Flasher SDK

Flasher Software Development Kit

User Guide and Reference Manual

Document: UM08044
Software Version: 2.4
Revision: 0
Date: January 15, 2026

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

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

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: January 15, 2026

Manual version	Revision	Date	By	Description
2.4	0	260105	KT	• Fixed a link in an annotation of chapter Where to go from here?. • Fixed `ControlTIF` sample app. • Corrected the sample file name used in `ControlLED` example. • Fixed typos.
2.4	0	260115	KT	• Revised wording throughout the manual. • Fixed typos.
2.4	0	251216	AKU	Fixed usage sample of S32CC.
2.4	0	251211	TFI	Updated description of API function `TIF_JTAG_TransferBits()` as the return value now indicates how the transfer will be executed.
2.4	0	251217	LG	• Updated description of API function `SYS_FILE_Open()`. • Updated description of API function `SYS_FILE_ReadChunk()`.
2.4	0	251203	TFI	Added API function `SYS_FILE_Write()`.
2.4	0	251203	AKU	Updated “Writing Apps - Overview” graphic.
2.2	0	251127	AKU	Fixed typos.
2.2	0	251125	AG	• Reimplemented API function printf() to use a line buffer.
2.2	0	251113	KT	• Fixed filenames in Testing the sandbox. • Fixed filenames in Deploying an App. • Capitalized chapter header of Manual versions.
2.2	0	251110	JST	• Fixed wrong name of pseudo-variable `TIF_Speed`. • Added API function `TIF_CORESIGHT_Configure()`. • Added API function `TIF_CORESIGHT_DAP_Read()`. • Added API function `TIF_CORESIGHT_DAP_Write()`. • Added API function `TIF_CORESIGHT_ReadAP()`. • Added API function `TIF_CORESIGHT_ReadDP()`. • Added API function `TIF_CORESIGHT_WriteAP()`. • Added API function `TIF_CORESIGHT_WriteDP()`. • Added pseudo-variable `TIF_CORESIGHT_IndexAPBAPToUse`. • Added pseudo-variable `TIF_CORESIGHT_IndexAHBAPToUse`.
2.2	0	251104	AB	• Updated description of API function `SYS_REPORT_CONDITIONAL()`. • Updated description of API function `SYS_REPORT1_CONDITIONAL()`. • Updated description of API function `SYS_REPORTF_CONDITIONAL()`.
2.2	0	251022	KT	Changed the name from Flasher Device Support Kit (DSK) to Flasher Software Development Kit (SDK).
2.2	0	251023	AG	• Added API function `SYS_GetTime_us()`. • Added API function `strchr()`. • Added API function `strstr()`. • Added API function `UTIL_BASE64_Decode()`. • Added API function `UTIL_BASE64_DecodeInPlace()`.
2.2	0	250929	AG	Added API function `TIF_JTAG_TransferBits()`.
2.0	0	250725	AKU	Complete overhaul of the manual.
1.2	2	241217	MM	• Removed deprecated information about wiring diagrams in DDFs. • Updated information regarding `DefaultRead`. • Added information about the `FlashBankInit()` and `FlashBankDeInit()` App entry points.

Manual version	Revision	Date	By	Description
1.2	1	240904	MM	Updated description of the “Controller (Config)” DDF attribute.
1.2	0	240514	MM	Initial Version.

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 *The C Programming Language* by Kernighan and Ritchie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

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.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction

This chapter provides a brief overview of the Flasher SDK.

1.1 What is the Flasher SDK?

The Flasher Software Development Kit (SDK) is a framework for developing target device specific applications for the SEGGER Flasher ecosystem. Typical use cases are:

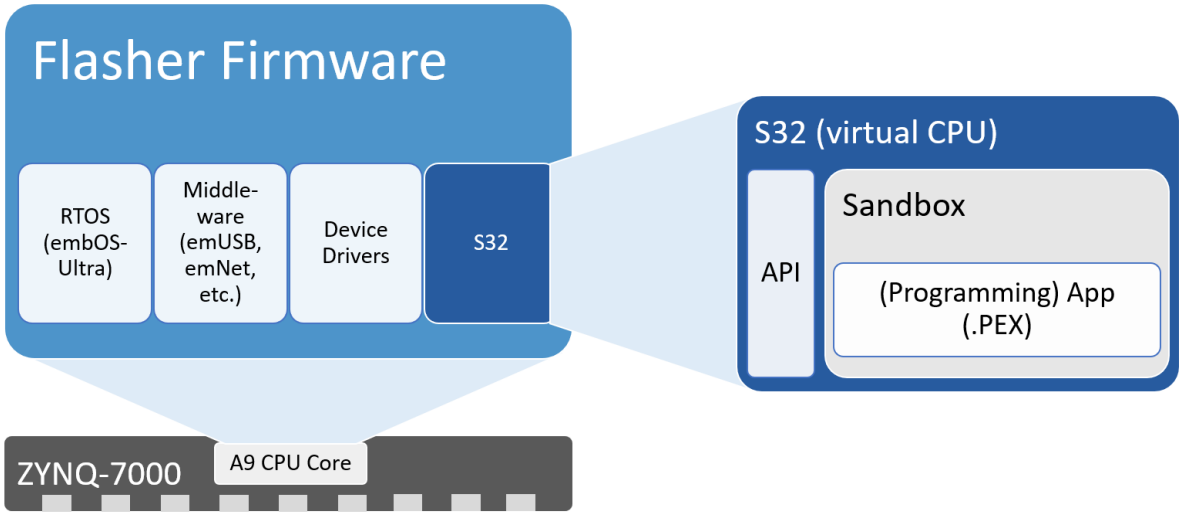
- programming of internal flash memory of an MCU or SoC
- programming of an external rewritable non-volatile memory device
- programming an FPGA or CPLD
- running device tests and board tests during production
- developing diagnostic applications to support engineers in the field

The key component for each application is an app which is executed on the Flasher hardware by SEGGER's secure S32 CPU. The Flasher SDK includes a compiler for the S32 to build the apps, which are written in a subset of the C programming language. The Flasher firmware also includes a variety of helpful API functions that can be called by the app to interact with the Flasher, modify data, or communicate with the target device via a specific target interface.

The most common target interfaces (e.g. UART, SPI, I2C, SWD, JTAG, etc.) are already supported. However, should a custom interface to the target device be required, it can be implemented by the user as part of the app, using two high-performance 8-bit softcores on the Flasher called "Speedy".

Unlike the J-Link DSK, the Flasher SDK is not limited to specific architectures and cores. It allows the creation of any possible application, not limited to programming support.

The image below shows an overview of the different hardware and software components inside the Flasher. To break down the complexity, you will be guided through the components step by step throughout the manual. In the end, you will be able to implement your own apps and add support devices.



1.2 What is included?

The following table shows the contents of the Flasher SDK:

Directory / File	Content
.\Doc	Release notes and Flasher SDK manual
.\Sample	Sample Flasher App projects
.\Template	Template app projects
.\Tool	Development tools for the creation of apps
.\Tool\Inc	Directory with the include files required by the S32 C compiler for compiling app source code files
.\Tool\Installer	Scripts and files to generate a Flasher Device Pack installer
.\Tool\FlasherAppBuilderCL.exe	Development environment tool for building and testing apps
.\Tool\S32CC.exe	S32 C compiler for compiling app source code files
.\Tool\SpeedyASM.exe	Speedy assembler for compiling Speedy applications

1.3 Requirements

The following hardware and software is required to work with the Flasher SDK:

- Computer with Microsoft Windows 10 or later (x64)
- *Flasher Software and Documentation Pack*
- SEGGER Flasher with hardware version 5 or later
- Text editor
- Python 3.6.5 or newer and NSIS-3.0b1 to create an installer for a Flasher Device Pack (optional)

> **Note**
>
> Please make sure to update your Flasher to the latest firmware provided with the Flasher Software Package and Documentation Pack to ensure full compatibility.

1.4 List of abbreviations

The following abbreviations are used in this manual. Note that only the abbreviations and what they stand for are listed here, along with some disabiguation: for more information, please refer to the glossary.

Abbreviation	Description
API	Application programming interface.
App	Application. In the context of this manual, a user-written application which is executed by the Flasher hardware.
cJTAG	Compact JTAG.
CLI	Command-line interface.
CPLD	Complex programmable logic device.
CPU	Central processing unit.
DDF	Device definition file.
DSK	Device support kit.
SDK	Software development kit. In the context of this manual, the Flasher SDK.
DUT	Device under test.
FDP	Flasher device pack.
FPGA	Field-programmable gate array.
I2C	Inter-Integrated Circuit (bus).
ICE	In-circuit emulator.
JTAG	Joint Test Action Group.
LED	Light-emitting diode.
PEX	Portable Executable. In the context of this manual, a file containing the instructions and data of an app.
RAM	Random Access Memory.
SPI	Serial peripheral interface.
SWD	Serial wire debug.
TAP	Test access port.
TIF	Target interface.
UNI	Universal. In the context of this manual, a file containing the specification of how to program a device.
UART	Universal asynchronous receiver-transmitter.

Chapter 2

Writing apps

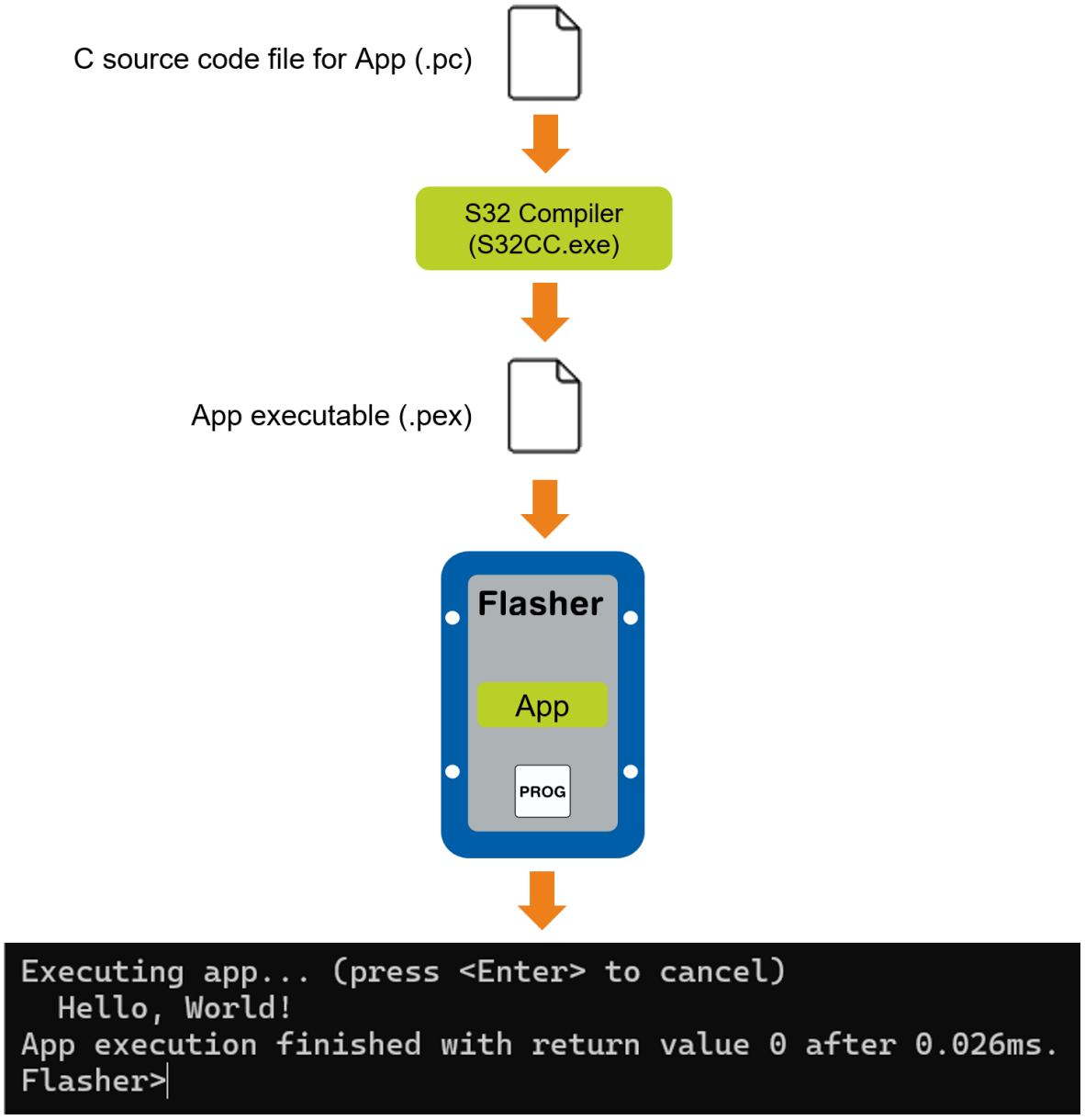
This chapter is a guide to help you to get started with developing apps that can be run on the Flasher.

2.1 Overview

Flashers have a built in, highly enhanced virtual CPU called S32, which allows running S32 executables (apps) on the Flasher in a sandboxed environment. The sandbox prevents accessing and manipulating data outside of the app's memory space.

App source code files have the extension `.pc` and are written in a subset of the C programming language. For compilation of apps, the Flasher SDK comes with SEGGER's S32 C compiler (S32CC.exe). The corresponding app executable files have the extension `.pex`. A dedicated address space of up to 64KB is guaranteed for each app by the Flasher.

The following image summarizes the code to execution process of an app:



2.2 Your first app

As a first step, to get familiar with the concept, let's compile and run a simple app on the Flasher that prints "Hello, World!".

Make sure that your Flasher is connected to your PC via USB or Ethernet. Refer to the *Flasher online documentation* for more information.

The source code for printing "Hello, World!" is already provided with the Flasher SDK and located in the `Sample/HelloWorld` directory. App source files have the `.pc` extension. Open the `HelloWorld.pc` file in a text editor to see the sample implementation:

```
int main(void) {
  printf("Hello, World!\n");
  return 0;
}
```

As you can see, the code is written in C language.

Next, start the Flasher App Builder Utility which is part of the Flasher SDK. The executable (`FlasherAppBuilderCL.exe`) is located in the `Tool` directory.

Change the working directory to `Sample/HelloWorld` directory of Flasher SDK by using the `cd` command.

```
Flasher>cd ../Sample/HelloWorld
```

To test the application, type go and hit <Enter> to build the app, download it to the Flasher and start execution. The Flasher App Builder will automatically connect an available Flasher or show a selection dialog if multiple Flashers are connected to your PC. If there is a single app source file in the working directory, no further user input is required.

```
Flasher>go
Flasher connection not established yet but required for command.
Connecting to Flasher via USB...O.K.
Firmware: J-Link / Flasher Compact V7 compiled May 14 2025 16:02:23
Hardware version: V7.00
Flasher uptime (since boot): 0d 00h 00m 02s
S/N: 1017000000
License(s): JFlash, GDB
USB speed mode: High speed (480 MBit/s)
VTref=3.200V
Looking for source file in working directory 'C:/FlasherSDK/Sample/HelloWorld'...
Using source file 'HelloWorld.pc'.
Selected build configuration: 'Debug'
Successfully compiled 'HelloWorld_D.pex'.
Successfully loaded 'HelloWorld_D.pex'.
Executing app... (press <Enter> to cancel)
  Hello, World!
App execution finished with return value 0 after 0.273ms.
Flasher>
```

Congratulations! You have successfully programmed, built, and run your first app on the Flasher.

Besides `cd` and `go`, the Flasher App Builder has more commands which e.g. allow to just build the executable or run specific functions. You can use the `?` command to show a list of all available commands within the Flasher App Builder.

2.3 Using API functions

The Flasher firmware includes a variety of helpful API functions that can be called by an app to interact with the Flasher, modify data, or communicate with the target device via a specific target interface.

In this example we change the "Hello, World!" implementation to show the system uptime of the Flasher. For this, open the `GetUpTime.pc` sample in a text editor which uses the additional API function call `SYS_GetTime()`. We store the return value of the function in a variable and add `printf()` to display the system time:

```
int main(void) {
  int UpTime;

  UpTime = SYS_GetTime();
  printf("Flasher system uptime: %dms.\n", UpTime);
  return 0;
}
```

Change the working directory to the `Sample/GetUpTime` directory. Using the `go` command in the Flasher App Builder, the output will now show the system uptime:

```
Flasher>cd ../Sample/GetUpTime
Flasher>go
[...]
Executing app... (press <Enter> to cancel)
  Flasher system uptime: 1270669ms.
App execution finished with return value 0 after 0.027ms.
```

The `go` command combines building, download and execution of the app. However, to see the updated system uptime, we only need to run the app again. For this, we can use the `run` command to just run a loaded app. Accordingly, a subsequent use of the `run` command will show the incrementing system uptime on each execution:

```
Flasher>run
[...]
Executing app... (press <Enter> to cancel)
  Flasher system uptime: 1274531ms.
App execution finished with return value 0 after 0.024ms.
```

The App can be extended with further API functions as desired. All available API functions are described in *App API reference* on page 39.

2.4 Controlling LEDs

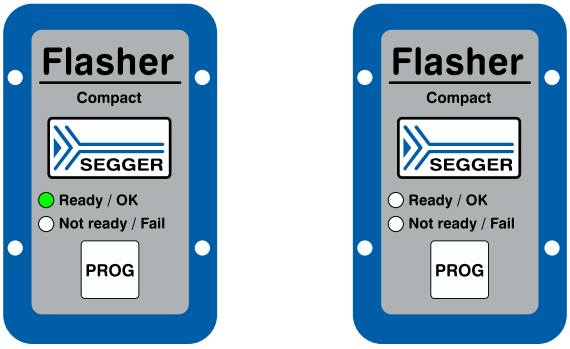
In order to provide visual feedback about the status of an app running on the Flasher, the API includes a function to control the LED indicators in the Flasher hardware. The `ControlLED.pc` sample in the `Sample/ControlLED` directory uses the `SYS_SetLED()` and `SYS_Sleep_ms()` functions create a LED blinking sequence.

```
int main(void) {
  U32 i;
  //
  // Alternating LED blink sequence
  //
  for (i = 0; i < 5; i++) {
    SYS_SetLED(0, 1);   // Turn on LED 0
    SYS_Sleep_ms(250);  // Wait for 250ms
    SYS_SetLED(0, 0);   // Turn off LED 0
    SYS_Sleep_ms(250);  // Wait for 250ms
  }
  return 0;
}
```

Using the `go` command in the Flasher App Builder, the “Ready / OK” LED blink alternately 5 times:

```
Flasher>cd ../ControlLED
Flasher>go
[...]
Executing app... (press <Enter> to cancel)
App execution finished with return value 0 after 2499.407ms.
```

For example, on the Flasher Compact, this looks like shown below:



2.5 Controlling the target interface

The extensive Flasher App API can also be used to control the interface to the target device. For this example, let's assume we want to communicate with our target device through a UART interface.

We use a simple app that sends and receives two bytes (`Sample/ControlTIF/ControlTIF.pc`):

```
int main(void) {
  static const U8 aOut[2] = {0xA5, 0x5A};
  U8 aIn[2];
  int r;
  //
  // Instruct the Flasher to assume a 3.3V target reference voltage
  //
  SYS_ExecCommand("VTrefTmp=3300");
  //
  // Initialize UART
  //
  TIF_PIN_SetFunc(TIF_PIN_PIN7, TIF_PIN_FUNC_UART_TX);
  TIF_PIN_SetFunc(TIF_PIN_PIN9, TIF_PIN_FUNC_UART_RX);
  TIF_UART_Init(115200);  // 8 data bits, one stop bit, and no parity
  //
  // Write data
  //
  TIF_UART_Write(&aOut[0], sizeof(aOut));
  //
  // Wait some time so the firmware can buffer the incoming data
  //
  SYS_Sleep_ms(10);
  //
  // Read data
  //
  r = TIF_UART_Read(&aIn[0], sizeof(aIn));
  if (r < (int)sizeof(aIn)) {
    printf("Error while receiving data.\n");
    r = -1;
    goto Done;
  }
  if (memcmp(aIn, aOut, sizeof(aIn))) {
    printf("Data error - received 0x%X, 0x%X\n", aIn[0], aIn[1]);
    r = -1;
    goto Done;
  }
  printf("Data received 0x%X, 0x%X\n", aIn[0], aIn[1]);
  printf("Data sent and received successfully!\n");
  r = 0;
Done:
  TIF_UART_DeInit();
  return r;
}
```

To successfully drive the Flasher's pins, a reference voltage needs to be supplied to the VTref pin. In order to make this example as simple as possible, we do not want to attach an external voltage. Therefore, we set the VTref voltage to a temporary value specified in millivolts using the `SYS_ExecCommand()` function.

The next step is to assign the UART Tx and Rx signals to arbitrary pins and initialize the UART module. In this case the line protocol is 115,200 baud with 8 data bits, one stop bit, and no parity.

> **Note**

> The Flasher supports multiple interfaces by default (see *Target interface* on page 112). Any other interface can be implemented using Speedy.

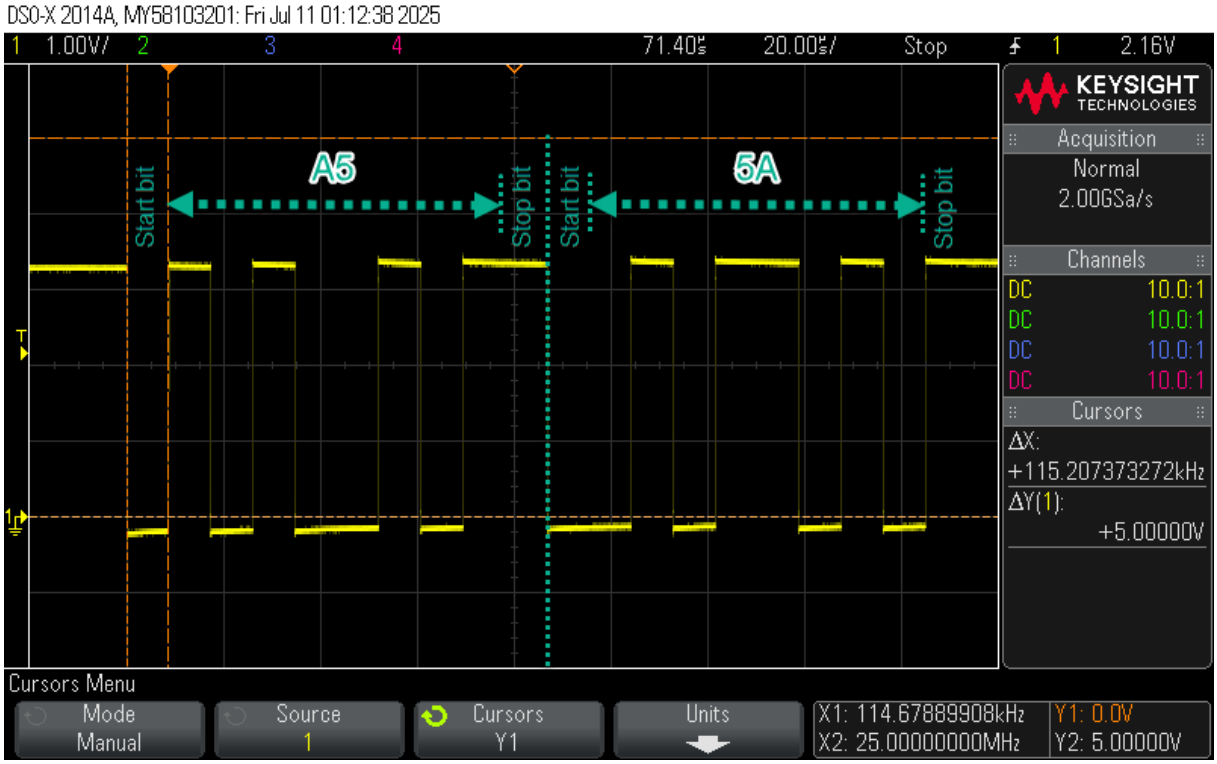
If you build, load, and run this app on the Flasher using the Flasher App Builder's go command, the output will be as follows:

```
Flasher>cd ../Sample/ControlTIF
Flasher>go
[...]
  "Timeout while receiving data."
App execution finished with return value 0 after 14.994ms.
```

We received the error message because the UART Tx pin (7) is not connected to the UART Rx pin (9) and no data could be received. To fix this, connect pin 7 of the Flasher's target interface connector to pin 9 using a female-female jumper wire or similar.

```
Flasher>run
[...]
  Data sent and received successfully!
App execution finished with return value 0 after 14.222ms.
```

The image below shows a scope shot of the Tx pin during the execution of the app. You can see the two bytes being transferred at 115,200 baud (see cursors).



2.6 Testing the sandbox

An app running on the Flasher is only allowed to use the memory space assigned by the Flasher firmware. The Flasher firmware provides that memory for as long as the app is active. Any intent by the app to access memory outside of this assigned memory space is intercepted and causes app execution to be terminated.

To demonstrate this, we use the sandbox sample (`Sandbox.pc`) located in `Sample/Sandbox`:

```
int main(void) {
  U32 *p;
  U32 v;

  printf("Sandbox check\n");
  p = (void*) 0x5555;
  v = *p;
  return 0;
}
```

We added an invalid memory access, as the address `0x5555` is outside of the app's memory space. Using the go command again, we will still see the "Sandbox check" output. However, the app stops with an access violation error:

```
Flasher>cd ../Sample/Sandbox
Flasher>go
[...]
Executing app... (press <Enter> to cancel)
  Sandbox check
App stopped with error code -9 (Access violation).
  R0: 0x00005555
  PC: 0x0000002A
  SP: 0x00000038
  SR: 0x00000000
```

To further analyze what happened, you can take a look at the `Sandbox_D.lst` list file the S32 compiler generates in the working directory:

```
000016             main:
[...]
    6:  p = (void*) 0x5555;

000022  000F 5555        MOVW  R0, #0x5555
000026  0C0C             STR   R0, [SP, #4]            ; assign 'p'

    7:  v = *p;

000028  00A8             LDR   R0, [R0]                ; dereference pointer
00002A  040C             STR   R0, [SP, #0]            ; assign 'v'
```

Pointer `p` is assigned to a value outside the memory space assigned to the app. We are then trying to dereference the pointer and store the result in variable `v`. This causes the access violation and aborts execution of the app.

2.7 Deploying an app

So far, we have used the Flasher App Builder to run an app on the Flasher. But the real use-case for apps is to run them stand-alone without any host software being involved.

Downloading an app into the Flasher and executing it is fairly easy. Switch back to the "ControlLED" sample and use the `make` command of the Flasher App Builder to build the executable. Using `make` will also print program size information of the app.

```
Flasher>cd ../Sample/ControlLED
Flasher>make
[...]
Program size:
     60 bytes code
      8 bytes data
      0 bytes strings
     16 bytes stack
      0 bytes heap
  ----------------------------------
     84 bytes total runtime image size

Compilation complete: 0 errors, 0 warnings, 0 remarks.

Successfully compiled 'ControlLED_D.pex'.
```

After the successful build of the executable, we download it onto the Flasher by using the `put` command.

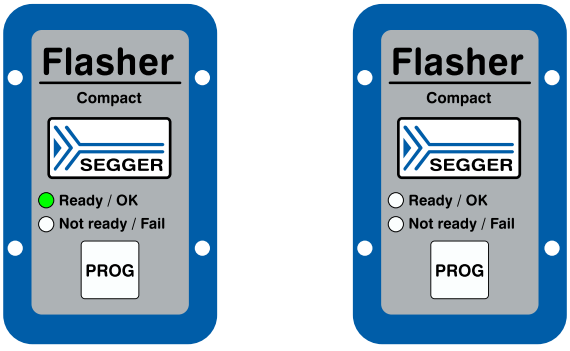
```
Flasher>put ControlLED_D.pex
Reading file 'ControlLED_D.pex'...
Writing file 'ControlLED_D.pex' on Flasher (0x90 bytes)...
OK.
```

The app executable file is now located in the root directory of the Flasher's filesystem and ready to run. Apps can be executed by pressing the "PROG" button on the Flasher programmer if it is the only executable found in the root directory. We can quickly check if that's the case by using the `rdir` command:

```
Flasher>rdir
Directory of \

22/08/2025  14:45                   144 ControlLED_D.pex
```

Press the "PROG" button on the Flasher programmer and you will see the "Ready / OK" LED blink 5 times as expected. Congratulations! You have successfully ran your first app fully standalone on the Flasher.



2.8 Timing & scheduling

Before diving into deeper topics, there is one more important thing to know about apps running on the Flasher: The timing and scheduling.

Since apps are executed by the S32 running on the main CPU of the Flasher, apps can be subject to interrupts of different kinds. For example, it could be the terminal which is receiving data (e.g. via Telnet) or sending data due to calls to e.g. `SYS_Report()` made by the programming app itself.

In most cases, these interrupts are negligible, because they are short and only affect the S32. Speedy modules, which are responsible for the low-level communication with the target, are not affected.

As an example, a call to `TIF_SPI_ReadWrite()` that is supposed to send and receive multiple bytes at once will be executed by Speedy without interruption. However, the next call to `TIF_SPI_ReadWrite()` made by the programming app could be slightly delayed if the S32 is interrupted between these two function calls. This could lead to complications if the programming app has to adhere to timings imposed by the high-level protocol of the target interface.

In this case, `SYS_SetMaxPause()` can help, as it ensures that the execution of the programming app is not preempted by other tasks for a specified amount of time.

2.9 Where to go from here?

Now that you have learned the basics how to program, build, test and run apps in standalone mode, there is much more to discover:

- Learn about *Creating device support* on page 31 to create custom device support
- Browse the extensive *App API reference* on page 39
- Get to know the *Speedy* on page 300
- Explore the feature set of the *Flasher App Builder* on page 327

Chapter 3

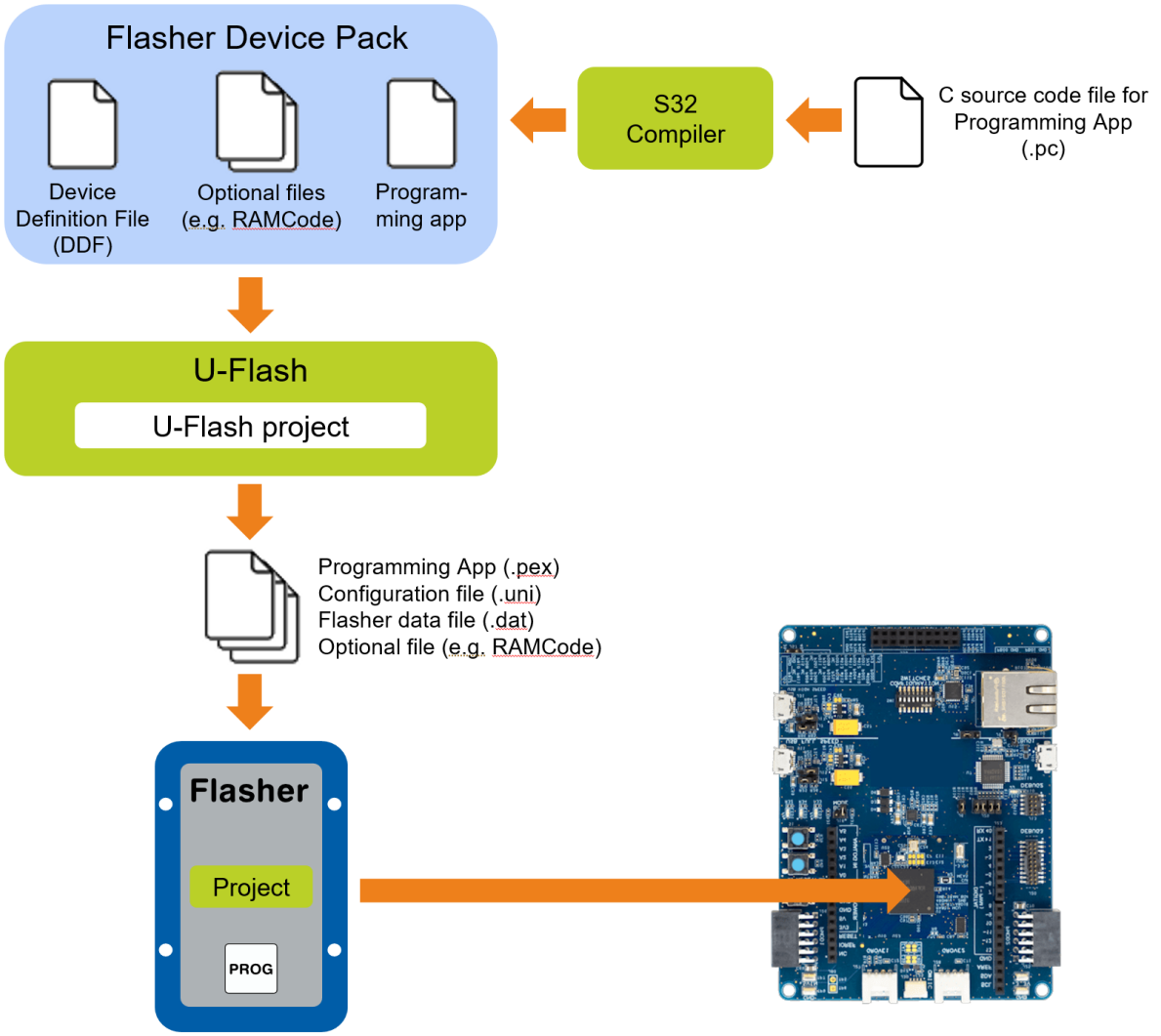
Creating device support

The following sections guide you through the process of adding Flasher device support for any device.

3.1 Overview

The Flasher SDK provides all required tools and information to add support for a new target device to U-Flash and eventually the Flasher. Basic device support includes options to erase, program and optionally read a target device's memory (usually, the flash memory).

Device Support for one or multiple devices is bundled as a Flasher Device Pack (FDP), which will be installed on a PC and makes the devices available in U-Flash. FDPs consist of a programming app (formerly known as flash loader), a Device Definition File (DDF), and some other optional files. As an overview, the following graphic shows the workflow to create the custom device support for the Flasher:



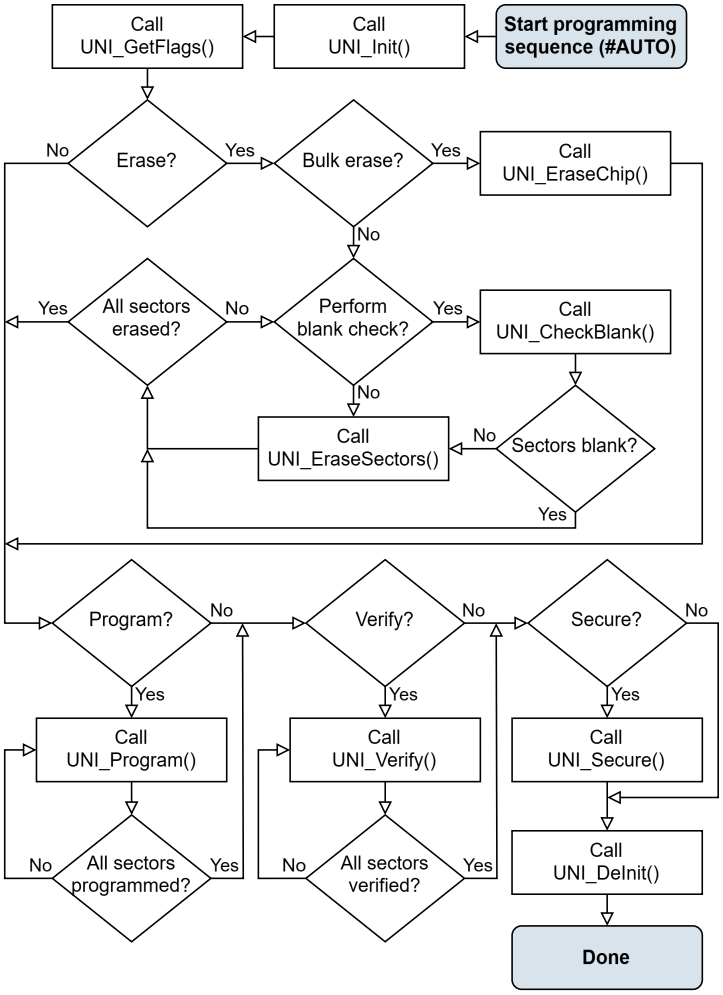
3.2 Programming app

Programming Apps are the basis for a Flasher Device Pack and as such become important when implementing custom device support. The concept of a programming app does not differ from the general concept of apps. While apps have one entry points called by the Flasher (`main()`), programming apps have multiple entry points for operations like erase, program, verify. These entry points are called by the Flasher firmware and are indicated by the `UNI_` prefix. Refer to *Programming app entry points* on page 193 for a complete list of all entry point functions.

Typically, a programming app communicates with the target device via a target interface (e.g. UART, SPI, I2C) to transfer the data and issue flash commands (erase, program, etc.). The Flasher supports multiple interfaces by default (see *Target interface* on page 112). Any other interface can be implemented using Speedy.

3.2.1 Programming sequence

An entry point function is executed only when the corresponding task needs to be performed. Whether a task is performed or not depends on the project configuration and the action triggered by the user. The following flowchart depicts how the Flasher calls the programming app's entry point functions during a programming sequence initiated by the `PROG` button or the `#AUTO` terminal command.



Note

The flowchart does not contain any information about the call sequence of the entry points in the event that a function returns an error.

While some entry points are mandatory, others are optional may not be required. Refer to the API reference Progamming app entry points to get an overview of mandatory and optional functions.

If a requested function is not available in the programming app when called by the Flasher, the programming attempt will fail and an error will be returned. To indicate the specific reason, not implemented entry point functions should always return -1 and include a meaningful error message.

```
int UNI_Read(U32 Addr, U32 NumBytes) {
  SYS_REPORT_ERR("The #READ functionality is not supported for this device.");
  return -1;
}
```

Refer to the chapters *Programming app entry points* on page 193 and *xdata_size* on page 298 for more information how to allocate xdata memory and use it. Data is shared between the Flasher firmware and the programming app via the xdata area.

3.2.2 Implementation and testing

For an easy start to implement a programming app, use the `Manufacturer_Device` template provided in the the `Template/Manufacturer_Device` directory of the Flasher SDK. It already includes the basic entry points as well as a `main()` function for testing. The `main()` function will only be included in debug builds of the programming app (see *Flasher App Builder* on page 327 for more information about build configurations).

The `main()` function is used to simulate a programming process by implicitly calling the corresponding entry point functions of the programming app. This is done in the same order as they are called when triggering programming via the `PROG` button or the `#AUTO` terminal command. As no information about the device is known at this time, static information is passed to the entry point functions. The following part of the implementation generates test data and stores it in the xdata area which is used in the `UNI_Program()` function.

```
//
// Generate test data
//
p = __S32_XDataBase;
for (i = 0; i < 0x100; i++) {
  *p++ = (U8)i;
}
[...]
r = UNI_Program(0x0, 0x100);
[...]
```

> **Note**
>
> The `main()` function may be fully customized to fit your testing purposes.
>
> The xdata area must not be referenced outside of its boundaries (see `__S32_XDataBase`, `__S32_XDataLimit`, *xdata_size* on page 298).

Extended programming app templates for different purposes can be found in the `Template` directory.

3.3 Device Definition File

Besides the programming app a device configuration is needed to specify the device name, flash banks as well as specific settings. This configuration is specified in the Device Definition File.

The DDF is an XML file that contains information about supported devices as well as all information required by the programming app. U-Flash uses the DDFs to list all supported devices in its device selection.

Each device entry in the DDF defines one or more devices and contains information as the manufacturer, device name, interfaces, their flash banks, etc. DDFs can provide much more information than that and are responsible for the available project configurations that are shown in U-Flash. Additional elements can be used to specify information that can then be loaded and used by the programming app. By default, configurations are shown in U-Flash and can be modified, depending on the type of the configuration.

3.3.1 Creating a Device Definition File

To follow up on *Implementation and testing* on page 34, we use DDF template of the `Manufacturer_Device` template project. It is located in the `Template/Manufacturer_Device` directory of the Flasher SDK.

The template contains a bare minimum implementation with two additional configuration fields which are used in the examples for the `SYS_CFG_GetU32()` and `SYS_CFG_GetData()` function reference. Refer to the *DDF reference* on page 208 for an overview of all available configuration options.

Extended DDF and programming app templates for different purposes can be found in the `Template` directory.

3.3.2 Testing the configuration

To make sure that the configuration data is correctly loaded in the programming app, it is possible to set up a test environment. With a Flasher Device Pack, the Flasher needs the following files on its file system to execute a standalone programming operation:

- Programming app (PEX file)
- Configuration (UNI file)
- Program data (DAT file)
- Optional files (e.g. RAMCode)

The UNI and DAT files are solely generated by U-Flash. While the DAT file contains the program data in an format which is optimized for the Flasher's programming processes, the UNI file contains the device configuration. The latter is generated from the DDF and the additional information and settings configured in U-Flash. `SYS_LoadUNI()` loads the UNI file from the Flasher's file system so that the programming app can access its configuration. Testing the configuration requires to

- have a configuration file located on the Flasher's file system
- load the configuration file from the programming app

To enable testing, replace the `UNI_Init()` function in the template with the following code to read the two additional configurations values:

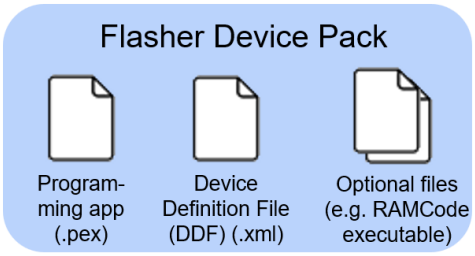
```
int UNI_Init(void) {
  U32 v;
  char acFileName[32];
  int r;

  r  = SYS_CFG_GetU32("CheckBoxConfig", &v);
  r |= SYS_CFG_GetData("FileConfig", &acFileName[0], sizeof(acFileName));
  if (r != 0) {
    return -1;
  }
  SYS_Reportf("CheckBoxConfig value: %u", v);
```



```
  SYS_Reportf("FileConfig value: %s", &acFileName[0]);
 return 0;
}
```

Next, we need to create a Flasher Device Pack. As initially mentioned, a FDP consists of a Programming app, a Device Definition File, and potentially some other optional files. Optional files may be required in rare cases, e.g. RAMCodes if a device requires to run code snippets from its RAM.



The structure of a FDP is fairly simple, as it just combines the mentioned files in one folder as shown below:

Directory / File	Description
`*.xml`	Device Definition File
`FlashLoader/*.pex`	Programming app
`Addons/*.*`	Optional files (see <Addon>)

To install the FDP, copy the folder containing the files to the following directories. U-Flash scans these directories for Device Definition Files and adds the included devices to the device list:

Operating System	Path
Windows	`%AppData%\Local\SEGGER\Flasher\SDK\`
Linux	`/home/<username>/.config/SEGGER/Flasher/SDK`
macOS	`/home/<username>/Library/Application Support/SEGGER/Flasher/SDK`

As the FDP is now installed to U-Flash, the implementation can be tested using the following sequence:

- Open U-Flash
- Select your device
- Select a data file (required by U-Flash, can be dummy data)
- Configure the project and specify “Test CheckBox” and “Test File”
- Generate a “FLASHER.uni” configuration file (File -> Save Flasher config file…)
- Close U-Flash
- Open the Flasher App Builder
- Download the `FLASHER.uni` file to the Flasher using the put command
- Change the working directory and run the programming app using the go command.

The programming app will print the configuration specified in U-Flash, e.g.:

```
Flasher>put FLASHER.uni
Flasher>cd ../Template/Manufacturer_Device
Flasher>go
[...]
CheckBoxConfig value: 0
FileConfig value: Test.bin
```


3.3.3 Testing the device support

After successfully testing the configuration, change the build configuration in the Flasher App Builder to "Release" (see *Flasher App Builder* on page 327). Include both PEX output files in the Flasher Device Pack and install it.

Now you are ready for the final test of the device support. Make sure to test every implemented feature and try different verbosity levels as timing may be different for each PEX file.

3.4 Creating a Flasher Device Pack installer

The Flasher SDK contains a pre-configured script to create a Windows installer for Flasher Device Packs. This allows to easily share custom device support with others.

Refer to `Tools/Installer/README.txt` of the Flasher SDK for a description how to create a Windows installer.

Chapter 4

App API reference

This chapter describes the app API for apps and programming apps.

4.1 C library functions

Function	Description
printf()	Outputs a formatted string to the terminal.
strlen()	Determines the length of a given string.
strnlen()	Determines the length of a given string.
strncpy_s()	Copies the first <Cnt> characters of <sSrc> to <pDest>.
strcpy_s()	Copies characters of <sSrc> to <pDest>.
strcat_s()	Appends characters of <sSrc> to <pDest>.
strncat_s()	Appends the first <Cnt> characters of <sSrc> to <pDest>.
strcmp()	Compares two strings.
strncmp()	Compares up to <Cnt> chars of <s1> to those of <s2>.
strchr()	Finds the specified character in the string and returns a pointer to it.
strstr()	Finds a given substring inside a given string.
memset()	Writes ByteValue into the first Cnt bytes into the memory pointed to by pDest.
memmove()	Copies Cnt bytes from pSrc to pDest.
memcpy()	Copies Cnt bytes from pSrc to pDest.
memcmp()	Compares NumBytes bytes of pData1 to NumBytes bytes of pData2.

4.1.1 printf()

Description

Outputs a formatted string to the terminal.

The string is buffered in a line buffer (max. 128 characters) until a newline ('\n') is sent or the buffer runs full.

Syntax

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

Parameters

Parameter	Description
sFormat	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

4.1.2 strlen()

Description

Calculates the length of a string.

Syntax

```
U32 strlen(const char* s);
```

Parameters

Parameter	Description
s	Input string.

Return value

Length of s in characters.

4.1.3 strnlen()

Description

Determines the length of a string. Stops prematurely when `MaxLen` characters have been checked.

Syntax

```
U32 strnlen(const char* s,
            U32   MaxLen);
```

Parameters

Parameter	Description
`s`	Input string
`MaxLen`	Max. number of characters to check

Return value

The length of the string or `MaxLen`, whatever is less.

4.1.4 strncpy_s()

Description

Copies the first Cnt characters of {sSrc} to {pDest}.

Syntax

```
char* strncpy_s(      char* pDest,
                      U32   DestLen,
                const char* sSrc,
                      U32   Cnt);
```

Parameters

Parameter	Description
pDest	Pointer to object that receives the retrieved value.
DestLen	Capacity of object that receives the retrieved string, in characters.
sSrc	Source string.
Cnt	Maximum number of chars to copy from sSrc.

Return value

Pointer to destination, pDest.

Additional information

The destination string is null-terminated, if the destination buffer is valid and its size is greater than 0.

4.1.5 strcpy_s()

Description

Copies characters of sSrc to pDest.

Syntax

```
char* strcpy_s(      char* pDest,
                     U32   DestLen,
               const char* sSrc);
```

Parameters

Parameter	Description
pDest	Pointer to buffer that receives the retrieved value.
DestLen	Size of buffer that receives the retrieved string, in characters.
sSrc	Source string.

Return value

Pointer to destination, pDest.

Additional information

The destination string is null-terminated, if the destination buffer is valid and its size is greater than 0.

4.1.6 strcat_s()

Description

Appends characters of sSrc to pDest, plus a terminating zero character ('\0').

Syntax

```
char* strcat_s(      char* pDest,
                     U32   DestLen,
               const char* sSrc);
```

Parameters

Parameter	Description
pDest	Pointer to object that receives the retrieved value.
DestLen	Capacity of object that receives the retrieved string, in characters.
sSrc	Source string.

Return value

Pointer to destination, pDest.

Additional information

The destination string is null-terminated, if the destination buffer is valid and its size is greater than 0.

4.1.7 strncat_s()

Description

Appends the first `Cnt` characters of `sSrc` to `pDest`, plus a terminating zero character (`'\0'`). If the length of `sSrc` is less than `Cnt`, only the content up to the terminating zero character or `pDestLen` is copied, whichever is smaller.

Syntax

```
char* strncat_s(      char* pDest,
                      U32   DestLen,
                const char* sSrc,
                      U32   Cnt);
```

Parameters

Parameter	Description
`pDest`	Pointer to object that receives the retrieved value.
`DestLen`	Capacity of object that receives the retrieved string, in characters.
`sSrc`	Source string.
`Cnt`	Max. number of characters to append.

Return value

Pointer to destination, `pDest`.

Additional information

The destination string is null-terminated, if the destination buffer is valid and its size is greater than 0.

4.1.8 strcmp()

Description

Compares two strings.

Syntax

```
int strcmp(const char* s1,
           const char* s2);
```

Parameters

Parameter	Description
s1	String to compare.
s2	String to compare.

Return value

Value	Description
< 0	First char that differs has lower value in s1 than in s2
= 0	Both strings are equal
> 0	First char that differs has greater value in s1 than in s2

Additional information

Comparison ends when either:

- a character differs.
- a string terminator (`'\0'`) is hit.

4.1.9 strncmp()

Description

Compares up to `Cnt` characters of `s1` to those of `s2`. Comparison ends when either:

- a character differs
- a string terminator (`'\0'`) is hit.
- `Cnt` characters have been compared.

Syntax

```
int strncmp(const char*    s1,
            const char*    s2,
                  unsigned Cnt);
```

Parameters

Parameter	Description
`s1`	String to compare.
`s2`	String to compare.
`Cnt`	Maximum number of characters to compare.

Return value

Value	Description
< 0	First char that differs has lower value in `s1` than in `s2`.
= 0	Both strings are equal.
> 0	First char that differs has greater value in `s1` than in `s2`.

4.1.10 strchr()

Description

Finds the given character c in the given string s and returns a pointer to it.

Syntax

```
char* strchr(const char* s,
                   int   c);
```

Parameters

Parameter	Description
s	Pointer to string to search in.
c	Character to search for.

Return value

Value	Description
≠ NULL	Pointer in s to the character found.
= NULL	Character has not been found.

4.1.11 strstr()

Description

Finds given substring s2 inside given string s1.

Syntax

```
char* strstr(const char* s1,
             const char* s2);
```

Parameters

Parameter	Description
s1	String to search in.
s2	Substring to search for.

Return value

Value	Description
≠ NULL	A pointer to the first occurrence in s1 of the entire sequence of characters specified in s2.
= NULL	The sequence specified in s2 is not present in s1.

4.1.12 memset()

Description

Writes `ByteValue` into the first `NumBytes` bytes of memory pointed to by `pDest`.

Syntax

```
int memset(void*    pDest,
           int      ByteValue,
           unsigned NumBytes);
```

Parameters

Parameter	Description
`pDest`	Pointer to the destination.
`ByteValue`	Value that should be set.
`NumBytes`	Number of bytes that should be set.

Return value

Returns the destination pointer, `pDest`.

4.1.13 memmove()

Description

Copies `Cnt` bytes from `pSrc` to `pDest`. Destination and source may overlap.

Syntax

```
void* memmove(void*    pDest,
              void*    pSrc,
              unsigned NumBytes);
```

Parameters

Parameter	Description
`pDest`	Pointer to the destination
`pSrc`	Pointer to the source
`NumBytes`	Number of bytes to move

Return value

Returns the destination pointer, `pDest`.

4.1.14 memcpy()

Description

Copies `Cnt` bytes from `pSrc` to `pDest`. Destination and source may overlap.

Syntax

```
void* memcpy(void*    pDest,
             void*    pSrc,
             unsigned NumBytes);
```

Parameters

Parameter	Description
`pDest`	Pointer to the destination
`pSrc`	Pointer to the source
`NumBytes`	Number of bytes to move

Return value

Returns the destination pointer, `pDest`.

Additional information

ISO standard C requires the objects pointed to by `pSrc` and `pDest` not overlap. This implementation allows them to overlap, and as such is still compliant to the ISO C standard.

4.1.15 memcmp()

Description

Compares `NumBytes` bytes of `pData1` to `NumBytes` bytes of `pData2`.

Syntax

```
int memcmp(const char*    pData1,
           const char*    pData2,
                 unsigned NumBytes);
```

Parameters

Parameter	Description
`pData1`	Pointer to block of memory
`pData2`	Pointer to block of memory
`NumBytes`	Number of bytes to compare

Return value

Value	Description
= 0	The contents are equal
< 0	The first byte that does not match has a lower value in `pData1` than in `pData2`
> 0	The first byte that does not match has a greater value in `pData1` than in `pData2`

4.2 Utility functions

Function	Description
COUNTOF()	Computes the number of elements in a statically allocated array.
UTIL_MIN()	Returns the lesser of two values.
UTIL_MAX()	Returns the greater of two values.
UTIL_IsDigit()	Checks if a character is a digit.
UTIL_ValToStr()	Converts a numeric value to a string.
UTIL_ParseVal()	Parses a numeric value from a string.
UTIL_ReverseBytes()	Reverses each byte in a buffer.
UTIL_ReverseBits()	Reverses the specified number of bits.
UTIL_WriteBitOff()	Writes data to a bit position in a buffer.
UTIL_Reverse32()	Reverses 32 bits.
UTIL_Reverse16()	Reverses 16 bits.
UTIL_Reverse8()	Reverses 8 bits.
UTIL_MulDiv()	Performs ((x * n) / d) calculation.
UTIL_CalcCRC()	Calculates a CRC32/16/8 over the given data stream.
UTIL_CalcChecksum()	Calculates a simple checksum over the given data stream.
UTIL_BASE64_Decode()	Decode a BASE64-encoded string.
UTIL_BASE64_DecodeInPlace()	Simplified version of UTIL_BASE64_Decode() where source and destination buffer are the same.

4.2.1 COUNTOF()

Description

Computes the number of elements in a statically declared array.

Syntax

```
U32 COUNTOF(T a[]);
```

Parameters

Parameter	Description
a	Array for which the number of elements was requested.

Return value

Number of elements in the array.

Additional information

> **Note**
>
> This is a macro.

4.2.2 UTIL_MIN()

Description

Returns the lesser of two values.

Syntax

```
T UTIL_MIN(T a,
           T b);
```

Parameters

Parameter	Description
a	Value a.
b	Value b.

Additional information

> **Note**
>
> This is a macro.

4.2.3 UTIL_MAX()

Description

Returns the greater of two values.

Syntax

```
T UTIL_MAX(T a,
           T b);
```

Parameters

Parameter	Description
a	Value a.
b	Value b.

Additional information

> **Note**
>
> This is a macro.

4.2.4 UTIL_IsDigit()

Description

Checks if the given character is a decimal digit [0..9].

Syntax

```
U32 UTIL_IsDigit(char c);
```

Parameters

Parameter	Description
c	Character to check.

Return value

Value	Description
= 1	Is a digit.
= 0	Is not a digit.

4.2.5 UTIL_ValToStr()

Description

Converts numerical value `Val` to a string in the given representation of value with `NumDigits` digits.

Syntax

```
int UTIL_ValToStr(U32   Val,
                  U32   NumDigits,
                  char* pDest,
                  U32   DestLen,
                  U32   Base);
```

Parameters

Parameter	Description
`Val`	Value to convert.
`NumDigits`	Number of digits.
`pDest`	Pointer to object that receives the retrieved value.
`DestLen`	Capacity of object that receives the retrieved string, in characters.
`Base`	Base of desired representation.

Return value

Value	Description
≥ 0	OK.
< 0	Error, `Base` is invalid.

4.2.6 UTIL_ParseVal()

Description

Parses a numeric value from a string.

Syntax

```
int UTIL_ValToStr(U32   Val,
                  U32   NumDigits,
                  char* pDest,
                  U32   DestLen,
                  U32   Base);
```

Parameters

Parameter	Description
`s`	String to parse.
`Base`	Base of numerical value in string.
`pDest`	Pointer to object that receives the retrieved value.

Return value

Value	Description
= 0	OK.

Value	Description
< 0	Error.

4.2.7 UTIL_ReverseBytes()

Description

Reverses each byte in the provided buffer individually.

Syntax

```
void UTIL_ReverseBytes(U8* pData,
                       U32 NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to a buffer with bytes to be reversed.
NumBytes	The number of bytes to be reversed.

4.2.8 UTIL_ReverseBits()

Description

Reverses the specified number of bits and returns the result.

Syntax

```
U32 UTIL_ReverseBits(U32 Data,
                     U32 NumBits);
```

Parameters

Parameter	Description
Data	The data to be reversed.
NumBits	The number of bits to reverse.

Return value

The result of the reversed data. If fewer than 32 bits are reversed, the result is shifted to the right by (32 - NumBits).

4.2.9 UTIL_WriteBitOff()

Description

Writes a given number of bits to a memory address with a specified offset.

Syntax

```
void UTIL_WriteBitOff(U8* pAddr,
                      U32 BitOff,
                      U32 Value,
                      U32 NumBits);
```

Parameters

Parameter	Description
pAddr	The memory address used in conjunction with BitOff to calculate the memory location where Value should be written to.
BitOff	The bit offset used in conjunction with pAddr to calculate the memory location where Value should be written to.
Value	The value to write to the location calculated with pAddr and BitOff.
NumBits	Number of bits to write.

4.2.10 UTIL_Reverse32()

Description

Reverse bit order of 32-bit value.

Syntax

```
int UTIL_Reverse32(U32 Data);
```

Parameters

Parameter	Description
Data	Value to bit-reverse.

Return value

The bit-reversed value.

4.2.11 UTIL_Reverse16()

Description

Reverse bit order of 16-bit value.

Syntax

```
int UTIL_Reverse16(U16 Data);
```

Parameters

Parameter	Description
Data	Value to bit-reverse.

Return value

The bit-reversed value.

4.2.12 UTIL_Reverse8()

Description

Reverse bit order of 8-bit value.

Syntax

```
int UTIL_Reverse8(U8 Data);
```

Parameters

Parameter	Description
Data	Value to bit-reverse.

Return value

The bit-reversed value.

4.2.13 UTIL_MulDiv()

Description

Perform fractional multiply. Calculates (x * n) / d) where the intermediate product of x and n is calculated in extended precision such that it does not overflow.

Syntax

```
int UTIL_MulDiv(int x,
                int n,
                int d);
```

Parameters

Parameter	Description
x	Multiplier
n	Multiplicand
d	Divisor

Return value

The result of the calculation.

4.2.14 UTIL_CalcCRC()

Description

Calculate a CRC over a byte string.

Syntax

```
U32 UTIL_CalcCRC(const U8* pData,
                       U32 NumBytes,
                       U32 InitVal,
                       U32 Polynomial,
                       U32 Flags);
```

Parameters

Parameter	Description
pData	Pointer to object containing the data to CRC.
NumBytes	Number of bytes to calculate the CRC over.
InitVal	Initial value for the calculation.
Polynomial	Polynomial for the calculation.
Flags	Flags for the calculation.

Flags

Value	Description
UTIL_CALC_CRC_FLAG_CRC32	Polynomial is 33 bits forming a 32-bit CRC
UTIL_CALC_CRC_FLAG_CRC16	Polynomial is 17 bits forming a 16-bit CRC
UTIL_CALC_CRC_FLAG_CRC8	Polynomial is 9 bits forming an 8-bit CRC
UTIL_CALC_CRC_FLAG_LSB_FIRST	Bit order for the CRC calculation
UTIL_CALC_CRC_FLAG_MSB_FIRST	Bit order for the CRC calculation
UTIL_CALC_CRC_FLAG_SHIFT_RIGHT	Direction of the bit shift during the CRC calculation
UTIL_CALC_CRC_FLAG_SHIFT_LEFT	Direction of the bit shift during the CRC calculation

Return value

The calculated CRC.

4.2.15 UTIL_CalcChecksum()

Description

Calculates a simple checksum over a byte string.

Syntax

```
U32 UTIL_CalcChecksum(const U8* pData,
                      U32 NumBytes,
                      U32 InitVal);
```

Parameters

Parameter	Description
pData	Pointer to object containing the data to checksum.
NumBytes	Number of bytes to calculate the checksum over.
InitVal	Initial value for the calculation.

Return value

The calculated checksum which is the summation of each individual unsigned byte of the input string.

4.2.16 UTIL_BASE64_Decode()

Description

Decode a BASE64-encoded string.

Syntax

```
unsigned UTIL_BASE64_Decode(const U8*      pInput,
                            unsigned InputLen,
                            U8*      pOutput,
                            unsigned OutputLen);
```

Parameters

Parameter	Description
pInput	Pointer to character string to decode.
InputLen	Octet length of input string.
pOutput	Pointer to octet string that receives the encoding.
OutputLen	Capacity of the receiving octet string.

Return value

Number of octets written to the output octet string. Invalid encodings always return a zero length.

4.2.17 UTIL_BASE64_DecodeInPlace()

Description

Simplified version of `UTIL_BASE64_Decode()` where `pInput` is also used as `pOutput`. As decoding a BASE64-encoded string always results in a smaller output than input, the given input buffer can be reused.

Syntax

```
unsigned UTIL_BASE64_DecodeInPlace(U8*      pData,
                                   unsigned DataLen);
```

Parameters

Parameter	Description
`pData`	Pointer to character string to decode. Also used to store decoded string.
`InputLen`	Octet length of input string.

Return value

Number of octets written to the output octet string. Invalid encodings always return a zero length.

4.3 System functions

Function	Description
SYS_ExecCommand()	Passes a command to the Flasher which is then executed.
SYS_Exit()	Aborts execution of the app.

4.3.1 SYS_ExecCommand()

Description

Executes the given command string and returns the return value of the executed command.

Syntax

```
int SYS_ExecCommand(const char* sMsg);
```

Parameters

Parameter	Description
sMsg	Command string to execute.

Return value

Return value of the executed command.

Example

```
SYS_ExecCommand("SupplyPower = 1");
```


4.3.1.1 Command strings

The following table lists the available command strings for the `SYS_ExecCommand()` function.

Command	Description
SupplyPower	Enables or disables the 5V target power supply.
VTrefTmp	Sets VTRef's power good voltage threshold to a temporary value specified in millivolts.
SetSWDTargetId	Sets the target ID of the target to communicate with, in an SWD multi-drop system.
SetSWDInstanceId	Sets the instance ID of the target to communicate with, in an SWD multi-drop system.
SetcJTAGInitMode	Selects the cJTAG connect sequence to be used to enable the cJTAG interface.

4.3.1.1.1 SupplyPower

Description

Enables or disables the 5V target power supply.

Syntax

`SupplyPower = <En>`

Additional information

`En` set to 1 enables the 5V power supply; `En` set to 0 disables the 5V power supply.

4.3.1.1.2 VTrefTmp

Description

Sets VTRef's power-good voltage threshold to a temporary value specified in millivolts.

Syntax

`VTrefTmp = <Value>`

Additional information

`Value` is the power-good voltage threshold and must be in the range 1200 (1.2V) to 5000 (5.0V) inclusive.

4.3.1.1.3 SetSWDTargetId

Description

Sets the target ID of the target to communicate with, in an SWD multi-drop system. Used in conjunction with command string SetSWDInstanceId.

Syntax

`SetSWDTargetId = <TargetId> (where <TargetId> is a hexadecimal value)`

See also

SetSWDInstanceId on page 77

4.3.1.1.4 SetSWDInstanceId

Description

Sets the instance ID of the target to communicate with, in an SWD multi-drop system. Used in conjunction with command string SetSWDTargetId.

Syntax

`SetSWDInstanceId = <InstanceId>` (where <InstanceId> is a hexadecimal value)

4.3.1.1.5 SetcJTAGInitMode

Description

Selects the cJTAG connect sequence to be used to enable the cJTAG interface. More info about the different cJTAG connect sequences being available: https://kb.segger.com/J-Link_cJTAG_specifics#Connect_sequence

Syntax

`SetcJTAGInitMode = <Mode>` (where <Mode> is an unsigned integer)

Value	Description
0	Long-form activation sequence with JScan0 boot & OScan1 enter afterwards.
1	Short-form activation sequence with OScan1 boot.
2	Short-form activation sequence with OScan1 boot + special for Wiliot chips.

4.3.2 SYS_Exit()

Description

Stops the execution of the app and exits it.

Syntax

```
void SYS_Exit(int ExitCode);
```

Parameters

Parameter	Description
ExitCode	Exit code returned by the application (0 = success).

> **Note**
>
> We should also note that returning a value from an entry point in normal execution also acts in the same manner as SYS_Exit().

Example

```
SYS_Exit(0);
```

4.4 Output functions

Function	Description
SYS_SetLED()	Controls the state of a Flasher's hardware LED.
SYS_Report()	Outputs a string to the terminal.
SYS_Report1()	Outputs a string followed by a 32-bit hexadecimal value to the terminal.
SYS_Reportf()	Outputs a formatted string to the terminal.
SYS_REPORT()	Outputs a string to the terminal in descriptive builds.
SYS_REPORT1()	Outputs a string followed by a 32-bit hexadecimal value to the terminal in descriptive builds.
SYS_REPORTF()	Outputs a formatted string to the terminal in descriptive builds.
SYS_REPORT_CONDITIONAL()	Outputs a string to the terminal depending on the verbosity level in descriptive builds.
SYS_REPORT1_CONDITIONAL()	Outputs a string followed by a 32-bit hexadecimal value to the terminal depending on the verbosity level in descriptive builds.
SYS_REPORTF_CONDITIONAL()	Outputs a formatted string to the terminal depending on the verbosity level in descriptive builds.
SYS_REPORT_ERR()	Outputs an error string.
SYS_REPORT1_ERR()	Outputs an error string followed by a 32-bit hexadecimal value to the terminal.
SYS_REPORTF_ERR()	Outputs a formatted error string.
SYS_ReportLoaderCompileDate()	Outputs the app's compile date and time.

4.4.1 SYS_SetLED()

Description

Controls the state of a LED on the Flasher's hardware.

Syntax

```
int SYS_SetLED(U32 Index,
               U32 State);
```

Parameters

Parameter	Description
Index	Index of the LED.
State	State of the LED.

Return value

Value	Description
≥ 0	OK
-1	Error, invalid state.
-2	Error, LED index does not exist in hardware.

Additional information

The following table shows the mapping between the Index and the LEDs. Each index controls an individual hardware LED.

Index	LED
0	Ready / OK - green
1	Ready / OK - red
2	Not ready / Fail - green
3	Not ready / Fail - red

> **Note**
>
> The availability of the LEDs depend on the Flasher model. The Flasher Portable, for instance, does not have LEDs that can be controlled.

4.4.2 SYS_Report()

Description

Outputs a string to the terminal. Verbosity levels are not respected by this function.

Syntax

```
int SYS_Report(const char* sMsg);
```

Parameters

Parameter	Description
sMsg	The string to be sent to the terminal.

Return value

Always returns 0.

4.4.3 SYS_Report1()

Description

Outputs a string followed by a 32-bit hexadecimal value without ’0x’ prefix to the terminal. Verbosity levels are not respected by this function.

Syntax

```
int SYS_Report1(const char* sMsg,
                int   Value);
```

Parameters

Parameter	Description
sMsg	The string to be sent to the terminal.
Value	A 32-bit value to be shown in hexadecimal form.

Return value

Always returns 0.

4.4.4 SYS_Reportf()

Description

Outputs a formatted string to the terminal. Verbosity levels are not respected by this function.

Syntax

```
int SYS_Reportf(const char* sFormat,
                        ...   ...);
```

Parameters

Parameter	Description
sFormat	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

4.4.5 SYS_REPORT()

Description

Outputs a string to the terminal, if the verbosity level is at least `VERBOSITY_BASIC`. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORT(const char* sMsg);
```

Parameters

Parameter	Description
`sMsg`	The string to be sent to the terminal.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.6 SYS_REPORT1()

Description

Outputs a string followed by a 32-bit hexadecimal value to the terminal without a '0x' prefix, if the verbosity level is at least VERBOSITY_BASIC. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORT1(const char* sMsg,
                int   Value);
```

Parameters

Parameter	Description
sMsg	The string to be sent to the terminal.
Value	A 32-bit value to be shown in hexadecimal form .

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.7 SYS_REPORTF()

Description

Outputs a formatted string to the terminal, if the verbosity level is at least `VERBOSITY_BASIC`. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORTF(const char* sFormat,
                ...   ...);
```

Parameters

Parameter	Description
`sFormat`	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.8 SYS_REPORT_CONDITIONAL()

Description

SYS_REPORT_CONDITIONAL() is a wrapper for SYS_Report() respecting verbosity. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORT_CONDITIONAL(      U32   MinVerbosity,
                           const char* sMsg);
```

Parameters

Parameter	Description
MinVerbosity	Minimum verbosity level to display the message.
sMsg	The string to be sent to the terminal.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.9 SYS_REPORT1_CONDITIONAL()

Description

`SYS_REPORT1_CONDITIONAL()` is a wrapper for `SYS_Report1()` respecting verbosity. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORT1_CONDITIONAL(      U32   MinVerbosity,
                            const char* sMsg,
                                  U32   Value);
```

Parameters

Parameter	Description
`MinVerbosity`	Minimum verbosity level to display the message.
`sMsg`	The string to be sent to the terminal.
`Value`	A 32-bit value to be shown in hexadecimal form without '`0x`' prefix.

Return value

Always returns 0.

Additional information

The following table describes the permitted values for the `MinVerbosity` parameter.

Value
VERBOSITY_NONE
VERBOSITY_BASIC
VERBOSITY_DETAILED
VERBOSITY_DEBUG
VERBOSITY_FULL

Additional information

> **Note**
>
> This is a macro.

4.4.10 SYS_REPORTF_CONDITIONAL()

Description

SYS_REPORTF_CONDITIONAL() is a wrapper for SYS_Reportf() respecting verbosity. The information is discarded in non-descriptive builds.

Syntax

```
int SYS_REPORTF_CONDITIONAL(      U32   MinVerbosity,
                            const char* sFormat,
                                  ...   ...);
```

Parameters

Parameter	Description
MinVerbosity	Minimum verbosity level to display the message.
sFormat	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

Additional information

The following table describes the permitted values for the MinVerbosity parameter.

Value
VERBOSITY_NONE
VERBOSITY_BASIC
VERBOSITY_DETAILED
VERBOSITY_DEBUG
VERBOSITY_FULL

Additional information

> **Note**
>
> This is a macro.

4.4.11 SYS_REPORT_ERR()

Description

SYS_REPORT_ERR() is a macro for error output. This is never discarded, not even in release builds with VERBOSITY_NONE.

Syntax

```
int SYS_REPORT_ERR(const char* sMsg);
```

Parameters

Parameter	Description
sMsg	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.12 SYS_REPORT1_ERR()

Description

SYS_REPORT1_ERR() is a macro for error output. This is never discarded, not even in release builds with VERBOSITY_NONE.

Syntax

```
int SYS_REPORT1_ERR(const char* sMsg,
                    int   Value);
```

Parameters

Parameter	Description
sMsg	The string to be formatted and sent to the terminal.
Value	A 32-bit value to be shown in hexadecimal form without '0x' prefix.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.13 SYS_REPORTF_ERR()

Description

SYS_REPORTF_ERR() is a macro for error output. This is never discarded, not even in release builds with VERBOSITY_NONE.

Syntax

```
int SYS_REPORTF_ERR(const char* sMsg,
                            ...   ...);
```

Parameters

Parameter	Description
sMsg	The string to be formatted and sent to the terminal.

Return value

Always returns 0.

Additional information

> **Note**
>
> This is a macro.

4.4.14 SYS_ReportLoaderCompileDate()

Description

`SYS_ReportLoaderCompileDate()` reports the flash loader compilation date.

Syntax

```
void SYS_ReportLoaderCompileDate(void);
```


4.5 Timing functions

Function	Description
SYS_GetTime()	Returns the Flasher's current system time in milliseconds.
SYS_GetTime_us()	Returns the Flasher's current system time in microseconds.
SYS_Delay_us()	Actively waits for a specified time.
SYS_Sleep_ms()	Suspends the execution of the app for a specified time in milliseconds.
SYS_Sleep_us()	Suspends the execution of the app for a specified time in microseconds.
SYS_Sleep_ns()	Suspends the execution of the app for a specified time in nanoseconds.
SYS_SetMaxPause()	Sets a maximum pause to ensure that the app is not preempted by other tasks.

4.5.1 SYS_Delay_us()

Description

`SYS_Delay_us()` waits actively for the specified time before the app resumes execution.

Syntax

```
void SYS_Delay_us(U32 us);
```

Parameters

Parameter	Description
us	The time period to actively delay execution in microseconds.

Return value

Always returns 0.

Additional information

If `SYS_Delay_us()` is used to wait several milliseconds, it is recommended to use `SYS_Sleep_ms()` for the full milliseconds.

4.5.2 SYS_Sleep_ms()

Description

`SYS_Sleep_ms()` suspends the execution of the app for the specified time.

Syntax

```
void SYS_Sleep_ms(U32 ms);
```

Parameters

Parameter	Description
`ms`	The time period in milliseconds that the app is suspended.

Return value

Always returns 0.

4.5.3 SYS_Sleep_us()

Description

`SYS_Sleep_us()` suspends the execution of the app for the specified time.

Syntax

```
void SYS_Sleep_us(U32 us);
```

Parameters

Parameter	Description
us	The time period in microseconds that the app is suspended.

4.5.4 SYS_Sleep_ns()

Description

`SYS_Sleep_ns()` suspends the execution of the app for the specified time.

Syntax

```
void SYS_Sleep_ns(U32 ns);
```

Parameters

Parameter	Description
ns	The time period in nanoseconds that the app is suspended.

4.5.5 SYS_SetMaxPause()

Description

`SYS_SetMaxPause()` sets the maximum pause to ensure that the app is not preempted by other tasks for a period longer than the time specified by `MaxPause_us`. If `MaxPause_us` is greater than zero, the app is executed for the maximum pause time (at least 100 us) without other tasks being able to preempt the app execution. After this time, other tasks have the chance to preempt the app again for the same period of time. This is repeated until another maximum pause time is set or the app finished execution which automatically resets the maximum pause time. By default, the maximum pause time is set to zero, i.e. there is no guaranteed maximum pause time after which the app will definitely continue again.

Syntax

```
U32 SYS_SetMaxPause(U32 us);
```

Parameters

Parameter	Description
`MaxPause_us`	The maximum pause time to be used, which must be at least 100 microseconds. A value of zero will reset the maximum pause to zero. Values between 1 and 100 will set the maximum pause time to 100 us.

Return value

The previously set maximum pause time.

Additional information

This function is only required in very rare cases. Most Apps do not need to use this function.

4.5.6 SYS_GetTime()

Description

`SYS_GetTime()` returns the Flasher's uptime in milliseconds.

Syntax

```
int SYS_GetTime(void);
```

Return value

Uptime in milliseconds.

Additional information

When the Flasher is powered on, it starts counting elapsed milliseconds. The time is stored internally as 32-bit value and is increments each millisecond. After approximately 49 days the time value will overflow from `0xFFFFFFFF` to `0x00000000`. To avoid any complications caused by the overflow, apps must only work with time differences calculated using two timestamps. The difference of two timestamps is correct if and only if the period between the timestamps does not exceed `0x7FFFFFFF` milliseconds (approximately 24.9 days), even if an overflow occurred between the two timestamps.

This an example that calculates the difference of two timestamps that had a counter overflow between them:

```
t0 = 0xFFFFFFF0
t1 = 0x00000005
t1 - t0 = 0x00000015 => 21 milliseconds
```

Example

```
#define DURATION 20

int t0;
int tNow;

t0 = SYS_GetTime();
do {
  //
  // Repeat something for 20 milliseconds.
  //
  tNow = SYS_GetTime();
} while ((tNow - t0) < DURATION);
```


4.5.7 SYS_GetTime_us()

Description

`SYS_GetTime_us()` returns the Flasher's uptime in microseconds.

Syntax

```
U32 SYS_GetTime_us(void);
```

Return value

Uptime in microseconds.

Additional information

When the Flasher is powered on, it starts counting elapsed microseconds. The time is stored internally as 32-bit value and is increments each microsecond. After approximately 49 days the time value will overflow from `0xFFFFFFFF` to `0x00000000`. To avoid any complications caused by the overflow, apps must only work with time differences calculated using two timestamps. The difference of two timestamps is correct if and only if the period between the timestamps does not exceed `0x7FFFFFFF` microseconds (approximately 36 minutes), even if an overflow occurred between the two timestamps.

This an example that calculates the difference of two timestamps that had a counter overflow between them:

```
t0 = 0xFFFFFFF0
t1 = 0x00000005
t1 - t0 = 0x00000015 => 21 microseconds
```

Example

```
#define DURATION 20

U32 t0;
U32 tNow;

t0 = SYS_GetTime_us();
do {
  //
  // Repeat something for 20 microseconds.
  //
  tNow = SYS_GetTime_us();
} while ((tNow - t0) < DURATION);
```


4.6 File system functions

Function	Description
SYS_FILE_Open()	Opens a file on the Flasher’s file system.
SYS_FILE_Close()	Closes an opened file.
SYS_FILE_Read()	Reads from a file.
SYS_FILE_SetPos()	Sets the file pointer of an opened file.
SYS_FILE_GetSize()	Gets the size of an opened file.
SYS_FILE_ReadChunk()	Read a chunk from a file on the Flasher’s file system.

4.6.1 SYS_FILE_Open()

Description

`SYS_FILE_Open()` opens a file on the file system.

Syntax

```
SYS_FILE_HANDLE SYS_FILE_Open(const char* sFilename,
                              U32   Mode);
```

Parameters

Parameter	Description
`sFilename`	A string with the name of the file to open.
`Flags`	Or-combination of `SYS_FILE_FLAG_*` values.

Return value

Value	Description
≥ 0	Success, file handle.
< 0	Error.
= -1	Unspecified error.
= -2	One or more unknown flags passed.
= -3	Unsupported combination of flags passed.

Additional information

When `sFilename` starts with the directory delimiter '\' it is interpreted as an absolute path on the Flasher file system. Otherwise, it is interpreted as a path relative to the directory that the app executable is located in.

> **Note**
>
> When executing an app from Flasher's internal memory instead of the file system (i.e. using *Flasher App Runner*), `sFilename` is always interpreted as an absolute path.

The following table describes the permitted values for the `Flags` parameter.

Value	Description
`SYS_FILE_FLAG_READ`	Open file for reading in binary mode.
`SYS_FILE_FLAG_READ_DECOMPRESSED`	Open compressed file for reading in binary mode. The file must be compressed by the SEGGER SMASH2 compressor using a window size of 256 bytes. Not combinable with `SYS_FILE_FLAG_WRITE`.
`SYS_FILE_FLAG_WRITE`	Open file for writing in binary mode.

The following table describes which effect a certain `Flags` value has when used for `SYS_FILE_Open()`.

Flags	Deletes file's content if it exists?	Creates file if it does not exist?	File position after open
`SYS_FILE_FLAG_READ`	No	No	Beginning
`SYS_FILE_FLAG_READ_DECOM-PRESSED`	No	No	Beginning
`SYS_FILE_FLAG_WRITE`	Yes	Yes	End
`SYS_FILE_FLAG_READ | SYS_FILE_FLAG_WRITE`	No	Yes	End

SEGGER offers a SMASH2 compressor in emCompress-ToGo and emCompress-PRO.

Example

In the following examples, `%AppExeFolder%` serves as placeholder for the path to the folder the App executable is located in.

```
SYS_FILE_Open("\\Flasher.log", SYS_FILE_FLAG_READ);
```

- The leading delimiter (`\`) marks the path as an absolute path.
- **When executed from an app on the file system**:
  Opens `Flasher.log` in the root folder for reading.
- **When executed from an app in memory**:
  Opens `Flasher.log` in the root folder for reading.

```
SYS_FILE_Open("Serial.txt", SYS_FILE_FLAG_READ);
```

- No leading delimiter means Flasher will interpret this path as relative, if possible.
- **When executed from an app on the file system**:
  Opens `%AppExeFolder%\Serial.txt` for reading.
- **When executed from an app in memory**:
  Opens `\Serial.txt` in the root folder for reading.

```
SYS_FILE_Open("SubFolder\\MyFile.txt", SYS_FILE_FLAG_READ);
```

- No leading delimiter means Flasher will interpret this path as relative, if possible.
- **When executed from an app on the file system**:
  Opens `%AppExeFolder%\SubFolder\MyFile.txt` for reading.
- **When executed from an app in memory**:
  Opens `\SubFolder\MyFile.txt` for reading.

4.6.2 SYS_FILE_Close()

Description

Close a previously opened file.

Syntax

```
int SYS_FILE_Close(SYS_FILE_HANDLE hFile);
```

Parameters

Parameter	Description
hFile	Handle to an open file.

Return value

Value	Description
≥ 0	OK.
< 0	Error.
= -1	Unspecified error.
= -2	Invalid handle.

4.6.3 SYS_FILE_Read()

Description

Read from a previously opened file.

Syntax

```
int SYS_FILE_Read(SYS_FILE_HANDLE hFile,
                  void*           pBuffer,
                  U32             NumBytes);
```

Parameters

Parameter	Description
hFile	Handle to an open file.
pBuffer	A pointer to a buffer the file should be stored in.
NumBytes	The number of bytes to read.

Return value

Value	Description
≥ 0	OK, number of bytes read (may be smaller than NumBytes in case EOF was hit)
< 0	Error.
= -1	Unspecified error.
= -2	Invalid handle.

4.6.4 SYS_FILE_Write()

Description

Write to a previously opened file.

Syntax

```
int SYS_FILE_Write(      SYS_FILE_HANDLE hFile,
                   const void*           pBuffer,
                         U32             NumBytes);
```

Parameters

Parameter	Description
hFile	Handle to an open file.
pBuffer	A pointer to a buffer that holds the data to be written to the file.
NumBytes	The number of bytes to write.

Return value

Value	Description
≥ 0	OK, number of bytes written (may be smaller than NumBytes)
< 0	Error.
= -1	Unspecified error.
= -2	Invalid handle.

4.6.5 SYS_FILE_SetPos()

Description

Moves the file pointer of a previously opened file.

Syntax

```
int SYS_FILE_SetPos(SYS_FILE_HANDLE hFile,
                    int             Offset,
                    SYS_FILE_ORIGIN Origin);
```

Parameters

Parameter	Description
hFile	Handle to an open file.
Offset	The position within the file
Origin	Origin of type PCODE_SYS_FILE_ORIGIN_*.

Return value

Value	Description
≥ 0	OK.
< 0	Error.
= -1	Unspecified error.
= -2	Invalid handle.
= -3	Invalid origin.

Additional information

The following table describes the permitted values for the Origin parameter.

Value	Description
SYS_FILE_ORIGIN_BEGIN	Start of the file.
SYS_FILE_ORIGIN_CURRENT	Current position of the file.
SYS_FILE_ORIGIN_END	End of the file.

4.6.6 SYS_FILE_GetSize()

Description

Returns the size, in bytes, of a previously opened file.

Syntax

```
int SYS_FILE_GetSize(SYS_FILE_HANDLE hFile);
```

Parameters

Parameter	Description
hFile	Handle to an open file.

Return value

Value	Description
≥ 0	OK, file size in bytes.
< 0	Error.
= -1	Unspecified error.
= -2	Invalid handle.

4.6.7 SYS_FILE_ReadChunk()

Description

Reads from a file located on the Flasher.

Syntax

```
int SYS_FILE_ReadChunk(const char* sFilename,
                       void* pBuffer,
                       U32   Offset,
                       U32   Length,
                       U32*  pCRC);
```

Parameters

Parameter	Description
sFilename	A string with the name of the file.
pBuffer	A pointer to a buffer the file should be stored in.
Offset	The position within the file to start reading at.
Length	The number of bytes to read.
pCRC	A pointer to a U32 variable which can be used to pass a start value or the result of a previous CRC calculation. After CRC calculation the variable pCRC is pointing to contains the result of the calculation. If NULL is passed, no CRC calculation is performed.

Additional information

When sFilename starts with the directory delimiter '\' it is interpreted as an absolute path on the Flasher file system. Otherwise, it is interpreted as a path relative to the directory that the app executable is located in.

> **Note**
>
> When executing an app from Flasher's internal memory instead of the file system (i.e. using *Flasher App Runner*), sFilename is always interpreted as an absolute path.

Return value

Returns the number of bytes that were read.

4.7 Target interface

4.7.1 Interface control functions

Function	Description
TIF_Select()	Select target interface.
TIF_ReleaseReset()	Release target reset.
TIF_ActivateReset()	Assert target reset.
TIF_PIN_SetFunc()	Configure pin function.
TIF_PIN_GetState()	Sample pin state.
TIF_PIN_Strobe()	Strobe pin.
TIF_PIN_SetTCK()	Set state of TCK.
TIF_PIN_SetTMS()	Set state of TMS.
TIF_PIN_SetTDI()	Set state of TDI.

4.7.1.1 TIF_Select()

Description

Selects a built-in standard target interface.

Syntax

```
void TIF_Select(U32 InterfaceId);
```

Parameters

Parameter	Description
`InterfaceId`	Identifier of the target interface.

Additional information

The following table describes the permitted values for the `InterfaceId` parameter.

Value	Description
`TIF_JTAG`	JTAG interface.
`TIF_SWD`	SWD interface.
`TIF_SPI`	SPI interface.
`TIF_I2C`	I2C interface.

4.7.1.2 TIF_ReleaseReset()

Description

Releases the target reset by pulling the reset pin (pin 15) to a high signal level.

Syntax

```
void TIF_ReleaseReset(void);
```


4.7.1.3 TIF_ActivateReset()

Description

Activates the target reset by pulling the reset pin (pin 15) to a low signal level.

Syntax

```
void TIF_ActivateReset(void);
```


4.7.1.4 TIF_SetSpeed()

Description

Set the target interface communication speed.

Syntax

```
void TIF_SetSpeed(U32 Speed);
```

Parameters

Parameter	Description
Speed	The frequency, in hertz, to be used for the target communication.

4.7.1.5 TIF_PIN_SetFunc()

Description

Set the function of a pin.

Syntax

```
int TIF_PIN_SetFunc(TIF_PIN      Pin,
                    TIF_PIN_FUNC State);
```

Parameters

Parameter	Description
Pin	Target interface pin to affect.
State	The new state of the specified pin.

Return value

Value	Description
≥ 0	OK.
< 0	Error.

Additional information

The following table describes the permitted values for the Pin parameter.

Value	Description
TIF_PIN_PIN3	Pin 3.
TIF_PIN_PIN5	Pin 5.
TIF_PIN_PIN7	Pin 7.
TIF_PIN_PIN9	Pin 9.
TIF_PIN_PIN11	Pin 11.
TIF_PIN_PIN13	Pin 13.
TIF_PIN_PIN15	Pin 15.
TIF_PIN_PIN17	Pin 17.

The following table describes the permitted values for the State parameter.

Value	Description
TIF_PIN_FUNC_DEFAULT	Pin is under Flasher’s control.
TIF_PIN_FUNC_IN	Input.
TIF_PIN_FUNC_OUT_LOW	Output with low level (logical 0).
TIF_PIN_FUNC_OUT_HIGH	Output with high level (logical 1).
TIF_PIN_FUNC_UART_TX	Output for UART Tx signal.
TIF_PIN_FUNC_UART_RX	Input for UART Rx signal.
TIF_PIN_FUNC_UART_RXTX	Bidirectional for UART in half-duplex mode.

4.7.1.6 TIF_PIN_GetState()

Description

Returns the state of the specified pin.

Syntax

```
int TIF_PIN_GetState(TIF_PIN Pin);
```

Parameters

Parameter	Description
Pin	Target interface pin to get state from.

Additional information

The following table describes the permitted values for the Pin parameter.

Value	Description
TIF_PIN_PIN3	Pin 3.
TIF_PIN_PIN5	Pin 5.
TIF_PIN_PIN7	Pin 7.
TIF_PIN_PIN9	Pin 9.
TIF_PIN_PIN11	Pin 11.
TIF_PIN_PIN13	Pin 13.
TIF_PIN_PIN15	Pin 15.
TIF_PIN_PIN17	Pin 17.

Return value

Value	Description
= 1	Pin state is high (logical 1).
= 0	Pin state is low (logical 0).
< 0	Error.

4.7.1.7 TIF_PIN_Strobe()

Description

Create a strobe signal on a pin.

> **Note**
>
> Currently, only pin 5 can be used for the strobe signal.

Syntax

```
int TIF_PIN_Strobe(TIF_PIN Pin,
                   U32     Frequency,
                   U32     NumClocks);
```

Parameters

Parameter	Description
Pin	Target interface pin to affect.
Frequency	Frequency of the strobe in hertz.
NumClocks	Number of strobes (low-high pulse).

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

The following table describes the permitted values for the Pin parameter.

Value	Description
TIF_PIN_PIN3	Pin 3.
TIF_PIN_PIN5	Pin 5.
TIF_PIN_PIN7	Pin 7.
TIF_PIN_PIN9	Pin 9.
TIF_PIN_PIN11	Pin 11.
TIF_PIN_PIN13	Pin 13.
TIF_PIN_PIN15	Pin 15.
TIF_PIN_PIN17	Pin 17.

4.7.1.8 TIF_PIN_SetTCK()

Description

Set the output state of pin 9 (TCK).

Syntax

```
void TIF_PIN_SetTCK(int State);
```

Parameters

Parameter	Description
State	Output state =0 for low (logical 0); =1 for high (logical 1).

4.7.1.9 TIF_PIN_SetTMS()

Description

Set the output state of pin 7 (TMS).

Syntax

```
void TIF_PIN_SetTMS(int State);
```

Parameters

Parameter	Description
State	Output state =0 for low (logical 0); =1 for high (logical 1).

4.7.1.10 TIF_PIN_SetTDI()

Description

Set the output state of pin 5 (TDI).

Syntax

```
void TIF_PIN_SetTDI(int State);
```

Parameters

Parameter	Description
State	Output state =0 for low (logical 0); =1 for high (logical 1).

4.7.2 JTAG interface functions

The following chapter describes the API functions for the JTAG target interface.

Pin mapping

The following table shows the fixed pin mapping for the JTAG target interface.

Signal	Target interface pin
TDI	Pin 5.
TMS	Pin 7.
TCK	Pin 9.
TDO	Pin 13.

Example

The following example extracts the first ID code after TAP reset from the attached.

```
int main(void) {
  int Pos;
  //
  // Select JTAG target interface.
  //
  TIF_Select(TIF_JTAG);
  TIF_SetSpeed(1000000);  // Interface speed 1 MHz.
  //
  // Clock TAP to Run-Test/Idle from any state.
  //   5 x TMS=1
  //   1 x TMS=0
  //
  TIF_JTAG_Store(0x1F, 0, 6);
  //
  // Perform a DR scan of 32 bits.  Remember where
  // data scan starts within JTAG buffer.
  //
  Pos = TIF_JTAG_WriteDR(0, 32);
  //
  // Read IDCODE from scanned-out data correspomding
  // to the data scanned in.
  //
  printf("IDCODE: %08X\n", TIF_JTAG_GetU32(Pos));
  return 0;
}
```

Functions

Function	Description
TIF_JTAG_GetU32()	Gets 32 bits JTAG data from the bitstream output by the DUT (Device under test).
TIF_JTAG_GetU32Rev()	Retrieves a reversed U32 from the bit-stream output by the DUT (Device under test).
TIF_JTAG_ReadWriteBits()	Stores the provided data in the JTAG buffer and sends all data contained in the buffer.
TIF_JTAG_Reset()	Resets the TAP controller's state machine and puts the TAP into Run-Test/Idle state by issuing 5 clock cycles while TMS is held

Function	Description
	HIGH followed by one clock cycle while TMS is LOW.
TIF_JTAG_SetupChain()	Setups the JTAG chain, meaning it specifies all important parameters, so the Flasher knows which device to talk to, how many dummy bits are necessary and how long the IRLen, of the device we shall talk to, is.
TIF_JTAG_StartDR()	Brings the state machine of the selected device in the JTAG-chain in SHIFT-DR state.
TIF_JTAG_Store()	Stores the provided data in the JTAG buffer.
TIF_JTAG_StoreClocks()	Stores a given number of clock cycles in the JTAG buffer with TMS being held LOW.
TIF_JTAG_StoreDR()	Stores data of variable length in the JTAG buffer and remains in UPDATE-DR state.
TIF_JTAG_StoreDRRev()	Stores data of variable length in reversed bit order in the JTAG buffer and remains in UPDATE-DR state.
TIF_JTAG_StoreIR()	Stores a JTAG instruction in the JTAG buffer and remains in UPDATE-IR state.
TIF_JTAG_TransferBits()	Transmits the provided JTAG data.
TIF_JTAG_Write()	Stores the provided data in the JTAG buffer and sends all data contained in the buffer.
TIF_JTAG_WriteClocks()	Stores a given number of clock cycles in the JTAG buffer with TMS being held LOW and writes all data contained in the buffer.
TIF_JTAG_WriteDR()	Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer.
TIF_JTAG_WriteDRCont()	Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer.
TIF_JTAG_WriteDREnd()	Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer.
TIF_JTAG_WriteDRRev()	Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer.
TIF_JTAG_WriteIR()	Stores the JTAG instruction in the JTAG buffer and writes all data contained in the buffer.

4.7.2.1 TIF_JTAG_GetU32()

Description

Gets 32 bits JTAG data from the bitstream output by the DUT (Device under test).

Syntax

```
U32 TIF_JTAG_GetU32(int BitPos);
```

Parameters

Parameter	Description
BitPos	Specifies the bit position where reading of the U32 data should be started.

Return value

Returns 32 bits of data starting at the bit position specified by BitPos.

Additional information

This function clears the JTAG buffer before data is read.

4.7.2.2 TIF_JTAG_GetU32Rev()

Description

Retrieves a reversed U32 from the bitstream output by the DUT (Device under test).

Syntax

```
U32 TIF_JTAG_GetU32Rev(int BitPos);
```

Parameters

Parameter	Description
BitPos	Specifies the bit position where reading of the U32 data should be started.

Return value

Returns 32 bits of bit-reversed data starting at the bit position specified by BitPos.

Additional information

This function clears the JTAG buffer before data is read.

4.7.2.3 TIF_JTAG_ReadWriteBits()

Description

The function `TIF_JTAG_ReadWriteBits()` stores the provided data in the JTAG buffer and sends all data contained in the buffer. The received data will be stored in the provided TDO buffer.

Syntax

```
int TIF_JTAG_ReadWriteBits(const U8* pTDI,
                           const U8* pTMS,
                                 U8* pTDO,
                                 U32 NumBits);
```

Parameters

Parameter	Description
`pTDI`	Pointer to a buffer with data to be written on TDI.
`pTMS`	Pointer to a buffer with data to be written on TMS.
`pTDO`	Pointer to a buffer the data read via TDO will be stored in.
`NumBits`	Number of bits to transfer.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

4.7.2.4 TIF_JTAG_Reset()

Description

Reset the TAP controller’s state machine and puts the TAP into Run-Test/Idle state by issuing 5 clock cycles while TMS is held HIGH followed by one clock cycle while TMS is LOW.

Syntax

```
int TIF_JTAG_Reset(void);
```

Return value

Always returns 0.

4.7.2.5 TIF_JTAG_SetupChain()

Description

Set up the JTAG chain to address DUT. This specifies all important parameters so the Flasher knows which device to talk to, how many dummy bits are necessary and the width of the instruction register of the device to shall talk to.

Syntax

```
void TIF_JTAG_SetupChain(int IRPre,
                         int DRPre,
                         int IRPost,
                         int DRPost,
                         int IRLenDevice);
```

Parameters

Parameter	Description
IRPre	Number of IR (instruction register) bits before DUT.
DRPre	Number of DR (data register) bits before DUT.
IRPost	Number of IR (instruction register) bits after DUT.
DRPost	Number of DR (data register) bits after DUT.
IRLenDevice	IR (instruction register) length of the DUT.

4.7.2.6 TIF_JTAG_StartDR()

Description

Brings the state machine of the selected device in the JTAG-chain in SHIFT-DR state.

Syntax

```
int TIF_JTAG_StartDR(void);
```

Return value

Value	Description
≥ 0	Bit position in JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

Data is stored in the JTAG buffer but not written. This behavior differs from `TIF_JTAG_StartDR()` executed by the J-Link DLL, because the J-Link DLL immediately writes the data contained in the JTAG buffer after it was stored there.

After calling this function, `TIF_JTAG_WriteDRCont()` and `TIF_JTAG_WriteDREnd()` can be used to write data on TDI. `TIF_JTAG_WriteDREnd()` has to be the last function call which puts the TAP controller into UPDATE-DR state.

4.7.2.7 TIF_JTAG_Store()

Description

Stores the provided data in the JTAG buffer.

Syntax

```
int TIF_JTAG_Store(U32 TMS,
                   U32 TDI,
                   U32 NumBits);
```

Parameters

Parameter	Description
TMS	Bitmask to output on TMS.
TDI	Bitmask to output on TDI.
NumBits	NumBits to store for each pin. Maximum 32 bits.

Return value

Always returns 0.

4.7.2.8 TIF_JTAG_StoreClocks()

Description

Stores a given number of clock cycles in the JTAG buffer with TMS being held LOW.

Syntax

```
int TIF_JTAG_StoreClocks(int NumClocks);
```

Parameters

Parameter	Description
NumClocks	Number of clock cycles to store in the JTAG buffer.

Return value

Always returns 0.

4.7.2.9 TIF_JTAG_StoreDR()

Description

Stores data of variable length in the JTAG buffer and remains in UPDATE-DR state.

Syntax

```
int TIF_JTAG_StoreDR(U32 Data,
                     int NumBits);
```

Parameters

Parameter	Description
Data	Bitmask to output on TDI.
NumBits	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

This function expects that the JTAG chain has already been configured before.

4.7.2.10 TIF_JTAG_StoreDRRev()

Description

Stores data of variable length in the JTAG buffer and remains in UPDATE-DR state.

Syntax

```
int TIF_JTAG_StoreDRRev(U32 Data,
                        int NumBits);
```

Parameters

Parameter	Description
Data	Bitmask to output on TDI.
NumBits	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

This function expects that the JTAG chain has already been configured before.

4.7.2.11 TIF_JTAG_StoreIR()

Description

Stores a JTAG instruction in the JTAG buffer and remains in UPDATE-IR state.

Syntax

```
int TIF_JTAG_StoreIR(U32 Instruction);
```

Parameters

Parameter	Description
`Instruction`	The instruction to be stored in the JTAG buffer.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

This function expects that the JTAG chain has already been configured before.

4.7.2.12 TIF_JTAG_TransferBits()

Description

Transmits the provided JTAG data. This function returns as soon as data transmission has been scheduled, but the transfer may have not completed yet.

This function may be called multiple times in a row without waiting for the previous transfer to complete, to schedule multiple transfers in a row. All pointers passed to this function must stay valid until the scheduled transfer completed.

In order to wait until all scheduled transfers have completed, use the `TIF_JTAG_WAIT_FOR_TRANSFER_BITS_DONE()` macro.

Syntax

```
int TIF_JTAG_TransferBits(const U8* pTDI,
                          const U8* pTMS,
                                U8* pTDO,
                                U32 NumBits);
```

Parameters

Parameter	Description
pTDI	Pointer to TDI data to be transmitted. Transmitted LSB first
pTMS	Pointer to TMS data to be transmitted. Transmitted LSB first
pTDO	Optional, may be `NULL`. Pointer to buffer where TDO data received by this function is stored to. May point to the same memory as pTDI or pTMS (if they point to writable memory) if their data is no longer needed after the transmission completed. This avoids the need of a separate buffer for TDO. Buffer is filled LSB first.
NumBits	Number of bits to transmit.

Return value

Value	Description
= 0	OK. New transfer(s) scheduled. Will be appended gapless to ongoing transfers.
= 1	OK. New transfer(s) scheduled. There will be gaps between the transfers.
= 2	OK. No new transfers scheduled. Waited for ongoing transfers to finish (if there were any).
< 0	Error. (Can usually only happen if adaptive clocking is active and a timeout occurred)

Examples

Scheduling of 2 transfers

```
  U8 aTDI0[128];
  U8 aTMS0[128];
  U8 aTDI1[128];
  U8 aTMS1[128];
  U32 NumBits;
  U8* pTDI;
  U8* pTMS;
  U8* pTDO;
  //
  // a) Schedule 1st sequence.
  //    Reuse TDI buffer for TDO data as we do not need TDI
  //    of this sequence anymore afterwards.
```



```
// b) Load data for 2nd sequence from file while 1st sequence
//    is transmitted in the background.
// c) Schedule 2nd sequence.
//    We are not interested in TDO data of the 2nd sequence
// d) Wait until all scheduled transfers completed
//
pTDI = &aTDI0[0];
pTMS = &aTMS0[0];
pTDO = &aTDI0[0];
_LoadJTAGDataFromSTAPLFile(pTDI, pTMS, &NumBits);
TIF_JTAG_TransferBits(pTDI, pTMS, pTDO, NumBits);           // a)
pTDI = &aTDI1[0];
pTMS = &aTMS1[0];
pTDO = NULL;
_LoadJTAGDataFromSTAPLFile(pTDI, pTMS, &NumBits);           // b)
TIF_JTAG_TransferBits(pTDI, pTMS, pTDO, NumBits);           // c)
TIF_JTAG_WAIT_FOR_TRANSFER_BITS_DONE();                     // d)
```


4.7.2.13 TIF_JTAG_Write()

Description

Stores the provided data in the JTAG buffer and sends all data contained in the buffer.

Syntax

```
int TIF_JTAG_Write(U32 TMS,
                   U32 TDI,
                   U32 NumBits);
```

Parameters

Parameter	Description
TMS	Bitmask to output on TMS.
TDI	Bitmask to output on TDI.
NumBits	NumBits to store for each pin. Maximum 32 bits.

Return value

Always returns 0.

4.7.2.14 TIF_JTAG_WriteClocks()

Description

Stores a given number of clock cycles in the JTAG buffer with TMS being held LOW and writes all data contained in the buffer.

Syntax

```
int TIF_JTAG_WriteClocks(int NumClocks);
```

Parameters

Parameter	Description
NumClocks	Number of clock cycles to write.

Return value

Always returns 0.

4.7.2.15 TIF_JTAG_WriteDR()

Description

Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer. Remains in the UPDATE-DR state.

Syntax

```
int TIF_JTAG_WriteDR(int NumClocks);
```

Parameters

Parameter	Description
Data	Bitmask to output on TDI.
NumBits	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

4.7.2.16 TIF_JTAG_WriteDRCont()

Description

Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer. Remains in the UPDATE-DR state.

Syntax

```
int TIF_JTAG_WriteDRCont(int NumClocks);
```

Parameters

Parameter	Description
`Data`	Bitmask to output on TDI.
`NumBits`	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in the SHIFT-DR state which is achieved by calling `TIF_JTAG_StartDR()`. Thus, `TIF_JTAG_WriteDRCont()` can be called multiple times to write data. The last data has to be written by calling `TIF_JTAG_WriteDREnd()` in order to bring the TAP controller into UPDATE-DR state.

4.7.2.17 TIF_JTAG_WriteDREnd()

Description

Stores the data of variable length in the JTAG buffer and writes all data contained in the buffer. Remains in the UPDATE-DR state.

Syntax

```
int TIF_JTAG_WriteDREnd(U32 Data,
                        int NumBits);
```

Parameters

Parameter	Description
`Data`	Bitmask to output on TDI.
`NumBits`	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in the SHIFT-DR state which is achieved by calling `TIF_JTAG_StartDR()`.

4.7.2.18 TIF_JTAG_WriteDRRev()

Description

Stores the data of variable length bit reversed in the JTAG buffer and writes all data contained in the buffer. Remains in the UPDATE-DR state.

Syntax

```
int TIF_JTAG_WriteDRRev(U32 Data,
                        int NumBits);
```

Parameters

Parameter	Description
Data	Bitmask to output on TDI.
NumBits	Number of bits to store.

Return value

Returns the bit position the data was appended to in the JTAG buffer.

Additional information

Requires the TAP controller to be in Run-Test/Idle state.

4.7.2.19 TIF_JTAG_WriteIR()

Description

Stores the JTAG instruction in the JTAG buffer and writes all data contained in the buffer. Remains in the UPDATE-IR state.

Syntax

```
int TIF_JTAG_WriteIR(U32 Instruction);
```

Parameters

Parameter	Description
`Instruction`	The JTAG instruction to write.

Return value

Returns the bit position the instruction was appended to in the JTAG buffer.

4.7.3 SWD interface functions

The following chapter describes the API functions for the SWD target interface.

Pin mapping

The following table shows the fixed pin mapping for the SWD target interface.

Signal	Target interface pin
SWDIO	Pin 7.
SWCLK	Pin 9.

Example

```
int main(void) {
  const U8 aIn[33] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  // SWD Line reset.
    0x9E, 0xE7,                                // Switching sequence.
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  // SWD Line reset.
    0xB6, 0xED,                                // Sw seq. luminary (deprecated).
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,  // SWD Line reset.
    0x00, 0x00,                                // SWD ready for a start bit.
    0xA5, 0x00, 0x00, 0x00, 0x00, 0x00         // Read Id Register.
  };
  const U8 aDir[33] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF,
    0xFF, 0x00, 0x00, 0x00, 0x00, 0xF0
  };
  U8 _aDataOut[33];
  U32 Off;
  U32 v;
  int r;
  //
  // Select SWD target interface.
  //
  TIF_Select(TIF_SWD);
  TIF_SetSpeed(1000000);  // Interface speed 1 MHz.
  //
  // Trigger SWD transfer.
  //
  r = TIF_SWD_ReadWriteBits(&aIn[0], &aDir[0], &_aDataOut[0], sizeof(aIn) * 8);
  if (r != 0) {
    printf("Error, transfer failed.\n");
    return -1;
  }
  //
  // Extract IDCODE from response.
  //
  Off = (3 * 9);  // Skip 2x line-reset-switching-sequence,
 1x line-reset-make-sure-ready-bit.
  Off += 1;       // Skip "read ID register" command.
  v  = _aDataOut[Off + 0] <<  0;
  v |= _aDataOut[Off + 1] <<  8;
  v |= _aDataOut[Off + 2] << 16;
  v |= _aDataOut[Off + 3] << 24;
  //
  // The first 3 bits after the command are the response
 (OK/WAIT/FAULT) on the DAP level and are not required.
```



```
  //
  v >>= 3;
  //
  // Load the final 3 bits from the next byte and merge them into the IDCODE.
  //
  v |= (_aDataOut[Off + 4] & 7) << 29;
  printf("IDCODE: 0x%08X\n", v);
  return 0;
}
```

Functions

Function	Description
TIF_SWD_ReadWriteBits()	Reads and writes data via SWD.

4.7.3.1 TIF_SWD_ReadWriteBits()

Description

Exchange data via SWD.

Syntax

```
int TIF_SWD_ReadWriteBits(const U8* pDataIn,
                          const U8* pDirection,
                                U8* pDataOut,
                                int NumBits);
```

Parameters

Parameter	Description
pDataIn	Pointer to buffer with data to write.
pDirection	Pointer to buffer with direction information. Ones indicate that data should be written, while zeros indicate that data should be read.
pDataOut	Pointer to buffer that receives data.
NumBits	Total number of bits to be read and written.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

4.7.4 CoreSight interface functions

The following chapter describes the API functions for the CoreSight target interface. It represents a high-level interface built upon the JTAG and SWD protocols, providing a unified method for communication with targets (mostly ARM). The CoreSight target interface abstracts the low-level details of the underlying transport mechanisms, allowing developers to easily access the target's memory, registers, and system components in a consistent and standardized manner.

More information can be found in our *Knowledge Base article* about CoreSight.

Example

The following example configures CoreSight using the JTAG interface and reads the DP CTLR/STAT register.

```
int main(void) {
  U32 v;
  int r;
  //
  // Select JTAG target interface.
  //
  TIF_Select(TIF_JTAG);
  TIF_SetSpeed(1000000);  // Interface speed 1 MHz.
  //
  // Simple setup: TDI -> Cortex-M (4-bits IRLen) -> TDO
  //
  TIF_CORESIGHT_Configure("IRPre=0;DRPre=0;IRPost=0;DRPost=0;IRLenDevice=4");
  r = JLINK_CORESIGHT_DAP_Read(TIF_CORESIGHT_DP_REG_CTRL_STAT, 0, &v);
  if (r < 0) {
    printf("Error on accessing DP-CTRL/STAT: %d\n", r);
    return -1;
  }
  printf("DP-CTRL/STAT: 0x%.8X\n", v);
  return 0;
}
```

Functions

Function	Description
TIF_CORESIGHT_Configure()	Configures the CoreSight target interface.
TIF_CORESIGHT_DAP_Read()	Reads from a CoreSight AP/DP register.
TIF_CORESIGHT_DAP_Write()	Writes to a CoreSight AP/DP register.
TIF_CORESIGHT_ReadAP()	Reads from a specific AP register.
TIF_CORESIGHT_ReadDP()	Reads froma specific DP register.
TIF_CORESIGHT_WriteAP()	Writes to a specific AP register.
TIF_CORESIGHT_WriteDP()	Writes to a specific DP register.

4.7.4.1 TIF_CORESIGHT_Configure()

Description

Accepts a configuration string used to prepare the target and Flasher for CoreSight functionality. The configuration string can include multiple setup parameters, each separated by semicolons This function outputs the appropriate target interface switching sequence for the currently active target interface, unless this behavior is disabled through a setup parameter. It must be called again whenever the JTAG chain changes (for example, in systems with dynamically changing JTAG chains such as those using a TI ICEPick) to reinitialize the JTAG chain configuration.

For JTAG

This function outputs the switching sequence from SWD to JTAG. This also triggers a TAP reset on the target (TAP controller goes through -> Reset -> Idle state). The IRPre, DRPre, IRPost, DRPost parameters describe which device inside the JTAG chainis currently selected for communicating with.

For SWD

This function outputs the switching sequence from JTAG to SWD and clears the “overrun mode enable” bit in the SW-DP CTRL/STAT register, because in SWD mode J-Link always assumes that overrun detection mode is disabled. Make sure that this bit is NOT set by accident when writing the SW-DP CTRL/STAT register via the `TIF_CORESIGHT_Write` functions.

Syntax

```
int TIF_CORESIGHT_Configure(const char* sConfig);
```

Function parameters

Parameter	Description
`sConfig`	Configuration string which consists of multiple parameters separated by semicolon (`;`) in the format `[name]=[value]`.

Configuration parameters

Parameter	Type	Description
IRPre	DecValue	Sum of IRLen of all JTAG devices in the JTAG chain, closer to TDO than the actual one Flasher shall communicate with.
DRPre	DecValue	Number of JTAG devices in the JTAG chain, closer to TDO than the actual one, Flasher shall communicate with.
IRPost	DecValue	Sum of IRLen of all JTAG devices in the JTAG chain, following the actual one, Flasher shall communicate with.
DRPost	DecValue	Number of JTAG devices in the JTAG chain, following the actual one, Flasher shall communicate with.
IRLenDevice	DecValue	IRLen of the actual device, Flasher shall communicate with.
PerformTIFInit	DecValue	0: Do not output switching sequence etc. once `TIF_CORESIGHT_Configure()` completes.

Return value

Value	Description
≥ 0	OK.
< 0	Error.
= -2	Not supported by the current CPU via selected target interface.

Example

```
//
// JTAG
// Simple setup: TDI -> Cortex-M (4-bits IRLen) -> TDO
//
TIF_Select(TIF_JTAG);
TIF_SetSpeed(1000000);        // Interface speed 1 MHz.
TIF_CORESIGHT_Configure("IRPre=0;DRPre=0;IRPost=0;DRPost=0;IRLenDevice=4");
//
// SWD
// For SWD, no special parameters are required (keep empty)
//
TIF_Select(TIF_SWD);
TIF_SetSpeed(1000000);        // Interface speed 1 MHz
TIF_CORESIGHT_Configure("");
```


4.7.4.2 TIF_CORESIGHT_DAP_Read()

Description

Reads a specific AP/DP register. For JTAG, this function ensures that AP/DP is selected automatically. Also this function will return valid data unless there is an error. It automatically performs necessary register read-accesses which usually only return data on the second access.

Syntax

```
int TIF_CORESIGHT_DAP_Read(int  RegIndex,
                           int  APnDP,
                           U32* pData);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the AP/DP register to read.
APnDP	0: DP register, 1: AP register
pData	Pointer to buffer for data read.

Return value

Value	Description
≥ 0	OK. (Number of repetitions needed before read was accepted / returned valid data).
< 0	Error.

Example

```
//
// Read AP[0], IDR (register 3, bank 15)
//
U32 Data;
TIF_CORESIGHT_DAP_Write(TIF_CORESIGHT_DP_REG_SELECT, 0, 0x000000F0);
TIF_CORESIGHT_DAP_Read(TIF_CORESIGHT_AP_REG_IDR, 1, &Data);
```


4.7.4.3 TIF_CORESIGHT_DAP_Write()

Description

Writes to a CoreSight AP/DP register. This function performs a full-qualified write which means that it tries to write until the write has been accepted or too many WAIT responses have been received.

Syntax

```
int TIF_CORESIGHT_DAP_Write(int  RegIndex,
                            int  APnDP,
                            U32* Data);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the AP/DP register to write.
APnDP	0: DP register, 1: AP register
Data	Data to be written.

Return value

Value	Description
≥ 0	OK. (Number of repetitions needed before write was accepted).
< 0	Error.
= -2	Not supported by the current CPU via selected target interface.

Example

```
//
// Write DP SELECT register: Select AP 0 bank 15
//
TIF_CORESIGHT_DAP_Write(TIF_CORESIGHT_DP_REG_SELECT, 0, 0x000000F0);
```


4.7.4.4 TIF_CORESIGHT_ReadAP()

Description

Reads a specific AP register. For JTAG, this function ensures that AP is selected automatically. Also this function will return valid data unless there is an error. It automatically performs necessary register read-accesses which usually only return data on the second access.

Syntax

```
int TIF_CORESIGHT_ReadAP(int RegIndex);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the AP register to read.

Return value

Value	Description
≥ 0	Read value from register.
< 0	Error.

4.7.4.5 TIF_CORESIGHT_ReadDP()

Description

Reads a specific DP register. For JTAG, this function ensures that DP is selected automatically. Also this function will return valid data unless there is an error. It automatically performs necessary register read-accesses which usually only return data on the second access.

Syntax

```
int TIF_CORESIGHT_ReadDP(int RegIndex);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the DP register to read.

Return value

Value	Description
≥ 0	Read value from register.
< 0	Error.

4.7.4.6 TIF_CORESIGHT_WriteAP()

Description

Writes a specific AP register. For JTAG, this function ensures that AP is selected automatically.

Syntax

```
int TIF_CORESIGHT_WriteAP(int RegIndex,
                          U32 Data);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the AP register to write.
Data	Data to be written to the register.

Return value

Value	Description
≥ 0	OK. (Number of repetitions needed before write was accepted)
< 0	Error.
= -2	Not supported by the current CPU via selected target interface.

4.7.4.7 TIF_CORESIGHT_WriteDP()

Description

Writes a specific DP register. For JTAG, this function ensures that DP is selected automatically.

Syntax

```
int TIF_CORESIGHT_WriteDP(int RegIndex,
                          U32 Data);
```

Parameters

Parameter	Description
RegIndex	Specifies the index of the DP register to write.
Data	Data to be written to the register.

Return value

Value	Description
≥ 0	OK. (Number of repetitions needed before write was accepted)
< 0	Error.
= -2	Not supported by the current CPU via selected target interface.

4.7.5 SPI interface functions

The following chapter describes the API functions for the SPI target interface.

Pin mapping

The following table shows the fixed pin mapping for the SPI target interface.

Signal	Target interface pin
SCK (serial clock)	Pin 9.
nCS (chip select)	Pin 7 or pin 17.
MOSI (Master Out Slave In)	Pin 13.
MISO (Master In Slave Out)	Pin 5.

Functions

Function	Description
`TIF_SPI_Read()`	Reads data via SPI.
`TIF_SPI_Write()`	Writes data via SPI.
`TIF_SPI_ReadWrite()`	Transfers data via SPI.

4.7.5.1 TIF_SPI_Read()

Description

Reads data from SPI.

Syntax

```
void TIF_SPI_Read(U8* pDataIn,
                  int NumBytes,
                  U32 Flags);
```

Parameters

Parameter	Description
pDataIn	Pointer to buffer that receives data.
NumBytes	Number of bytes to read.
Flags	A bit mask containing a combination of the TIF_SPI_FLAG_* definitions to configure the SPI communication.

Return value

Always returns 0.

Additional information

The line protocol is set by Flags. To configure the SPI, combine one chip select start state, one chip select end state. If chip select is used, also add a one chip select line on to the combination.

Value	Description
Chip select start state	
TIF_SPI_FLAG_CS_START_UNCHANGED	nCS not touched at start of transfer.
TIF_SPI_FLAG_CS_START_LOW	nCS = LOW at start of transfer.
TIF_SPI_FLAG_CS_START_HIGH	nCS = HIGH at start of transfer.
Chip select end state	
TIF_SPI_FLAG_CS_END_UNCHANGED	nCS not touched at end of transfer.
TIF_SPI_FLAG_CS_END_LOW	nCS = LOW at end of transfer.
TIF_SPI_FLAG_CS_END_HIGH	nCS = HIGH at end of transfer.
Chip select line	
TIF_SPI_FLAG_CS_INDEX_0	nCS0 (Pin 17).
TIF_SPI_FLAG_CS_INDEX_1	nCS1 (Pin 7).

4.7.5.2 TIF_SPI_ReadWrite()

Description

Exchanges data via SPI.

Syntax

```
int TIF_SPI_ReadWrite(const U8* pDataOut,
                      U8* pDataIn,
                      U32 NumBytes,
                      U32 Flags);
```

Parameters

Parameter	Description
pDataOut	Pointer to buffer with data to write.
pDataIn	Pointer to buffer that receives data.
NumBytes	Number of bytes to transfer.
Flags	A bit mask containing a combination of the TIF_SPI_FLAG_* definitions to configure the SPI communication.

Return value

Always returns 0.

Additional information

The line protocol is set by Flags. To configure the SPI, combine one chip select start state, one chip select end state. If chip select is used, also add a one chip select line on to the combination.

Value	Description
Chip select start state	
TIF_SPI_FLAG_CS_START_UNCHANGED	nCS not touched at start of transfer.
TIF_SPI_FLAG_CS_START_LOW	nCS = LOW at start of transfer.
TIF_SPI_FLAG_CS_START_HIGH	nCS = HIGH at start of transfer.
Chip select end state	
TIF_SPI_FLAG_CS_END_UNCHANGED	nCS not touched at end of transfer.
TIF_SPI_FLAG_CS_END_LOW	nCS = LOW at end of transfer.
TIF_SPI_FLAG_CS_END_HIGH	nCS = HIGH at end of transfer.
Chip select line	
TIF_SPI_FLAG_CS_INDEX_0	nCS0 (Pin 17).
TIF_SPI_FLAG_CS_INDEX_1	nCS1 (Pin 7).

4.7.5.3 TIF_SPI_Write()

Description

Write data via SPI.

Syntax

```
int TIF_SPI_Write(const U8* pDataOut,
                        int NumBytes,
                        U32 Flags);
```

Parameters

Parameter	Description
pDataOut	Pointer to buffer with data to write.
NumBytes	Number of bytes to write.
Flags	A bit mask containing a combination of the TIF_SPI_FLAG_* definitions to configure the SPI communication.

Return value

Always returns 0.

Additional information

The line protocol is set by Flags. To configure the SPI, combine one chip select start state, one chip select end state. If chip select is used, also add a one chip select line on to the combination.

Value	Description
Chip select start state	
TIF_SPI_FLAG_CS_START_UNCHANGED	nCS not touched at start of transfer.
TIF_SPI_FLAG_CS_START_LOW	nCS = LOW at start of transfer.
TIF_SPI_FLAG_CS_START_HIGH	nCS = HIGH at start of transfer.
Chip select end state	
TIF_SPI_FLAG_CS_END_UNCHANGED	nCS not touched at end of transfer.
TIF_SPI_FLAG_CS_END_LOW	nCS = LOW at end of transfer.
TIF_SPI_FLAG_CS_END_HIGH	nCS = HIGH at end of transfer.
Chip select line	
TIF_SPI_FLAG_CS_INDEX_0	nCS0 (Pin 17).
TIF_SPI_FLAG_CS_INDEX_1	nCS1 (Pin 7).

4.7.6 I2C interface functions

The following chapter describes the API functions for the I2C target interface.

Pin mapping

The following table shows the fixed pin mapping for the I2C target interface.

Signal	Target interface pin
SCL (serial clock)	Pin 9.
SDA (serial data)	Pin 7.

Functions

Function	Description
`TIF_I2C_Init()`	Initialize the I2C interface.
`TIF_I2C_DeInit()`	Deinitialize the I2C interface.
`TIF_I2C_TransferBytes()`	Transfer data over the I2C interface.
`TIF_I2C_WaitDeviceReady()`	Waits until the device is ready again after a started programming action.

4.7.6.1 TIF_I2C_Init()

Description

Initializes the I2C interface. The interface uses TCK (pin 9) as SCL (serial clock) and TMS (pin 7) as SDA (serial bidirectional data line).

Syntax

```
int TIF_I2C_Init(U32 Speed,
                 U8  Reserved0,
                 U32 Reserved1);
```

Parameters

Parameter	Description
Speed	The I2C clock frequency, in hertz, to be used.
Reserved0	Reserved.
Reserved1	Reserved.

Return value

Value	Description
= 0	OK.
= 1	Error.

Additional information

This function must be called after selecting the target interface using `TIF_Select()`.

4.7.6.2 TIF_I2C_DeInit()

Description

Deinitialize the I2C interface.

Syntax

```
void TIF_I2C_DeInit(void);
```


4.7.6.3 TIF_I2C_TransferBytes()

Description

Transfers data via I2C.

It writes the data from the output buffer and then reads the data which is stored in the input buffer. Data is only written or read if the number of bytes of the corresponding buffer is greater than 0. Before data is written from the buffer or read and stored in the input buffer, a start condition is signaled followed by the address. Then the specified amount of data is written or read, followed by a stop condition.

Syntax

```
int TIF_I2C_TransferBytes(      U8  Addr,
                          const U8* pDataOut,
                                U32 NumBytesOut,
                                U8* pDataIn,
                                U32 NumBytesIn);
```

Parameters

Parameter	Description
Addr	Target address within bits [7:1]. LSB (R/W bit) is ignored and set automatically.
pDataOut	Pointer to object that contains data to write. May be NULL if NumBytesOut is zero.
NumBytesOut	Number of bytes to write.
pDataIn	Pointer to object that receives the data read. May be NULL if NumBytesIn is zero.
NumBytesIn	Number of bytes to read.

Return value

Value	Description
= 0	OK.
= 1	Error.

4.7.6.4 TIF_I2C_WaitDeviceReady()

Description

Wait until the device is ready again after a started programming action.

Syntax

```
int TIF_I2C_WaitDeviceReady(U8  Addr,
                            U32 Timeout_ms);
```

Parameters

Parameter	Description
Addr	Target address.
Timeout_ms	Timeout in milliseconds.

Return value

Value	Description
= 0	OK.
= 1	Error.

4.7.7 UART interface functions

The following chapter describes the API functions for the UART target interface.

Pin mapping

The pin mapping for the UART target interface can be individually configured using the `TIF_PIN_SetFunc()` function.

Example

```
int main(void) {
  U8  aOut[2] = {0xA5, 0x5A};
  U8  aIn[2];
  int r;
  //
  // Instruct the Flasher to assume a 3.3V target reference voltage
  //
  SYS_ExecCommand("VTrefTmp=3300");
  //
  // Initialize UART
  //
  TIF_PIN_SetFunc(TIF_PIN_PIN7, TIF_PIN_FUNC_UART_TX);
  TIF_PIN_SetFunc(TIF_PIN_PIN9, TIF_PIN_FUNC_UART_RX);
  TIF_UART_Init(115200);  // 8 data bits, one stop bit, and no parity
  //
  // Write data
  //
  TIF_UART_Write(&aOut[0], sizeof(aOut));
  //
  // Wait some time so the firmware can buffer the incoming data
  //
  SYS_Sleep_ms(10);
  //
  // Read data
  //
  r = TIF_UART_Read(&aIn[0], sizeof(aIn));
  if (r < (int)sizeof(aIn)) {
    printf("Timeout while receiving data.\n");
    return -1;
  }
  if (memcmp(aIn, aOut, sizeof(aIn))) {
    printf("Data error - received 0x%X, 0x%X\n", aIn[0], aIn[1]);
    return -1;
  }
  printf("Data sent and received successfully!\n");
  return 0;
}
```

Functions

Function	Description
`TIF_UART_Init()`	Initialize the UART interface.
`TIF_UART_InitEx()`	Initialize the UART interface, extended.
`TIF_UART_DeInit()`	Deinitialize the UART interface.
`TIF_UART_Read()`	Read data from the UART receive buffer.
`TIF_UART_Write()`	Write data to the UART transmit buffer.

4.7.7.1 TIF_UART_Init()

Description

Initialize the UART interface. The line protocol is fixed to eight data bits, no parity, one stop bit.

Syntax

```
int TIF_UART_Init(int BaudRate);
```

Parameters

Parameter	Description
BaudRate	The baud rate, in bits per second, to be used.

Return value

Always returns 0.

Additional information

Before calling `TIF_UART_Init()`, the modes of the pins to be used must be set using `TIF_PIN_SetFunc()`.

> **Note**
>
> When the UART is used as a target interface, `TIF_Select()`, and `TIF_SetSpeed()` must not be used.

4.7.7.2 TIF_UART_InitEx()

Description

Initialize the UART interface, extended.

Syntax

```
int TIF_UART_InitEx(U32 BaudRate,
                    U32 ModeFlags,
                    U32 BufSize);
```

Parameters

Parameter	Description
BaudRate	The baud rate to be used.
ModeFlags	A bit mask containing a combination of the TIF_UART_MODE_FLAG_* constants to configure the UART.
BufSize	The size of the receive buffer to allocate, in bytes (max. 10 KB).

Return value

Value	Description
≥ 0	OK.
< 0	Error.
= -2	Error, maximum size of the Rx buffer exceeded.
= -3	Error, insufficient memory.

Additional information

Before calling TIF_UART_InitEx(), the modes of the pins to be used must be set using TIF_PIN_SetFunc().

The line protocol is set by ModeFlags. To select a mode, combine one line discipline selection, one parity bit selection, and one stop bit selection from the table below using the standard bit-or operator "|":

Flag	Description
Line discipline	
TIF_UART_MODE_FLAG_FULL_DUPLEX	Full-duplex mode, Rx and Tx separate.
TIF_UART_MODE_FLAG_HALF_DUPLEX	Half-duplex mode, Rx and Tx combined.
Parity bit	
TIF_UART_MODE_FLAG_PARITY_NONE	No parity bit.
TIF_UART_MODE_FLAG_PARITY_EVEN	Even parity bit.
TIF_UART_MODE_FLAG_PARITY_ODD	Odd parity bit.
Stop bits	
TIF_UART_MODE_FLAG_1_STOP_BIT	One stop bit.
TIF_UART_MODE_FLAG_2_STOP_BIT	Two stop bits.

See also

TIF_PIN_SetFunc() on page 117

4.7.7.3 TIF_UART_DeInit()

Description

Deinitialize the UART interface.

Syntax

```
int TIF_UART_DeInit(void);
```

Return value

Always returns 0.

4.7.7.4 TIF_UART_Read()

Description

Read data from UART.

Syntax

```
int TIF_UART_Read(U8* pData,
                  U32 NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to object that receives the data read.
NumBytes	Number of bytes to read.

Return value

The number of bytes read from the receive buffer which will be no more than NumBytes.

4.7.7.5 TIF_UART_Write()

Description

Write data UART.

The data written are transmitted asynchronously over the UART interface. If the transmit buffer is not full and can accept all data, this function returns immediately. If the transmit buffer cannot accept all data, this function will block until there is sufficient free space in the transmit buffer to store the remaining data.

Syntax

```
int TIF_UART_Write(const U8* pData,
                   U32 NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to a buffer with data to be written.
NumBytes	Number of bytes to write.

Return value

The number of bytes written into the transmit buffer.

4.8 Speedy functions

Function	Description
SPEEDY_Start()	Starts the specified Speedy core by releasing it from reset.
SPEEDY_Stop()	Stops the specified Speedy core by asserting its reset signal.
SPEEDY_SetFreq()	Sets the desired frequency to be used by the Speedy.
SPEEDY_TransferDCC()	Transfers data to and from the specified Speedy core.
SPEEDY_WriteProg()	Loads a Speedy program into the specified Speedy core.
SPEEDY_ReadDCCNB()	Reads data from the Speedy core in a non-blocking manner.

4.8.1 SPEEDY_Start()

Description

Starts the specified Speedy core by releasing it from reset.

Syntax

```
void SPEEDY_Start(unsigned int CoreIndex);
```

Parameters

Parameter	Description
`CoreIndex`	The index of the Speedy core to be used. 0: Speedy core 0. 1: Speedy core 1.

Additional information

The Speedy core starts execution at address `0x0` if it is released from reset.

4.8.2 SPEEDY_Stop()

Description

Stops the specified Speedy core by asserting its reset signal.

Syntax

```
void SPEEDY_Stop(unsigned int CoreIndex);
```

Parameters

Parameter	Description
CoreIndex	The index of the Speedy core to be used. 0: Speedy core 0. 1: Speedy core 1.

4.8.3 SPEEDY_SetFreq()

Description

Sets the desired frequency to be used by the Speedy. Both Speedy cores share their clock source.

Syntax

```
U32 SPEEDY_SetFreq(U32 Frequency);
```

Parameters

Parameter	Description
Frequency	The frequency, in hertz, to be used by the Speedy core.

Return value

Returns the frequency that was set.

Additional information

If the specified frequency cannot be set, the next-lower frequency is used.

4.8.4 SPEEDY_TransferDCC()

Description

Transfers data to and from the specified Speedy core.

Syntax

```
void SPEEDY_TransferDCC(      unsigned int CoreIndex,
                        const void*        pDataDown,
                              U32          NumBytesDown,
                              void*        pDataUp,
                              U32          NumBytesUp);
```

Parameters

Parameter	Description
CoreIndex	The index of the Speedy core to be used. 0: Speedy core 0. 1: Speedy core 1.
pDataDown	A pointer to a buffer with the data to write. Can be NULL if NumBytesDown is 0.
NumBytesDown	The number of bytes to write. Set to 0 if no data is to be sent.
pDataUp	A pointer to a buffer used to store the read data. Can be NULL if NumBytesUp is 0.
NumBytesUp	The number of bytes to read. Set to 0 if no data is to be read.

Additional information

This function blocks execution until the specified amount of bytes are read and written.

4.8.5 SPEEDY_WriteProg()

Description

Loads a Speedy program into the specified Speedy core.

Syntax

```
int SPEEDY_WriteProg(      unsigned int CoreIndex,
                     const U16*         paInst,
                           unsigned int NumInst);
```

Parameters

Parameter	Description
CoreIndex	The index of the Speedy core to be used. 0: Speedy core 0. 1: Speedy core 1.
paInst	A pointer to a buffer containing the instructions of the Speedy program.
NumInst	The number of 16-bit instructions in the buffer. The instruction count may not exceed the maximum program size of the Speedy cores. Core 0 supports up to 512 instructions and core 1 supports 256 instructions.

Return value

Value	Description
≥ 0	OK.
< 0	Error.
= -1	Unspecified error.
= -2	Program image too big.
= -3	Invalid core index.

Additional information

Core must be stopped before loading a program into it.

4.8.6 SPEEDY_ReadDCCNB()

Description

Reads data from the Speedy core in a non-blocking manner.

Syntax

```
U32 SPEEDY_ReadDCCNB(unsigned int CoreIndex,
                     void*        pDataUp,
                     U32          MaxNumBytesUp);
```

Parameters

Parameter	Description
CoreIndex	The index of the Speedy core to be used. 0: Speedy core 0. 1: Speedy core 1.
pDataUp	A pointer to a buffer used to store the read data.
MaxNumBytesUp	The maximum number of bytes to read.

Return value

Returns the number of bytes read.

4.9 Language constructs

This chapter describes language constructs as pseudo-variables, image variables, type definitions and macros which can be used with Flasher Apps.

4.9.1 Pseudo-variables

Pseudo-variables are syntactic sugar for accessing special variables made available by the Flasher. This data can then be read or written just like a normal variable.

> **Note**
>
> It is not possible to read or write to to pseudo-variables through a pointer or reference it with the & operator.

The following pseudo-variables are provided by the Flasher:

Index	Declaration	Read/ Write	Description
1	`U32 TIF_JTAG_IRPre`	rw	Sets or returns the sum of JTAG IRLen of all TAPs preceding the device.
2	`U32 TIF_JTAG_DRPre`	rw	Sets or returns the number of JTAG TAPs preceding the device.
3	`U32 TIF_JTAG_IRPost`	rw	Sets or returns the sum of the JTAG IRLen of all TAPs following the device.
4	`U32 TIF_JTAG_DRPost`	rw	Sets or returns the number of JTAG TAPs following the device.
5	`U32 TIF_JTAG_IRLen`	rw	Sets or returns the JTAG IRLen of the device.
6	`U32 TIF_JTAG_TotalIRLen`	r	Returns the currently set total IR length of the JTAG chain.
8	`U32 TIF_Speed`	rw	Sets or returns the target interface speed in kHz.
9	`U32 TIF_JTAG_ResetPin`	rw	Sets or clears pin 15.
10	`U32 TIF_JTAG_TRSTPin`	rw	Sets or clears pin 3.
11	`U32 TIF_JTAG_TCKPin`	rw	Sets or clears pin 9.
12	`U32 TIF_JTAG_TMSPin`	rw	Sets or clears pin 7.
13	`U32 TIF_JTAG_TDIPin`	rw	Sets or clears pin 5.
19	`U32 TIF_CORESIGHT_CoreBaseAddr`	rw	Sets the core base address to for the CoreSight components.
22	`U32 SYS_ActiveTIF`	r	Returns the currently active target interface.
23	`U32 TIF_CORESIGHT_IndexAHBAP-ToUse`	rw	Sets the index of the AHB-AP to use when connecting to a device.
24	`U32 TIF_CORESIGHT_IndexAPBAP-ToUse`	rw	Sets the index of the APB-AP to use when connecting to a device.
33	`U32 SYS_TargetVoltage`	r	Returns the voltage on the line connected to VTREF.

4.9.2 Image variables

Image variables provide access to specific memory regions of the Flasher which are known at compile time.

> **Note**
>
> It is not possible to write to image variables through a pointer or reference it with the `&` operator.

The following image variables are available in apps for the Flasher:

Declaration	Description
`U8* __S32_XDataBase`	Base address of the XData area.
`U8* __S32_XDataLimit`	End address of the XData area (XData base address + XData size).

4.9.3 Type definitions and macros

The following type definitions and macros can be used by apps:

Macro	Description
Type definitions	
U8	Shorthand for an unsigned 8-bit type.
U16	Shorthand for an unsigned 16-bit type.
U32	Shorthand for an unsigned 32-bit type.
I8	Shorthand for a signed 8-bit type.
I16	Shorthand for a signed 16-bit type.
I32	Shorthand for a signed 32-bit type.
Macros	
LOW	Defined as ’0’. Can be use with TIF_JTAG_*Pin pseudo-variables to set the pin to LOW.
HIGH	Defined as ’1’. Can be use with TIF_JTAG_*Pin pseudo-variables to set the pin to HIGH.
FALSE	Defined as ’0’. Can be used in boolean expressions.
TRUE	Defined as ’1’. Can be used in boolean expressions.
NULL	Null pointer constant.

4.10 Programming app

4.10.1 Programming app functions

Function	Description
SYS_LoadUNI()	Loads a configuration for testing.
SYS_CFG_GetU32()	Reads a U32 value of a key from the configuration file.
SYS_CFG_GetData()	Reads data of a key from the configuration file.
SYS_CFG_GetSubindexCnt()	Returns the number of indices available for a key in the configuration file.
SYS_CFG_GetIndexedU32()	Reads an indexed U32 value of a key from the configuration file.
SYS_CFG_GetIndexedString()	Reads an indexed string of a key from the configuration file.
SYS_CFG_GetFlashBankU32()	Reads an U32 value from a flash bank information from the configuration file.
SYS_CFG_GetFlashBankString()	Reads a string from a flash bank information from the configuration file.

4.10.1.1 SYS_LoadUNI()

Description

Loads a configuration file from the Flasher's file system. It can be used for testing programming apps with the Flasher App Builder Utility.

Syntax

```
int SYS_LoadUNI(void);
```

Return value

Value	Description
≥ 0	OK.
< 0	Error, failed to load configuration file.

Additional information

> **Note**
>
> The configuration file must be called FLASHER.uni and be stored in the root directory of the Flasher file system.

Example

```
int main(void) {
  SYS_LoadUNI();  // Load configuration file.
  return 0;
}
```


4.10.1.2 SYS_CFG_GetU32()

Description

Reads a U32 value of a key from the configuration file.

Syntax

```
int SYS_CFG_GetU32(const char* sKey,
                   U32*  pValue);
```

Parameters

Parameter	Description
sKey	Pointer to the buffer containing the name of the key.
pValue	Pointer to the destination buffer.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

> **Note**
>
> The key comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<Config Display="TestGroup">
  <Group>
    <Config Name="CheckBoxConfig" Display="Test Checkbox" Value="0"
 Hint="Checkbox sample.">
      <Edit Style="CheckBox"/>
    </Config>
  </Group>
</Config>
[...]
```

App source code:

```
[...]
U32 v;
int r;

r = SYS_CFG_GetU32("CheckBoxConfig", &v);
[...]
```


4.10.1.3 SYS_CFG_GetData()

Description

Reads data of a key from the configuration file.

Syntax

```
int SYS_CFG_GetData(const char* sKey,
                    U32*  pValue);
```

Parameters

Parameter	Description
sKey	Pointer to the buffer containing the name of the key.
pValue	Pointer to the destination buffer.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

> **Note**
>
> The key comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<Config Display="TestGroup">
  <Group>
    <Config Name="FileConfig" Display="Test File" Value="" Hint="File sample.">
      <Edit Style="File" Filter="*.*"/>
    </Config>
  </Group>
</Config>
[...]
```

App soure code:

```
[...]
char acFileName[32];
int r;

r = SYS_CFG_GetData("FileConfig", &acFileName[0], sizeof(acFileName));
[...]
```


4.10.1.4 SYS_CFG_GetSubindexCnt()

Description

Retrieves the count for an indexed value from the configuration file.

Syntax

```
int SYS_CFG_GetSubindexCnt(const char* sKey,
                           U32*  pDest);
```

Parameters

Parameter	Description
sKey	PoName of key to retrieve value from.
pDest	Pointer to object that receives the subindex count.

Return value

Value	Description
≥ 0	OK.
< 0	Error.

Additional information

> **Note**
>
> The key comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

For an example usage of `SYS_CFG_GetSubindexCnt()` refer to `SYS_CFG_GetIndexedU32()` or `SYS_CFG_GetIndexedString()`.

4.10.1.5 SYS_CFG_GetIndexedU32()

Description

Retrieve an indexed U32 value from the configuration file.

Syntax

```
int SYS_CFG_GetIndexedU32(const char* sKey,
                                U32   Index,
                                U32*  pDest);
```

Parameters

Parameter	Description
sKey	Name of key to retrieve value from.
Index	Key index to retrieve value from.
pDest	Pointer to object that receives the retrieved value.

Return value

Value	Description
≥ 0	OK.
< 0	Error.

Additional information

> **Note**
>
> The key comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<Config Display="TestGroup">
  <Group>
    <Config Name="TableTest" Display="Table Test" Hint="Table sample.">
      <Table StartIdx="0" EndIdx="5">
        <Config Name="DrowDownConfig" Display="DropDown Config" Value="0xFF"
 Hint="DropDown sample.">
          <Edit Style="DropDown">
            <Option Name="Entry 1" Value="0x0"/>
            <Option Name="Entry 2" Value="0x1"/>
            <Option Name="Entry 3" Value="0xFF"/>
          </Edit>
        </Config>
      </Table>
    </Config>
  </Group>
</Config>
[...]
```


App soure code:

```
int main(void) {
  U32 NumEntries;
  U32 v;
  U32 i;
  int r;

  r = SYS_CFG_GetSubindexCnt("DrowDownConfig", &NumEntries);
  if (r != 0) {
    return -1;
  }
  i = 0;
  while (NumEntries) {
    r = SYS_CFG_GetIndexedU32("DrowDownConfig", i, &v);
    if (r != 0) {
      return -1;
    }
    printf("Selection %u: '%u'.\n", i, v);
    NumEntries--;
    i++;
  };
  return 0;
}
```


4.10.1.6 SYS_CFG_GetIndexedString()

Description

Retrieve an indexed string from the configuration file.

Syntax

```
int SYS_CFG_GetIndexedString(const char* sKey,
                                    U32   Index,
                                    char* pDest,
                                    U32   DestLen);
```

Parameters

Parameter	Description
sKey	Name of key to retrieve value from.
Index	Key index to retrieve value from.
pDest	Pointer to object that receives the retrieved value.
DestLen	Capacity of object that receives the retrieved string, in characters.

Return value

Value	Description
≥ 0	OK.
< 0	Error.

Additional information

> **Note**
>
> The key comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<Config Display="TestGroup">
  <Group>
    <Config Name="TableTest" Display="Table Test" Hint="Table sample.">
      <Table StartIdx="0" EndIdx="5">
        <Config Name="FileConfig" Display="Test File" Value="" Hint="File
 sample.">
          <Edit Style="File" Filter="*.*"/>
        </Config>
      </Table>
    </Config>
  </Group>
</Config>
[...]
```


App soure code:

```
int main(void) {
  char acFileName[32];
  U32 NumEntries;
  U32 i;
  int r;

  r = SYS_CFG_GetSubindexCnt("DrowDownConfig", &NumEntries);
  if (r != 0) {
    return -1;
  }
  i = 0;
  while (NumEntries) {

  r = SYS_CFG_GetIndexedString("FileConfig", i, &acFileName[0], sizeof(acFileName));
    if (r != 0) {
      return -1;
    }
    printf("Filename of file %u: '%s'.\n", i, acFileName);
    NumEntries--;
    i++;
  };
  return 0;
}
```


4.10.1.7 SYS_CFG_GetFlashBankU32()

Description

Retrieves a U32 value (specified by key name) for a flash bank (specified by name).

Syntax

```
int SYS_CFG_GetFlashBankU32(const char* sFlashBank,
                            const char* sItem,
                                  U32*  pDest);
```

Parameters

Parameter	Description
sFlashBank	Pointer to object that contains the name of flash bank.
sItem	Pointer to object that contains the key to retrieve value for.
pDest	Pointer to object that receives the retrieved value.

Return value

Value	Description
= 0	OK.
< 0	Error.

Additional information

> **Note**
>
> The flash bank and item comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<FlashBankInfo Name="Flash Bank 1" BaseAddr="0x00000000" Size="0x00010000"
 MinWriteSize="0x20" Loader="Device.pex" RAMCode="RAMCode.bin">
  <SectorGroup Count="256" Size="0x100"/>
</FlashBankInfo>
[...]
```

App soure code:

```
[...]
U32 v;
int r;

r = SYS_CFG_GetFlashBankU32("Flash Bank 1", "MinWriteSize", &v);
[...]
```


4.10.1.8 SYS_CFG_GetFlashBankString()

Description

Retrieves a string value (specified by key name) for a flash bank (specified by name).

Syntax

```
int SYS_CFG_GetFlashBankString(const char* sFlashBank,
                               const char* sItem,
                                     char* pDest,
                                     U32   DestLen);
```

Parameters

Parameter	Description
sFlashBank	Pointer to object that contains the name of flash bank.
sItem	Pointer to object that contains the key to retrieve value for.
pDest	Pointer to object to store the retrieved string.
DestLen	Capacity of object that receives the retrieved string, in characters.

Return value

Value	Description
= 0	OK.
< 0	Error.

Additional information

> **Note**
>
> The flash bank and item comparison is not case-sensitive. However, it is recommended to use the exact name without capitalization differences.

Example

Device Definition File:

```
[...]
<FlashBankInfo Name="Flash Bank 1" BaseAddr="0x00000000" Size="0x00010000"
 MinWriteSize="0x20" Loader="Device.pex" RAMCode="RAMCode.bin">
  <SectorGroup Count="256" Size="0x100"/>
</FlashBankInfo>
[...]
```

App soure code:

```
[...]
char acRAMCodeFileName[32];
int r;

r = SYS_CFG_GetFlashBankString("Flash Bank
 1", "RAMCode", &acRAMCodeFileName[0], sizeof(acRAMCodeFileName));
[...]
```


4.10.2 Programming app entry points

Function	Description
UNI_Init()	Performs all preparations required by subsequent function calls.
UNI_DeInit()	Always called as the last function to perform deinitialization.
UNI_CheckBlank()	Determines whether the specified sectors are blank or not.
UNI_EraseSectors()	Erases the specified sectors.
UNI_EraseChip()	Performs a bulk/mass/chip erase.
UNI_Program()	Programs the specified memory section.
UNI_Verify()	Verifies the specified memory section.
UNI_Read()	Reads from the specified memory section.
UNI_PrepareOperation()	Performs all preparations required by subsequent calls for the respective operation
UNI_RestoreOperation()	Performs restoration after all calls for the respective operation
UNI_PrepareFlashBank()	Performs all preparations required by subsequent calls for the respective operation and flashbank.
UNI_RestoreFlashBank()	Performs restoration after all calls for the respective operation and flash bank are done.
UNI_GetFlags()	Provides flags at runtime to the Flasher to request non-standard, special behavior.
UNI_Secure()	Secures the target device.

4.10.2.1 UNI_Init()

Description

Perform all preparations required by subsequent calls of programming app entry point functions.

Syntax

```
int UNI_Init(void);
```

Return value

Value	Description
= 0	OK.
≠ 0	Error.

4.10.2.2 UNI_DeInit()

Description

Called (always) as the last function to perform deinitialization. It must ensure that the communication with the target is properly terminated after the execution of the programming app.

Syntax

```
int UNI_DeInit(void);
```

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

Unlike the other pins, the reset pin retains its state after the programming app has finished execution. Use this function to control whether the target device stays in reset or not by setting the final state of the reset pin, e.g. with `TIF_ReleaseReset()` or `TIF_ActivateReset()`.

> **Note**
>
> `UNI_DeInit()` is also called if a previous entry point function returned an error, aborting the programming process.

4.10.2.3 UNI_CheckBlank()

Description

Determines whether sectors are blank.

Syntax

```
int UNI_CheckBlank(U32 SectorAddr,
                   U32 SectorIndex,
                   U32 NumSectors,
                   U32 SectorSize);
```

Parameters

Parameter	Description
SectorAddr	Start address of the first sector to check.
SectorIndex	Index of the first sector to check.
NumSectors	Number of consecutive sectors to check.
SectorSize	Size of each sector to check.

Return value

Value	Description
< 0	Error.
= 0	Sectors are blank.
= 1	At least one sector is not blank.
= (2 + x)	Specifies the first sector that is not blank with x being the index of the sector.

Additional information

This function is called depending on the available erase types specified in the DDF as well as the active configuration stored in the configuration file.

The following table shows the available erase types used by the `@EraseType` and `@Default-Erase` attributes and the entry points they depend on.

Erase type	Entry points
None	
Bulk	`UNI_EraseChip()`
Sector	`UNI_EraseSectors()`
Dirty	`UNI_CheckBlank()`, `UNI_EraseSectors()`

4.10.2.4 UNI_EraseSectors()

Description

Erase sectors.

Syntax

```
int UNI_EraseSectors(U32 SectorAddr,
                     U32 SectorIndex,
                     U32 NumSectors,
                     U32 SectorSize);
```

Parameters

Parameter	Description
SectorAddr	Start address of the first sector to erase.
SectorIndex	Index of the first sector to erase.
NumSectors	Number of consecutive sectors to erase.
SectorSize	Size of each sector to erase.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

This function is called depending on the available erase types specified in the DDF as well as the active configuration stored in the configuration file. For more information refer to *UNI_CheckBlank()* on page 196.

4.10.2.5 UNI_EraseChip()

Description

Bulk-erase chip. Also known as “mass erase” or “chip erase”.

Syntax

```
int UNI_EraseChip(void);
```

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

This function is called depending on the available erase types specified in the DDF as well as the active configuration stored in the configuration file. For more information refer to *UNI_CheckBlank()* on page 196.

4.10.2.6 UNI_Program()

Description

Program data into memory.

Syntax

```
int UNI_Program(U32 Addr,
                U32 NumBytes);
```

Parameters

Parameter	Description
Addr	Address of the memory section to program.
NumBytes	Size of the memory section to program.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

The data to program is provided via the __S32_XDataBase image variable. The Flasher firmware ensures the following:

- Addr is valid
- NumBytes is greater 0
- NumBytes is a multiple of the @MinWriteSize specified in the DDF, and maximal xdata_size

Example

```
int UNI_Program(U32 Addr, U32 NumBytes) {
  U8* p;
  int r;

  p = __S32_XDataBase;
  do {
    r = _SendByte(*p++);
    if (r < 0) {
      return -1;
    }
  } while (--NumBytes);
  return 0;
}
```

See also

Image variables on page 180 *Pragmas* on page 298

4.10.2.7 UNI_Verify()

Description

Verify programmed memory against intended content.

Syntax

```
int UNI_Verify(U32 Addr,
               U32 NumBytes);
```

Parameters

Parameter	Description
Addr	Address of the memory section to program.
NumBytes	Size of the memory section to verify.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

The verify data is provided via the __S32_XDataBase image variable. The Flasher firmware ensures the following:

- Addr is valid
- NumBytes is greater 0
- NumBytes is a multiple of the @MinWriteSize specified in the DDF, and maximal xdata_size

Example

```
int UNI_Verify(U32 Addr, U32 NumBytes) {
  U8* p;
  U8 Byte;
  int r;

  p = __S32_XDataBase;
  do {
    r = _ReadByte(&Byte);
    if (r < 0) {
      return -1;
    }
    if (Byte != *p++) {
      return -1;
    }
  } while (--NumBytes);
  return 0;
}
```

See also

Image variables on page 180 *Pragmas* on page 298

4.10.2.8 UNI_Read()

Description

Read memory.

Data from the specified memory section is read back and stored to the extended data area provided by the Flasher.

Syntax

```
int UNI_Read(U32 Addr,
             U32 NumBytes);
```

Parameters

Parameter	Description
Addr	Address of the memory section to read.
NumBytes	Size of the memory section to read.

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

The verify data is provided via the __S32_XDataBase image variable. The Flasher firmware ensures the following:

- Addr is valid
- NumBytes is greater 0
- NumBytes is a multiple of the @MinWriteSize specified in the DDF, and maximal xdata_size

Example

```
int UNI_Read(U32 Addr, U32 NumBytes) {
  U8* p;
  int r;

  p = __S32_XDataBase;
  do {
    r = _ReadByte(p++);
    if (r < 0) {
      return -1;
    }
  } while (--NumBytes);
  return 0;
}
```

See also

Image variables on page 180 *Pragmas* on page 298

4.10.2.9 UNI_PrepareOperation()

Description

Prepare for following operation.

This function performs all preparations required by subsequent calls for the respective operation (e.g. Erase, Program, Verify, …). This function is optional.

Syntax

```
int UNI_PrepareOperation(UNI_OPERATION_TYPE Operation);
```

Parameters

Parameter	Description
`Operation`	Operation of type `UNI_OPERATION_TYPE`

Return value

Value	Description
= 0	OK.
≠ 0	Error.

> **Note**
>
> This function is optional. If defined `UNI_RestoreOperation()` must be defined as well.

4.10.2.10 UNI_RestoreOperation()

Description

Restore state after operation.

This function performs restoration after all calls for the respective operation (e.g. Erase, Program, Verify, …) are done.

Syntax

```
int UNI_RestoreOperation(UNI_OPERATION_TYPE Operation);
```

Parameters

Parameter	Description
`Operation`	Operation of type `UNI_OPERATION_TYPE`

Return value

Value	Description
= 0	OK.
≠ 0	Error.

> **Note**
>
> This function is optional. If defined `UNI_PrepareOperation()` must be defined as well.

4.10.2.11 UNI_PrepareFlashBank()

Description

Prepare for following operation on a flash bank.

This function performs all preparations required by subsequent calls for the respective operation (e.g. Erase, Program, Verify, …) and flash bank.

Syntax

```
int UNI_PrepareFlashBank(const char*              sBankName,
                         U32                      BankBaseAddr,
                         UNI_OPERATION_TYPE       Operation);
```

Parameters

Parameter	Description
sBankName	Name of the flash bank
BankBaseAddr	Base address of the flash bank
Operation	Operation of type UNI_OPERATION_TYPE

Return value

Value	Description
= 0	OK.
≠ 0	Error.

> **Note**
>
> This function is optional. If defined UNI_RestoreFlashBank() must be defined as well.

4.10.2.12 UNI_RestoreFlashBank()

Description

Restore state after operation on flash bank.

This function performs restoration after all calls for the respective operation (e.g. Erase, Program, Verify, …) and flash bank are done.

Syntax

```
int UNI_RestoreFlashBank(const char*              sBankName,
                         U32                      BankBaseAddr,
                         UNI_OPERATION_TYPE       Operation);
```

Parameters

Parameter	Description
sBankName	Name of the flash bank
BankBaseAddr	Base address of the flash bank
Operation	Operation of type UNI_OPERATION_TYPE

Return value

Value	Description
= 0	OK.
≠ 0	Error.

> **Note**
>
> This function is optional. If defined UNI_PrepareFlashBank() must be defined as well.

4.10.2.13 UNI_GetFlags()

Description

Return special-behavior-request flags.

This function provides flags, at runtime, to the Flasher in order to request non-standard, specialized behavior. The flags are stored as 32-bit blocks in the XData area (`__S32_XDataBase`), similar to `UNI_Read()`.

Syntax

```
int UNI_GetFlags(void);
```

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Additional information

The following table describes the available flags which cane be provided.

Value	Description
`FLASHER_PCODE_FLAG_BITPOS_PROG_DO_NOT_CROSS_SECTOR_BOUNDARIES`	`UNI_Program()` and `UNI_Verify()` will be called only for memory sections that do not cross sector boundaries.

Note

The available flags describe bit positions and not bit masks.

Note

This function is optional. In most cases it is not required and can be omitted.

4.10.2.14 UNI_Secure()

Description

Secure a device.

This function is called when the command #SECURE is sent to the Flasher and is supposed to secure the target device. This function is deprecated due to difficulties with some devices to secure the device after it already has been fully programmed and given the fact that there are usually multiple ways to secure a device.

> **Note**
>
> Implementing this function is not recommended because securing a device can be accomplished in many ways and the outcome is not clear. Instead, the provided data file should determine if a device will be secured. Alternatively, the DDF can contain configurations that can be activated and modified to overwrite e.g. fuses or configuration words that would otherwise originate from the data file to flash.

Syntax

```
int UNI_Secure(void);
```

Return value

Value	Description
= 0	OK.
≠ 0	Error.

Chapter 5

DDF reference

This chapter is a reference for all DDF elements.

5.1 Elements

Each subsection defines the elements that can be used as child elements for the element referenced in the section's title.

The following keys are used to denote options and multiplicity:

no keys
Elements without any keys are mandatory, they must exactly once, in exact that order.

elemA, elemB
The comma(**,**) denotes a sequence. The *elemA* must be followed by an *elemB*.

(...)
This denotes a grouping. All definitions and keys apply to the content inside that group.

elemA | elemB
The pipe (**|**) denotes a choice. Either *elemA* or *elemB* must appear here.

elem?
The question mark(**?**) denotes an option. The *elem* may appear here.

elem*
The asterisk (*****) denotes an zero-or-more option. The *elem* can appear here as often as required.

elem+
The plus sign (**+**) denotes an one-or-more option. The *elem* must appear here at least once.

elem{min, max}
The braces (**{...}**) denotes an min-to-max option. The *elem* must appear here at least *min* times, but at most *max* times.

DDFs have `<DataBase>` as the root element. Within `<DataBase>`, the `<Device>` element is used to define one or multiple devices for a specific target interface and programming app.

A minimal skeleton DDF for an example device might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DataBase>
  <Device>
    <ChipInfo Vendor="Manufacturer" Family="Family" Core="Core" Name="Device"
 Interface="Interface"/>
    <DefaultTasks Fixed="0" EraseType="None Sector Dirty Bulk"
 DefaultErase="Bulk" DefaultProgram="1" DefaultRead="-1" DefaultVerify="1"
 DefaultSecure="0"/>
    <FlashBankInfo Name="Flash Bank 1" BaseAddr="0x00000000" Size="0x00010000"
 MinWriteSize="0x20" Loader="DEVICE.PEX">
      <SectorGroup Count="256" Size="0x100"/>
    </FlashBankInfo>
  </Device>
</DataBase>
```


5.1.1 Addon

Description

Declares additional files needed by a programming app.

Child of

`<Device>`

Permitted children

`<File>+`

Permitted attributes

None.

5.1.2 Alternative

Description

Defines a device derivation.

Child of

`<Alternatives>`

Permitted children

`<Config>` (Alternative)*

Permitted attributes

`@Name` (Device)
`@MemoryMap`?

Additional information

It is possible to create a single `<Device>` entry that supports multiple devices that are using the same programming app and configurations, but whose configuration values may differ. This is done by using `<Alternatives>` which allows to add an `<Alternative>` child element for each additionally supported device. The name of the additional devices need to be specified by the `@Name` (Device) attribute of `<Alternative>`. If no children are added to `<Alternative>`, the alternative device is an exact copy of the original device specified in `<ChipInfo>`, but with another `@Name` (Device). Existing configurations can be overridden for alternative devices by adding `<Config>` child elements to `<Alternative>` which are using the same configuration name as the one to override. The `@Value`, `@Hide` and `@Hint` attributes of the `<Config>` in `<Alternative>` can then be set as desired.

5.1.3 Alternatives

Description

Optional element to specify alternative names (and minor differences) for the defined device.

Child of

`<Device>`

Permitted children

`<Alternative>+`

Permitted attributes

None.

See also

Alternative on page

5.1.4 Bitfield

Description

Creates a configuration value as a view of with named bit(-groups).

Child of

`<Config>` (Device)

Permitted children

`<BitGroup>+`

Permitted attributes

`@Size` (Bitfield)

5.1.5 BitGroup

Description

Creates a named bit group inside a `<Bitfield>`.

> **Note**
>
> `<Edit>` children must be one of type CheckBox, DropDown, Hex or Binary.

Child of

`<Bitfield>`

Permitted children

`<Edit>`

Permitted attributes

`@Name` (Config)
`@Pos`
`@Value`
`@Controller` (Config)?
`@Hide`?
`@Hint`?

5.1.6 c

Description

Defines the default values for named configuration value for a row of the `<Edit>` element with `@Style`="Table".

Child of

`<row>`

Permitted children

text node

Permitted attributes

`@N`

5.1.7 ChipInfo

Description

Defines meta-information for a device.

Child of

`<Device>`

Permitted children

None.

Permitted attributes

`@Vendor`
`@Core`
`@Family`
`@Name` (Device)
`@Interface`
`@Descr?`
`@OptDatFile?`

5.1.8 Config (Alternative)

Description

Defines or Redefines a constant value.

Child of

`<Alternative>`

Permitted children

None.

Permitted attributes

`@Name` (Config)
`@Value`
`@Hide?`
`@Hint?`

5.1.9 Config (Device)

Description

Defines a configuration with its default configuration value for the programming app as well as its input field format used by U-Flash. Configuration values can be modified in U-Flash if not marked as “hidden”. When a Flasher project is created by U-Flash, the configuration values are stored in the UNI file.

Child of

`<Device>`

Permitted children

(`<Edit>` | `<Group>` | `<Table>` | `<Bitfield>` | `<Range>`)?

Permitted attributes

`@Name` (Config)
`@Display`?
`@Value`
`@Controller` (Config)?
`@Hide`?
`@Hint`?

5.1.10 DataBase

Description

Root element.

Child of

None.

Children

`<Device>` | <DevicesFile>

Attributes

None.

5.1.11 DefaultTasks

Description

Defines the supported erase types, whether #READ is supported and the default tasks performed by #AUTO.

Child of

<Device>

Permitted children

None.

Permitted attributes

```
@Fixed?
@EraseType?
@DefaultErase?
@DefaultProgram?
@DefaultRead?
@DefaultVerify?
@DefaultSecure?
```


5.1.12 Device

Description

Encloses device definition.

Child of

`<DataBase>`

Permitted children

`<Addon>?`
`<Alternatives>?`
`<ChipInfo>`
`<Config>` (Device)*
`<DefaultTasks>?`
`<FlashBankInfo>*,`
`<Information>*`
`<MemoryMap>*`

Permitted attributes

`@Version?`

5.1.13 Edit

Description

Defines an editable value.

Child of

`<Config>` (Device)

Permitted children

Depends on attribute `@Style`.

Permitted attributes

`@Style`; presence of other attributes depends on this.

Style	Description	Permitted children	Permitted attributes
DropDown	Defines a value with drop-down input	@Option	Style='DropDown'
Hex	Defines a value with hexa-decimal input	None.	Style='Hex', `@Min`, `@Max`, `@Bytes`
Decimal	Defines a value with decimal input	None.	Style='Decimal', `@Min`, `@Max`, `@Unit?`
Bin	Defines a value with binary input	None.	Style='Bin', `@Len`, `@Min?`, `@Max?`
CheckBox	Defines a value with check-box input	None.	Style='CheckBox', (`@Mes-sage`, `@MsgLvl`, `@MsgEmi-tOn`)?, `@Tristate?`
TextEx	Defines a value with text in-put	None.	Style='TextEx', `@Min?`, `@Max?`, `@Pattern?`
File	Defines a value with file name picker	None.	Style='File', `@Filter?`
Button	Defines an action button in U-Flash	None.	Style='Button'

5.1.14 File

Description

Describes an additional file required by a programming app.

Child of

<Addon>

Permitted children

None.

Permitted attributes

@RelPath?
@TargetPath (Addon)

5.1.15 FlashBankInfo

Description

This element is used to define the flash banks of a device. Each <FlashBankInfo> element defines a single contiguous memory area of the device with its own attributes. Only memory areas defined by this element are stored in the DAT file when it is created by U-Flash from a data file (e.g. HEX file). That is, memory areas that are used in a data file but do not match any <FlashBankInfo> are discarded.

Child of

<Device>

Permitted children

<SectorGroup>+

Permitted attributes

@Name (FlashBankInfo)
@BaseAddr
@MinWriteSize,
@Size (FlashBankInfo)?
@Blank?
@Loader?
@Aliased?
@Flags?

Additional information

There are several properties that can be applied to each flash bank using the @Flags attribute. The most important one is <NoTestData>, which ensures that U-Flash won't generate data for the flash bank when "Generate test data file…" or "Generate data file for read…" is used. It is recommended to use <NoTestData> for flash banks that contain data for e.g. fuses that might lock access to the device if programmed with random data. The <ReadOnly> flag can be used together with <NoTestData> to allow U-Flash to still generate data for the flash bank when "Generate data file for read…" is used. This is due to the fact that #READ reads only memory areas for which data is available in the data file. "Generate data file for read…" will then use the @Blank value of the respective flash bank to fill the data file.

5.1.16 Group

Description

Visually groups configuration values.

The group box contains a check box if the parent `<Config>` has its `@Value` set to 0 or 1. If the `@Value` is set to 1, the check box is selected by default. The content of the group cannot be edited when the check box is not selected. If no value is assigned to `@Value` (i.e. `@Value=“”`), no check box is shown and its content is editable.

Child of

`<Config>` (Device)

Permitted children

`<Config>+`

Permitted attributes

`@Controller` (Group)?
`@Controlled`?

5.1.17 Information

Description

Additional information for the device.

Child of

`<Device>`

Permitted children

None.

Permitted attributes

None.

5.1.18 MemoryMap

Description

Used to define one or more memory maps for a <Device>.

<Alternative> devices can use the @MemoryMap attribute to select the <MemoryMap>s they support. <MemoryMap>s can be excluded for the <Device> they are defined in using the @Exclude attribute.

Child of

<Device>

Permitted children

<FlashBankInfo>+

Permitted attributes

@Name (MemoryMap)
@Id?
@Exclude?

Additional information

Some device's memory areas may be represented by different memory maps. In this case it is possible to define multiple memory maps for specific devices by using <MemoryMap> elements. These elements then contain the according <FlashBankInfo> elements as described in *Defining flash banks* on page . To distinguish the memory maps, each <MemoryMap> receives a @Name (MemoryMap) that is shown in U-Flash.

```
<MemoryMap Name="16 MB Flash">
  <FlashBankInfo Name="2nd_Stage_Bootloader" BaseAddr="0x00000000" Size="0x8000"
 MinWriteSize="0x20" Loader="EXAMPLE.PEX">
    <SectorGroup Count="8" Size="0x1000"/>
  </FlashBankInfo>
  <FlashBankInfo Name="Partition_Table" BaseAddr="0x00008000" Size="0x8000"
 MinWriteSize="0x20" Loader="EXAMPLE.PEX">
    <SectorGroup Count="8" Size="0x1000"/>
  </FlashBankInfo>
  <FlashBankInfo Name="Application" BaseAddr="0x00010000" Size="0xFF0000"
 MinWriteSize="0x20" Loader="EXAMPLE.PEX">
    <SectorGroup Count="4080" Size="0x1000"/>
  </FlashBankInfo>
</MemoryMap>
<MemoryMap Name="Flat Memory Model">
  <FlashBankInfo Name="Flash (1)" BaseAddr="0x00000000" Size="0x80000000"
 MinWriteSize="0x20" Loader="EXAMPLE.PEX">
    <SectorGroup Count="524228" Size="0x1000"/>
  </FlashBankInfo>
  <FlashBankInfo Name="Flash (2)" BaseAddr="0x80000000" Size="0x80000000"
 MinWriteSize="0x20" Loader="EXAMPLE.PEX">
    <SectorGroup Count="524228" Size="0x1000"/>
  </FlashBankInfo>
</MemoryMap>
```

By default, all memory maps are used by the devices defined in the <Device> entry. But <MemoryMap> can also be used to define multiple memory maps that are not used by all devices defined in a <Device> entry.

<Device> entries can define multiple memory maps which can be assigned to specific devices defined in this <Device> entry. This avoids the creation of multiple <Device> entries just because some devices have a slightly different memory map, e.g. due to larger flash.

The defined memory maps are active by default for all devices defined in the <Device> entry. Memory maps that shall not be used by the device specified by the @Name (Device) attribute in <ChipInfo> require the @Exclude attribute to be set to true. To assign the memory maps to the alternative devices, an @Id attribute must be added to the memory maps so that they can be referenced. The @Id attribute must be unique among all <Device> entries in the DDF. The @Ids can then be used in the @MemoryMap attribute of <Alternative>. Multiple @Ids are used in @MemoryMap by separating them with semicolons.

The following example <Device> entry contains three devices (i.e. the main device and two alternative devices), each one using another memory map.

```
<Device>
  <ChipInfo Vendor="Example Manufacturer" Core="Example Core" Family="Example
 Family" Name="Device 1" Interface="Example IF"/>
  <Alternatives>
    <Alternative Name="Device 2" MemoryMap="MAP_64KB_512B">
      <Config Name="DEVICEID1" Value="0x96"/>
      <Config Name="DEVICEID2" Value="0x15"/>
    </Alternative>
    <Alternative Name="Device 3" MemoryMap="MAP_128KB_512B">
      <Config Name="DEVICEID1" Value="0x97"/>
      <Config Name="DEVICEID2" Value="0x0A"/>
    </Alternative>
  </Alternatives>
  <MemoryMap Name="32KB Flash, 512B EEPROM" Id="MAP_32KB_512B">
    <FlashBankInfo Name="Flash"  BaseAddr="0x00000000" Size="0x00008000"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="64" Size="0x200"/>
    </FlashBankInfo>
    <FlashBankInfo Name="EEPROM" BaseAddr="0x00810000" Size="0x00000200"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="1" Size="0x200"/>
    </FlashBankInfo>
  </MemoryMap>
  <MemoryMap Name="64KB Flash, 512B EEPROM" Id="MAP_64KB_512B" Exclude="true">
    <FlashBankInfo Name="Flash"  BaseAddr="0x00000000" Size="0x00010000"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="128" Size="0x200"/>
    </FlashBankInfo>
    <FlashBankInfo Name="EEPROM" BaseAddr="0x00810000" Size="0x00000200"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="1" Size="0x200"/>
    </FlashBankInfo>
  </MemoryMap>
  <MemoryMap Name="128KB Flash, 512B EEPROM" Id="MAP_128KB_512B" Exclude="true">
    <FlashBankInfo Name="Flash"  BaseAddr="0x00000000" Size="0x00020000"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="256" Size="0x200"/>
    </FlashBankInfo>
    <FlashBankInfo Name="EEPROM" BaseAddr="0x00810000" Size="0x00000200"
 MinWriteSize="0x20" Blank="0xFF" Loader="EXAMPLE.PEX">
      <SectorGroup Count="1" Size="0x200"/>
    </FlashBankInfo>
  </MemoryMap>
</Device>
```


5.1.19 Note

Description

This creates a text that is shown in the information section.

Child of

`<Information>`

Permitted children

any*

Permitted attributes

None.

5.1.20 Option

Description

Creates an entry inside `<Edit>` elements with `@Style`="DropDown".

Child of

`<Edit>`

Permitted children

None.

Permitted attributes

`@Name` (Option)
`@Value`

5.1.21 Range

Description

Creates a pair of configuration values with a common prefix denoting a *_min* and a *_max* value.

Child of

`<Config>` (Device)

Permitted children

depends on attribute `@Style`

Permitted attributes

`@Style`; presence of other attributes depends on this.

Style	Description	Permitted children	Permitted attributes
DropDown	Defines a drop-down input	@Option	Style='DropDown'
Hex	Defines a drop-down input	None.	Style='Hex', `@Min`, `@Max`, `@Bytes`
Decimal	Defines decimal input	None.	Style='Decimal', `@Min`, `@Max`, `@Unit?`
Bin	Defines a binary input	None.	Style='Bin', `@Len`, `@Min?`, `@Max?`

5.1.22 row

Description

Defines the default values for configuration values at row's index in `<Edit>` elements with `@Style=`"Table".

Child of

`<Edit>`

Permitted children

`<c>+`

Permitted attributes

None.

5.1.23 SectorGroup

Description

Specifies the count and the size of a group of sectors in the flash bank. If the flash bank contains sectors with different sizes, multiple <SectorGroup> elements need to be used. The order of the <SectorGroup> is important: the first sector group starts at the base addess of the flash bank, the second one continues after the first sector group, and so on.

Child of

<FlashBankInfo>

Permitted children

None.

Permitted attributes

@Count
@Size (SectorGroup)

5.1.24 Table

Description

Creates an extensible table view of configuration values.

> **Note**
>
> Configuration children must only be of type `<Edit>`, `<Range>` or `<Bitfield>`

Child of

`<Config>` (Device)

Permitted children

```
<Config>+
<row>+
```

Permitted attributes

```
@StartIdx?
@EndIdx?
```


5.1.25 XMLComment

Description

A comment within an `<Information>` element creates a remark text that is shown in the information section. The text may contain HTML markup, but it is not guaranteed that it will render as expected, this depends on the browser.

Child of

`<Information>`

Permitted children

None.

Permitted attributes

None.

5.2 Attributes

5.2.1 Aliased

Description

This attribute defines an alias address. This is useful when mapping storage areas, for example, in a cached and a non-cached area. Programming should be done in non-cached address areas. In such a case, the cached area should be defined with Aliased to the non-cached area.

Where used

- `<FlashBankInfo>`: optional

Type

32-bit hexadecimal value

Values

`0x00000000…0x7FFFFFFF`

Default (when omitted)

same as BaseAddr

5.2.2 BaseAddr

Description

This attribute defines the start address of the memory area.

Elements and usage

- DataBase/Device/FlashBankInfo: required

Type

32-bit hexadecimal value

Values

`0x00000000…0x7FFFFFFF`

5.2.3 Blank

Description

This attribute defines the value of a memory cell in erased state.

Elements and usage

- DataBase/Device/FlashBankInfo: optional

Type

8/16/32-bit hexadecimal value

Values

- `0x00…0xFF`
- `0x0000…0xFFFF`
- `0x00000000…0xFFFFFFFF`

Default (when omitted)

undefined

5.2.4 Bytes

Description

This attribute defines the length of a hexadecimal configuration value.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='Hex']: required
- DataBase/Device/*/Config/Range[@Style='Hex']: required

Type

Style='Hex': decimal number

Values

Style='Hex': 1…4

5.2.5 Controlled

Description

This attribute determines which configuration values within the group are made editable by `@Controller` (Group) and which value must be assumed for this purpose.

Elements and usage

- DataBase/Device/Config/Group: optional

Type

list of key-value-pairs

Values

Semicolon-separated list of key-value-pairs:

```
<controlled config name>=<controller value>
```

Default (when omitted)

not set

Note

Shall be used in conjunction with `@Controller` (Group).

5.2.6 Controller (Config)

Description

This attribute is used to specify a condition that controls whether the <Config> using the Controller attribute is editable in U-Flash. A condition consists of a list of one or more configuration name and value pairs in the form of <config name#<value>. These pairs can be logically combined with ';' (boolean OR) and '+' (boolean AND). If the <Config> referenced by the @Controller (Config) attribute has the specified value, the condition is true and the configuration is editable in U-Flash.

Elements and usage

- DataBase/Device/Config: optional
- DataBase/Device/Alternatives/Alternative/Config: optional
- DataBase/Device/Config/Group/Config: optional
- DataBase/Device/Config/Table/Config: optional
- DataBase/Device/Config/Bitfield/BitGroup: optional

Type

key-value-pair-list

Values

```
<config name>=<value>[(;<config name>=<value>)|(+<config name>=<value>)]
```

Default (when omitted)

not set

Additional information

@Controller (Config)s provide a way to make specific configurations editable in U-Flash depending on the configuration values of other configurations. Configuration groups (<Group>) that have a check box automatically make their content editable or not depending on the state of the check box. Therefore, a controller is not required for such groups. However, if a configuration should be editable depending on the value of another configuration, the @Controller (Config) attribute has to be used.

The following example shows how the @Controller (Config) attribute can be used.

- The configuration "Example Config 2" is editable when the check box "Example Config 1" is selected
- The configuration "Example Config 5" is editable when "Example Config 3" is set to 0x10 or "Example Config 4" is set to 10
- The configuration "Example Config 6" is editable when "Example Config 3" is set to 0x55 and "Example Config 4" is set to 32

```
<Config Name="Example Config 1" Value="0">
  <Edit Style="CheckBox"/>
</Config>
<Config Name="Example Config 2" Value="0" Controller="Example Config 1=1">
  <Edit Style="Decimal" Min="0" Max="100"/>
</Config>
<Config Name="Example Config 3" Value="0x00">
  <Edit Style="Hex" Min="0x00" Max="0xFF" Bytes="1"/>
</Config>
<Config Name="Example Config 4" Value="0">
  <Edit Style="Decimal" Min="0" Max="100"/>
</Config>
<Config Name="Example Config 5" Value="0" Controller="Example Config
 3=0x10;Example Config 4=10">
  <Edit Style="Decimal" Min="0" Max="100"/>
</Config>
```



```
<Config Name="Example Config 6" Value="0" Controller="Example Config
 3=0x55+Example Config 4=32">
  <Edit Style="Decimal" Min="0" Max="100"/>
</Config>
```


5.2.7 Controller (Group)

Description

This attribute determines which other configuration value makes configuration values inside this group editable.

Elements and usage

- DataBase/Device/Config/Group: optional

Type

id string

Values

```
<config name>
```

Default (when omitted)

not set

Note

Shall be used in conjunction with `@Controlled`.

5.2.8 Core

Description

Should be the name assigned by the silicon manufacturer or designer for the core design, for instance “M16C/80” or “Cortex-M0+”.

Elements and usage

- DataBase/Device/ChipInfo: required

Type

Character string

Values

Any non-empty character string.

5.2.9 Count

Description

Specifies the number of sectors within the sector group.

Elements and usage

- DataBase/Device/FlashBankInfo/SectorGroup: required

Type

positive integer

Values

> 0

5.2.10 DefaultErase

Description

This attribute defines the pre-selected erase method for the algorithm.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

erase method

Values

One of:

- `None` (do not erase),
- `Sector` (erase required sectors),
- `Dirty` (erase required, non empty sectors),
- `Bulk` (mass erase)

Please note that this value **must** be included in the list of `@EraseType`. Otherwise, a change of this attribute cannot be restored without discarding the project.

Default (when omitted)

no value; first value of list `@EraseType` is used.

5.2.11 DefaultProgram

Description

This attribute defines the default setting for program task on automatic mode.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

DisableSetUnset

Values

One of:

- `0` (checkbox not ticked and editable; task will not be performed)
- `1` (checkbox ticked and editable; task will be performed)
- `-1` (checkbox not ticked and not editable; task will not be performed)
- `-2` (checkbox ticked and not editable; task will be performed)

Default (when omitted)

`0`

5.2.12 DefaultRead

Description

This attribute defines whether #READ is supported. #READ is not part of the default tasks and thus is not executed by #AUTO. It also isn’t shown in the tasks group in the project configuration of U-Flash.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

DisableSetUnset

Values

One of:

- `# 0` #READ is supported.
- `= -1` #READ is not supported.

Default (when omitted)

`0`

5.2.13 DefaultSecure

Description

This attribute defines the default setting for secure task on automatic mode.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

DisableSetUnset

Values

One of:

- `0` (checkbox not ticked and editable; task will not be performed)
- `1` (checkbox ticked and editable; task will be performed)
- `-1` (checkbox not ticked and not editable; task will not be performed)
- `-2` (checkbox ticked and not editable; task will be performed)

Default (when omitted)

`0`

5.2.14 DefaultVerify

Description

This attribute defines the default setting for verify task on automatic mode.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

DisableSetUnset

Values

One of:

- `0` (checkbox not ticked and editable; task will not be performed)
- `1` (checkbox ticked and editable; task will be performed)
- `-1` (checkbox not ticked and not editable; task will not be performed)
- `-2` (checkbox ticked and not editable; task will be performed)

Default (when omitted)

`0`

5.2.15 Descr

Description

This attribute can be used to provide the user with a short list of the main features of the unit.

Elements and usage

- DataBase/Device/ChipInfo: optional

Type

Character string

Values

Any character string.

Default (when omitted)

empty string

5.2.16 Display

Description

This attribute defines the text shown in UI for the configuration value. It decouples the screen text from config value name.

Elements and usage

- DataBase/Device/Config: optional
- DataBase/Device/Alternatives/Alternative/Config: optional
- DataBase/Device/Config/Group/Config: optional
- DataBase/Device/Config/Table/Config: optional
- DataBase/Device/Config/Bitfield/BitGroup: optional

Type

Character string

Values

Any non-empty character string.

Default (when omitted)

Same as `@Name` (Config).

5.2.17 EndIdx

Description

This attribute defines the highest possible index for tabulated configuration values.

Elements and usage

- DataBase/Device/*/Config/Table: optional

Type

An decimal number

Values

`0` ≤ `@StartIdx` ≤ `2147483647`

Default (when omitted)

`2147483647`

5.2.18 EraseType

Description

This attribute lists all supported erase methods of the device.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

List of erase methods

Values

Space separated list of one or more values out of:

- `None` (do not erase),
- `Sector` (erase required sectors),
- `Dirty` (erase required, non empty sectors),
- `Bulk` (mass erase)

Default (when omitted)

```
None Sector Dirty Bulk
```


5.2.19 Exclude

Description

This attribute specifies whether the `<MemoryMap>` is used by the `<Device>` it is defined in.

Elements and usage

- DataBase/Device/MemoryMap: optional

Type

Boolean

Values

`true`, `false`

Default (when omitted)

`false`

5.2.20 Family

Description

Should be the name assigned by the silicon manufacturer or designer for the core design. E.g. “M16C/80” or “Cortex-M0+”.

Elements and usage

- DataBase/Device/ChipInfo: required

Type

Character string

Values

Any non-empty character string.

Default (when omitted)

empty string

5.2.21 Filter

Description

This attribute defines the filter list for file selection dialog.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='File']: optional

Type

`@Style='File'`: List of file name pattern

Values

`@Style='File'`: A space separated list of file name pattern. Normal wildcard rules apply: '?' for single letter, '*' for one or more letters.

Default (when omitted)

`*.*`

5.2.22 Fixed

Description

This attribute completely disables changing the “Task” settings in the project settings of U-Flash.

Elements and usage

- DataBase/Device/DefaultTasks: optional

Type

SetUnset

Values

`0, 1`

Default (when omitted)

`0`

5.2.23 Flags

Description

This attribute defines flags for specific flash banks that determine the way U-Flash handles these flash banks.

Elements and usage

- DataBase/Device/FlashBankInfo: optional

Type

Character string

Values

List of flags separated by space characters.

Flag	Description
<NoFitCheck>	When converting a data file into DAT files, U-Flash checks if the data file contains data outside of the defined flash banks. This flag disables this check. Disabling this check may be useful if a DDF has a flash bank defined for large memory ranges. For example, if a 2GB flash bank is defined, running the check may require a lot of RAM and its rather unlikely that the data file contains data outside the allowed 2GB.
<NoTestData>	With <NoTestData> users will not be able to include the respective flash bank when generating test data files with random data or data files for read with U-Flash. This is useful to prevent users from generating data for e.g. flash banks that contain option bytes or fuses that might irreversibly lock the device.
<Readable>	If <Readable> is used together with <NoTestData> then the user can include the respective flash bank when generating data files for read but not when generating test data files.

Default (when omitted)

empty string

5.2.24 Hide

Description

This attribute defines in which view the config value is shown.

Elements and usage

- DataBase/Device/Config: optional
- DataBase/Device/Alternatives/Alternative/Config: optional
- DataBase/Device/Config/Group/Config: optional
- DataBase/Device/Config/Table/Config: optional
- DataBase/Device/Config/Bitfield/BitGroup: optional

Type

Enumeration

Values

One of:

- `Hide` (Never show in U-Flash)
- `Hide-Simple` (Do not show in U-Flash’s basic view)
- `Hide-Never` (Always show in U-Flash)

Default (when omitted)

`Hide-Never`

5.2.25 Hint

Description

This attribute defines the a helpful hint text shown in U-Flash to explain function of this configuration value.

Elements and usage

- DataBase/Device/Config: optional
- DataBase/Device/Alternatives/Alternative/Config: optional
- DataBase/Device/Config/Group/Config: optional
- DataBase/Device/Config/Table/Config: optional
- DataBase/Device/Config/Bitfield/BitGroup: optional

Type

Character string

Values

Any non-empty character string.

Default (when omitted)

empty string

5.2.26 Id

Description

This attribute gives <MemoryMap> elements a unique identity. <MemoryMap> IDs can be referenced with @MemoryMap. The @Id attribute must be unique among all <Device> entries in the DDF.

Elements and usage

- DataBase/Device/MemoryMap: optional

Type

character string

Values

any unique string.

5.2.27 Interface

Description

The name of the interface (possibly in conjunction with the protocol/application) used for programming should be clearly stated here. E.g. “BSL via UART” for using UART to communicate with the built-in boot strap loader.

Elements and usage

- DataBase/Device/ChipInfo: required

Type

Character string

Values

Any non-empty character string.

Note

This value — in connection with `@Name` (Device) — must be unique across all devices!

5.2.28 Len

Description

This attribute defines the length of a binary configuration value.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='Bin']: required
- DataBase/Device/*/Config/Range[@Style='Bin']: required

Type

Style='Bin': decimal number

Values

Style='Bin': 1…32

5.2.29 Loader

Description

This attribute defines the name of the PEX file to be used.

Elements and usage

- DataBase/Device/FlashBankInfo: optional

Type

31 char PEX file name.

Values

Any valid file name.

Default (when omitted)

undefined

Note

Must not be omitted for the first defined memory bank. It is not permissible to omit the attribute only for individual banks. It is recommended to use it for each bank.

5.2.30 Max

Description

This attribute defines the maximal value or maximal length, depending on `@Style`.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='Hex']: required
- DataBase/Device/*/Config/Edit[@Style='Decimal']: required
- DataBase/Device/*/Config/Edit[@Style='Bin']: optional
- DataBase/Device/*/Config/Edit[@Style='TextEx']: optional
- DataBase/Device/*/Config/Range[@Style='Hex']: required
- DataBase/Device/*/Config/Range[@Style='Decimal']: required
- DataBase/Device/*/Config/Range[@Style='Bin']: optional

Type

- Style='Hex': hexadecimal number
- Style='Decimal': decimal number
- Style='Bin': binary number
- Style='TextEx': decimal number

Values

- Style='Hex', `@Bytes='1'`: `0x00 ≤ @Max ≤ 0xFF`
- Style='Hex', `@Bytes='2'`: `0x0000 ≤ @Max ≤ 0xFFFF`
- Style='Hex', `@Bytes='3'`: `0x000000 ≤ @Max ≤ 0xFFFFFF`
- Style='Hex', `@Bytes='4'`: `0x00000000 ≤ @Max ≤ 0xFFFFFFFF`
- Style='Decimal': `0 ≤ @Max ≤ 4294967295`
- Style='Bin': `0b0 ≤ @Max ≤ 0b11111111111111111111111111111111`
- Style='TextEx': `0 ≤ @Max ≤ 128`

Default (when omitted and permitted)

the respective maximum value

5.2.31 MemoryMap

Description

This attribute references the <MemoryMap>s used by <Alternative> devices.

Elements and usage

- DataBase/Device/Alternatives/Alternative: optional

Type

Character string

Values

A list containing the @Ids of the <MemoryMap>s, separated by spaces.

Default (when omitted)

5.2.32 Message

Description

This attribute defines a message to be displayed in a message box with severity `@MsgLvl` when the value of the CheckBox-styled configuration value is set to the value specified in `@MsgEmitOn`. The mentioned attributes may only be used together.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='CheckBox']: optional

Type

Style='CheckBox': character string

Values

Style='CheckBox': any character string

Default (when omitted)

empty string

5.2.33 Min

Description

This attribute defines the minimal value or minimal length, depending on `@Style`.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='Hex']: required
- DataBase/Device/*/Config/Edit[@Style='Decimal']: required
- DataBase/Device/*/Config/Edit[@Style='Bin']: optional
- DataBase/Device/*/Config/Edit[@Style='TextEx']: optional
- DataBase/Device/*/Config/Range[@Style='Hex']: required
- DataBase/Device/*/Config/Range[@Style='Decimal']: required
- DataBase/Device/*/Config/Range[@Style='Bin']: optional

Type

- Style='Hex': hexadecimal number
- Style='Decimal': decimal number
- Style='Bin': binary number
- Style='TextEx': decimal number

Values

- Style='Hex', `@Bytes='1'`: `0x00 ≤ @Min ≤ 0xFF`
- Style='Hex', `@Bytes='2'`: `0x0000 ≤ @Min ≤ 0xFFFF`
- Style='Hex', `@Bytes='3'`: `0x000000 ≤ @Min ≤ 0xFFFFFF`
- Style='Hex', `@Bytes='4'`: `0x00000000 ≤ @Min ≤ 0xFFFFFFFF`
- Style='Decimal': `0 ≤ @Min ≤ 4294967295`
- Style='Bin': `0b0 ≤ @Min ≤ 0b11111111111111111111111111111111`
- Style='TextEx': `0 ≤ @Min ≤ 128`

Default (when omitted and permitted)

`0`

5.2.34 MinWriteSize

Description

This attribute specifies the minimum amount of bytes the programming app is able to program in the target device. The Flasher can then call `UNI_Program()` for multiple consecutive memory sections, each one of the size specified by MinWriteSize.

Elements and usage

- DataBase/Device/Config: required

Type

32-bit hexadecimal value

Values

`0x00000000…0x7FFFFFFF`

5.2.35 MsgEmitOn

Description

This attribute defines the value to which a CheckBox-styled configuration value must be set to display a `@Message` in a message box with severity `@MsgLvl`. The above attributes may only be used together.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='CheckBox']: optional

Type

Style='CheckBox': Integer

Values

Style='CheckBox': `0`, `1`, `2`

Default (when omitted)

not set

5.2.36 MsgLvl

Description

This attribute defines severity `@MsgLvl` of the message box the `@Message` will be displayed in, when the value of the CheckBox-styled configuration value is set to the value specified in `@MsgEmitOn`. The mentioned attributes may only be used together.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='CheckBox']: optional

Type

Style='CheckBox': Enumeration

Values

Style='CheckBox': one of

- `Question`
- `Information`
- `Warning`
- `Critical`

Default (when omitted)

not set

5.2.37 N

Description

This attribute names the configuration value to set the default value for.

Elements and usage

- DataBase/Device/*/Config/Table/row/c: required

Type

character string

Values

Any valid configuration value name in the row

5.2.38 Name (Config)

Description

This attribute defines the name of the configuration value and also the key used by the programming app to load the configuration value from the UNI file.

Used by

- DataBase/Device/Config: required
- DataBase/Device/Alternatives/Alternative/Config: required
- DataBase/Device/Config/Group/Config: required
- DataBase/Device/Config/Table/Config: required
- DataBase/Device/Config/Bitfield/BitGroup: required

Type

Character string

Values

Any non-empty character string.

Note

This value must be unique for this device.

5.2.39 Name (Device)

Description

It should be the (most common) device name assigned to the device by the silicon manufacturer or developer. Be as specific as necessary, as generic as possible. For example, “TC334LP-32F” instead of “SAK-TC334LP-32F200F AA”.

Elements and usage

- DataBase/Device/ChipInfo: required
- DataBase/Device/Alternatives/Alternative: optional

Type

Character string

Values

Any non-empty character string.

Note

This value — in connection with `@Interface` — must be unique across all devices!

5.2.40 Name (FlashBankInfo)

Description

This attribute names a memory area on the device. It has only a descriptive function, but must be unique for the defined device.

Elements and usage

- DataBase/Device/FlashBankInfo: required

Type

Character string

Values

Any non-empty character string

5.2.41 Name (MemoryMap)

Description

This attribute defines the name shown in U-Flash for the memory map.

Elements and usage

- DataBase/Device/MemoryMap: required

Type

Character string

Values

Any non-empty character string

Additional information

5.2.42 Name (Option)

Description

This attribute defines the string that will be displayed for a DropDown-styled configuration value. It decouples the functional `@Value` from its displayed value.

Elements and usage

- DataBase/Device/Config/Edit[@Style='DropDown']/Option: required

Type

Character string

Values

Any character string.

Note

This value must be unique for this drop-down list.

5.2.43 OptDatFile

Description

This attribute allows to create and provision a project without a data (DAT) file.

Elements and usage

- DataBase/Device/ChipInfo: optional

Type

Boolean

Values

`true`, `false`

Default (when omitted)

`false`

5.2.44 Pattern

Description

This attribute allows to define a RegEx to validate the input into the text field.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='TextEx']: optional

Type

Style='TextEx': RegEx string

Values

Style='TextEx': any valid RegEx string

Default (when omitted)

empty string

5.2.45 Pos

Description

This attribute defines the position and location of <BitGroup> in a <Bitfield>. It is in the form <zero-based-2-digit-number of MSB>:<zero-based-2-digit-number of LSB>. A notation of Pos="5:3" denotes a <BitGroup> comprising 3 bits, starting with bit #3 ($\#2^3=8_d$) through bit #5 ($\#2^5=32_d$).

Elements and usage

- DataBase/Device/*/Bitfield/BitGroup: required

Type

Range denominator

Values

Colon separated tuple of zero based, zero padded, 2 digit decimal values. #MSB':'#LSB

5.2.46 RelPath

Description

This attribute defines the source file name (on PC) of an additional file if it is part of a project configuration.

Elements and usage

- DataBase/Device/Addon/File: required

Type

An OS file path

Values

any valid file path relative to U-Flash.

5.2.47 Size (Bitfield)

Description

This attribute defines the size of the bit field in bytes.

Elements and usage

- DataBase/Device/*/Config/Bitfield: required

Type

integer

Values

`0…4`

5.2.48 Size (FlashBankInfo)

Description

This attribute defines the size of the memory area.

Elements and usage

- DataBase/Device/FlashBankInfo: optional

Type

32-bit hexadecimal value

Values

`0x00000000…0x7FFFFFFF`

Default (when omitted)

`0x7FFFFFFF`

Note

This value, when defined, must not be 0.

5.2.49 Size (SectorGroup)

Description

This attribute defines the size for all sectors of the `<SectorGroup>`.

Elements and usage

- DataBase/Device/FlashBankInfo/SectorGroup: required

Type

32-bit hexadecimal value

Values

`0x00000000…0x7FFFFFFF`

Default (when omitted)

`0x7FFFFFFF`

Note

This value, when defined, must not be 0.

5.2.50 StartIdx

Description

This attribute defines the first index for tabulated configuration values.

Elements and usage

- DataBase/Device/*/Config/Table: optional

Type

A decimal number

Values

`0…2147483647`

Default (when omitted)

`0`

5.2.51 Style

Description

This attribute defines the type of input for the interactive editing of configuration values by the user. It also defines the allowed child elements and attributes.

Elements and usage

- DataBase/Device/Config/Edit: required

Type

Enumeration

Values

One of:

- `DropDown`
- `Hex`
- `Decimal`
- `Bin`
- `CheckBox`
- `TextEx`
- `File`
- `Button`

5.2.52 TargetPath (Addon)

Description

This attribute defines the target name (on Flasher) of an additional file if it is part of a project configuration.

Elements and usage

- DataBase/Device/Addon/File: required

Type

File name of length less than 32, preferably in DOS 8.3 format.

Values

any valid file name

5.2.53 Tristate

Description

This attribute defines a checkbox that defines three states: checked, unchecked, and a third state, e.g. to not change the dependent state.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='CheckBox']: optional

Type

Style='CheckBox': boolean

Values

Style='CheckBox': `true`

Fixed value

true

5.2.54 Unit

Description

This attribute defines a unit in which the decimal configuration can be meaningfully interpreted by the user. It has only informational character.

Elements and usage

- DataBase/Device/*/Config/Edit[@Style='Decimal']: optional

Type

Style='Decimal': character string

Values

Style='Decimal': any character string

Default (when omitted)

empty string

5.2.55 Value

Description

This attribute defines the initial value for this configuration value. The format of the value is generic and cannot be checked by U-Flash unless an `<Edit>` node is present. In this case, the value is a constant.

Elements and usage

- DataBase/Device/Config: required
- DataBase/Device/Alternatives/Alternative/Config: required
- DataBase/Device/Config/Group/Config: required
- DataBase/Device/Config/Table/Config: required
- DataBase/Device/Config/Bitfield/BitGroup: required
- DataBase/Device/Config/Edit[@Style='DropDown']/Option: required

Type

Character string

Values

Any non-empty character string.

The value is interpreted according to the `@Style` attribute. This attribute also defines the expected and accepted format of the content.

5.2.56 Vendor

Description

Should be the common (short) name of the silicon vendor, for instance “ISSI” rather than “Integrated Silicon Solution Inc”.

Elements and usage

- DataBase/Device/ChipInfo: required

Type

Character string

Values

Any non-empty character string.

5.2.57 Version

Description

Defines a version number to distinguish the evolutionary stages of an algorithm from each other.

Used by

- `<Device>`: optional

Type

Character string

Values

`[RLP][0-9A-F]+`

- `R`: Reviewed
- `L`: Local
- `P`: Published

followed by a hexadecimal value indicating a monotonically increasing version number.

Default (when omitted)

`R0`

Note

Please note that versions beginning with “R” are reserved for Flasher device packs reviewed or published by SEGGER.

Chapter 6

S32 Compiler

This chapter contains the documentation of the S32 C compiler.

6.1 Introduction

The S32CC executable for Windows is located in the `Tools` directory of the Flasher SDK. It is used to compile a single app source code file into a portable executable file (.pex) which is the final executable used by the Flasher. A linker is not required.

The compiler adheres to the C90 standard with some C99 extensions (see *Extensions* on page 296). However, there are some restrictions and deviations that are listed in *Restrictions* on page 296 and *Deviations* on page 297.

6.2 Usage

The S32CC is called from the command line as follows:

```
S32CC [<options>] <source file>
```

A list with all available options can be shown by calling the S32CC without any arguments.

To compile a programming app 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 declarations for the programming app API described in *app API reference* on page .

Assuming the S32CC is called from within the `Tools` directory of the Flasher SDK and the `Inc` directory is located in the `Tools` directory, S32CC is called as follows to compile the provided programming app source code template `Manufacturer_Device.pc`:

```
S32CC -JInc ../Template/Manufacturer_Device/Manufacturer_Device.pc
```

When the compiler finishes compilation successfully, there are two new files located next to the source code file:

- The portable executable (PEX) file `Manufacturer_Device.pex` containing the code and data for the app.
- A listing file `Manufacturer_Device.lst` containing a listing of the S32 instructions and data compiled to run the app.

6.3 Extensions

- C++ single-line comments (i.e. `//`) are supported.
- Data within functions doesn't have to be defined at the beginning of the function but can be defined anywhere within the function.
- 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.

6.4 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 is not (`x = y = z;`).

- The comma operator in expressions is not supported.
- Functions can take no more than 7 parameters.

6.5 Deviations

The supported C programming language is subject to following deviations:

- Octal constants are flagged as a warning and treated as decimal.
- As there is no linker, the `extern` specifier serves another purpose and is used only with pseudo-variables. It prevents taking the address of variables and accessing them through pointers.

6.6 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.

6.6.1 Predefined macros

Following macros are predefined by the compiler:

Macro	Type	Description	Example Output
`__DATE__`	string	Date the programming app source code file was compiled.	Feb 20 2025
`__TIME__`	string	Time the programming app source code file was compiled.	12:02:05
`__FILE__`	string	Name of the programming app source code file.	MyCode.pc
`__LINE__`	integer	The line number the macro was used in.	80
`__S32C__`	integer	The full version number containing the major, minor and patch number (Mmmpp).	21202
`__S32C_MAJOR__`	integer	The major version number.	2
`__S32C_MINOR__`	integer	The minor version number.	12
`__S32C_PATCHLEVEL__`	integer	The patch level.	2
`__S32C_VERSION__`	string	The version string (“M.mm.pp”).	2.12.2

6.6.2 Pragmas

The following pragmas are supported by the S32 C compiler.

Pragma	Explanation
`xdata_size`	Specifies the size of the memory allocated for the xdata area.

6.6.2.1 xdata_size

Description

This pragma specifies the size of the memory allocated for the xdata area pointed to by `__S32_XDataBase`. The xdata area is used by the Flasher to provide the data to be flashed to the programming app, or is expected to contain the data that was read from the target device after a `UNI_Read()` call. The `xdata_size` pragma must be set to a value equal or greater than the largest `@MinWriteSize` of all flash banks specified for the devices supported by the programming app.

Syntax

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

Parameter	Description
`Size`	Size of the xdata area to be allocated.

Example

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

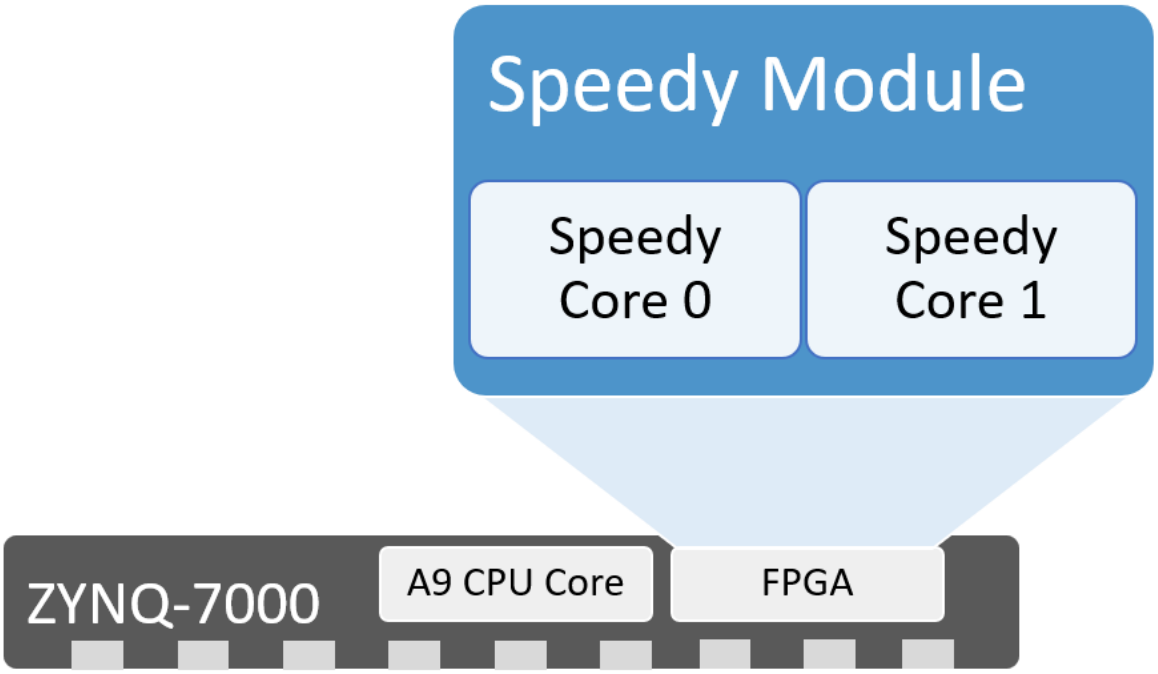

Chapter 7

Speedy

This chapter contains the documentation for the Speedy module. A guide on how to use Speedy can be found in the chapter *Creating custom TIFs* on page 302.

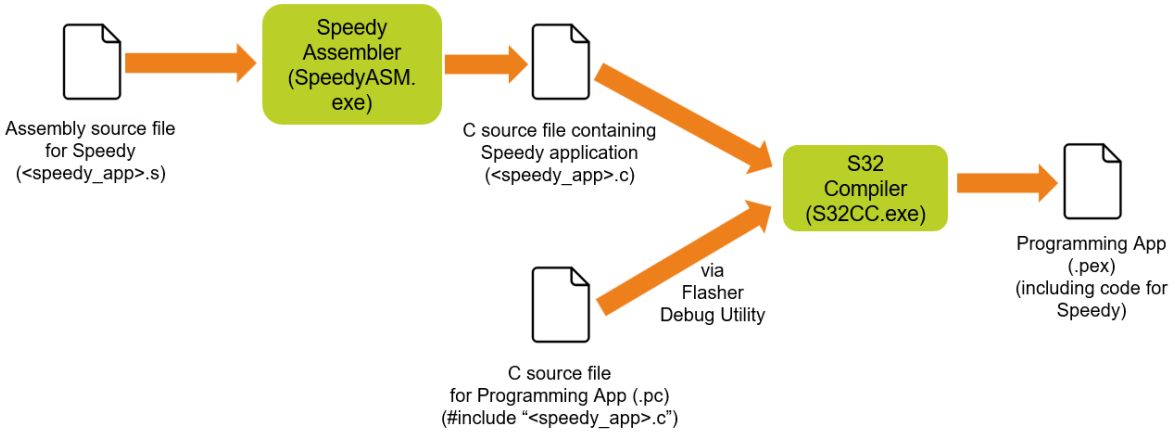
7.1 Introduction

The Speedy module consists of two high-speed 8-bit cores. Its purpose is to provide a dedicated processing unit for the communication with the target device. By decoupling the target communication from the Flasher's main CPU, communication with the target is not affected by the main core or an app running on it. Both Speedy cores can run at up to 200 MHz with single-cycle execution, delivering up to 200 MIPS of performance. With the deterministic timing of Speedy running at 200MHz, pins can be accurately toggled with a precision of 5ns.



When a programming app initializes a target interface, the Flasher loads the appropriate Speedy application for the target interface into Speedy. It is also possible to create custom target interfaces by developing applications for Speedy that can be loaded by the programming app into Speedy.

Speedy applications are written in assembly language using the Speedy instruction set. The assembly language source file is translated using the Speedy assembler (included in the Flasher SDK), which outputs a C array in a separate file that can be included by the programming app source code to load the array with the Speedy instructions into Speedy. This allows customers to create target interfaces for any target device without requiring the Flasher to support the target interface. Also, see the illustration below.



7.2 Basic information

Below is a list with some basic information about the Speedy module:

- Two independent 8-bit cores
- Adjustable frequency from 1 MHz up to 200 MHz (both Speedy cores share the same clock source)
- 32 general-purpose registers
- All instructions are 16-bit
- Instructions are executed in a single cycle, except for branches, which require three cycles if the branch is taken
- Program size of 512 instructions for core 0 and 256 instructions for core 1
- After reset, Speedy starts execution at location zero, the first instruction loaded into Speedy
- Single-level return stack used with JSR and RET instructions
- No multiplier or divider
- No RAM and therefore no stack pointer and no push or pop instructions
- No interrupts

7.3 Creating custom TIFs

The creation of a custom target interface consists of two parts:

- Developing the Speedy application that handles the low-level communication with the target.
- Implementing the API functions that handle the communication and data transfer between the programming app and Speedy.

When talking about the communication and data transfer between the programming app and Speedy, "programming app" is used to refer to the programming application that is compiled and executed by the S32. The Speedy application, on the other hand, is written in assembly language using the Speedy instruction set and runs independently on the Speedy core. The Speedy instruction set, the assembly language and the assembler are documented in the chapter *Speedy* on page 300.

7.4 Communication with Speedy

The Flasher manages two FIFO buffers that are used to transfer data between the programming app and Speedy: An up buffer for sending data from Speedy to the programming app, and a down buffer for sending data from the programming app to Speedy.

Speedy can send and receive data via the I/O space, which contains 8-bit registers used to place a byte in the up buffer (`OUT_TX_DATA`) and fetch a byte from the down buffer (`IN_RX_DATA`). To ensure that the data is placed in the up buffer only when it is not full, the `IN_TX_STAT` register must be checked. The `IN_RX_STAT` register must be used to check if the programming app has placed some data in the down buffer.

The following assembly code waits for data from the programming app and sends it back to the programming app.

```
  OUT_TX_DATA = 0x3E
  IN_TX_STAT  = 0x3D
  IN_RX_DATA  = 0x3E
  IN_RX_STAT  = 0x3F
Start:
  // Wait until there's data in the down buffer.
  WWL IN_RX_STAT
  // Fetch the data.
  IN  R0, IN_RX_DATA
  // Wait until there's free space in the up buffer.
  WWH IN_TX_STAT
  // Send the received data back to the flash loader.
  OUT R0, OUT_TX_DATA
  // Jump to start.
  JMP Start
```

The app can send and receive data with the API functions `SPEEDY_TransferDCC()` and `SPEEDY_ReadDCCNB()`. `SPEEDY_TransferDCC()` sends and receives the specified amount of bytes and blocks until all data to send to the Speedy is stored in the down buffer and all data to receive from the Speedy is fetched from the up buffer. `SPEEDY_ReadDCCNB()`, on the other hand, allows to read only the data that is currently available in the up buffer and returns immediately.

The following function sends the string “Hello World” to the Speedy core 0 and waits for Speedy to return the same string.

```
#define STRING  "Hello World"
static void EchoSample(void) {
  U8 DataUp[sizeof(STRING)];

  //
  // Flush the up buffer by reading all data available in the up buffer.
  //
  while (SPEEDY_ReadDCCNB(0, &DataUp, 1) > 0) {
    SYS_Reportf("Removed 0x%X from the up buffer.", DataUp);
}
  //
  // Send "Hello World" and wait for Speedy to return the sent data.
  //
  SPEEDY_TransferDCC(0, STRING, sizeof(STRING), &DataUp[0], sizeof(DataUp));
}
```

7.5 Controlling Flasher pins

Just like the communication with the programming app, the pins of the Flasher are controlled via the I/O space. The I/O space provides multiple registers for configuring, setting, and clearing the pins. But before the pins can be used, each pin must be initialized correctly.

To access and configure pins, a bit mask is used to operate on the desired pins in the respective registers. Each bit in the mask refers to a specific pin. Some pins can be used only as input but not as output. The following table depicts the pin assignment used in the bit mask and the supported direction of the pins.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pin15	Pin 13	Pin 17	Pin 11	Pin 3	Pin 5	Pin 7	Pin 9
In	In	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out

The registers used for initializing pins and setting their state provide dedicated set, clear and toggle registers that allow to operate only on specific pins and preserve the state and settings of the other pins. Pins are initialized by setting the desired direction of the pin with the `OUT_PIN_DIR` registers. For the direction settings to take affect they must be forced for the pins that are used by Speedy. This is done with the `OUT_FORCE_DIR` register.

The following code shows how to initialize pin 11 for input and pin 5 for output.

```
  OUT_PIN_DIR_SET   = 0x11
  OUT_PIN_DIR_CLR   = 0x12
  OUT_FORCE_DIR_SET = 0x21
  // Load masks into general purpose registers
  MOV   R0, #0x10   // Mask for pin 11
  MOV   R1, #0x04   // Mask for pin 5
  // Clear to configure pin 11 for input
  OUT   R0, OUT_PIN_DIR_CLR
  // Set to configure pin 5 for output
  OUT   R1, OUT_PIN_DIR_SET
  // Force direction settings for both pins
  OUT   R0, OUT_FORCE_DIR_SET
  OUT   R1, OUT_FORCE_DIR_SET
```

After the pins are initialized, their state can be read with the `IN_PIN` register or changed with the `OUT_PIN` registers. When using the instruction `IN Rd, <InAddr>` to read a register located in the I/O space, bit 0 of the register read is loaded into the carry flag and bit 1 is loaded into the zero flag. This allows for some specialties like performing a conditional branch depending on the pin state.

```
  IN_PIN = 0x00
  // Read pin states.
  // State of pin 9 is loaded into the C-flag
  IN   R0, IN_PIN
  // Branch to PinIsHigh when C-flag is set, i.e. pin 9 is HIGH
  BCS  PinIsHigh
PinIsLow:
  // Do something
PinIsHigh:
  // Do something
```

To allow this feature to be used with pins that are not accessed via bit 0 and bit 1 like pin 7 and pin 9, the I/O space provides rotated views of `IN_PIN`.

```
  IN_PIN_ROR4 = 0x04
  // Read pin states.
  // State of pin 11 is loaded into the C-flag
  IN   R0, IN_PIN_ROR4
  // Branch when C-flag is set, i.e. pin 11 is HIGH
  BCS  PinIsHigh
PinIsLow:
  // Do something
PinIsHigh:
  // Do something
```

7.6 Initializing the Speedy core

Speedy is initialized by loading the machine code of a Speedy application into one of the Speedy cores and starting it. The machine code is generated by the Speedy assembler

which outputs a C file with an array that contains the machine code. The following assembly code toggles pin 9 in an endless loop. If the assembly code from chapter *Communication with Speedy* on page 302 is saved in a file named `SpeedyApp.s` and passed to the `SpeedyASM.exe` located in the `Tools` directory, a C file named `SpeedyApp.c` will be created with the following content.

```
static const U16 _aSpeedyApp_Out[] = {
  /* 0000: */   0x97E0,   //   Start:    WWL IN_RX_STAT
  /* 0001: */   0x87C0,   //             IN  R0, IN_RX_DATA
  /* 0002: */   0x97A1,   //             WWH IN_TX_STAT
  /* 0003: */   0xE7C0,   //             OUT R0, OUT_TX_DATA
  /* 0004: */   0xD000,   //             JMP Start
};
```

This file can then be included in the app source code file and used to initialize Speedy. Including the file allows the Speedy application to be modified and translated again without the need to update the app source code.

```
#include "SpeedyApp.c"

void UNI_Init(void) {
  unsigned int NumInstructions;

  // Speedy core must be stopped before loading the machine code
  SPEEDY_Stop(0);
  // Set the frequency of the Speedy to 200 MHz
  SPEEDY_SetFreq(200000000);
  // The machine code consists of 16-bit instruction
  // So we can divide the size of the array by two
  NumInstructions = sizeof(_aSpeedyApp_Out) / 2;
  // Load the machine code into the Speedy
  SPEEDY_WriteProg(0, _aSpeedyApp_Out, NumInstructions);
  // Start the Speedy core
  SPEEDY_Start(0);
}

void UNI_DeInit(void) {
  SPEEDY_Stop(0);
}
```

The above code initializes the Speedy core 0 by stopping it, setting the desired frequency, loading the machine code into Speedy and starting the code. Finally, the Speedy core 0 is stopped again in `UNI_DeInit()`.

7.7 Using Efficient Branches

Speedy supports 2 types of branches, normal branches and delayed branches, which have a "D" at the end of their mnemonic name, e.g. `BEQ` and `BEQD`. Each branch taken, whether delayed or not, requires 3 cycles until Speedy continues with the instruction at the destination address. With normal branches, the 2 additional cycles required behave like NOP instructions. Delayed branches, on the other hand, cause Speedy to execute the two instructions following the branch instruction. This can be used to write more efficient code.

Normal branch

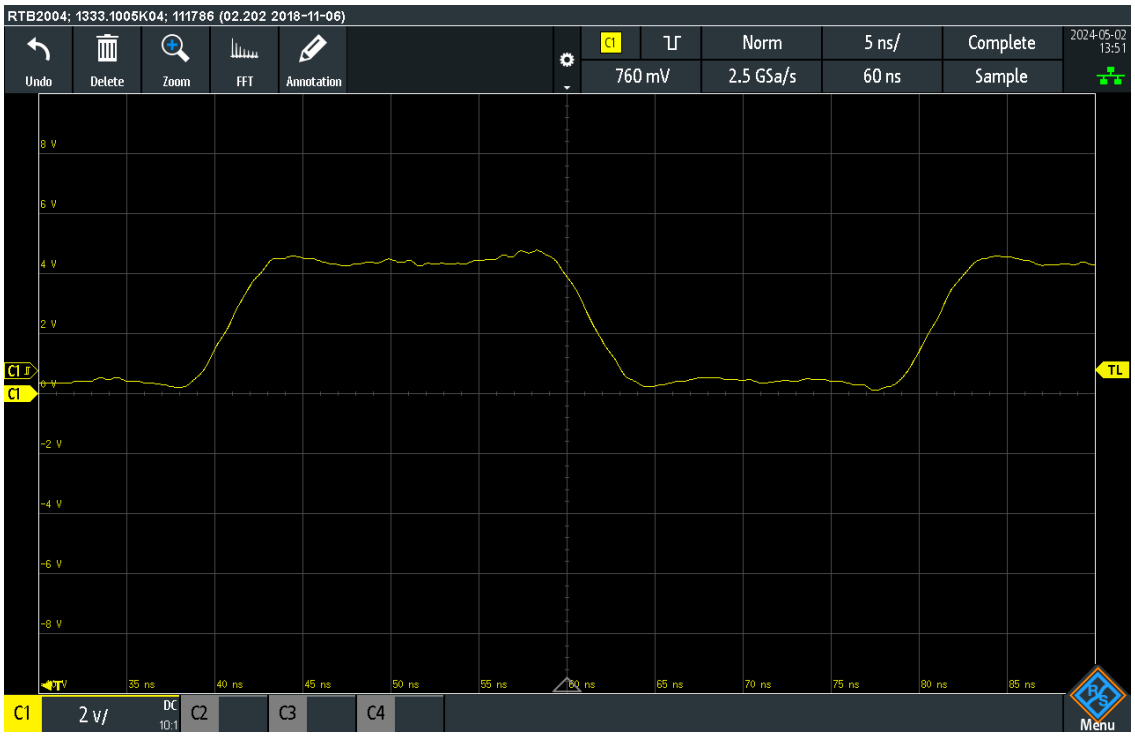
For example, take a look at the following Speedy assembly code. It uses normal branch instructions to toggle pin 9 as fast as possible.

```
  OUT_PIN_TGL       = 0x03
  OUT_PIN_DIR_SET   = 0x11
  OUT_FORCE_DIR_SET = 0x21
  MOV   R0, #0x01             // Mask for pin 9
  OUT   R0, OUT_FORCE_DIR_SET  // Force direction setting for pin 9
```



```
  OUT   R0, OUT_PIN_DIR_SET    // Configure pin 9 for output
Start:
  OUT   R0, OUT_PIN_TGL
  JMP   Start
```

The loop for toggling pin 9 uses only 2 instructions but requires 4 cycles for each iteration. One cycle for the OUT instruction and 3 cycles for the branch. The following screenshot of the oscilloscope shows pin 9 being toggled by the Speedy while running at 200 MHz. Since the time scale of the oscilloscope is set to 5 ns, and each Speedy cycle takes 5 ns, it can be easily seen that the pin is toggled every 4 cycles.

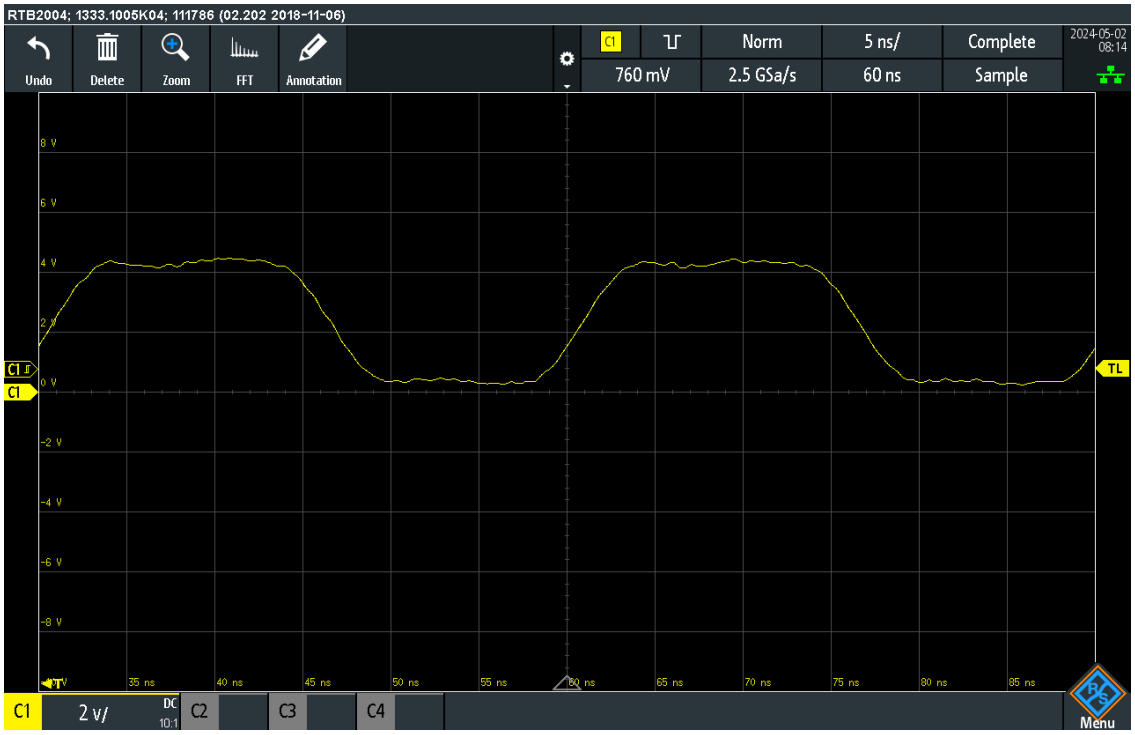


Delayed branch

Now, let's use a delayed branch to toggle the pin.

```
  OUT_PIN_TGL       = 0x03
  OUT_PIN_DIR_SET   = 0x11
  OUT_FORCE_DIR_SET = 0x21
  MOV   R0, #0x01              // Mask for pin 9
  OUT   R0, OUT_FORCE_DIR_SET  // Force direction setting for pin 9
  OUT   R0, OUT_PIN_DIR_SET    // Configure pin 9 for output
Start:
  JMPD  Start
  OUT   R0, OUT_PIN_TGL
  NOP
```

Instead of JMP the JMPD instruction is used, which executes the following two instructions, i.e. OUT and NOP. The NOP is necessary to avoid Speedy from executing a random instruction that is located in the memory. Although the code requires 3 instructions, it takes only 3 cycles, because the OUT instruction is executed in the first delay slot of JMPD. The following screenshot confirms that only 3 cycles are used for one iteration.



7.8 Registers

Each Speedy core contains the following set of registers.

Name	Size	Description
PC	10 bits	Program counter
D	16 bits	Delay register
Rx	8 bits	32 General purpose registers (R0 .. R31)
RA	10 bits	Return address register. This register is used implicitly by the JSR and RET instructions.

7.9 Flags

Each Speedy core has 2 flags, the carry flag (C) and the zero flag (Z). The zero flag is set if the result of an instruction is zero. The carry flag reflects the output of the ALU or the bit shifted out by shift or rotate instructions. The *Instruction set* on page 316 provides detailed information about the instructions that update the flags and how they are updated.

7.10 I/O address space

This chapter describes the implemented special function registers in the I/O space accessible by the IN and OUT instructions. The functionality of the registers changes depending on whether they are read with IN or written with OUT. I/O addresses can be passed either directly as literals to the IN and OUT instructions, or by using Definitions. The identifiers for the definitions can be freely chosen and do not need to match the names used in the following description of the I/O space.

IN I/O space

Addr	Name	Description
0x00	IN_PIN	Current state of the pins (1 = HIGH, 0 = LOW)
0x01	IN_PIN_ROR1	IN_PIN rotated right by 1 bit
0x02	IN_PIN_ROR2	IN_PIN rotated right by 2 bits
0x03	IN_PIN_ROR3	IN_PIN rotated right by 3 bits
0x04	IN_PIN_ROR4	IN_PIN rotated right by 4 bits
0x05	IN_PIN_ROR5	IN_PIN rotated right by 5 bits
0x06	IN_PIN_ROR6	IN_PIN rotated right by 6 bits
0x3D	IN_TX_STAT	Indicates whether the Tx buffer is full
0x3E	IN_RX_DATA	Contains data received from the programming app
0x3F	IN_RX_STAT	Indicates whether the Rx buffer contains data to read

OUT I/O space

Addr	Name	Description
0x00	OUT_PIN	Sets the state of the pins to 1 (HIGH) or 0 (LOW)
0x01	OUT_PIN_SET	If set to 1 the respective pin is set
0x02	OUT_PIN_CLR	If set to 1 the respective pin is cleared
0x03	OUT_PIN_TGL	If set to 1 the respective pin is toggled
0x08	OUT_PIN_9	Sets pin 9 to 1 (HIGH) or 0 (LOW)
0x09	OUT_PIN_7	Sets pin 7 to 1 (HIGH) or 0 (LOW)
0x0A	OUT_PIN_5	Sets pin 5 to 1 (HIGH) or 0 (LOW)
0x0B	OUT_PIN_3	Sets pin 3 to 1 (HIGH) or 0 (LOW)
0x0C	OUT_PIN_11	Sets pin 11 to 1 (HIGH) or 0 (LOW)
0x0D	OUT_PIN_17	Sets pin 17 to 1 (HIGH) or 0 (LOW)
0x10	OUT_PIN_DIR	Configures the pins for input or output
0x11	OUT_PIN_DIR_SET	If set to 1 the respective pin is configured for output
0x12	OUT_PIN_DIR_CLR	If set to 1 the respective pin is configured for input
0x13	OUT_PIN_DIR_TGL	If set to 1 direction of the respective pin is changed
0x20	OUT_FORCE_DIR	Forces the Flasher to use the direction configuration specified for the respective pin in OUT_PIN_DIR
0x21	OUT_FORCE_DIR_SET	If set to 1 forces the Flasher to use the direction configuration specified for the respective pin in OUT_PIN_DIR
0x22	OUT_FORCE_DIR_CLR	If set to 1 the Flasher controls the pin direction
0x23	OUT_FORCE_DIR_TGL	If set to 1 the force direction configuration is changed for the respective pin
0x3E	OUT_TX_DATA	Sends 8 bit of data to the programming app

7.10.1 IN I/O space

7.10.1.1 IN_PIN

Address: 0x00
Size: 8 bits
Description:
Returns the current state of the pins which is either 1 for HIGH or 0 for LOW. Since IN loads Bit[0] into the C-flag and Bit[1] into the Z-flag it is possible to perform a conditional branch depending on the state of pin 9 or pin 7 after reading IN_PIN.

Bit	7	6	5	4	3	2	1	0
Field	PIN_15	PIN_13	PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.1.2 IN_PIN_ROR1

Address: 0x01
Size: 8 bits
Description:
Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 1 bit so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_9	PIN_15	PIN_13	PIN_17	PIN_11	PIN_3	PIN_5	PIN_7

7.10.1.3 IN_PIN_ROR2

Address: 0x02
Size: 8 bits
Description:
Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 2 bits so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_7	PIN_9	PIN_15	PIN_13	PIN_17	PIN_11	PIN_3	PIN_5

7.10.1.4 IN_PIN_ROR3

Address: 0x03
Size: 8 bits
Description:
Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 3 bits so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_5	PIN_7	PIN_9	PIN_15	PIN_13	PIN_17	PIN_11	PIN_3

7.10.1.5 IN_PIN_ROR4

Address: 0x04
Size: 8 bits
Description:

Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 4 bits so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_3	PIN_5	PIN_7	PIN_9	PIN_15	PIN_13	PIN_17	PIN_11

7.10.1.6 IN_PIN_ROR5

Address: 0x05
Size: 8 bits
Description:
Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 5 bits so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9	PIN_15	PIN_13	PIN_17

7.10.1.7 IN_PIN_ROR6

Address: 0x06
Size: 8 bits
Description:
Same as *IN_PIN* on page 309 with the difference that the pins are rotated right by 6 bits so that other pins are loaded into the C-flag and Z-flag that can be used for a conditional branch.

Bit	7	6	5	4	3	2	1	0
Field	PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9	PIN_15	PIN_13

7.10.1.8 IN_TX_STAT

Address: 0x3D
Size: 8 bits
Description:
Indicates whether the Tx buffer is full.
If it is full, the Speedy needs to wait until there is some free space in the Tx buffer before writing data to OUT_TX_DATA.
IN_TX_STAT = 1: Tx buffer is full.
IN_TX_STAT = 0: Tx buffer has space for at least one byte.

7.10.1.9 IN_RX_DATA

Address: 0x3E
Size: 8 bits
Description:
Contains the next byte of the Rx buffer received from the programming app. Check IN_RX_STAT to see if the Rx buffer contains data to read.

7.10.1.10 IN_RX_STAT

Address: 0x3F
Size: 8 bits
Description:
Indicates whether the Rx buffer contains data to read. If it contains data, the data can be read from IN_RX_DATA. IN_RX_STAT is automatically cleared to 0 when the Rx buffer is empty.
IN_RX_STAT = 1: Rx buffer contains at least one byte to read.

`IN_RX_STAT` = 0: Rx buffer is empty.

7.10.2 OUT I/O space

7.10.2.1 OUT_PIN

Address: 0x00
Size: 8 bits
Description:
Sets the state of the pins to 1 (HIGH) or 0 (LOW).

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.2 OUT_PIN_SET

Address: 0x01
Size: 8 bits
Description:
Write 1 to set the state of the pin to HIGH.
Write 0 if the state of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.3 OUT_PIN_CLR

Address: 0x02
Size: 8 bits
Description:
Write 1 to set the state of the pin to LOW.
Write 0 if the state of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.4 OUT_PIN_TGL

Address: 0x03
Size: 8 bits
Description:
Write 1 to toggle the state of the pin.
Write 0 if the state of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.5 OUT_PIN_9

Address: 0x08
Size: 8 bits
Description:
Sets or clears pin 9 by writing 1 or 0 into this register.

7.10.2.6 OUT_PIN_7

Address: 0x09
Size: 8 bits
Description:

Sets or clears pin 7 by writing 1 or 0 into this register.

7.10.2.7 OUT_PIN_5

Address: 0x0A
Size: 8 bits
Description:
Sets or clears pin 5 by writing 1 or 0 into this register.

7.10.2.8 OUT_PIN_3

Address: 0x0B
Size: 8 bits
Description:
Sets or clears pin 3 by writing 1 or 0 into this register.

7.10.2.9 OUT_PIN_11

Address: 0x0C
Size: 8 bits
Description:
Sets or clears pin 11 by writing 1 or 0 into this register.

7.10.2.10 OUT_PIN_17

Address: 0x0D
Size: 8 bits
Description:
Sets or clears pin 17 by writing 1 or 0 into this register.

7.10.2.11 OUT_PIN_DIR

Address: 0x10
Size: 8 bits
Description:
Configures the direction of the respective pin.
If set to 1 the respective pin is configured as an output pin. If set to 0 the respective pin is configured as an input pin.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.12 OUT_PIN_DIR_SET

Address: 0x11
Size: 8 bits
Description:
Write 1 to configure the respective pin for output.
Write 0 if the direction of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.13 OUT_PIN_DIR_CLR

Address: 0x12
Size: 8 bits
Description:
Write 1 to configure the respective pin for input.
Write 0 if the direction of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.14 OUT_PIN_DIR_TGL

Address: 0x13
Size: 8 bits
Description:
Write 1 to toggle the direction of the respective pin.
Write 0 if the direction of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.15 OUT_FORCE_DIR

Address: 0x20
Size: 8 bits
Description:
Forces the Flasher to use the pin directions configured by the Speedy in OUT_PIN_DIR. This is required with custom Speedy applications.
If set to 1 the pin direction configuration overrides the configurations of the Flasher. If set to 0 the Flasher controls the pin direction.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.16 OUT_FORCE_DIR_SET

Address: 0x21
Size: 8 bits
Description:
Write 1 to force the configured pin direction for the respective pin.
Write 0 to not change the force direction configuration.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.17 OUT_FORCE_DIR_CLR

Address: 0x22
Size: 8 bits
Description:
Write 1 to let the FLasher control the pin direction for the respective pin.
Write 0 to not change the force direction configuration.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.18 OUT_FORCE_DIR_TGL

Address: 0x23
Size: 8 bits
Description:
Write 1 to toggle the force direction configuration for the respective pin.
Write 0 if the direction of the pin should not be changed.

Bit	7	6	5	4	3	2	1	0
Field			PIN_17	PIN_11	PIN_3	PIN_5	PIN_7	PIN_9

7.10.2.19 OUT_TX_DATA

Address: 0x3E
Size: 8 bits
Description:
Sends 8 bits of data to the programming app.
This register must no be written as long as IN_TX_STAT indicates that the Tx buffer is full.

7.11 Instruction set

7.11.1 Move

7.11.1.1 MOV D, Rr:Rd

Description

Moves a 16-bit value from two general purpose registers into the D register. Rr is loaded into the upper byte of D and Rd is loaded into the lower byte of D.

Operation

```
D = (Rr << 8) | (Rd << 0);
```

7.11.1.2 MOV Rd, Rr

Description

Copies Rr to Rd.

Operation

```
Rd = Rr;
```

Flags

```
Z = (Result == 0);
```

7.11.1.3 MOV Rd, Imm8

Description

Loads the 8-bit immediate value into Rd.

Operation

```
Rd = Imm8;
```

Flags

```
Z = (Result == 0);
```

7.11.2 Arithmetic

7.11.2.1 SUB Rd, Rr

Description

Subtracts Rr from Rd and stores the result in Rd. The carry flag is set when the result is positive and cleared if it is negative.

SUB can be used to add two numbers by subtracting the two’s complement of the number to add. Furthermore, Literal replacement can be used to let the assembler calculate the two’s complement of the subtrahend. This is done by specifying the subtrahend as a negative number.

The following instruction adds 7 to R0:

```
  SUB R0, #-7
```

Operation

```
Rd = Rd - Rr;
```

Flags

```
C = (Rd - Rr) >= 0;
Z = (Result == 0);
```

7.11.3 Logic

7.11.3.1 AND Rd, Rr

Description

Performs a bitwise AND of Rd and Rr and stores the result in Rd.

Operation

```
Rd = Rd & Rr;
```

Flags

```
Z = (Result == 0);
```

7.11.3.2 OR Rd, Rr

Description

Performs a bitwise OR of Rd and Rr and stores the result in Rd.

Operation

```
Rd = Rd | Rr;
```

Flags

```
Z = (Result == 0);
```

7.11.3.3 EOR Rd, Rr

Description

Performs a bitwise XOR of Rd and Rr and stores the result in Rd.

Operation

```
Rd = Rd ^ Rr;
```

Flags

```
Z = (Result == 0);
```


7.11.3.4 LSL Rd

Description

Shifts Rd one bit to the left and stores the result in Rd.

Operation

```
Rd = Rd << 1;
```

Flags

```
C = Rd[7];
Z = (Result == 0);
```

7.11.3.5 LSR Rd

Description

Shifts Rd one bit to the right and stores the result in Rd.

Operation

```
Rd = Rd >> 1;
```

Flags

```
C = Rd[0];
Z = (Result == 0);
```

7.11.3.6 ROL Rd

Description

Rotates the bits in Rd one bit to the left and stores the result in Rd. The carry flag is shifted in from the right and bits shifted out are stored in the carry flag.

Operation

```
Tmp   = C;
C     = Rd[7];
Rd    = Rd << 1;
Rd[0] = Tmp;
```

Flags

```
C = Rd[7];
Z = (Result == 0);
```

7.11.3.7 ROR Rd

Description

Rotates the bits in Rd one bit to the right and stores the result in Rd. The carry flag is shifted in from the left and bits shifted out are stored in the carry flag.

Operation

```
Tmp   = C;
```



```
C     = Rd[0];
Rd    = Rd >> 1;
Rd[7] = Tmp;
```

Flags

```
C = Rd[0];
Z = (Result == 0);
```

7.11.4 I/O

7.11.4.1 IN Rd, <InAddr>

Description

Reads one byte via the I/O bus from the specified address and stores it in Rd.

Operation

```
Rd = [<InAddr>];
```

Flags

```
C = Rd[0];
Z = Rd[1];
```

7.11.4.2 OUT Rd, <OutAddr>

Description

Writes Rd via the I/O bus to the specified address.

Operation

```
[<OutAddr>] = Rd;
```

7.11.4.3 WWL <InAddr>

Description

Waits while the first bit of the data read via the I/O bus from the specified address is 0.

Operation

```
while (<InAddr>[0] == 0);
```

7.11.4.4 WWLT <InAddr>

Description

Waits while the D register is greater than 0 and the first bit of the data read via the I/O bus from the specified address is 0. The D register is decremented with each cycle. When the D register reaches 0 and the Speedy continues with the next instruction on the next cycle, D is reloaded with its initial value.

Operation

```
Tmp = D;
```



```
while ((<InAddr>[0] == 0) && (D-- != 0));
if (D == 0) D = Tmp;
```

7.11.4.5 WWH <InAddr>

Description

Waits while the first bit of the data read via the I/O bus from the specified address is 1.

Operation

```
while (<InAddr>[0] == 1);
```

7.11.4.6 WWHT <InAddr>

Description

Waits while the D register is greater than 0 and the first bit of the data read via the I/O bus from the specified address is 1. The D register is decremented with each cycle. When the D register reaches 0 and the Speedy continues with the next instruction on the next cycle, D is reloaded with its initial value.

Operation

```
Tmp = D;
while ((<InAddr>[0] == 1) && (D-- != 0));
if (D == 0) D = Tmp;
```

7.11.5 Control flow

7.11.5.1 BCC <Addr>

Description

Takes the branch if the C-flag is 0. This instruction requires 3 cycles if the branch is taken and 1 cycle if the branch is not taken.

Operation

```
PC = (C == 0) ? <Addr> : PC + 1;
```

7.11.5.2 BCCD <Addr>

Description

Takes the branch if the C-flag is 0. The next two instructions after `BCCD` will be executed regardless of whether the branch is taken or not.

Operation

```
PC = (C == 0) ? <Addr> : PC + 1;
```

7.11.5.3 BCS <Addr>

Description

Takes the branch if the C-flag is 1. This instruction requires 3 cycles if the branch is taken and 1 cycle if the branch is not taken.

Operation

```
PC = (C == 1) ? <Addr> : PC + 1;
```

7.11.5.4 BCSD <Addr>

Description

Takes the branch if the C-flag is 1. The next two instructions after `BCSD` will be executed regardless of whether the branch is taken or not.

Operation

```
PC = (C == 1) ? <Addr> : PC + 1;
```

7.11.5.5 BEQ <Addr>

Description

Takes the branch if the Z-flag is 1. This instruction requires 3 cycles if the branch is taken and 1 cycle if the branch is not taken.

Operation

```
PC = (Z == 1) ? <Addr> : PC + 1;
```

7.11.5.6 BEQD <Addr>

Description

Takes the branch if the Z-flag is 1. The next two instructions after `BEQD` will be executed regardless of whether the branch is taken or not.

Operation

```
PC = (Z == 1) ? <Addr> : PC + 1;
```

7.11.5.7 BNE <Addr>

Description

Takes the branch if the Z-flag is 0. This instruction requires 3 cycles if the branch is taken and 1 cycle if the branch is not taken.

Operation

```
PC = (Z == 0) ? <Addr> : PC + 1;
```

7.11.5.8 BNED <Addr>

Description

Takes the branch if the Z-flag is 0. The next two instructions after `BNED` will be executed regardless of whether the branch is taken or not.

Operation

```
PC = (Z == 0) ? <Addr> : PC + 1;
```


7.11.5.9 JMP <Addr>

Description

Takes the branch to the specified address. This instruction requires 3 cycles.

Operation

```
PC = <Addr>;
```

7.11.5.10 JMPD <Addr>

Description

Takes the branch to the specified address. The next two instructions after `JMPD` will be executed before the branch is taken.

Operation

```
PC = <Addr>;
```

7.11.5.11 JSR <Addr>

Description

Stores the next instruction, i.e. the return address, in RA and takes the branch into the subroutine. This instruction requires 3 cycles.

Operation

```
LR = PC + 1;
PC = <Addr>;
```

7.11.5.12 RET

Description

Returns from a subroutine by jumping to the address stored in RA. The return address is automatically loaded into RA by the JSR instruction.

Operation

```
PC = RA;
```

7.11.6 Miscellaneous

7.11.6.1 CMP Rd, Rr

Description

Compares two registers by subtracting Rr from Rd and updating the flags accordingly.

Operation

```
Tmp = Rd - Rr;
```


Flags

```
C = (Rd - Rr) >= 0;
Z = (Result == 0);
```

7.11.6.2 DLY

Description

Delays the Speedy’s execution by decrementing the D register each cycle by one until it is zero. If D is zero, the Speedy continues with the next instruction and reloads D with the value it contained when the DLY instruction was started. Max. delay is 65536 cycles which equates to 327.68 us at 200 MHz.

Operation

```
Tmp = D;
while (D--);
D = Tmp;
```

7.11.6.3 NOP

Description

No operation.

Operation

```
PC = PC;
```


7.12 Assembler

The Speedy assembler (`SpeedyASM.exe`) located in the `Tools` directory is used to translate an assembly file for the Speedy. The assembler outputs 2 files, a binary file with the Speedy machine code and a C file containing an array that is initialized with the Speedy machine code. The content of the C file is used in the programming app source code file to load the Speedy machine code into the Speedy by passing the array with the machine code to `SPEEDY_WriteProg()`.

The file `SpeedyApplication.s`, which is located in the `Template` directory can be used as a starting point for writing custom Speedy applications.

7.12.1 Comments

The assembler supports C/C++ comments.

```
/*
  C comment
*/
  NOP  // C++ comment
```

7.12.2 Labels

In order to jump to specific instructions or subroutines, the address of the instruction or subroutine must be passed to the branch instructions. Since the addresses of instructions are not clearly visible in an assembly file and since the position of instructions may change, labels are used to mark specific locations in the code. These labels can then be passed instead of actual addresses to the branch instructions.

```
MyLabel:
  NOP
  NOP
  JMP MyLabel
```

7.12.3 Macros

The Speedy assembler is able to process macros.

```
MACRO Read Reg, Port
  WWL .Port.
  IN  .Port.+1, .Reg.
ENDM

Read R0, 1
Read R2, 7
```

Macro parameters are expanded by surrounding them with periods. Arguments passed to macros are separated by commas. If a single argument already contains a comma, the argument must be quoted.

```
DoSomething "R1, R2"
ReadReg r0, "X+1"
```

7.12.4 Repetitions

The assembler is able to repeat a set of instructions within a REPT-ENDR block. The following code results in 10 consecutive NOP instructions.

```
REPT 10
  NOP
```



```
ENDR
```

Inside a repetition block `.COUNT.` and `.LABEL.` can be used to generate numbers. `.COUNT.` expands to a number starting at zero that is incremented with every repetition of the same block. `.LABEL.` expands to a number starting at zero that is unique for all repetitions of the same block and is incremented with each subsequent repetition block.

```
   REPT 2
     BNE L.LABEL..COUNT.
     NOP
L.LABEL..COUNT.:
     NOP
   ENDR

   REPT 2
     BNE L.LABEL..COUNT.
     NOP
L.LABEL..COUNT.:
     NOP
   ENDR
```

Resulting output:

```
static const U16 _aSpeedyApplication_Out[] = {
  /* 0000: */   0xA802,   //             BNE L10
  /* 0001: */   0x0000,   //             NOP
  /* 0002: */   0x0000,   //   L10:      NOP
  /* 0003: */   0xA805,   //             BNE L11
  /* 0004: */   0x0000,   //             NOP
  /* 0005: */   0x0000,   //   L11:      NOP
  /* 0006: */   0xA808,   //             BNE L20
  /* 0007: */   0x0000,   //             NOP
  /* 0008: */   0x0000,   //   L20:      NOP
  /* 0009: */   0xA80B,   //             BNE L21
  /* 000A: */   0x0000,   //             NOP
  /* 000B: */   0x0000,   //   L21:      NOP
};
```

7.12.5 Literal replacement

The Speedy instruction set supports immediate values only with the MOV instruction. All other instructions operate on registers or addresses. However, the assembler allows the usage of literal values as it replaces the literal by a general purpose register that is previously loaded with the value of the literal. For each literal value a general purpose register is reserved by the assembler and initialized with the value of the literal at the start of the application. This is only possible as long as enough unused general purpose registers are available for the assembler to use. If not enough registers are available, an error is thrown by the assembler. If the assembler succeeds translating the assembly file, it prints the number of registers used by the program and the number of registers used by the assembler as constants for literals.

The following program uses 2 registers (R0 and R1) and 3 registers (R29-R31) are used by the assembler for the literals.

```
  SUB R0, #1
  SUB R1, #2
  SUB R1, #3
```

```
SEGGER Speedy Macro Assembler 1.1.0 compiled Sep 22 2021 23:50:30
Copyright (c) 2021 SEGGER Microcontroller GmbH    www.segger.com

Register use
  Program    R0-R1 (2 registers)
```

```
  Constants  R29-R31 (3 registers)
  Free       R2-R28 (27 registers)

Code use
  Program    3 words
  Constants  3 words
  Total      6 words

No errors
```

7.12.6 Entry point

The Speedy core starts execution at the first instruction of the application code. In most cases it is the first instruction used in the assembly file. If Literal replacement is used, either explicitly or implicitly via Pseudo instructions, the code for the initialization of the constant registers is placed before the first instruction in the assembly file.

7.12.7 Pseudo instructions

The assembler provides the pseudo instructions `INC` and `DEC` for incrementing and decrementing a register by one. Both instructions expand to `SUB` instructions using literals which require the assembler to reserve registers for the literal values.

```
  INC R0  // Expands to SUB R0, #0xFF
  DEC R1  // Expands to SUB R1, #0x01
```

7.12.8 Definitions

Definitions can be used to hide literal values like I/O addresses behind identifiers. The following code shows the usage of definitions for addresses to facilitate the access to the I/O space.

```
  // Definitions for I/O addresses.
  PORT_SET_PIN     = 0x01
  PORT_CLR_PIN     = 0x02
  PORT_DCC_DN_FIFO = 0x3E
  // Load mask with pins to set and clear from Down FIFO.
  in   R1, PORT_DCC_DN_FIFO
  // Set pins.
  out  R1, PORT_SET_PIN
  // Clear pins.
  out  R1, PORT_CLR_PIN
```


Chapter 8

Flasher App Builder

This chapter contains the documentation for the Flasher App Builder Utility.

8.1 Introduction

Flasher App Builder (FlasherAppBuilderCL.exe) is the development environment for Flasher Apps. It can make, load, run, and debug Flasher Apps.

General

When running an app on the Flasher, the PEX file is loaded directly from the host computer and thus doesn't need to reside in the Flasher's internal storage. When a PEX file is loaded by Flasher App Builder, an address space for the app is allocated, which persists until a new app is loaded. This means that all functions executed by Flasher App Builder operate on the same data, and functions can rely on the data modified by previous functions executed by Flasher App Builder.

Build configurations

The Flasher App Builder includes two build configurations for debug and release builds. If the debug configuration is active, the Flasher App Builder will build a single executable file for debugging purposes.

Executable file name	Description

`<Filename>_D.pex`	Descriptive build including debug code. Defines:
	`DESCRIPTIVE = 1`
	`DEBUG = 1`

If the release configuration is active, the Flasher App Builder will build two executable files exluding debug code.

Executable file name	Description
`<Filename>.pex`	Non-descriptive build without debug code.
`<Filename>_D.pex`	Descriptive build without debug code. Defines: `DESCRIPTIVE = 1`

While descriptive builds (`*_D.pex`) include data and code for all verbosity levels, non-descriptive builds (*.pex) only include data and code for verbosity level "None".

Programming app specifics

Programming apps require to load configuration data from the programming app's configuration file (*.uni). To allow easily allow testing of programming apps, the `SYS_LoadUNI()` function can be used to load a configuration file stored on the Flasher while app execution.

8.2 Commands

Command	Description
Flasher connection	
`connect`	Connect to a Flasher.
`disconnect`	Disconnect the Flasher.
File management	
`cd`	Show or change the local working directory.
`dir`	Show local files.
`rdir`	Show remote files.
`put`	Download file to the Flasher.
Build + run	
`sc`	Show or select build configuration.
`make`	Compile an app source file
`load`	Load an app into the Flasher’s RAM
`run`	Run a function of the loaded app.
`go`	Make, load and run the specified app, function main(void).
Debug	
`mem`	Read data from the app’s memory space.
`w1`	Write 8-bit item(s) to the app’s memory space.
`w2`	Write 16-bit item(s) to the app’s memory space.
`w4`	Write 32-bit item(s) to the app’s memory space.
Miscellaneous	
`?`	Show information about all or specific commands.
`exit`	Close the Flasher connection and quit.

8.2.1 connect

Description

Connect to a Flasher (default is USB). If only one Flasher is connected via USB, it will automatically be selected.

Syntax

```
connect [usb [<SN | Nickname>] | ip [<IPAddr | RemoteServerString>]]
```

Parameters

Parameter	Description
usb	Connect via USB (via SN or nickname).
ip	Connect via IP (via IP address or remote server name).

Example

```
Flasher>connect
Flasher>connect usb 167000000
Flasher>connect usb MyFlasher
Flasher>connect ip 192.168.1.15
Flasher>connect ip MyFlasherCon
```


8.2.2 disconnect

Description

Disconnects the currently connected Flasher.

Syntax

```
disconnect
```


8.2.3 cd

Description

Show or change the local working directory.

Syntax

```
cd [<Path>]
```

Parameters

Parameter	Description
Path	Path of the working directory.

Example

```
Flasher>cd
C:\FlasherSDK\Sample\HelloWorld
Flasher>cd ../Sample/HelloWorld
```


8.2.4 put

Description

Download a file to the file system of the connected Flasher.

Syntax

```
put [<Destination on Flasher>] <File>
```

Parameters

Parameter	Description
Destination on Flasher	Destination on the Flasher file system for the file.
File	File name of the file to be downloaded to the Flasher.

Example

```
Flasher>put HelloWorld_D.pex
```


8.2.5 sc

Description

Select the build configuration, “Debug” or “Release”.

Syntax

```
sc [<Name>]
```

Parameters

Parameter	Description
Name	Build configuration to select (options are “Debug” or “Release”).

Example

```
Flasher>sc
Selected build configuration: 'Debug'
Flasher>sc Release
```


8.2.6 make

Description

Compiles an app source code file.

Using this function, program size information will be printed.

Syntax

```
make [[<Options>] <File>]
```

Parameters

Parameter	Description
options	S32CC compiler options (see *S32 Compiler* on page 295).
File	App source code filename.

Example

```
Flasher>make
Flasher>make HelloWorld.pc
```


8.2.7 load

Description

Loads an app into the Flasher's RAM.

While the app is loaded, functions can be executed, and the app memory can be read and written.

Syntax

```
load <File>
```

Parameters

Parameter	Description
File	Path to the app executable to load.

Example

```
Flasher>load
Flasher>load HelloWorld_D.pex
```


8.2.8 run

Description

Run a function of the loaded app. The default function is `main(void);`

The Flasher then returns the execution time of the function, as well as usually the S32 registers R0, PC, SP and SR. R0 contains the value returned by the executed function. The called function must have external linkage.

Syntax

```
run [<Func> [<IntParaN> ...]] (N < 16)
```

Parameters

Parameter	Description
Func	The name of the function to be called
IntParaN	An integer argument passed to the function. Multiple arguments can be passed to functions (N < 16). Arguments are interpreted as decimal number. To use hexadecimal number, use the "0x" prefix.

Example

```
void Test(U32 Delay) {
  SYS_Sleep_ms(Delay);
}
```

```
Flasher>run Test 500
```


8.2.9 go

Description

Make, load, and run the specified app, starting at function main(void)

Syntax

```
go [<File>]
```

Parameters

Parameter	Description
File	App source code file to be compiled, loaded into the Flasher, and executed

Example

```
go C:/FlasherSDK/Samples/UART/UART.pc
```


8.2.10 mem

Description

Read app memory.

Reads the specified number of bytes at the specified address from the programming app's memory. The address must be within the programming app's address space. No data is read if the address points outside the programming app's address space. The Flasher stops reading data when it reaches the end of the programming app's address space.

Syntax

```
mem <address> <num bytes>
```

Parameters

Parameter	Description
address	The address from which the data is to be read. This parameter is interpreted as a hexadecimal value.
num bytes	The number of bytes to read from <address>. This parameter is interpreted as a hexadecimal value.

Example

The following example will read the entire memory allocated for the programming app.

```
mem 0 10000
```


8.2.11 w1

Description

Write memory in bytes.

Writes at least one byte into the programming app's memory. If only a nibble is provided, the most significant bits will be filled with zeros.

The address must be within the programming app's address space. The Flasher stops writing data when it reaches the end of the programming app's address space.

Syntax

```
w1 <address> <data> [data]...
```

Parameters

Parameter	Description
address	The address where the data is to be written. This parameter is interpreted as a hexadecimal value.
data	The byte to be written to the address. Multiple bytes to write can be provided. This parameter is interpreted as a hexadecimal value.

Example

The following example writes the value `0x12` at `0xA0` and the value `0x98` at `0xA1`.

```
w1 a0 12 98
```


8.2.12 w2

Description

Write memory in halfwords.

Writes one or more words consisting of two bytes into the programming app’s memory. The data is written with little endianness into the memory. That means that the least significant byte is stored at the highest memory address. If not a whole word is provided, the most significant bits will be filled with zeros

The address must be within the programming app’s address space. The Flasher stops writing data when it reaches the end of the programming app’s address space. Words crossing the address space boundary are cut off.

Syntax

```
w2 <address> <data> [data]...
```

Parameters

Parameter	Description
address	The address where the data is to be written. This parameter is interpreted as a hexadecimal value.
data	The word consisting of two bytes to be written to the address. Multiple words to write can be provided. This parameter is interpreted as a hexadecimal value.

Example

The following example writes the value `0x1234` at address `0xA0` and the value `0x9876` at address `0xA2`.

```
w2 a0 1234 9876
```


8.2.13 w4

Description

Write memory is words.

Writes one or more double words consisting of four bytes into the programming app's memory. The data is written with little endianness into the memory. That means that the least significant byte is stored at the highest memory address. If not a whole double word is provided, the most significant bits will be filled with zeros

The address must be within the programming app's address space. The Flasher stops writing data when it reaches the end of the programming app's address space. Double words crossing the address space boundary are cut off.

Syntax

```
w4 <address> <data> [data]...
```

Parameters

Parameter	Description
address	The address where the data is to be written. This parameter is interpreted as a hexadecimal value.
data	The double word consisting of four bytes to be written to the address. Multiple double words to write can be provided. This parameter is interpreted as a hexadecimal value.

Example

The following example writes the value `0x12345678` at `0xA0` and the value `0x98765432` at `0xA4`.

```
w4 a0 12345678 98765432
```


8.2.14 help (?)

Description

Shows information about all or specific commands.

Syntax

```
? [<Command>]
```

Parameters

Parameter	Description
Command	Command to show information about

Example

```
Flasher>? run
----------------------
Command long name: run
Command short name: run
Syntax: run [<Func> [<IntPara0> ... <IntParaN>]] (N < 16)
Function: Run a function of the loaded app (Default is main(void))
Requires connection to Flasher: yes
----------------------
```


8.2.15 exit

Description

Close the connection to the Flasher and quit.

Syntax

```
exit
```

Parameters

None

Chapter 9

Glossary

This chapter explains important terms used throughout this manual.

TIF
Target interface, the interface between the Flasher and the target.

Application Program Interface
A specification of a set of procedures, functions, data structures, and constants that are used to interface two or more software components together.

Big-endian
Memory organization where the least-significant byte of a word is at a higher address than the most significant byte. See also *Little-endian*.

Cache cleaning
The process of writing dirty data in a cache to main memory.

Coprocessor
An additional processor that is used for certain operations, for example, for floating-point math calculations, signal processing, or memory management.

Dirty data
When referring to a processor data cache, data that has been written to the cache but has not been written to main memory. Only write-back caches can have dirty data, because a write-through cache writes data to the cache and to main memory simultaneously. The process of writing dirty data to main memory is called cache cleaning.

Dynamic Linked Library (DLL)
A collection of programs, any of which can be called when needed by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

EmbeddedICE
The additional hardware provided by debuggable ARM processors to aid debugging.

Host
A computer which provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

ICache
Instruction cache.

ICE Extension Unit
A hardware extension to the EmbeddedICE logic that provides more breakpoint units.

ID
Identifier.

IEEE 1149.1
The IEEE Standard which defines TAP. Commonly (but incorrectly) referred to as JTAG.

Image
An executable file that has been loaded onto a processor for execution.

In-circuit emulator
a device enabling access to and modification of the signals of a circuit while that circuit is operating. Commonly abbreviated to *ICE*.

Instruction Register
When referring to a TAP controller, a register that controls the operation of the TAP.

IR
See *Instruction Register*.

Joint Test Action Group (JTAG)
The name of the standards group which created the IEEE 1149.1 specification.

Little-endian
Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.

Memory coherency
A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult

when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.

Memory management unit (MMU)
Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.

Memory Protection Unit (MPU)
Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.

Multi-ICE
Multi-processor EmbeddedICE interface. ARM registered trademark.

nSRST
Abbreviation of System Reset. The electronic signal which causes the target system other than the TAP controller to be reset. This signal is known as nSYSRST in some other manuals. See also *nTRST*.

nTRST
Abbreviation of TAP Reset. The electronic signal that causes the target system TAP controller to be reset. This signal is known as nICERST in some other manuals. See also nSRST.

Open collector
A signal that may be actively driven low by one or more drivers, and is otherwise passively pulled high. Also known as a "wired AND" signal.

Processor core
The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Program Status Register (PSR)
Contains some information about the current program and some information about the current processor. Often, therefore, also referred to as Processor Status Register. Is also referred to as *Current PSR (CPSR)*, to emphasize the distinction between it and th *Saved PSR (SPSR)*. The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

Remapping
Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.

Scan chain
Scan chain: definition
{A group of one or more registers from one or more TAP controllers connected between TDI and TDO, through which test data is shifted.}

RAMCode
Code that is executed directly from RAM to execute specific functions on the target itself.

TAP controller
Logic on a device which allows access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

Target
The actual processor (real silicon or simulated) on which the application program is running.

TCK
The electronic clock signal which times data on the TAP data lines TMS, TDI, and TDO.

TDI
The electronic signal input to a TAP controller from the data source (upstream). Usually this is seen connecting the Multi-ICE Interface Unit to the first TAP controller.

TDO
The electronic signal output from a TAP controller to the data sink (downstream). Usually this is seen connecting the last TAP controller to the Multi-ICE Interface Unit.

Test Access Port (TAP)
The port used to access a device's TAP Controller. Comprises TCK, TMS, TDI, TDO and (optionally) nTRST.

Transistor-transistor logic (TTL)
A type of logic design in which two bipolar transistors drive the logic output to one or zero. LSI and VLSI logic often used TTL with high logic level approaching +5V and low approaching 0V.

Word
A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Chapter 10

References

This chapter lists documents, which we think may be useful to gain deeper understanding of technical details.

Reference	Title	Comments
[JTAG]	IEEE Std 1149.1-2001 (Revision of IEEE Std 1149.1-1990) IEEE Standard Test Access Port and Boundary-Scan Architecture	Defines the JTAG standard. A bit difficult to read.
[SWD]	ARM Debug Interface v5 (Issue A, February 2006)	Specifies the ARM Debug Interface v5 Architecture (includes the SWD protocol).

Chapter 11

Indexes

11.1 Subject index

