date the source code file was compiled.	"Feb 20 2025"
<code>__TIME__</code>	Time the source code file was compiled.	"12:02:05"
<code>__FILE__</code>	Name of the source code file.	"App.c"
<code>__LINE__</code>	The line number the macro was used in.	80
<code>__S32C__</code>	The full version number containing the major, minor and patch number (Mmmp).	21202
<code>__S32C_MAJOR__</code>	The major version number.	2
<code>__S32C_MINOR__</code>	The minor version number.	12
<code>__S32C_PATCHLEVEL__</code>	The patch level.	3
<code>__S32C_VERSION__</code>	The version string ("M.mm.pp").	"2.12.3"

4.4.2 Pragmas

The following pragmas are supported by the S32 C compiler.

Pragma	Explanation
<code>xdata_size</code>	Specifies the size of the memory allocated for the xdata area.
<code>stack_size</code>	Specifies the size of the memory allocated for the stack area.
<code>export</code>	Specifies a function to be exported from the application.

4.4.2.1 xdata_size

Syntax

```
#pragma S32C xdata_size (Size)
```

Description

This pragma specifies the size of the memory allocated for the xdata area pointed to by `__S32_XDataBase`. The xdata area can be used by the program to provide the data to an application, or for the application to return data to the program.

Parameter	Description
<code>Size</code>	Size of the xdata area to be allocated.

Example

```
#pragma S32C xdata_size (0x800)
```

4.4.2.2 stack_size

Syntax

```
#pragma S32C stack_size (size)
```

Description

This pragma is not usually required because the S32 compiler analyzes each entry point to determine maximum stack depth and sets this automatically. However, additional space can be added “for safety reasons” by using this pragma and override the compiler’s calculation.

The compiler will disregard stack sizes smaller than that calculated by static analysis, and therefore it is not possible to produce an application that will fail because of lack of stack space. It is also possible to increase the allocated stack above that calculated by the compiler by a given amount that is specified using a '+’.

```
#pragma S32C stack_size(4096) // Set stack size to 4096
#pragma S32C stack_size(0x1000) // Same as above
#pragma S32C stack_size(+256) // Calculated stack size increased by 256 bytes.
```

In the case where there is direct or indirect recursion or when using function pointers, an exact stack bound cannot be calculated. The compiler calculates the maximum stack requirement of all functions, including those with direct or indirect recursion up to the point of recursion, and uses this as the calculated bound. A warning is generated when the compiler detects such recursion, and the cycle is described in the compiler’s map file.

In this case, it is the user’s responsibility to determine the minimum stack space required by whatever means are at hand, and instruct the compiler by using this pragma. If the declared stack space is smaller than that actually required, overwriting of other data or the program will occur, but sandboxing ensures that no write occurs to data outside of the sandbox.

4.4.2.3 export

Syntax

```
#pragma S32C export("name")
#pragma S32C export(name)
```

Description

The function with the given name is placed into the export list contained in the application’s executable such that it can be located using `S32_FindExport()`.

Chapter 5

Program API reference

This chapter explains the API functions of emApps that are needed for loading and execution. The emApps API is kept as simple as possible to provide a straightforward way to integrate emApps into a product.

5.1 Data types

Type	Description
S32_EXEC_GEO	Layout geometry of a PEX file.
S32_EXEC_GEO_REGION	Describes a region of the PEX file.
S32_EXEC_CONTEXT	Execution context.
S32_SERVICE_BINDING	Association of service name and its implementation.
S32_SERVICE_FUNC	Prototype for service function implementation.
S32_EXPORT_INFO	Exported function information.
S32_EXPORT_FUNC	Prototype for exported function enumeration.

5.1.1 S32_EXEC_GEO

Description

Layout geometry of a PEX file.

Type definition

```
typedef struct {
    S32_SERVICE_FUNC *const * pIdxFn;
    S32_U32 IdxCnt;
    const S32_SERVICE_BINDING * pImpFn;
    S32_U32 ImpCnt;
    S32_U32 XDataLen;
    S32_U32 StackLen;
    S32_U32 ExportCnt;
    S32_EXEC_GEO_REGION FileRgn;
    S32_EXEC_GEO_REGION ExportAddrRgn;
    S32_EXEC_GEO_REGION ExportNameRgn;
    S32_EXEC_GEO_REGION ImportPatchRgn;
    S32_EXEC_GEO_REGION ImportNameRgn;
    S32_EXEC_GEO_REGION ExecRgn;
} S32_EXEC_GEO;
```

Structure members

Member	Description
pIdxFn	Pointer to dispatch table for function imported by index. Included for backwards compatibility with no support.
IdxCnt	Number of functions imported by index. Included for backwards compatibility with no support.
pImpFn	Pointer to dispatch table for function imported by name.
ImpCnt	Number of functions imported by index.
XDataLen	Size of additional memory beyond execution image.
StackLen	Size of stack at end of image.
ExportCnt	Number of exported functions.
FileRgn	Covers entire PEX file.
ExportAddrRgn	Export address table.
ExportNameRgn	Export name table.
ImportPatchRgn	Import patch table.
ImportNameRgn	Import name table.
ExecRgn	Execution region (does not include stack and xdata, i.e. only the data delivered in the PEX file).

5.1.2 S32_EXEC_GEO_REGION

Description

Describes a region of the PEX file.

Type definition

```
typedef struct {  
    S32_U32 Off;  
    S32_U32 Len;  
    S32_U8 * pData;  
} S32_EXEC_GEO_REGION;
```

Structure members

Member	Description
Off	Byte offset from start of PEX file to region.
Len	Size of region in bytes.
pData	Native pointer to data associated with region.

5.1.3 S32_EXEC_CONTEXT

Description

Execution context.

Type definition

```
typedef struct S32_EXEC_CONTEXT_s {
    S32_U32      aReg[];
    S32_U32      PC;
    S32_U32      AluOut;
    S32_U8       * pMem;
    S32_U32      ML;
    S32_SERVICE_FUNC *const * pIdxFn;
    S32_U32      IdxCnt;
    const S32_SERVICE_BINDING * pImpFn;
    S32_U32      ImpCnt;
    S32_I32      XR;
    void         * pUserCtx;
} S32_EXEC_CONTEXT;
```

Structure members

Member	Description
aReg	[0-14]: general purpose registers. [15]: stack pointer.
PC	Program counter, sandbox address, next instruction to execute.
AluOut	Last ALU output.
pMem	Native pointer to base of S32 application memory space.
ML	Highest sandbox address of S32 application memory space + 1.
pIdxFn	Pointer to dispatch table for functions imported by index.
IdxCnt	Number of functions imported by index.
pImpFn	Pointer to dispatch table for functions imported by name.
ImpCnt	Number of functions imported by name.
XR	Exception register holding exception raised during execution.
pUserCtx	Pointer to user context associated with execution context.

Additional information

Fields in this structure are considered private. Access functions are provided, e.g. [S32_ReadReg\(\)](#), to extract useful information from the context. SEGGER does not guarantee that the fields in this structure preserve their name, order, type, or presence between emApps versions.

5.1.4 S32_SERVICE_BINDING

Description

Association of service name and its implementation.

Type definition

```
typedef struct {  
    const char * sName;  
    S32_SERVICE_FUNC * pExec;  
} S32_SERVICE_BINDING;
```

Structure members

Member	Description
sName	Service name
pExec	Service execution function

5.1.5 S32_SERVICE_FUNC

Description

Prototype for service function implementation.

Type definition

```
typedef S32_U32 S32_SERVICE_FUNC(S32_EXEC_CONTEXT * pCtx);
```

5.1.6 S32_EXPORT_INFO

Description

Exported function information.

Type definition

```
typedef struct {  
    const char * sName;  
    S32_I32      Index;  
    S32_U32      Addr;  
    S32_U32      ParaCnt;  
} S32_EXPORT_INFO;
```

Structure members

Member	Description
sName	Pointer to zero-terminated function name.
Index	Exported function index, starting at zero.
Addr	Entry point address into the execution region.
ParaCnt	Number of parameters function expects.

5.1.7 S32_EXPORT_FUNC

Description

Prototype for exported function enumeration.

Type definition

```
typedef S32_I32 S32_EXPORT_FUNC(
    S32_EXEC_GEO * pGeo,
    const char * sName,
    unsigned Index,
    unsigned ParaCnt,
    S32_U32 CodeAddr,
    void * pUserCtx);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
sName	Pointer to zero-terminated function name.
Index	Zero-based index of exported function.
ParaCnt	Number of parameters expected by function.
CodeAddr	Sandbox address of function entry point.
pUserCtx	Pointer to user-provided context.

5.2 General functions

The table below lists the functions that return information related to the emApps product.

Function	Description
S32_GetVersionText()	Get emApps product version.
S32_GetCopyrightText()	Get emApps copyright attribution.
S32_GetErrorText()	Get printable text for error.
S32_CheckConfig()	Check configuration of emApps product.

5.2.1 S32_GetVersionText()

Description

Get emApps product version.

Prototype

```
char *S32_GetVersionText(void);
```

Return value

Pointer to a zero-terminated string that contains the version number of the product.

5.2.2 S32_GetCopyrightText()

Description

Get emApps copyright attribution.

Prototype

```
char *S32_GetCopyrightText(void);
```

Return value

Pointer to a zero-terminated string that contains the copyright string.

5.2.3 S32_GetErrorText()

Description

Get printable text for error.

Prototype

```
char *S32_GetErrorText(S32_I32 Status);
```

Parameters

Parameter	Description
Status	Error or exception code.

Return value

Pointer to string describing error or exception.

5.2.4 S32_CheckConfig()

Description

Check configuration of emApps product.

Prototype

```
S32_I32 S32_CheckConfig(void);
```

Return value

= 0 Configuration check passed.
≠ 0 Configuration check failed.

Additional information

If this function fails, check the configuration of the S32 to ensure that data types are correct and, if using the assembly language executor, that calculated offsets into the execution context are correct.

5.3 Loading functions

The table below lists the functions that are used to prepare an application for execution.

Function	Description
S32_InitGeo()	Initialize application geometry.
S32_LoadHeader()	Validate application file header.
S32_LoadFile()	Load entire application file.

5.3.1 S32_InitGeo()

Description

Initialize application geometry.

Prototype

```
S32_I32 S32_InitGeo(      S32_EXEC_GEO      * pGeo,  
                        const S32_SERVICE_BINDING * pImp,  
                        unsigned                ImpCnt);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
pImp	Pointer to function-imported-by-name table.
ImpCnt	Number of entries in function-imported-by-name table.

Return value

≥ 0 Success.
< 0 Failure status.

5.3.2 S32_LoadHeader()

Description

Validate application file header.

Prototype

```
S32_I32 S32_LoadHeader( S32_EXEC_GEO * pGeo,
                       const S32_U8   * pFile,
                       S32_U32       FileLen);
```

Parameters

Parameter	Description
pGeo	Pointer to object receiving application geometry.
pFile	Pointer to PEX file header.
FileLen	Total length of PEX file, in bytes.

Return value

- ≥ 0 Success, number of bytes of additional memory required to hold the stack and xdata areas.
- < 0 Failure.

Additional information

The PEX file header consists of eight 32-bit words, and this is all that is examined by `S32_LoadHeader()`. If [FileLen](#) indicates the PEX file does not contain a complete header, an error is returned.

If the file contains a complete header, it is examined and validated as far as it can be. It is not necessary to provide the entire file at this time, only the header part (i.e. 32 bytes). Full validation is deferred to `S32_LoadFile()`. Note that there is no security risk associated with partial validation of the header, everything that is required during a call to `S32_LoadHeader()` undergoes validation.

On successful completion, the number of bytes required to hold the stack and xdata areas is returned, which can be used to allocate memory for the execution image in memory before calling `S32_LoadFile()`.

See also

[S32_LoadFile](#) on page 70.

5.3.3 S32_LoadFile()

Description

Load entire application file.

Prototype

```
S32_I32 S32_LoadFile(S32_EXEC_GEO * pGeo,
                   S32_U8      * pFile);
```

Parameters

Parameter	Description
<code>pGeo</code>	Pointer to object containing application geometry.
<code>pFile</code>	Pointer to PEX file content plus additional memory allocated for stack and xdata areas.

Return value

≥ 0 Success, PEX file is valid.
 < 0 Failure.

Additional information

Once the PEX file header is validated successfully, the entire execution image is presented for validation in preparation for application execution. `S32_LoadFile()` validates that the file is structurally sound, that import-by-name functions exist before execution, and that all in-file structural “section offsets” point within and are entirely contained by the section content pointed to.

See also

`S32_LoadHeader` on page 69.

5.4 Image inquiry functions

The table below lists the functions that are used to query information on the loaded application.

Function	Description
S32_FindImport()	Find index of import-by-name function.
S32_FindExport()	Find address of exported function.
S32_FindExportEx()	Find address of exported function, extended.
S32_IterateImports()	Iterate over import table.
S32_IterateExports()	Iterate over export table.
S32_NativePtr()	Convert sandbox address to native pointer.
S32_SandboxAddr()	Convert native pointer to sandbox address.
S32_XDataAddr()	Get sandbox xdata address.
S32_XDataLen()	Get sandbox xdata size.

5.4.1 S32_FindImport()

Description

Find index of import-by-name function.

Prototype

```
S32_I32 S32_FindImport( S32_EXEC_GEO * pGeo,  
                      const char * sName);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
sName	Pointer to zero-terminated function name.

Return value

- ≥ 0 Index of imported-by-name function in API table.
- < 0 Imported function not present.

5.4.2 S32_FindExport()

Description

Find address of exported function.

Prototype

```
S32_I32 S32_FindExport(      S32_EXEC_GEO * pGeo,  
                          const char * sName);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
sName	Pointer to zero-terminated function name.

Return value

- ≥ 0 Success, sandbox address of function.
- < 0 Failure status, function does not exist or PEX file is damaged.

5.4.3 S32_FindExportEx()

Description

Find address of exported function, extended.

Prototype

```
S32_I32 S32_FindExportEx(    S32_EXEC_GEO * pGeo,  
                           const char * sName,  
                           S32_EXPORT_INFO * pInfo);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
sName	Pointer to zero-terminated function name.
pInfo	Pointer to object that receives additional exported function information. This pointer must not be null.

Return value

- ≥ 0 Success, sandbox address of function.
- < 0 Failure status, function does not exist or PEX file is damaged.

5.4.4 S32_IterateImports()

Description

Iterate over import table.

Prototype

```
S32_I32 S32_IterateImports(S32_EXEC_GEO * pGeo,  
                          S32_IMPORT_FUNC * pfVisitor,  
                          void * pUserCtx);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
pfVisitor	Pointer to visitor function, called for each import.
pUserCtx	Pointer to user-provided context, passed to visitor function.

Return value

≥ 0 Success.
< 0 Failure status.

5.4.5 S32_IterateExports()

Description

Iterate over export table.

Prototype

```
S32_I32 S32_IterateExports(S32_EXEC_GEO * pGeo,  
                          S32_EXPORT_FUNC * pfVisitor,  
                          void * pUserCtx);
```

Parameters

Parameter	Description
pGeo	Pointer to application geometry.
pfVisitor	Pointer to visitor function, called for each export.
pUserCtx	Pointer to user-provided context, passed to visitor function.

Return value

≥ 0 Success.
< 0 Failure status.

5.4.6 S32_NativePtr()

Description

Convert sandbox address to native pointer.

Prototype

```
void *S32_NativePtr(S32_EXEC_CONTEXT * pCtx,  
                  S32_U32 Addr);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Application (sandbox) address.

Return value

Pointer that is a native address. Note that a sandbox address of zero, corresponding to a null pointer, is converted to a pointer to the start of memory and not to a native null pointer.

It is expected that sandbox addresses are checked for acceptability outside of this function. [S32_AcceptStr\(\)](#) and [S32_AcceptMem\(\)](#) ensure that accesses through null pointers are detected and, therefore, any string or memory block that passes through [S32_AcceptStr\(\)](#) or [S32_AcceptMem\(\)](#) is guaranteed to point to a valid memory within the application which can be converted to a native pointer for use.

5.4.7 S32_SandboxAddr()

Description

Convert native pointer to sandbox address.

Prototype

```
S32_U32 S32_SandboxAddr(S32_EXEC_CONTEXT * pCtx,  
                        void * pVoid);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
pVoid	Native address.

Return value

Sandbox address. A null pointer is converted to the sandbox address zero.

5.4.8 S32_XDataAddr()

Description

Get sandbox xdata address.

Prototype

```
S32_U32 S32_XDataAddr(S32_EXEC_GEO * pGeo);
```

Parameters

Parameter	Description
pGeo	Pointer to object containing application geometry.

Return value

Base address of application's xdata area.

5.4.9 S32_XDataLen()

Description

Get sandbox xdata size.

Prototype

```
S32_U32 S32_XDataLen(S32_EXEC_GEO * pGeo);
```

Parameters

Parameter	Description
pGeo	Pointer to object containing application geometry.

Return value

Size in bytes of application's xdata area.

5.5 Execution functions

The table below lists the functions that are used to control execution.

Function	Description
S32_PrepareByName()	Prepare context, by function name.
S32_PrepareByAddr()	Prepare context, by address.
S32_Exec()	Start execution.
S32_SetUserContext()	Associate user context with an execution context.
S32_GetUserContext()	Query user context associated with an execution context.

5.5.1 S32_Exec()

Description

Start execution.

Prototype

```
S32_I32 S32_Exec(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

= S32_ERROR_NONE No execution error.
≠ S32_ERROR_NONE Termination condition.

Additional information

This function starts executing instructions until an exception is raised. When a top-level function returns, a “break” exception is raised causing termination.

5.5.2 S32_PrepareByName()

Description

Prepare context, by function name.

Prototype

```
S32_I32 S32_PrepareByName(      S32_EXEC_GEO    * pGeo,  
                               S32_EXEC_CONTEXT * pCtx,  
                               const char       * sFunc);
```

Parameters

Parameter	Description
pGeo	Pointer to object containing application geometry.
pCtx	Pointer to execution context to prepare.
sFunc	Exported function name to be executed.

Return value

≥ 0 Success, context is prepared, number of parameters function expects.
< 0 Failure.

Additional information

If preparation fails, an exceptional condition is registered in the execution context and an attempt to start execution will immediately fail.

5.5.3 S32_PrepareByAddr()

Description

Prepare context, by address.

Prototype

```
S32_I32 S32_PrepareByAddr(S32_EXEC_GEO * pGeo,  
                        S32_EXEC_CONTEXT * pCtx,  
                        S32_U32 Addr);
```

Parameters

Parameter	Description
pGeo	Pointer to object containing application geometry.
pCtx	Pointer to execution context to prepare.
Addr	Address of function to be executed.

Return value

≥ 0 Success, context is prepared.
< 0 Failure.

Additional information

If preparation fails, an exceptional condition is registered in the execution context and an attempt to start execution will immediately fail.

5.5.4 S32_SetUserContext()

Description

Associate user context with an execution context.

Prototype

```
void S32_SetUserContext(S32_EXEC_CONTEXT * pCtx,  
                       void * pUserCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.
<code>pUserCtx</code>	Pointer to user context.

5.5.5 S32_GetUserContext()

Description

Query user context associated with an execution context.

Prototype

```
void *S32_GetUserContext(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Pointer to the user context associated with the execution context.

5.6 Service API support functions

The table below lists the functions that are used to implement service API functions that operate within a sandbox.

Function	Description
S32_GetArg()	Read function argument.
S32_AcceptMem()	Check address range against sandbox.
S32_AcceptStr()	Check string against sandbox.

5.6.1 S32_GetArg()

Description

Read function argument.

Prototype

```
S32_U32 S32_GetArg(S32_EXEC_CONTEXT * pCtx,  
                  int Para);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Para	Parameter index, zero being the first function parameter.

Return value

If the parameter lies outside of the sandbox address space, the return value has zero substituted and an access violation exception is raised in the execution context.

5.6.2 S32_AcceptMem()

Description

Check address range against sandbox.

Prototype

```
int S32_AcceptMem(S32_EXEC_CONTEXT * pCtx,  
                 S32_U32          Addr,  
                 S32_U32          Len);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address of first address of range.
Len	Size in bytes of the address range to test.

Return value

- = 0 Address range fails the test and is wholly or partly outside of the sandbox address space.
- ≠ 0 Address range passes the test and is wholly within the sandbox address space.

Additional information

If any of the address range lies outside of the sandbox address space, an access violation exception is raised in the execution context.

5.6.3 S32_AcceptStr()

Description

Check string against sandbox.

Prototype

```
int S32_AcceptStr(S32_EXEC_CONTEXT * pCtx,  
                S32_U32           Addr,  
                S32_U32           MaxLen);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address of the zero-terminated string.
MaxLen	Maximum number of bytes to examine for null terminator.

Return value

- = 0 String fails the test and is wholly or partly outside of the application's address space.
- ≠ 0 String passes the test and is wholly within the application's address space.

Additional information

If the string wholly or partially lies outside of the application's address space, an access violation exception is raised.

5.7 Machine state functions

The table below lists the functions that are used to read and write the application memory space.

Function	Description
Accessing registers	
S32_RdPC()	Read program counter register.
S32_RdML()	Read memory limit register.
S32_RdXR()	Read raised-exception status.
S32_RdReg()	Read general-purpose register.
Accessing memory	
S32_RdMem_U8()	Read sandbox memory, U8.
S32_RdMem_U16()	Read sandbox memory, U16.
S32_RdMem_U32()	Read sandbox memory, U32.
S32_WrMem_U8()	Write sandbox memory, U8.
S32_WrMem_U16()	Write sandbox memory, U16.
S32_WrMem_U32()	Write sandbox memory, U32.
S32_WrMem_Blk()	Write sandbox memory, block.
Utility functions	
S32_Push()	Push to stack.
S32_Raise()	Raise exception.
S32_Continue()	Query if execution should continue.

5.7.1 S32_RdPC()

Description

Read program counter register.

Prototype

```
S32_U32 S32_RdPC(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Program counter, offset into the execution area.

5.7.2 S32_RdML()

Description

Read memory limit register.

Prototype

```
S32_U32 S32_RdML(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Memory limit of S32 address space.

5.7.3 S32_RdXR()

Description

Read raised-exception status.

Prototype

```
S32_U32 S32_RdXR(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

If no exception has been raised, `S32_ERROR_NONE` (defined as zero) is returned. Any other value raised by `S32_Raise()` is taken to be an exceptional state and is the exception value returned.

5.7.4 S32_RdReg()

Description

Read general-purpose register.

Prototype

```
S32_U32 S32_RdReg(S32_EXEC_CONTEXT * pCtx,  
                 unsigned Index);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Index	Register index, 0 through 15.

Return value

Value of register.

5.7.5 S32_RdMem_U8()

Description

Read sandbox memory, U8.

Prototype

```
S32_U32 S32_RdMem_U8(S32_EXEC_CONTEXT * pCtx,  
                    S32_U32 Addr);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to read from.

Return value

U8 at the sandbox address if the address is valid, and zero if the sandbox address is not valid.

Additional information

If the datum lies outside of the sandbox address space, the return value has zero substituted and an access violation exception is raised in the execution context.

5.7.6 S32_RdMem_U16()

Description

Read sandbox memory, U16.

Prototype

```
S32_U32 S32_RdMem_U16(S32_EXEC_CONTEXT * pCtx,  
                     S32_U32 Addr);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to read from.

Return value

U16 at the sandbox address if the address is valid, and zero if the address is not valid.

Additional information

If the datum lies outside of the sandbox address space, the return value has zero substituted and an access violation exception is raised in the execution context.

5.7.7 S32_RdMem_U32()

Description

Read sandbox memory, U32.

Prototype

```
S32_U32 S32_RdMem_U32(S32_EXEC_CONTEXT * pCtx,  
                     S32_U32 Addr);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to read from.

Return value

U32 at the sandbox address if the address is valid, and zero if the address is not valid.

Additional information

If the datum lies outside of the sandbox address space, the return value has zero substituted and an access violation exception is raised in the execution context.

5.7.8 S32_WrMem_U8()

Description

Write sandbox memory, U8.

Prototype

```
void S32_WrMem_U8(S32_EXEC_CONTEXT * pCtx,  
                 S32_U32           Addr,  
                 S32_U32           Val);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to write to.
Val	Value to write.

Additional information

If the datum lies outside of the sandbox address space, the write is canceled and an access violation exception is raised in the execution context.

5.7.9 S32_WrMem_U16()

Description

Write sandbox memory, U16.

Prototype

```
void S32_WrMem_U16(S32_EXEC_CONTEXT * pCtx,  
                  S32_U32          Addr,  
                  S32_U32          Val);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to write to.
Val	Value to write.

Additional information

If the datum lies outside of the sandbox address space, the write is canceled and an access violation exception is raised in the execution context.

5.7.10 S32_WrMem_U32()

Description

Write sandbox memory, U32.

Prototype

```
void S32_WrMem_U32(S32_EXEC_CONTEXT * pCtx,  
                  S32_U32          Addr,  
                  S32_U32          Val);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to write to.
Val	Value to write.

Additional information

If the datum lies outside of the sandbox address space, the write is canceled and an access violation exception is raised in the execution context.

5.7.11 S32_WrMem_Blk()

Description

Write sandbox memory, block.

Prototype

```
void S32_WrMem_Blk(      S32_EXEC_CONTEXT * pCtx,
                        S32_U32          Addr,
                        const S32_U8      * pData,
                        S32_U32          DataLen);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Addr	Sandbox address to write to.
pData	Pointer to data to write.
DataLen	Octet length of data to write.

Additional information

If any of the the data lies outside of the sandbox address space, the write is canceled and an access violation exception is raised in the execution context.

5.7.12 S32_Push()

Description

Push to stack.

Prototype

```
void S32_Push(S32_EXEC_CONTEXT * pCtx,  
             S32_U32          Val);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Val	Value to push.

Additional information

If the address written when pushing lies outside the sandbox address space, the write is canceled and an access violation exception is raised in the execution context.

5.7.13 S32_Raise()

Description

Raise exception.

Prototype

```
void S32_Raise(S32_EXEC_CONTEXT * pCtx,  
              S32_I32          Code);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.
Code	Code to raise.

Additional information

This sets the execution environment into an exceptional state and causes immediate termination of the application.

5.7.14 S32_Continue()

Description

Query if execution should continue.

Prototype

```
int S32_Continue(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

- ≠ 0 No exception raised, execution can continue.
- = 0 Exception raised, execution should not continue.

Chapter 6

Application API reference

This chapter details some optional functions that can be added to the service API table to offer additional functions available to the application.

6.1 Runtime functions

The table below lists the functions that are used for C-language runtime support. If your application uses division or modulo operators, it will likely require these functions to be present in the service API table.

Function	Description
S32_API_S32_idiv()	Signed division.
S32_API_S32_imod()	Signed modulus.
S32_API_S32_udiv()	Unsigned division.
S32_API_S32_umod()	Unsigned modulus.

Application interface

The following is the interface provided to the application and may be required by the C compiler to implement division and modulus with unknown inputs:

```

/*****
 *
 *      Runtime functions
 */
__imp int      __S32_imod   (int, int);
__imp int      __S32_idiv   (int, int);
__imp unsigned __S32_umod   (unsigned, unsigned);
__imp unsigned __S32_udiv   (unsigned, unsigned);

```

Binding

The functions above can be bound using the following service table:

```

static const _aServices[] = {
  { "__S32_umod", S32_API_S32_umod },
  { "__S32_udiv", S32_API_S32_udiv },
  { "__S32_imod", S32_API_S32_imod },
  { "__S32_idiv", S32_API_S32_idiv },
};

```

6.1.1 S32_API_S32_idiv()

Description

Signed division.

Prototype

```
S32_U32 S32_API_S32_idiv(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Quotient.

Additional information

If the divisor is zero, a zero-divide exception is raised in the execution context.

This function implements the function `__S32_idiv()` that the compiler calls in order to perform signed division:

```
int __S32_idiv(int Num, int Div);
```

6.1.2 S32_API_S32_imod()

Description

Signed modulus.

Prototype

```
S32_U32 S32_API_S32_imod(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Modulus.

Additional information

If the divisor is zero, a zero-divide exception is raised in the execution context.

This function implements the function `__S32_imod()` that the compiler calls in order to perform signed remainder-after-division:

```
int __S32_imod(int Num, int Div);
```

6.1.3 S32_API_S32_udiv()

Description

Unsigned division.

Prototype

```
S32_U32 S32_API_S32_udiv(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Quotient.

Additional information

If the divisor is zero, a zero-divide exception is raised in the execution context.

This function implements the function `__S32_udiv()` that the compiler calls in order to perform unsigned division:

```
unsigned __S32_udiv(unsigned Num, unsigned Div);
```

6.1.4 S32_API_S32_umod()

Description

Unsigned modulus.

Prototype

```
S32_U32 S32_API_S32_umod(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Modulus.

Additional information

If the divisor is zero, a zero-divide exception is raised in the execution context.

This function implements the function `__S32_umod()` that the compiler calls in order to perform unsigned remainder-after-division:

```
unsigned __S32_umod(unsigned Num, unsigned Div);
```

6.2 Introspection functions

The table below lists the functions that can be used within an application to find services offered by the program.

Service functions

Function	Description
S32_API_S32_FindService()	Find service API function.
S32_API_S32_ExecService()	Execute service API function.

Application interface

The following is the interface provided to the application:

```

/*****
 *
 *      Introspection functions
 */
__imp int      SYS_FindService(const char *sName);
__imp unsigned SYS_ExecService(int ServiceHandle, ...);

```

Binding

The functions above can be bound for use on a Microsoft Windows operating system, using the following service table:

```

static S32_SERVICE_BINDING _aServices[] = {
    { "SYS_FindService", S32_API_S32_FindService },
    { "SYS_ExecService", S32_API_S32_ExecService },
};

```

6.2.1 S32_API_S32_FindService()

Description

Find service API function.

Prototype

```
S32_U32 S32_API_S32_FindService(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

≥ 0 Handle to service API function.
< 0 Service API not found.

Additional information

This function implements:

```
int SYS_FindService(const char *sName);
```

6.2.2 S32_API_S32_ExecService()

Description

Execute service API function.

Prototype

```
S32_U32 S32_API_S32_ExecService(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Value returned by executed service API function.

Additional information

This function implements:

```
int SYS_ExecService(int Handle, ...);
```

6.3 C library functions

The functions in this section can be installed into the service API table to provide fully sandboxed implementations of the equivalent C library functions.

The C source files for these functions are provided in the `Etc` folder.

Function	Description
S32_API_C_memset()	Fill memory.
S32_API_C_memcpy()	Copy memory.
S32_API_C_memmove()	Move memory.
S32_API_C_memchr()	Find character, in memory.
S32_API_C_strcpy()	Copy string.
S32_API_C_strcat()	Concatenate strings.
S32_API_C_strlen()	Calculate string length.
S32_API_C_strchr()	Find character, in string.
S32_API_C_strcmp()	Compare strings.
S32_API_C_putchar()	Write standard output, character.
S32_API_C_puts()	Write standard output, string.
S32_API_C_printf()	Write standard output, formatted.
S32_API_C_sprintf()	Write string, formatted.

6.3.1 S32_API_C_memset()

Description

Fill memory.

Prototype

```
S32_U32 S32_API_C_memset(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `memset()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `memset()`:

```
void * memset(void *pDst, int Val, unsigned DstLen);
```

6.3.2 S32_API_C_memcpy()

Description

Copy memory.

Prototype

```
S32_U32 S32_API_C_memcpy(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by memcpy().

Additional information

Implementation is fully sandboxed.

This function implements the standard C function memcpy():

```
void * memcpy(void *pDst, const void *pSrc, unsigned Len);
```

6.3.3 S32_API_C_memmove()

Description

Move memory.

Prototype

```
S32_U32 S32_API_C_memmove(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by memmove().

Additional information

Implementation is fully sandboxed.

This function implements the standard C function memmove():

```
void * memmove(void *pDst, const void *pSrc, unsigned Len);
```

6.3.4 S32_API_C_memchr()

Description

Find character, in memory.

Prototype

```
S32_U32 S32_API_C_memchr(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `memchr()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `memchr()`:

```
void * memchr(const void *pSrc, int Chr, unsigned Len);
```

6.3.5 S32_API_C_strcpy()

Description

Copy string.

Prototype

```
S32_U32 S32_API_C_strcpy(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `strcpy()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `strcpy()`:

```
char * strcpy(char *pDst, const char *pSrc);
```

6.3.6 S32_API_C_strcat()

Description

Concatenate strings.

Prototype

```
S32_U32 S32_API_C_strcat(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `strcat()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `strcpy()`:

```
char * strcat(char *pDst, const char *pSrc);
```

6.3.7 S32_API_C_strlen()

Description

Calculate string length.

Prototype

```
S32_U32 S32_API_C_strlen(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `strlen()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `strlen()`:

```
unsigned strlen(const char *pSrc);
```

6.3.8 S32_API_C_strchr()

Description

Find character, in string.

Prototype

```
S32_U32 S32_API_C_strchr(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `strchr()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `strchr()`:

```
char * strchr(const char *pSrc, int Chr, unsigned Len);
```

6.3.9 S32_API_C_strncmp()

Description

Compare strings.

Prototype

```
S32_U32 S32_API_C_strncmp(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `strncmp()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `strncmp()`:

```
int strncmp(const char *pSrc1, const char *pSrc2);
```

6.3.10 S32_API_C_putchar()

Description

Write standard output, character.

Prototype

```
S32_U32 S32_API_C_putchar(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by `putchar()`.

Additional information

Implementation is fully sandboxed.

This function implements the standard C function `putchar()`:

```
int putchar(int Ch);
```

6.3.11 S32_API_C_puts()

Description

Write standard output, string.

Prototype

```
S32_U32 S32_API_C_puts(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Value returned by puts().

Additional information

Implementation is fully sandboxed.

This function implements the standard C function puts():

```
int puts(const char *sSrc);
```

6.3.12 S32_API_C_printf()

Description

Write standard output, formatted.

Prototype

```
S32_U32 S32_API_C_printf(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to formatting context.

Return value

Value returned from printf().

This function implements the standard C function printf():

```
int printf(const char *sFmt, ...);
```

6.3.13 S32_API_C_sprintf()

Description

Write string, formatted.

Prototype

```
S32_U32 S32_API_C_sprintf(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to formatting context.

Return value

Value returned from sprintf().

This function implements the standard C function sprintf():

```
int sprintf(char *pDst, const char *sFmt, ...);
```

6.4 General library functions

The functions in this section can be installed into the service API table as generic functions. The C++ source files for these functions are provided in the `Etc` folder.

Function	Description
<code>S32_API_General_GetTime_ms()</code>	Get time, milliseconds.
<code>S32_API_General_GetTime_us()</code>	Get time, microseconds.
<code>S32_API_General_Sleep_ms()</code>	Sleep, duration in milliseconds.
<code>S32_API_General_Sleep_ns()</code>	Sleep, duration in nanoseconds.
<code>S32_API_General_MulDiv()</code>	Fractional multiply.
<code>S32_API_General_FOpen()</code>	Open file.
<code>S32_API_General_FClose()</code>	Close file.
<code>S32_API_General_FRead()</code>	Read file.
<code>S32_API_General_FWrite()</code>	Write file.

Installation

The functions above can be installed for use with the Flasher using the following API table:

```
static S32_SERVICE_BINDING _aServices[] = {
  { "UTIL_MulDiv",    S32_API_General_MulDiv    },
  { "SYS_GetTime_ms", S32_API_General_GetTime_ms },
  { "SYS_GetTime_us", S32_API_General_GetTime_us },
  { "SYS_Sleep_ms",   S32_API_General_Sleep_ms   },
  { "SYS_Sleep_ns",   S32_API_General_Sleep_ns   },
  { "FOpen",          S32_API_General_FOpen    },
  { "FClose",          S32_API_General_FClose    },
  { "FRead",           S32_API_General_FRead     },
  { "FWrite",          S32_API_General_FWrite     },
};
```

Application interface

To use these functions (as implemented by the SEGGER Flasher), use the following definitions:

```
#define FILE_MODE_READ    (1u<<0)
#define FILE_MODE_WRITE   (1u<<2)

__imp unsigned SYS_GetTime_ms (void);
__imp unsigned SYS_GetTime_us (void);
__imp void     SYS_Sleep_us    (unsigned Cnt);
__imp void     SYS_Sleep_ns    (unsigned Cnt);
__imp unsigned UTIL_MulDiv     (unsigned Val, unsigned Mul, unsigned Div);
__imp int      FOpen           (const char *sName, int Mode);
__imp int      FClose          (int Handle);
__imp int      FRead           (int Handle, void *pBuf, unsigned BufLen);
__imp int      FWrite          (int Handle, const void *pBuf, unsigned BufLen);
```

6.4.1 S32_API_General_GetTime_ms()

Description

Get time, milliseconds.

Prototype

```
S32_U32 S32_API_General_GetTime_ms(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Time in milliseconds.

Additional information

This function implements the prototype:

```
unsigned SYS_GetTime_ms(void);
```

6.4.2 S32_API_General_GetTime_us()

Description

Get time, microseconds.

Prototype

```
S32_U32 S32_API_General_GetTime_us(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Time in microseconds.

Additional information

This function implements the prototype:

```
unsigned SYS_GetTime_us(void);
```

6.4.3 S32_API_General_Sleep_ms()

Description

Sleep, duration in milliseconds.

Prototype

```
S32_U32 S32_API_General_Sleep_ms(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Zero, success.

Additional information

This function implements the prototype:

```
unsigned SYS_Sleep_ms(unsigned Cnt);
```

6.4.4 S32_API_General_Sleep_ns()

Description

Sleep, duration in nanoseconds.

Prototype

```
S32_U32 S32_API_General_Sleep_ns(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Zero, success.

Additional information

This function implements the prototype:

```
unsigned SYS_Sleep_ns(unsigned Cnt);
```

6.4.5 S32_API_General_MulDiv()

Description

Fractional multiply.

Prototype

```
S32_U32 S32_API_General_MulDiv(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

Val * Mul/Div using 64-bit arithmetic.

Additional information

If Div is zero, a zero-divide exception is raised in the execution context.

This function implements the prototype:

```
unsigned UTIL_MulDiv(void);
```

6.4.6 S32_API_General_FOpen()

Description

Open file.

Prototype

```
S32_U32 S32_API_General_FOpen(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

≥ 0 File handle.
< 0 Error opening file.

Additional information

This function implements the prototype:

```
unsigned FOpen(const char *sName, int Mode);
```

6.4.7 S32_API_General_FClose()

Description

Close file.

Prototype

```
S32_U32 S32_API_General_FClose(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

≥ 0 Success.
< 0 Write error or invalid file handle.

Additional information

This function implements the prototype:

```
unsigned FClose(int Handle);
```

6.4.8 S32_API_General_FRead()

Description

Read file.

Prototype

```
S32_U32 S32_API_General_FRead(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
<code>pCtx</code>	Pointer to execution context.

Return value

≥ 0 Success, number of bytes read.
< 0 Read error or invalid file handle.

Additional information

This function implements the prototype:

```
int FRead(int Handle, void *pData, unsigned DataLen);
```

6.4.9 S32_API_General_FWrite()

Description

Write file.

Prototype

```
S32_U32 S32_API_General_FWrite(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

≥ 0 Success, number of bytes written.
< 0 Write error or invalid file handle.

Additional information

This function implements the prototype:

```
int FWrite(int Handle, const void *pData, unsigned DataLen);
```

6.5 ISO 7816 functions

The table below lists the functions that are used to provide access to integrated circuit cards using the ISO 7816 interface. These services are implemented for Microsoft Windows only at this time. These functions are not provided in the base distribution of emApps.

Although typically used for integrated circuit cards (smart cards, SIMs, SAMs), any device may provide services through an ISO 7816 interface.

Service functions

Function	Description
S32_API_ISO7816_Init()	Initialize ISO 7816 interface.
S32_API_ISO7816_Exit()	Deinitialize ISO 7816 interface.
S32_API_ISO7816_PowerOn()	Power-on interface.
S32_API_ISO7816_PowerOff()	Power-off interface.
S32_API_ISO7816_WarmReset()	Warm-reset interface.
S32_API_ISO7816_ColdReset()	Cold-reset interface.
S32_API_ISO7816_GetATR()	Read ATR received from last reset.
S32_API_ISO7816_Case1()	Send Case 1 APDU.
S32_API_ISO7816_Case2()	Send Case 2 APDU.
S32_API_ISO7816_Case3()	Send Case 3 APDU.
S32_API_ISO7816_Case4()	Send Case 4 APDU.

Application interface

The following is the interface provided to the application:

```

/*****
 *
 *      ISO 7816 functions
 */
__imp int  ISO7816_Init      (void);
__imp void ISO7816_Exit     (void);
__imp int  ISO7816_PowerOn  (void);
__imp int  ISO7816_PowerOff (void);
__imp int  ISO7816_WarmReset(void);
__imp int  ISO7816_ColdReset(void);
__imp int  ISO7816_GetATR   (unsigned char *pATR, unsigned ATRLen);
__imp int  ISO7816_Case1    (int Cla, int Ins, int P1, int P2);
__imp int  ISO7816_Case2    (int Cla, int Ins, int P1, int P2,
                             int Lc, const unsigned char *pSendData);
__imp int  ISO7816_Case3    (int Cla, int Ins, int P1, int P2,
                             int Le, unsigned char *pRecvData);
__imp int  ISO7816_Case4    (int Cla, int Ins, int P1, int P2,
                             int Lc, const unsigned char *pSendData,
                             int Le, unsigned char *pRecvData);

```

Binding

The functions above can be bound for use on a Microsoft Windows operating system, using the following service table:

```

static const _aServices[] = {
  { "ISO7816_Init",      S32_API_ISO7816_Init    },
  { "ISO7816_Exit",     S32_API_ISO7816_Exit    },
  { "ISO7816_PowerOn",  S32_API_ISO7816_PowerOn },
  { "ISO7816_PowerOff", S32_API_ISO7816_PowerOff },
  { "ISO7816_Case1",   S32_API_ISO7816_Case1   },

```

```
{ "ISO7816_Case2",    S32_API_ISO7816_Case2    },  
{ "ISO7816_Case3",    S32_API_ISO7816_Case3    },  
{ "ISO7816_Case4",    S32_API_ISO7816_Case4    },  
{ "ISO7816_WarmReset", S32_API_ISO7816_WarmReset },  
{ "ISO7816_ColdReset", S32_API_ISO7816_ColdReset },  
{ "ISO7816_GetATR",   S32_API_ISO7816_GetATR   },  
};
```

6.5.1 S32_API_ISO7816_Init()

Description

Initialize ISO 7816 interface.

Prototype

```
S32_U32 S32_API_ISO7816_Init(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error, e.g. already initialized or can't find interface.
≥ 0 Success.

6.5.2 S32_API_ISO7816_Exit()

Description

Deinitialize ISO 7816 interface.

Prototype

```
S32_U32 S32_API_ISO7816_Exit(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

Zero, always succeeds.

Additional information

ISO7816_Exit() can be called without a matching call to ISO7816_Init(). If not already done so, power to the interface is removed.

6.5.3 S32_API_ISO7816_PowerOn()

Description

Power-on interface.

Prototype

```
S32_U32 S32_API_ISO7816_PowerOn(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 Success.

Additional information

After powering on, the answer to reset (ATR) is available through ISO7816_GetATR().

6.5.4 S32_API_ISO7816_PowerOff()

Description

Power-off interface.

Prototype

```
S32_U32 S32_API_ISO7816_PowerOff(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 Success.

Additional information

This function implements the prototype:

```
int IS07816_PowerOff(void);
```

6.5.5 S32_API_ISO7816_WarmReset()

Description

Warm-reset interface.

Prototype

```
S32_U32 S32_API_ISO7816_WarmReset(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error, e.g. interface not initialized or not powered on.
≥ 0 Success.

Additional information

The interface is reset without removing power. After resetting, the new answer to reset (ATR) is available through IS07816_GetATR().

6.5.6 S32_API_ISO7816_ColdReset()

Description

Cold-reset interface.

Prototype

```
S32_U32 S32_API_ISO7816_ColdReset(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error, e.g. interface not initialized or not powered on.
≥ 0 Success.

Additional information

The interface is reset without removing power. After resetting, the new answer to reset (ATR) is available through IS07816_GetATR().

6.5.7 S32_API_ISO7816_GetATR()

Description

Read ATR received from last reset.

Prototype

```
S32_U32 S32_API_ISO7816_GetATR(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 Success, number of bytes in ATR.

Additional information

This function implements the prototype:

```
int ISO7816_GetATR(U8 *pATR, unsigned ATRLen);
```

6.5.8 S32_API_ISO7816_Case1()

Description

Send Case 1 APDU.

Prototype

```
S32_U32 S32_API_ISO7816_Case1(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 R-APDU status word.

6.5.9 S32_API_ISO7816_Case2()

Description

Send Case 2 APDU.

Prototype

```
S32_U32 S32_API_ISO7816_Case2(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 R-APDU status word.

6.5.10 S32_API_ISO7816_Case3()

Description

Send Case 3 APDU.

Prototype

```
S32_U32 S32_API_ISO7816_Case3(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 R-APDU status word.

6.5.11 S32_API_ISO7816_Case4()

Description

Send Case 4 APDU.

Prototype

```
S32_U32 S32_API_ISO7816_Case4(S32_EXEC_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to execution context.

Return value

< 0 Error.
≥ 0 R-APDU status word.

Chapter 7

SEGGER formatter reference

7.1 Introduction

The SEGGER formatter can be used to implement all C library “formatted output” functions. It is also able to provide useful functionality that goes beyond that of the C standard which is particularly useful in embedded systems.

The way in which the formatter works is entirely parameterized by a context held in a `SEGGER_FORMAT_CONTEXT` object. The task is to initialize the context to match the expected semantics of the function being emulated and to provide the necessary mechanisms for extracting arguments as formatting progresses.

The following sections will describe how to achieve this.

7.2 Format control strings

The formatter implements a subset of the C library format specification. In particular:

Format	Description
Format conversions	
%c	Character.
%s	String.
%d, %i	Signed decimal.
%o	Unsigned octal.
%u	Unsigned decimal.
%x, %X	Unsigned hexadecimal.
Length modifiers	
l	long size modifier.
ll	long long size modifier.
h	short size modifier.
hh	char size modifier.
Flags	
0	Zero-pad value.
space	Space-separate values.
-	Left-adjust value.
+	Force sign.
#	Use alternate form.

Field width and precision are supported both as literals contained in the format string and as runtime inputs when specified by `*` in the format string.

The formatter does not implement the following format conversions:

Format	Description
%a, %A	Format floating-point value in hexadecimal form.
%e, %E	Format floating-point value in exponential form.
%f, %F	Format floating-point value in fixed form.
%g, %G	Format floating-point value in general form.

7.3 Using the formatter

7.3.1 Formatting contexts

There are two contexts used by the formatter:

- `SEGGER_FORMAT_CONTEXT` containing the control parameters that the formatter will use.
- `SEGGER_FORMAT_USER_CONTEXT` containing parameters related to argument extraction and parameters for the output function, and so on.

The structure `SEGGER_FORMAT_CONTEXT` is defined by the formatter and its is described below. The structure `SEGGER_FORMAT_USER_CONTEXT` is defined by the user and several examples are shown in order to implement the functions in the C library.

7.3.2 Implementing printf()

This section describes how to use the SEGGER formatter to implement a function that is equivalent to `printf()` from the C standard library,

7.3.2.1 The top-level code

The example here implements `HOST_printf()` using the SEGGER formatter. Here is the top-level code to set up and call the SEGGER formatter:

```
int HOST_printf(const char *sFmt, ...) {
    SEGGER_FORMAT_USER_CONTEXT UserCtx;
    SEGGER_FORMAT_CONTEXT      FmtCtx;
    int                         Res;
    //
    SEGGER_FORMAT_Init(&FmtCtx);      ❶
    FmtCtx.MaxLen = ~0u;              ❷
    FmtCtx.pUserCtx = &UserCtx;      ❸
    FmtCtx.pfGetVal = _HOST_GetVal;   ❹
    FmtCtx.pfGetStr = _HOST_GetStr;   ❺
    FmtCtx.pfFlush = _WrStdOut;       ❻
    //
    va_start(UserCtx.ap, sFmt);       ❼
    Res = SEGGER_FORMAT_Exec(&FmtCtx, sFmt); ❽
    va_end(UserCtx.ap);               ❾
    //
    return Res;
}
```

Some of the mechanisms behind the formatter will be ignored for the moment; they will, however, be described later.

❶ Initialize the formatting context

The formatting context is initialized by calling `SEGGER_FORMAT_Init()`. All members within the context are set to default values which are zero for all integer-style members and NULL for pointer-style members.

❷ Set the maximum number of characters output

The `printf()` function prints all its output and is not bounded; the `MaxLen` member is set to the largest unsigned value so that output is not truncated during formatting. `MaxLen` will be important when implementing functions such as `snprintf()` where output must be truncated.

❸ Establish the user context

The user context, together with some functions described below, is used to extract the incoming arguments that have been passed to `HOST_printf()`. This assignment links the formatting context to a specific user context provided for this purpose.

4 Provide a method to access incoming integer-style parameters

The formatting context uses the function pointed to by `pfGetVal` to get the next argument to format. The function pointed to by `pfGetVal` is passed the user context along with some additional information in order to accomplish this. Not to complicate matters, the specific implementation of `_HOST_GetVal()` used here will be considered a black box for now.

5 Provide a method to access incoming string-style parameters

Much like the integer-based arguments above, it will be necessary to access string parameters. When a string is necessary, the function pointed to by `pfGetStr` to get this. Again, the implementation of `_HOST_GetStr()` will be considered a black box.

6 Tell the formatter how to output

The formatter does not output formatted strings directly. Instead, it uses the function pointed to by `pfFlush` to output characters that must be printed. The implementation for this `printf()` is to use `fwrite()` to write the presented data to the standard output stream:

```
static int _HOST_WrStdOut(    SEGGER_FORMAT_CONTEXT * pCtx,
                            const char          * pData,
                            unsigned            DataLen) {
    return fwrite(pData, 1, DataLen, stdout);
}
```

7 Initialize the user context

The user context is initialized by calling `va_start()` to commence iteration over the incoming parameters. The definition of the user context for this is:

```
struct SEGGER_FORMAT_USER_CONTEXT_s {
    va_list ap;
};
```

8 Call the formatter

The formatter is called and formatting takes place. It is provided the formatting context and the format control string. The value returned is the number of characters that were output, or a negative value if there was any output error.

9 Wrap up

For compliance with the ISO standard, a call to `va_start()` must have a matching call to `va_end()`. Once done, the number of characters or error indication is returned to the caller.

7.3.2.2 Retrieving string arguments

The code for `_HOST_GetStr()` used above is very simple in this case:

```
static const char *_HOST_GetStr(SEGGER_FORMAT_USER_CONTEXT *pCtx, unsigned MaxLen) {
    (void)MaxLen;
    //
    return va_arg(pCtx->ap, const char *);
}
```

This simply extracts the string from the user context using `va_arg()`. The parameter `MaxLen` is ignored here, it simply encodes the precision argument provided in the format string. This parameter is important in some cases, especially when implementing a sandboxed environment.

7.3.2.3 Retrieving integer-style arguments

The code for `_HOST_GetVal()` used above is more complex than `_HOST_GetStr()` and requires explanation.

The prototype for the function to retrieve an integer is:

```
void SEGGER_FORMAT_GET_VAL_FUNC(SEGGER_FORMAT_USER_CONTEXT * pCtx,
                               SEGGER_FORMAT_VALUE         * pValue,
                               unsigned                    Flags);
```

The user context is provided in `pCtx`; the value retrieved must be stored into the object pointed to by `pValue` which is never NULL; the value is extracted according to the formatting flags in `Flags`.

The combination of these parameters is as follows.

`Flags` is a set of flags that encode the expected form of the incoming argument. Each flag is bitwise-or'd into the set. The combination of flags are described in the following table:

Flags	Argument is...
Signedness	
SEGGER_FORMAT_FLAG_SIGNED	Signed integer else an unsigned integer. This is combined with the length modifier.
Length	
SEGGER_FORMAT_FLAG_CHAR	char size (specified by <code>hh</code> modifier).
SEGGER_FORMAT_FLAG_SHORT	short size, (specified by <code>h</code> modifier).
SEGGER_FORMAT_FLAG_LONG	long size, (specified by <code>l</code> modifier).
SEGGER_FORMAT_FLAG_LONG_LONG	short size, (specified by <code>ll</code> modifier).
SEGGER_FORMAT_FLAG_PTR	void * size (specified by <code>%p</code> conversion).
None of the above set	int size (no length modifier).

Only one of the length flags will be set and enable correct interpretation of the incoming argument. The following implementation is highly generic and works on both 32-bit and 64-bit operating systems and with compilers that implement both `int` and `long` as 32 bits, compilers that implement `int` as 32 bits and `long` as 64 bits, and even compilers that implements both `int` and `long` as 64 bits.

```
void HOST_GetVal(SEGGER_FORMAT_USER_CONTEXT * pCtx,
                SEGGER_FORMAT_VALUE         * pValue,
                unsigned                    Flags) {
    if (Flags & SEGGER_FORMAT_FLAG_PTR) {
        pValue->u = (uintptr_t)va_arg(pCtx->ap, void *);
    } else if (Flags & SEGGER_FORMAT_FLAG_SIGNED) {
        if (Flags & SEGGER_FORMAT_FLAG_CHAR) {
            pValue->i = (signed char)va_arg(pCtx->ap, int);
        } else if (Flags & SEGGER_FORMAT_FLAG_SHORT) {
            pValue->i = (short)va_arg(pCtx->ap, int);
        } else if (Flags & SEGGER_FORMAT_FLAG_LONG) {
            pValue->i = va_arg(pCtx->ap, long);
        } else if (Flags & SEGGER_FORMAT_FLAG_LONG_LONG) {
            pValue->i = va_arg(pCtx->ap, long long);
        } else {
            pValue->i = va_arg(pCtx->ap, int);
        }
    } else {
        if (Flags & SEGGER_FORMAT_FLAG_CHAR) {
            pValue->u = (unsigned char)va_arg(pCtx->ap, unsigned);
        } else if (Flags & SEGGER_FORMAT_FLAG_SHORT) {
            pValue->u = (unsigned short)va_arg(pCtx->ap, unsigned);
        } else if (Flags & SEGGER_FORMAT_FLAG_LONG) {
            pValue->u = va_arg(pCtx->ap, unsigned long);
        }
    }
}
```

```
} else if (Flags & SEGGER_FORMAT_FLAG_LONGLONG) {  
    pValue->u = va_arg(pCtx->ap, unsigned long long);  
} else {  
    pValue->u = va_arg(pCtx->ap, unsigned);  
}  
}  
}
```

7.3.2.4 Testing the function

Rudimentary testing of `HOST_printf()` reveals everything is in order:

```
int main(void) {  
    HOST_printf("The square of |%d| is |%+4ld|, and the cube is |%-4lld|  
    \n", 3, 3L*3, 3LL*3*3);  
    return 0;  
}
```

Running this:

```
The square of |3| is | +9|, and the cube is |27 |
```

7.4 Preprocessor symbols

7.4.1 Formatting flags

Description

Flags set by format parsing.

Definition

```
#define SEGGER_FORMAT_FLAG_MINUS          (1 << 0)
#define SEGGER_FORMAT_FLAG_PLUS          (1 << 1)
#define SEGGER_FORMAT_FLAG_SPACE        (1 << 2)
#define SEGGER_FORMAT_FLAG_HASH         (1 << 3)
#define SEGGER_FORMAT_FLAG_ZERO         (1 << 4)
#define SEGGER_FORMAT_FLAG_HAVE_PRECISION (1 << 5)
#define SEGGER_FORMAT_FLAG_CAPS         (1 << 6)
#define SEGGER_FORMAT_FLAG_SIGNED       (1 << 7)
#define SEGGER_FORMAT_FLAG_CHAR         (1 << 8)
#define SEGGER_FORMAT_FLAG_SHORT        (1 << 9)
#define SEGGER_FORMAT_FLAG_LONG         (1 << 10)
#define SEGGER_FORMAT_FLAG_LONGLONG    (1 << 11)
#define SEGGER_FORMAT_FLAG_PTR          (1 << 12)
#define SEGGER_FORMAT_FLAG_NEGATIVE_INPUT (1 << 13)
```

Symbols

Definition	Description
SEGGER_FORMAT_FLAG_MINUS	'-' flag specified in format string (literally or by variable).
SEGGER_FORMAT_FLAG_PLUS	'+' flag specified in format string.
SEGGER_FORMAT_FLAG_SPACE	' ' flag specified in format string.
SEGGER_FORMAT_FLAG_HASH	'#' flag specified in format string.
SEGGER_FORMAT_FLAG_ZERO	'0' flag specified in format string.
SEGGER_FORMAT_FLAG_HAVE_PRECISION	Precision is specified in format string (literally or by variable).
SEGGER_FORMAT_FLAG_CAPS	'%X' format conversion, uppercase output.
SEGGER_FORMAT_FLAG_SIGNED	'%i' or '%d' format conversion, signed argument.
SEGGER_FORMAT_FLAG_CHAR	'hh' length modifier specified.
SEGGER_FORMAT_FLAG_SHORT	'h' length modifier specified.
SEGGER_FORMAT_FLAG_LONG	'l' length modifier specified.
SEGGER_FORMAT_FLAG_LONGLONG	'll' length modifier specified.
SEGGER_FORMAT_FLAG_PTR	'%p' format conversion.
SEGGER_FORMAT_FLAG_NEGATIVE_INPUT	Argument is considered negative.

7.5 Data types

Data type	Description
SEGGER_FORMAT_VALUE	Union containing value to format.
SEGGER_FORMAT_GET_STR_FUNC	Parameter acquisition, string object.
SEGGER_FORMAT_GET_VAL_FUNC	Parameter acquisition, integer-type object.
SEGGER_FORMAT_FLUSH_FUNC	Write formatted output.
SEGGER_FORMAT_CONTEXT	Formatting context.
SEGGER_FORMAT_USER_CONTEXT	Parameter acquisition context.

7.5.1 SEGGER_FORMAT_VALUE

Description

Union containing value to format.

Type definition

```
typedef union {  
    long long          i;  
    unsigned long long u;  
} SEGGER_FORMAT_VALUE;
```

Structure members

Member	Description
i	Signed value.
u	Unsigned value.

7.5.2 SEGGER_FORMAT_GET_STR_FUNC

Description

Parameter acquisition, string object.

Type definition

```
typedef const char * (SEGGER_FORMAT_GET_STR_FUNC)  
                    (SEGGER_FORMAT_USER_CONTEXT * pCtx,  
                     unsigned MaxLen);
```

Parameters

Parameter	Description
pCtx	Pointer to user context.
MaxLen	Maximum number of characters to examine for null terminator.

Return value

Pointer to string.

7.5.3 SEGGER_FORMAT_GET_VAL_FUNC

Description

Parameter acquisition, integer-type object.

Type definition

```
typedef void (SEGGER_FORMAT_GET_VAL_FUNC)(SEGGER_FORMAT_USER_CONTEXT * pCtx,  
                                           SEGGER_FORMAT_VALUE * pValue,  
                                           unsigned Flags);
```

Parameters

Parameter	Description
pCtx	Pointer to user context.
pValue	Pointer to object that receives the next argument.
pFlags	Pointer to flags that encode the argument length modifier.

Additional information

This function is called to extract the value of the next argument to format.

7.5.4 SEGGER_FORMAT_FLUSH_FUNC

Description

Write formatted output.

Type definition

```
typedef int (SEGGER_FORMAT_FLUSH_FUNC)(          SEGGER_FORMAT_USER_CONTEXT * pCtx,  
                                           const char * pData,  
                                           unsigned DataLen);
```

Parameters

Parameter	Description
pCtx	Pointer to user context.
pData	Pointer to object to write.
DataLen	Number of characters to write.

Return value

≥ 0 Success.
< 0 Error.

7.5.5 SEGGER_FORMAT_CONTEXT

Description

Formatting context.

Type definition

```
typedef struct SEGGER_FORMAT_CONTEXT_s {
    SEGGER_FORMAT_USER_CONTEXT * pUserCtx;
    char * pData;
    unsigned DataLen;
    unsigned Cnt;
    unsigned MaxLen;
    unsigned TrueCnt;
    SEGGER_FORMAT_GET_STR_FUNC * pfGetStr;
    SEGGER_FORMAT_GET_VAL_FUNC * pfGetVal;
    SEGGER_FORMAT_FLUSH_FUNC * pfFlush;
} SEGGER_FORMAT_CONTEXT;
```

Structure members

Member	Description
pUserCtx	Pointer to the user-provided context used to iterate over incoming arguments and hold any other formatting data.
pData	Pointer to a user-provided object that collects formatted output as formatting progresses, the collection buffer. It may also be NULL.
DataLen	Capacity of the collection buffer in characters, pointed to by pData .
Cnt	Number of characters currently written to the collection buffer. Cnt is always less than or equal to DataLen .
MaxLen	Maximum number of characters, in total, to pass pass to the flush method. Characters beyond the maximum length are dropped.
TrueCnt	Number of characters that a fully formatted operation would output.
pfGetStr	Method to deque a string parameter.
pfGetVal	Method to deque an integer or pointer parameter.
pfFlush	Method to flush converted data.

7.5.6 SEGGER_FORMAT_USER_CONTEXT

Description

Parameter acquisition context.

Type definition

```
typedef struct SEGGER_FORMAT_USER_CONTEXT_s {  
} SEGGER_FORMAT_USER_CONTEXT;
```

Additional information

This opaque type contains the context required to extract arguments passed to the formatting function as the format string is processed. Its structure is entirely defined by the user.

7.6 Functions

Function	Description
SEGGER_FORMAT_Init()	Initialize formatter context.
SEGGER_FORMAT_Exec()	Run formatter.

7.6.1 SEGGER_FORMAT_Init()

Description

Initialize formatter context.

Prototype

```
void SEGGER_FORMAT_Init(SEGGER_FORMAT_CONTEXT * pCtx);
```

Parameters

Parameter	Description
pCtx	Pointer to formatting context.

Additional information

This function initializes the formatting context pointed to by [pCtx](#). All members are set to zero or NULL as appropriate.

7.6.2 SEGGER_FORMAT_Exec()

Description

Run formatter.

Prototype

```
int SEGGER_FORMAT_Exec(    SEGGER_FORMAT_CONTEXT * pCtx,  
                          const char * sFormat);
```

Parameters

Parameter	Description
pCtx	Pointer to formatting context.
sFormat	Format control string.

Return value

Number of characters (without termination) that would have been stored if the buffer had been large enough.

Additional information

Performs formatting using the formatting context pointed to by [pCtx](#) under control of the formatting string [sFormat](#).