# STM8S Toolset & Firmware Library

STM8S Technical Training

# Purpose and objective

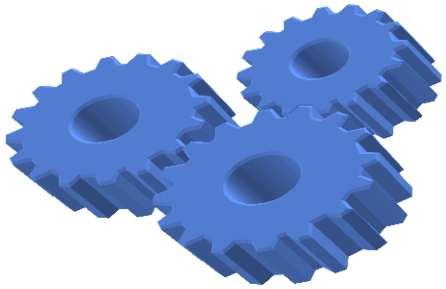
- The purpose of this chapter is to introduce you
  - New set of tools available to develop and debug your application
  - Firmware library available
- The objective is to help you to select the right tools for your needs

# Goal of the firmware library

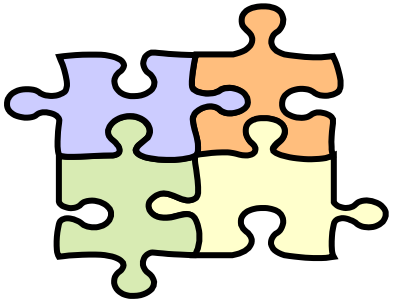
- To provide an easy way to start using STM8 microcontroller
- To be compliant with the industry standard of coding rules
- To validate the silicon in application environment

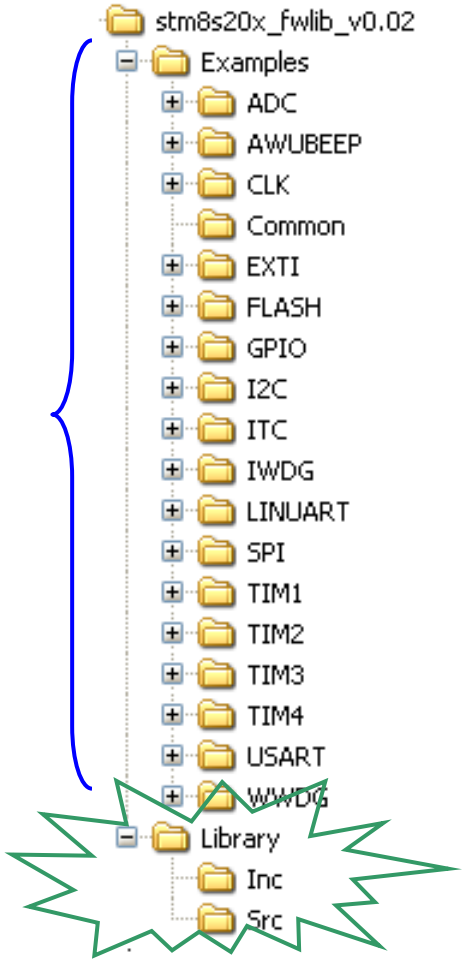
# STM8 Firmware Library Features

STM8 Library
- All in Strict ANSI-C
- MISRA C 2004 Compliant
- STVD tool chain + Cosmic compiler
- Self documented



STM8 Examples
- *Set of examples for each STM8 peripheral*
- Running on STMicroelectronics STM8 evalboard and can be easily tailored to any other hardware

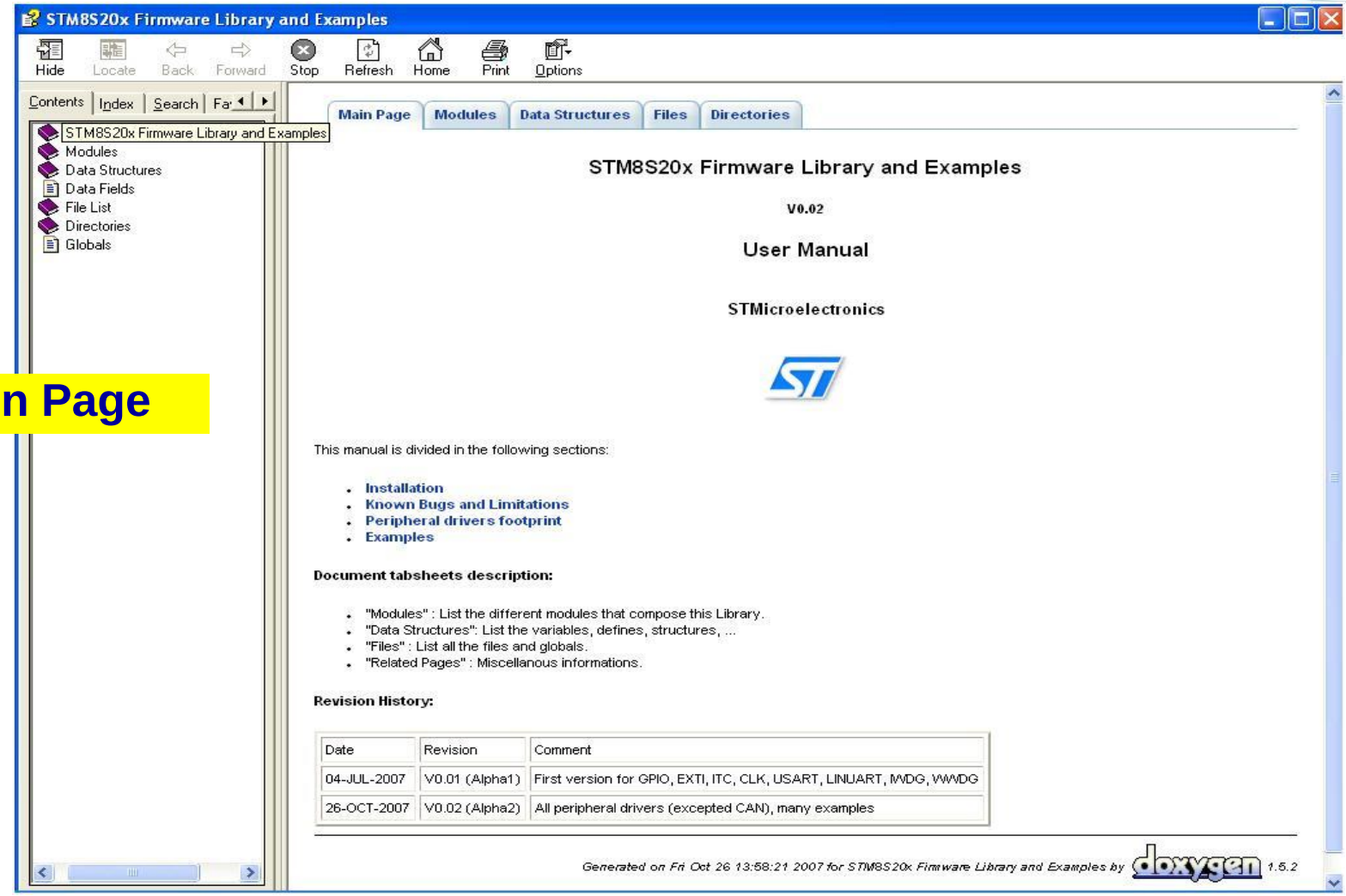




# STM8 Firmware Library User Manual (1/4)

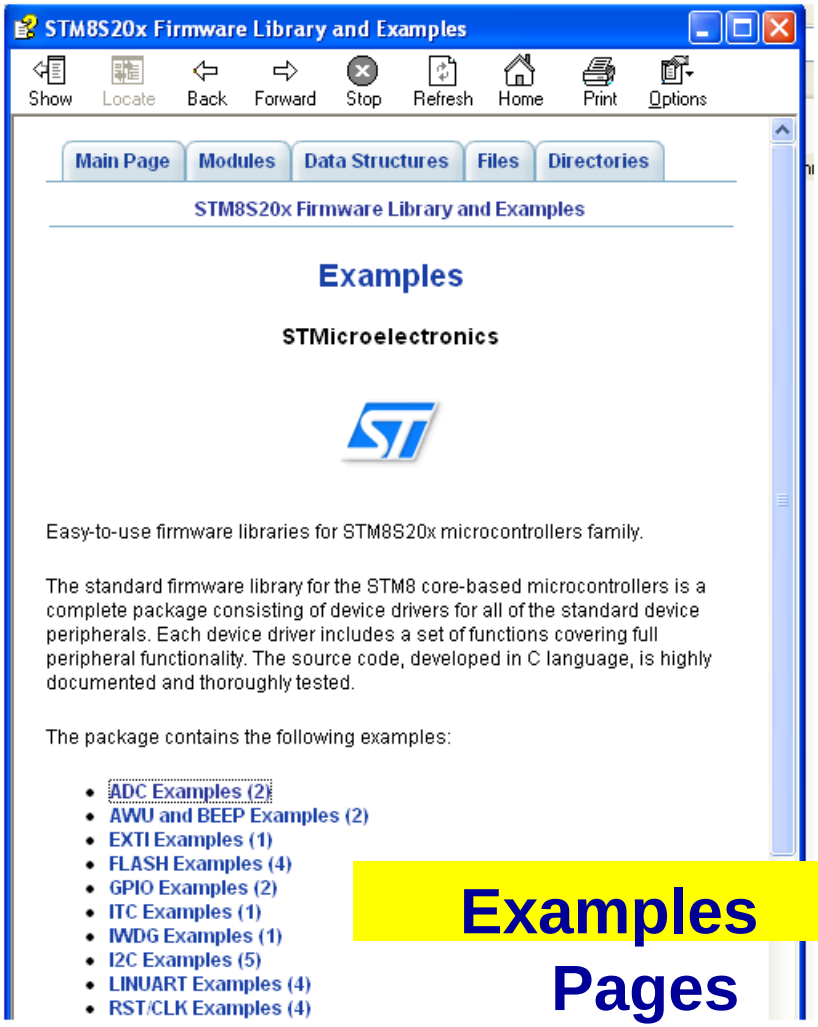
- CHM file directly created from comments present in the firmware
  - Thanks to Doxygen software
  - User Manual is fully in line with firmware comments
- All functions and examples are described
- Direct link to functions and variables description and to code

# STM8 Firmware Library User Manual (2/4)

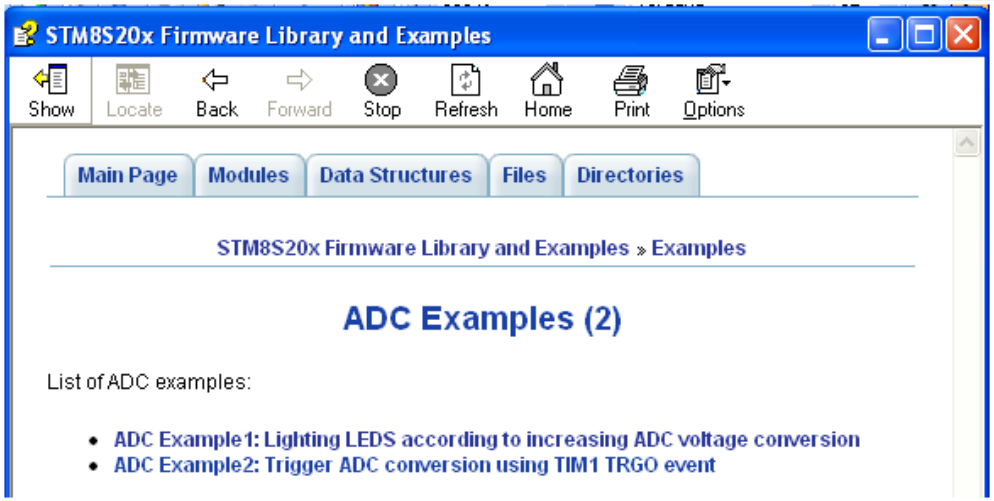



**Main Page**

# STM8 Firmware Library User Manual (3/4)

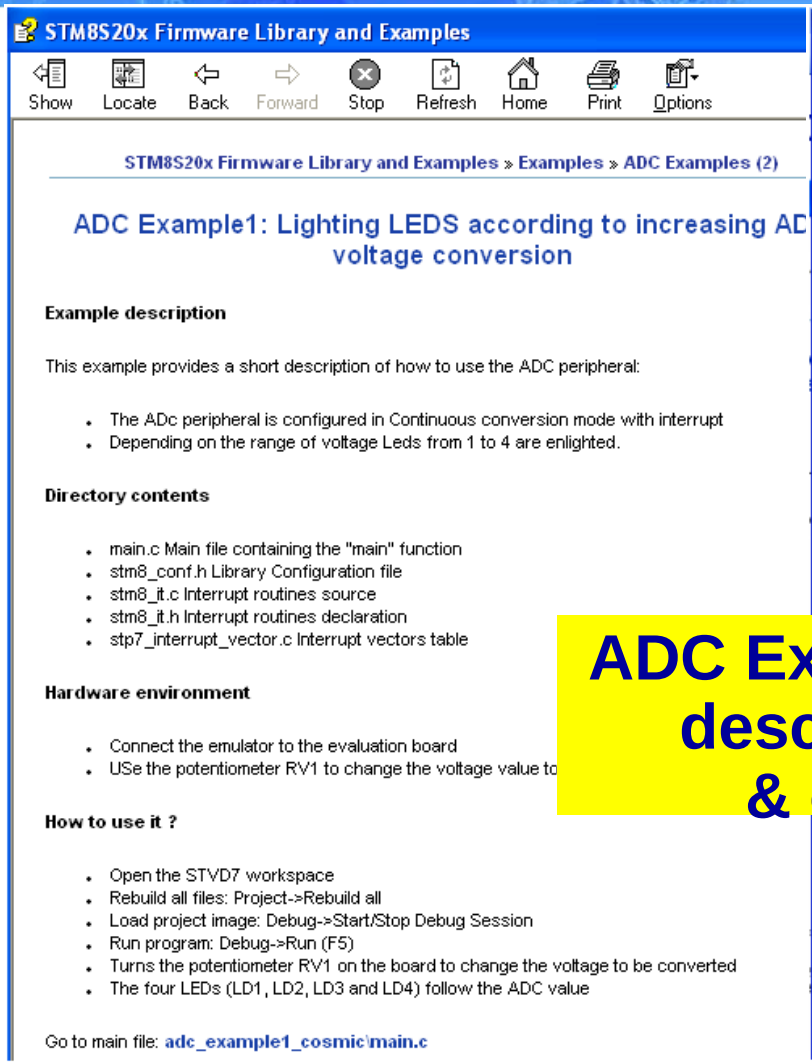


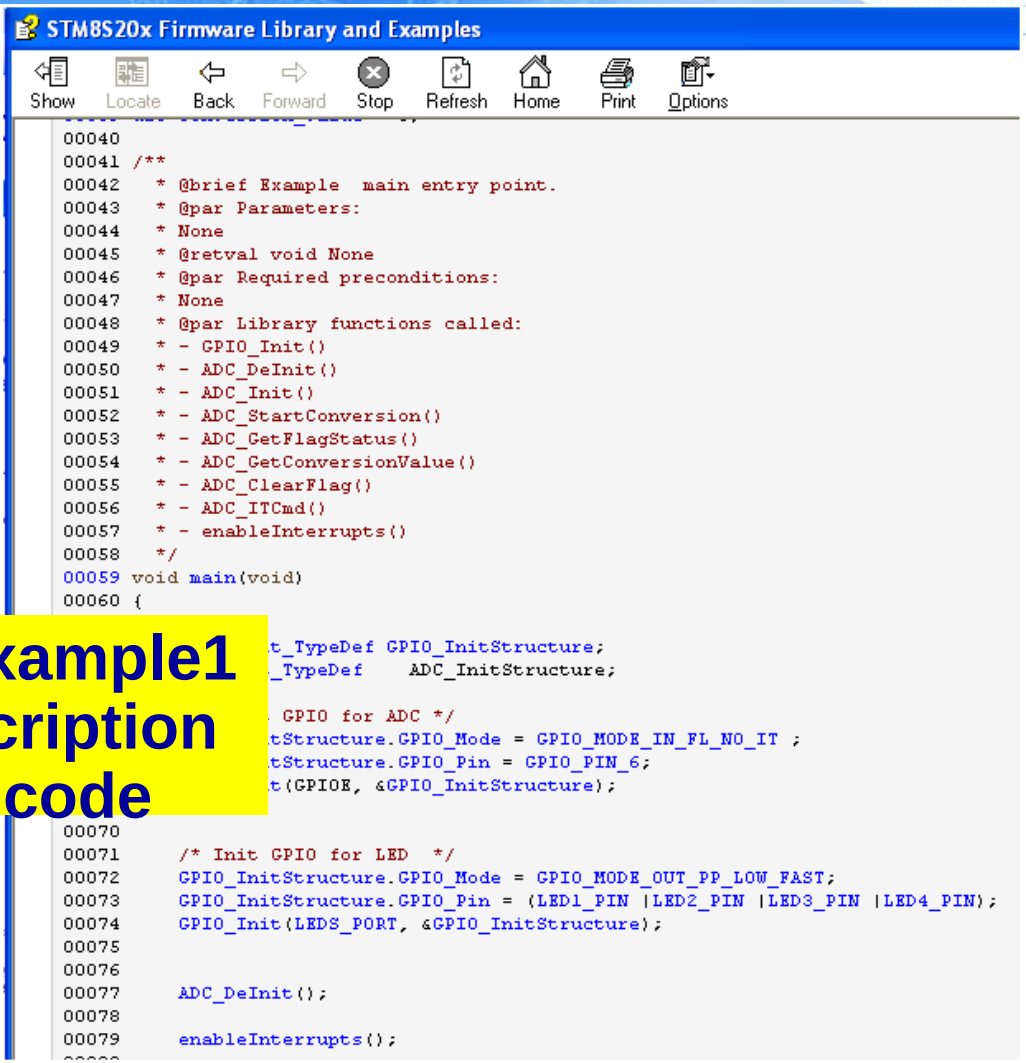

**Examples Pages**




**ADC Examples Pages**

# STM8 Firmware Library User Manual (4/4)



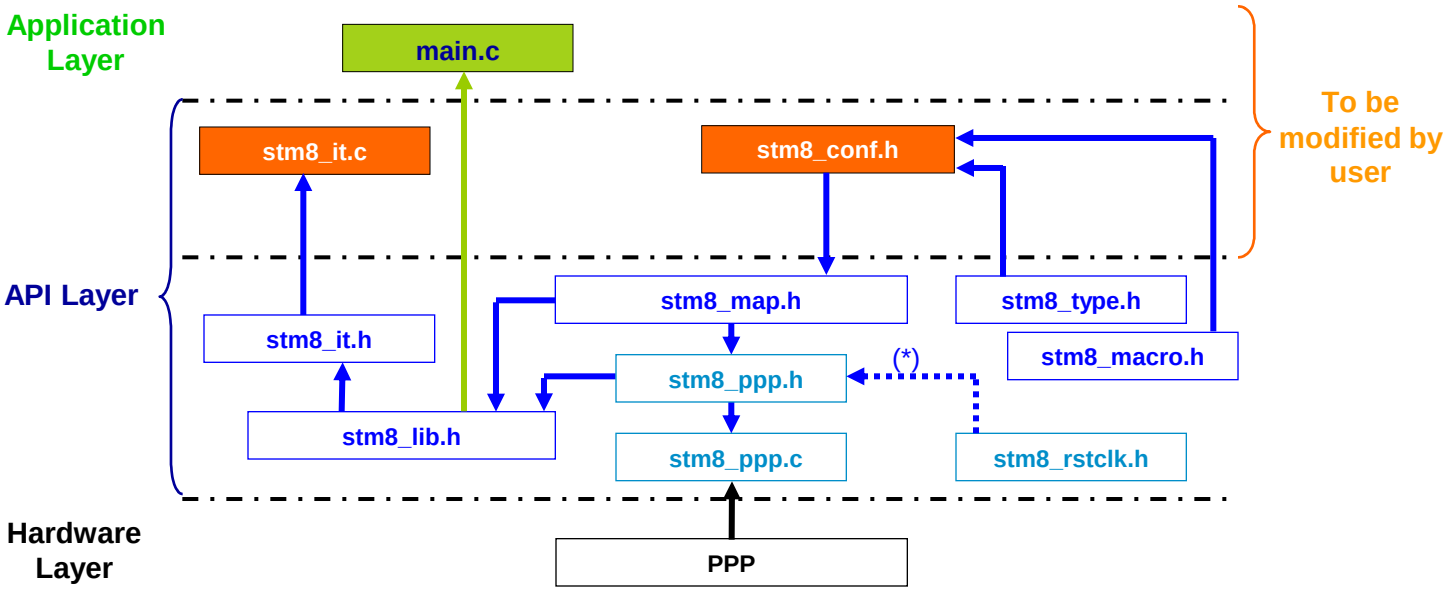

**ADC Example1 description & code**

# Coding conventions

- All firmware is coded in ANSI-C
  - Strict ANSI-C for all library peripheral files and examples
  - MISRA C 2004 compliant (checked by PC-LINT software)
- *PPP* is used to reference any peripheral acronym, e.g. *TIM1* for Timer1.
- Registers & Structures
  - STM8 registers are mapped in the microcontroller address space
    - FW library registers have the same names as in STM8 Datasheets & reference manuals.
  - All registers hardware accesses are performed through a C structures :
    - Work with only one base address and indirect addressing
    - Improve code re-use : e.g. the same structure to handle and initialize the same peripherals (i.e. GPIO)

# Library Structure

- Common types and constants
- Peripheral registers' addresses
  - Organized in structures with a base address
- Peripherals pointers initialization if #DEBUG defined
- Peripheral API code :
  - Bit fields / Masks
  - Peripheral user Structure
  - Low-level & API functions to perform basic operations offered by the peripheral
- Global headers (includes all)
- Configuration file
- Interrupt functions source code
- User application source code




(*) Some peripherals used other peripherals routines:
- rstclk to get the master clock frequency
- tim3 to calibrate the LSI

# Using the Library (1/5)

- **Common files have to be included into the project**: stm8_conf.h, stm8_map.h, stm8_type.h, stm8_macro.h, stm8_lib.h, stm8_it.c and stm8_it.h
- To use the peripheral PPP:
  - stm8_ppp.c and stm8_ppp.h files must be included into the project
  - **Edit the stm8_conf.h file** and uncomment the following lines :
    - #define _PPP (1)
- If you want **to debug your application**, you have to define the label **DEBUG** in the **stm8_conf.h file**:
  - This creates a pointer, in memory, to the peripheral structure, so debug become easier and dumping a peripheral variable provides all registers settings.
  - This creates also the “assert” mechanism used to handle errors. You can add your own error processing code inside the “assert_failed” routine (usually placed inside the main.c file)
- **Include only "**stm8_lib.h" **in your application source code**

# Using the Library (2/5)

- **stm8_conf.h**

To be modified by user

```
#define DEBUG (1)
#define _ADC /* include stm8_adc.h file and
   declare a constant that points to the
   registers structure */
```

- **main.c**

To be modified by user

```
#include "stm8_lib.h"

void main (void)
{
  /* main program */
}

void assert_failed(void)
{
 /* Enter your own code to manage errors */
}
```

Include stm8_lib.h only

Only if DEBUG constant is defined

- **stm8_lib.h**

Do not modify this file

```
#include "stm8_map.h"
#ifdef _ADC
  #include "stm8_adc.h"
#endif
```

Define _ADC in stm8_conf.h to include stm8_adc.h in your project

- **stm8_map.h**

Do not modify this file

```
#include "stm8_conf.h"
typedef volatile struct ADC_struct
{
  u8 CSR;       /*!< ADC control status register */
  u8 CR1;       /*!< ADC configuration register 1 */
  u8 CR2;       /*!< ADC configuration register 2 */
  u8 RESERVED; /*!< Reserved byte */
  u8 DRH;       /*!< ADC Data high */
  u8 DRL;       /*!< ADC Data low */
  u8 TDRH;      /*!< ADC Schmitt trig disable reg high */
  u8 TDRL;      /*!< ADC Schmitt trig disable reg low */
}
ADC_TypeDef;

#define  ADC_CSR_RESET_VALUE  ((u8)0x00)
#define  ADC_CR1_RESET_VALUE  ((u8)0x00)

#define ADC_CSR_EOC  ((u8)0x80) /*!< End of Conversion */
#define ADC_CSR_ITEN ((u8)0x20) /*!< Int Enable mask */

#define ADC_BaseAddress 0x5400

#ifdef _ADC
#define ADC ((ADC_TypeDef *) ADC_BaseAddress)
#endif
```

Registers list

Reset values

Bits definitions

Periph base address

# Using the Library (3/5)

## Functions with structure parameter:

- You have to declare a *PPP_InitTypeDef* structure:

  e.g: PPP_InitTypeDef PPP_InitStructure;

  – The **PPP_InitStructure** is a working variable located in data memory that allows you to initialize one or more instance of PPPs.

# Using the Library (4/5)

## Functions with structure parameter (cont'd):

- You have to fill the ***PPP_InitStructure*** variable with the allowed values of the structure member before calling the PPP_Init function.
  - Configuration of the whole structure:
    - PPP_InitStructure.member1 = val1;
    - ...
    - PPP_InitStructure.memberN = valN;

Note : The previous initialization could be merged in only one line like the following :

- PPP_InitTypeDef PPP_InitStructure = { val1, val2, ..., valn}

  This reduces and optimises code size.

  - Configuration of few structure's members:
    - PPP_StructInit(&PPP_InitStructure); → **initialize all members with default value**
    - PPP_InitStructure.memberX = valX;
    - PPP_InitStructure.memberY = valY;

# Using the Library (5/5)

- You have to initialize the PPP peripheral by calling the *PPP_Init* function.
- At this stage the PPP peripheral is initialized. In some peripherals, it can be necessary to call the *PPP_Cmd* function in order to enable the peripheral.
- To access the entire functionalities of the PPP peripheral, the user can use a set of dedicated functions. These functions are specific to the peripheral and for more details refer to STM8 Firmware Library User Manual.
- *The PPP_DeInit* function can be used to set all PPP peripheral registers to their reset values.
- The *PPP_StructInit* function can be used to set all structure members to a default value before calling the *PPP_Init* function.
- You can find various coding examples in each peripheral examples firmwares.

# STM8 Tools

STM8S Technical Training

# STM8 Tools

| IDE | Compiler | Debugger | Programmer |
|---|---|---|---|
| ST Toolsets<br>(Free)<br>www.st.com/mcu | COSMIC<br>(32KB free )<br>www.cosmicsoftware.com | ST-Link<br>STX_RLink | From third partner |
| EWSTM8 | IAR for STM8<br>(32KB free)<br>www.iar.com | ST-Link | |


# STICE Emulator (1)

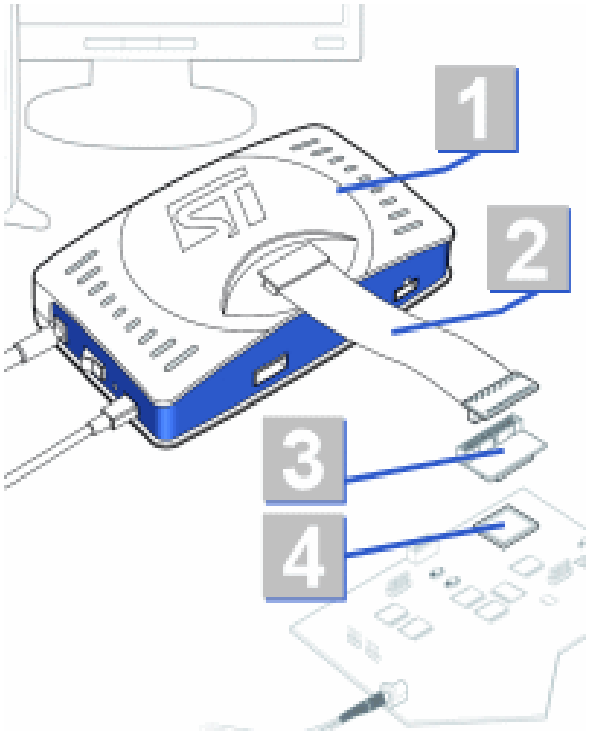
The *STICE* is ST's advanced emulation system for MCUs . It connects to the PC via USB interface and to the application board in place of the target microcontroller. It is supported by the free ST MCU Toolset: STVD IDE, STVP programmer and the STM8 Assembler.

| | |
|---|---|
| **FCPU** | 32kHz to 50MHz |
| **Memory-Read** | All memory and registers, also NEM and write-protected<br>Granularity 1byte |
| **Instruction Breakpoints** | Full Memory |
| **Advanced breakpoints** | Bus-event machine<br>4 user-configurable levels |
| **Data Breakpoints** | Yes |
| **RW on the fly** | All memory and registers |
| **Trace** | 128K with 10ns timestamp |
| **Coverage** | code & data |
| **Profiling** | timing & occurrence |
| **Voltage range** | 0.8V to 5.5V |
| **Standalone** | user mode, without PC |

| | |
|---|---|
| **Triggers** | 1 IN, 2 OUT |
| **Analyzer** | 8-bit probe |
| **In-circuit programming / debugging** | Via SWIM protocol |

# STICE Emulator (2)

**The STICE system permits Emulation and ICP / ICD**




## STICE BASE

| | |
|---|---|
| **1. Emulation System** | Order code: **STICE-SYS001**<br>▪ Emulator box with MEB S60C4 / 512KB, TEB GB12 (no PEB needed)<br>▪ Cables for USB, Power supply etc. |

## EMULATION

| | |
|---|---|
| **2. Connection Flex** | Order code: **CF/ FP 60 or 120**<br>▪ 60 - (up to 48-pin) or 120-wire flex cable |
| **3. Connection Adapter** | Order code: **AD / QFP80 ...QFP32**<br>▪ Adapts the Connection Flex to the socket |
| **4. Adapter Socket** | Order code: **AS / QFP80 ...QFP32**<br>▪ Package specific socket |

## In-circuit Programming / Debugging

Order code: **AD-ICD/ICP**

- In addition to STICE-SYS001 you need only the ICD/ICP Kit (includes cable, five 4-pin ERNI connectors for evaluation boards and an adapter for the SWIM cable)

# R-Link Debugger

**R-Link provides low-cost debugging and programming capabilities**


## R-Link BASE MODULE

**In-circuit Programming / Debugging**

Order code: **STX-RLINK**

- RLink extends programming and debugging capabilities to STM8 with addition of SWIM adapter.

*Note: RLink supports connections for a complete range of ST MCUs including JTAG for ARM core-based families and uPSD, plus ICC for ST7.*

# In-circuit Debugging (1)

Integrated STM8 debug-module enables low-cost, **non-intrusive** in-application **debugging** and fast **NVM programming** The DM is connected via the single wire interface SWIM and debugging hardware (RLink, STICE) to a PC's USB port.

| | |
|---|---|
| **Single wire debug interface** | ▪ Debug module activation and data exchange via a high speed, single wire interface - Effective SWIM data transmission speed of 145 bytes/ms |
| **Non-intrusive Debugging** | ▪ Debugging while executing the application program<br>▪ No need for resources (RAM, NVM) of the target product<br>▪ No restrictions on the use of flash addresses<br>▪ All Instructions and Interrupts remain usable during debugging |
| **Instruction Breakpoints** | ▪ Unlimited - Full Memory<br>▪ Software breakpoints |
| **Real-time Read/Write** | ▪ Read/Write of all memory and peripheral registers during application execution (no interrupt, no monitor – CPU stall for 1..2 clock cycles)<br>▪ Real-time read access to CPU registers |
| **Advanced Breakpoints** | ▪ 2 configurable advanced breakpoints - 23 different configurations)<br>▪ Data break-points |
| **Code Execution control** | ▪ Step by step execution |
| **Fast memory programming** | ▪ Programming and verification of 128Kbytes flash via SWIM in less than 5s |

# In-circuit Debugging (2)

**Debugging power of the integrated DM is close to a full feature emulator.**

| Feature | STice Emulator | Debug module & SWIM |
|---|---|---|
| **Memory size** | • Same as MCU | |
| **Timing Accuracy** | • Real time execution | |
| **Memory read / write** | • RAM, Flash and peripheral registers can be read/ written “on the fly”<br>• All other resources can be read / written when the application is stopped. | |
| **Instruction breakpoints** | • On all program memory instructions | |
| | • Hardware breakpoints | • Software breakpoints |
| **Advanced breakpoints** | • Powerful 4 level state machine<br>• INPUT: instruction and data bus events, external trigger<br>• OUTPUT: trace control and trigger | • 23 easy to use configurable predefined configurations<br>• Can only trigger breakpoints |
| **Peripheral & memory Accuracy** | • Analog part of peripherals are rebuilt<br>• Flash programming is not emulated | • 100% accurate (actual chip) |
| **Trace** | • Yes | • None |
| **Coverage** | • Code & data coverage monitors code execution and data accesses | • None |
| **Profiling** | • Analyze performance on whole code & data, counting event occurrences and durations | • None |

# Coverage… What's this ?

- New ST ICE Development Feature
- Coverage is also called "**test coverage**"
- **Code Coverage** tells if a portion of code has been executed or not.
- **Data Coverage** tells if a memory area has been accessed or not.

# Coverage… What for ?

- Detect holes in validation test plan
  - ⇨Improve application quality
- Measure test coverage
  - ⇨Follow a coding norm (such as MISRA)
- Detect dead code
  - ⇨Reduce code size => decrease product price or add other functionalities
- Detect unused and partially unused DATA
  - ⇨Reduce needed RAM => decrease product prize or add other functionalities

# Profiling… What's this ?

- New ST ICE Development Feature
- Shows where an application spends time
- Profiling with STVD & ST*ice* includes
  - Occurrence profiling (event counting)
  - Time profiling (duration)
- Occurrence profiling is available on code execution and data accesses
- Time profiling is available on code only

# Code Profiling… What for ?

- Find performance bottlenecks in written code
  - ⇨Improve global application time performances
  - ⇨Debug when time performance is critical (eg : interrupt handler, reset / initialization phase)
  - ⇨Optimize code effectivity in most used parts of code
  - ⇨Reduce clock frequency (and power consumption)
- Get statistics for a given test scenario
  - ⇨Count numbers of occurrence of resets, of each kind of interrupts or other functional events
  - ⇨Evaluate (and secure !) performance margin on time critical code

# Data Profiling… What for ?

- Optimize variable placement
  - ⇨Improve application time performances
- Optimize data structure
  - ⇨Detect unused and scarcely used members in structures and reduce needed RAM size
- Optimize stack usage
  - ⇨Reuse unused and scarcely used area for RAM and reduce needed RAM size

# STVD & STice

- Give for each scenario a **full picture**…
  - Coverage and Profiling
  - Code and Data
  - Examine the whole application
- ...**without any code modification**
  - No need to instrument the code
  - Observation doesn’t change the performances

# Bottlenecks Analysis

- Top-down process
  - Do profiling on the entire application
  - From the profiling results, find-out the longest time consuming code block
    => e.g. *func1().*
  - Then, focus on that block:
    ⇒ Get the tree of the called functions (e.g. *func1() -> func2() -> func3()*)
    ⇒ Find out in which sub-function/instruction you spent the most time (e.g. *func3()*).

**Don't need to know in advance where to look for…**

# Dynamic vs static analysis

- Static analysis: done at build time by compiler and linker
  - Small optimization done overall whole application
    - Functions never called in the source code can be removed by the linker

      *If (false) Manage_False_Function();*
    - Variables never used can be removed by the linker

- Dynamic analysis: done at run time
  - Improve significantly key parts
  - Functions called in conditions that never happen are detected by the coverage

    *If (external_cause1) Manage_external_cause1();*

- Profiler could be used with and/or without Compiler Optimazation

# Benefits for your application -1

- Improve application time performances
  - ⇨Detect bottlenecks in code
  - ⇨Optimize variable placement
- Reduce code size
  - ⇨Detect and remove dead code
- Reduce ram size
  - ⇨Detect and remove unused and scarcely used data
  - ⇨Detect unused and scarcely used members in structures
  - ⇨Evaluate and decrease needed stack size

# Benefits for your application - 2

- Improve validation coverage
  - ⇨Detect some holes in the validation plan
  - ⇨Get internal metrics (white box validation approach)
- Reduce power consumption
  - ⇨Improve global code efficiency and reduce clock frequency
  - ⇨Identify portion of code that could work with lower clock frequency

# Benefits for your application - 3

- Coverage and profiling features are not the only answer to any problem
- Please keep in mind, you can use it jointly with other tools
  - Compiler/linker provides static analysis
  - Other emulation features…

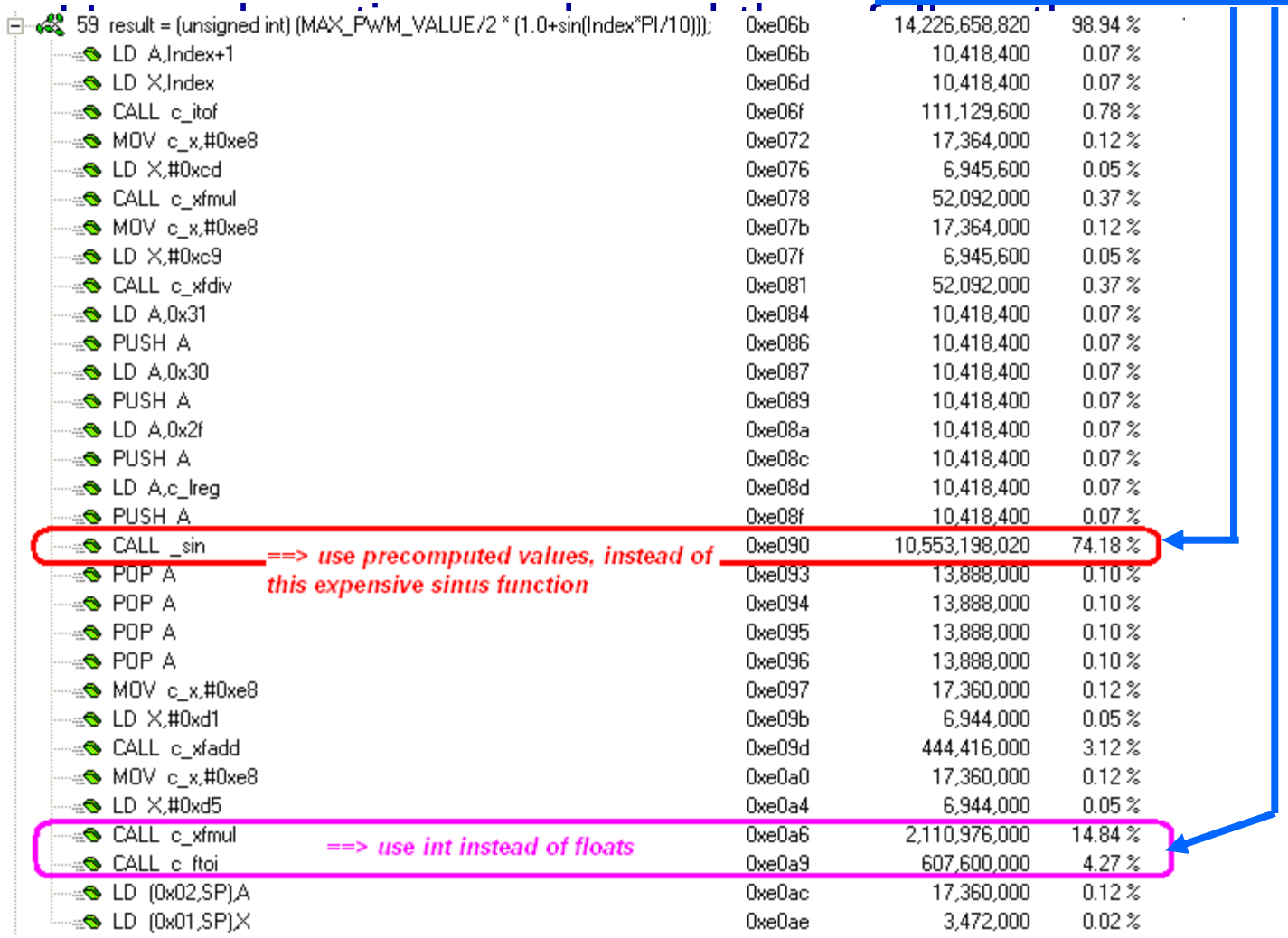
# What makes the profiler so powerful

- Data collection
  - Metrics on Code and Data for the whole application
  - Coverage, occurrence profiling and time profiling
  - No code instrumentation
  - Advanced profiling modes (subroutine mode !)
- Easy analysis
  - Consolidation views with sort capabilities
  - Navigation between function/line/instruction levels
  - Navigation between consolidated and source views
- Profiling session focused on scenarios
  - The profiler can be started and stopped at almost any moment $\Rightarrow$ can focus the analysis on a special sequence

# Competition

- Coverage and profiling are not very common in 8 bit world
- Quite common in 32 bit embedded world but based on trace analysis
  - Limited to a (very) small portion of code
- Common on PC / workstation
  - Typically done through automated code instrumentation

# How to find performance bottlenecks



| | | | |
|---|---|---|---|
| 59 result = (unsigned int) (MAX_PWM_VALUE/2 * (1.0+sin(Index*PI/10))); | 0xe06b | 14,226,658,820 | 98.94 % |
| LD A,Index+1 | 0xe06b | 10,418,400 | 0.07 % |
| LD X,Index | 0xe06d | 10,418,400 | 0.07 % |
| CALL c_itof | 0xe06f | 111,129,600 | 0.78 % |
| MOV c_x,#0xe8 | 0xe072 | 17,364,000 | 0.12 % |
| LD X,#0xcd | 0xe076 | 6,945,600 | 0.05 % |
| CALL c_xfmul | 0xe078 | 52,092,000 | 0.37 % |
| MOV c_x,#0xe8 | 0xe07b | 17,364,000 | 0.12 % |
| LD X,#0xc9 | 0xe07f | 6,945,600 | 0.05 % |
| CALL c_xfdiv | 0xe081 | 52,092,000 | 0.37 % |
| LD A,0x31 | 0xe084 | 10,418,400 | 0.07 % |
| PUSH A | 0xe086 | 10,418,400 | 0.07 % |
| LD A,0x30 | 0xe087 | 10,418,400 | 0.07 % |
| PUSH A | 0xe089 | 10,418,400 | 0.07 % |
| LD A,0x2f | 0xe08a | 10,418,400 | 0.07 % |
| PUSH A | 0xe08c | 10,418,400 | 0.07 % |
| LD A,c_lreg | 0xe08d | 10,418,400 | 0.07 % |
| PUSH A | 0xe08f | 10,418,400 | 0.07 % |
| CALL _sin | 0xe090 | 10,553,198,020 | 74.18 % |
| POP A | 0xe093 | 13,888,000 | 0.10 % |
| POP A | 0xe094 | 13,888,000 | 0.10 % |
| POP A | 0xe095 | 13,888,000 | 0.10 % |
| POP A | 0xe096 | 13,888,000 | 0.10 % |
| MOV c_x,#0xe8 | 0xe097 | 17,360,000 | 0.12 % |
| LD X,#0xd1 | 0xe09b | 6,944,000 | 0.05 % |
| CALL c_xfadd | 0xe09d | 444,416,000 | 3.12 % |
| MOV c_x,#0xe8 | 0xe0a0 | 17,360,000 | 0.12 % |
| LD X,#0xd5 | 0xe0a4 | 6,944,000 | 0.05 % |
| CALL c_xfmul | 0xe0a6 | 2,110,976,000 | 14.84 % |
| CALL c_ftoi | 0xe0a9 | 607,600,000 | 4.27 % |
| LD (0x02,SP),A | 0xe0ac | 17,360,000 | 0.12 % |
| LD (0x01,SP),X | 0xe0ae | 3,472,000 | 0.02 % |

*==> use precomputed values, instead of this expensive sinus function*

*==> use int instead of floats*

# References

- AN2752 Getting Start with STM8
- UM0482 STM8_128 EVAL Evaluation Board User Manual
- UM0470 STM8 SWIM Communication Protocol and Debug Module
- UM0817 STM8S Discovery User Manual
- STM8S Firmware Library

**CONFIDENTIALITY OBLIGATIONS:**

THIS DOCUMENT CONTAINS SENSITIVE INFORMATION.

IT IS CLASSIFIED “MICROCONTROLLERS, MEMORIES & SECURE MCUs (MMS) RESTRICTED AND ITS DISTRIBUTION IS SUBMITTED TO ST/MMS AUTHORIZATION

AT ALL TIMES YOU SHOULD COMPLY WITH THE FOLLOWING SECURITY RULES:

- DO NOT COPY OR REPRODUCE ALL OR PART OF THIS DOCUMENT
- KEEP THIS DOCUMENT LOCKED AWAY
- FURTHER COPIES CAN BE PROVIDED ON A “NEED TO KNOW BASIS”, PLEASE CONTACT YOUR LOCAL ST SALES OFFICE OR DOCUMENT WRITTER.