Boot-Up Profiling using the ARM System Trace Macrocell (STM) on a Renesas R-Car Gen3 SoC

Publish Date: 07/27/2020

Table of Contents

1 Introduction

This application note describes how to utilize the ARM System Trace Macrocell (STM) for timing measurement of the boot-up process of complex SoCs, such as the Renesas R-Car family.

1.1 Motivation

Complex SoCs typically run rather complex software architectures. Such SoCs often implement a mixture of real-time cores, such as ARM Cortex R7 or M7, and clusters of application cores, such as ARM Cortex A53 or A57. The real-time cores may run an AUTOSAR Classic stack, while the application cores run an AUTOSAR Adaptive stack, using a POSIX OS such as Linux.
The boot-up process of such system can be quite complex and may involve a mixture of SoC-specific bootloaders for the real-time core, often used as the master boot core, and various bootloaders for the application cores implemented according to the ARM Trusted Firmware (ATF) architecture.

Figure 1 shows a sample software architecture running on an R-Car SoC, including the various bootloader stages. In this example, the R7 core is used as master boot core. The boot-up sequence comprises the following steps:

1. Upon release from reset the R7 master boot core first executes a ROM-based bootloader, called Boot ROM. The Boot ROM determines which boot memory is used, e.g. QSPI FLASH, copies the 1st State Bootloader (R7 Initial Program Loader, IPL) from QSPI FLASH to on-chip RAM and subsequently branches to the R7 IPL.
2. The IPL, loads the software stack running on the real-time cores, e.g. an AUTOSAR Classic stack. In addition, it loads the ATF BL2 loader image to RAM, enables an application processor and triggers it to run the BL2.
3. The BL2 loads the U-Boot, which finally loads the Linux image from some storage device such as eMMC.

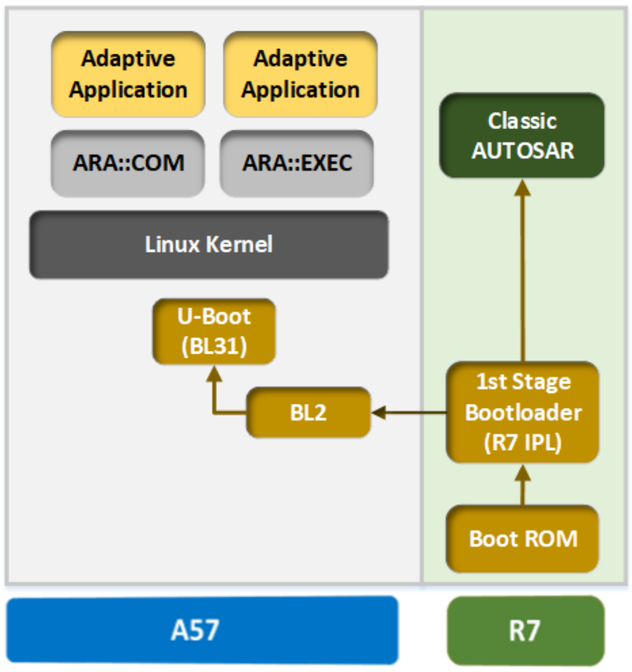


Figure 1: Sample Software Architecture of an R-Car SoC including Bootloader Stages

This boot-up procedure may even be changed or interleaved in order to meet certain system-specific timing requirements. This can lead to even more complex boot-up sequences. Thus, a good understanding of the timing behavior and the possibility for measuring the timing is essential for the optimization and verification of the boot-up process.

This application note describes how iSYSTEM trace tools, such as the iC5700 in conjunction with the winIDEA Trace Analyzer, can be used for the timing analysis of such a boot-up process. The final result could be a visual representation of the boot process as depicted in Figure 2.

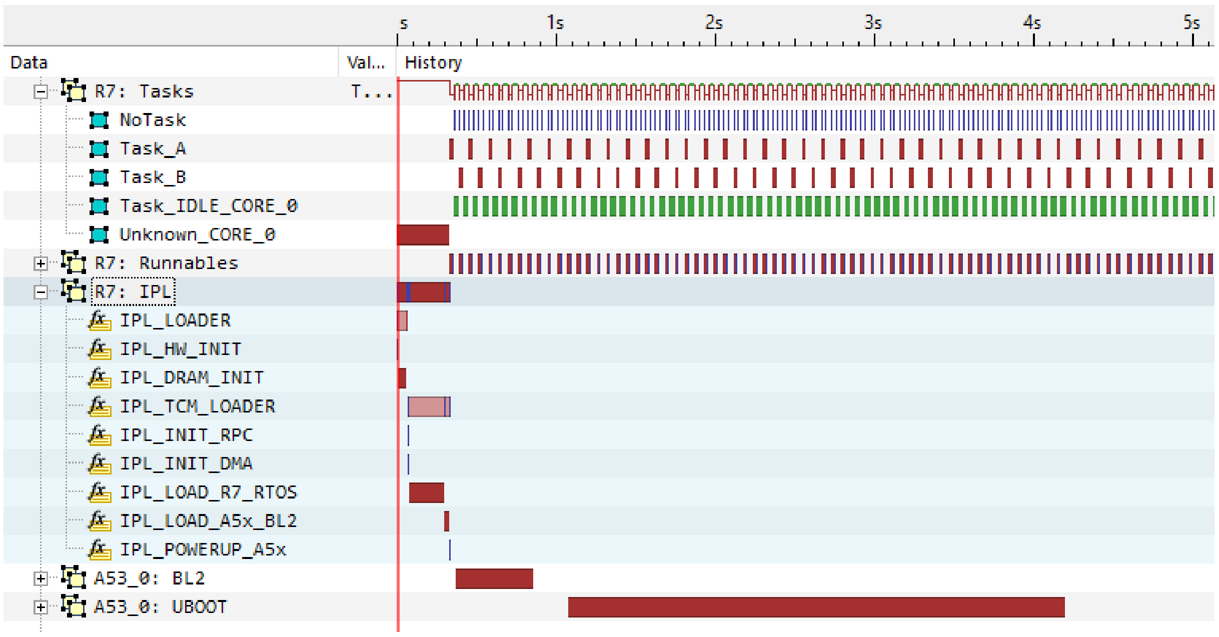


Figure 2: Sample R-Car Boot-up Trace Recording

1.2 Trace Concept

The trace concept is based on utilizing the on-chip hardware trace mechanisms implemented in the SoC. ARM-based SoCs implement a debug and trace infrastructure according to the ARM CoreSight architecture. Such an architecture is depicted in Figure 3.

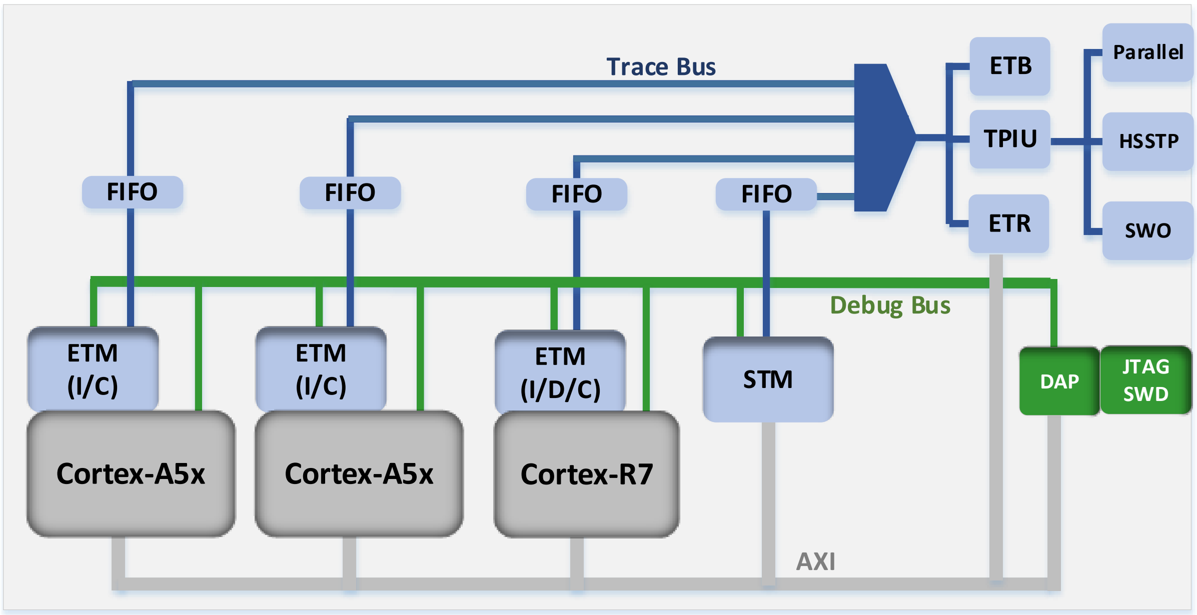


Figure 3: Generic SoC implementing the ARM CoreSight Debug and Trace Architecture

The ARM CoreSight trace architecture comprises the following components:

Table 1: ARM CoreSight Components

Module	Description
Cortex.A5x ETM	Cortex-A5x Embedded Trace Macrocell Provides Instruction and ContextID Trace
Cortex R/M ETM	Cortex-R/M Embedded Trace Macrocell Provides Instruction, Data und ContextID Trace
STM	System Trace Macrocell Data Trace via Instrumentation
ETB	Embedded Trace Buffer
HSSTP	ARM High Speed Serial Trace Port Trace Port with 2.5 / 5.0 Gbps Transfer Rate
SWO	Serial Wire Output Asynchronous Single Wire Trace Output Port

SoCs typically use a boot-up concept where one dedicated core is assigned as the master boot core. This means that only this boot core is operational after reset, whereas all other cores are still held in reset or not even powered or clocked yet. This typically also implies that the trace logic associated to each core, e.g. the Embedded Trace Macrocell (ETM) is also not operational yet. Thus, a trace tool is not able to access and configure the on-chip trace logic of all cores involved in the boot process right after reset, before the start of the boot process. This means, tracing the boot-up process via ETM trace would require stopping the cores after they have been released by the boot core, in order to configure their ETM module. This obviously has a major impact on the boot-up timing.
The STM module can be enabled by the trace tool right after reset and each core can contribute to the STM tracing (i.e. writing to the STM Stimulus port) as soon as it is operational. Therefore, STM trace is perfectly suited for this use-case.

2 R-Car Trace Architecture

Figure 4 depicts the overall trace architecture of an R-Car Gen3 device. STM trace uses the trace infrastructure high-lighted in red color.

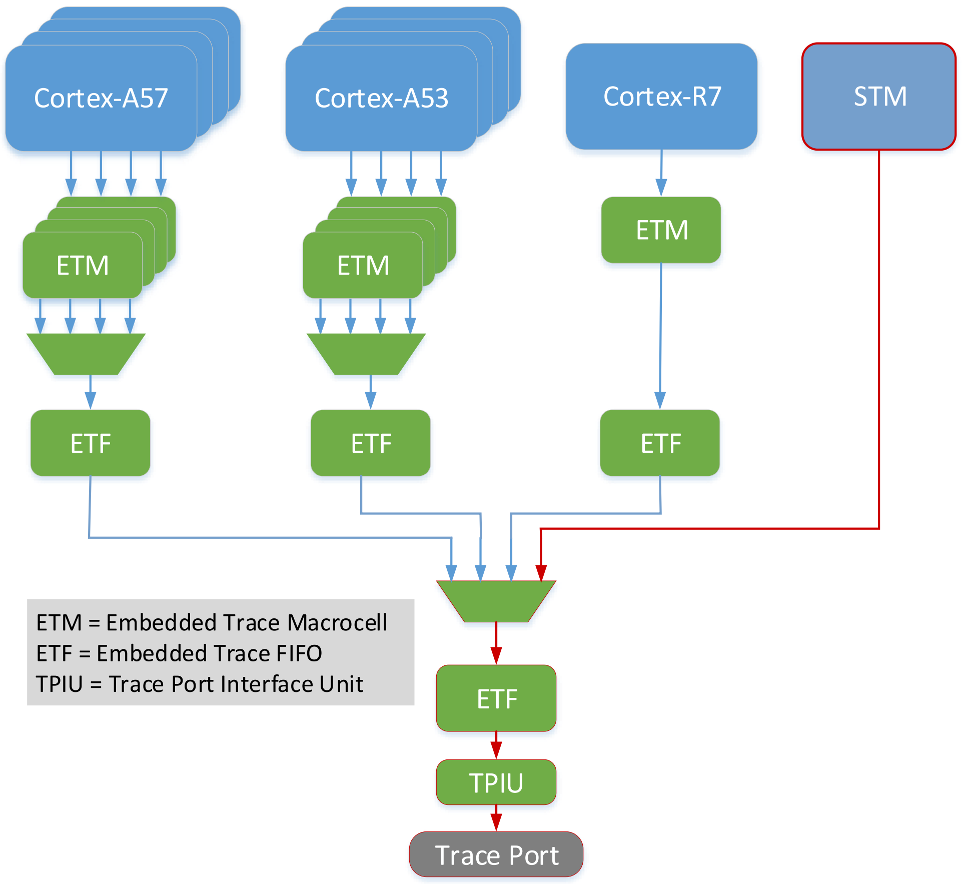

Cortex-A57
Cortex-A53
Cortex-R7
STM
ETM
ETM
ETM
ETF
ETF
ETF
ETM = Embedded Trace Macrocell
ETF = Embedded Trace FIFO
TPIU = Trace Port Interface Unit
ETF
TPIU
Trace Port

Figure 4: Simplified Chip-Level Trace Architecture of an R-Car Gen3 Device

The trace messages generated by the STM first pass through a FIFO (ETF) and are, optionally, forwarded to a trace port, e.g. the ARM High-Speed Serial Trace Port (HSSTP). Thus, there are two options for the storage of the STM trace message, either on-chip in the ETF or off-chip in the connected trace recording tool.
When storing the STM trace into the ETF, the storage is rather limited (e.g. 4k Byte), but may be sufficient for the analysis of a boot-up sequence. However, the STM trace can be read out via the standard JTAG debug interface.
On the other hand, streaming out the STM trace via the HSSTP interface allows a virtually unlimited trace duration, i.e. the STM trace can be extended to include not only the boot-up process but also the startup and operation of the OS and application software.

2.1 STM Architecture

The concept behind STM trace is that a core can perform data write transactions to a memory mapped area of the STM, residing on the AXI bus of the processor. This memory mapped area, called the Stimulus Port, is divided into multiple so-called Channels (256 bytes per channel, see address map in Figure 6). A write transaction to such an STM Stimulus Port Channel causes the STM to emit an STM message via the hardware trace port. The Channel number encoded in the STM message can be used by the trace recording tool to differentiate between different messages types. An STM message may contain a data field with a length of up to 64 bits, a timestamp and also a marker to allow for multi-message protocols, e.g. for sending out strings. This versatility allows for signaling

various types of information and events such as OS task state transitions or for function/runnable entry/exit signaling, etc.
Figure 5 shows the chip-level architecture STM architecture implemented on Renesas R-Car SoCs. It illustrates how the STM may be integrated within a System-On-Chip (SoC).

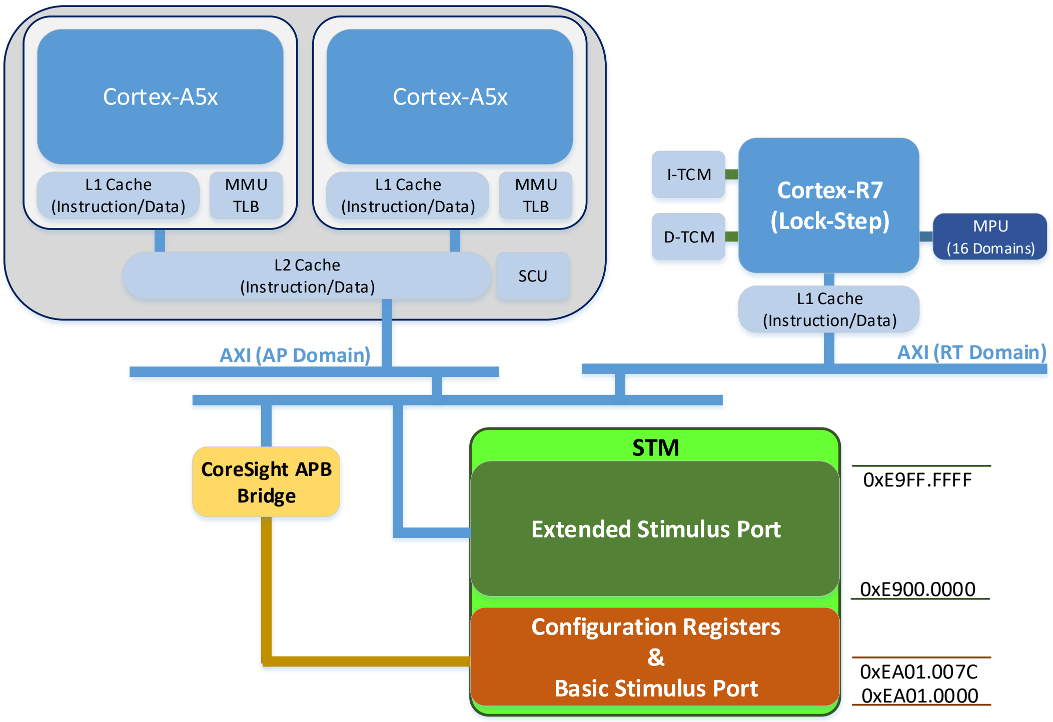
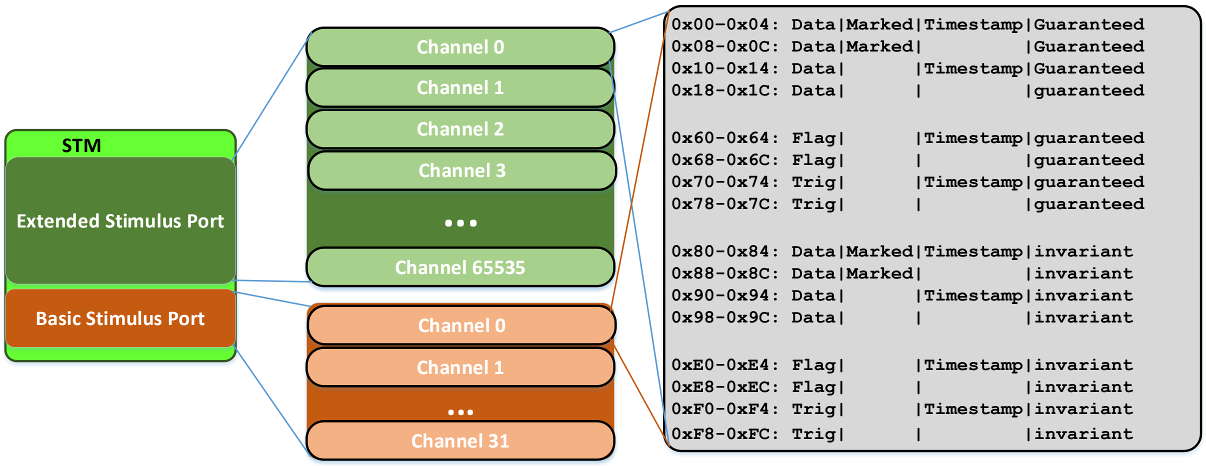


Figure 5: Chip-Level STM Architecture of the Renesas R-Car SoC

The STM actually provides two stimulus ports, the basic stimulus port and the extended stimulus port. The basic stimulus port is mapped to the ARM peripheral bus (APB) and offers 32 channels. The extended stimulus port is mapped to the ARM high-speed bus matrix (AXI) and offers 64k channels. Within each channel, a write access to specific 64-bit aligned address locations triggers the generation of specific STM message types. For instance, a write access to channel offset address 0x10 is treated as a blocking write bus transaction (guaranteed) and generates a STM message with a data payload of up to 64 bits and includes a timestamp.
Figure 6 illustrates the channel allocation within a basic and extended stimulus port of a STM. It also shows the mapping of STM message types to a specific address location within each channel.



```
0x00-0x04: Data|Marked|Timestamp|Guaranteed
0x08-0x0C: Data|Marked|         |Guaranteed
0x10-0x14: Data|      |Timestamp|Guaranteed
0x18-0x1C: Data|      |         |guaranteed

0x60-0x64: Flag|      |Timestamp|guaranteed
0x68-0x6C: Flag|      |         |guaranteed
0x70-0x74: Trig|      |Timestamp|guaranteed
0x78-0x7C: Trig|      |         |guaranteed

0x80-0x84: Data|Marked|Timestamp|invariant
0x88-0x8C: Data|Marked|         |invariant
0x90-0x94: Data|      |Timestamp|invariant
0x98-0x9C: Data|      |         |invariant

0xE0-0xE4: Flag|      |Timestamp|invariant
0xE8-0xEC: Flag|      |         |invariant
0xF0-0xF4: Trig|      |Timestamp|invariant
0xF8-0xFC: Trig|      |         |invariant
```

Figure 6: The STM Stimulus Ports are divided into multiple Channels.

2.2 STM Memory Access

The STM stimulus port is mapped into memory accessible by each core of the processor. The STM memory locations can be accessed via standard data write transactions of the CPU, e.g. store instructions, without any prior initialization required by the application software. The necessary STM configuration can be performed by a debug/trace tool attached to the processor via its JTAG debug interface.
However, in case the system either uses memory protection (MPU) or memory virtualization (MMU), the STM memory range may not be accessible by any component of the application software. The underlying OS and/or hypervisor may first need to grant access to the STM hardware.

2.2.1 Memory Virtualization via MMU

The physical address space of the STM stimulus port must be added to the virtual address space of the context (kernel or user space) which intends to perform data write accesses to the STM stimulus port. The POSIX compliant system call MMAP() may be used, for instance in Linux OS based systems, to map the STM hardware memory range into the virtual address space of a user application.

2.3 On-Chip STM Time Stamp Generation

Figure 7 shows a simplified block diagram of the STM time stamping clock generation on the Renesas R-Car devices. The so called Generic Counter of the Application Processor sub-system provides the 32-bit time stamp value used by the Embedded Trace Macrocells (ETM) of all on-chip processors and the System Trace Macrocell (STM). The Generic Counter is driven by the clock generated from the external crystal. The crystal clock is divided by 2 before feeding the Generic Counter. Driving the Generic Counter directly from an external crystal ensures that trace time stamping remains operational also in low-power modes which typically disable on-chip clock generators such as PLLs.

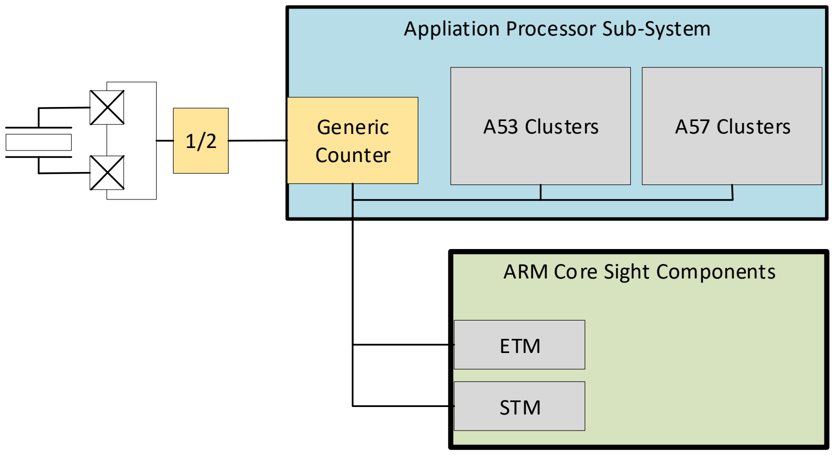
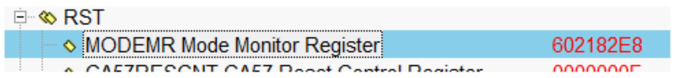


Figure 7: Simplified Block Diagram of the STM Time Stamp Clock Generation

Sample STM Time Stamp Clock Configuration:
Figure 8 shows a winIDEA Special Function Register (SFR) window listing the MODEMR Register of the RESET Module, relevant to derive the STM time stamp clock. The MODEMR register bits MD13, MD14, MD17 and MD19 can be used to derive the frequency of the external crystal. This information is needed to set the correct “Cycle Duration” in the “Hardware – CPU Options… - Analyzer” dialog.



RST
MODEMR Mode Monitor Register 602182E8

```
  6        0        2        1        8        2        E        8

0110 | 0000 | 0010 | 0001 | 1000 | 0010 | 1110 | 8000
```

MD13: 0

MD14: 0

MD17: 0

MD19: 0

```
=> XTAL Frequency = 16.66 MHz (see table)
=> Generic Counter Frequency = XTAL / 2
=> Trace Cycle Duration = 1 / Generic Counter Frequency
   Trace Cycle Duration = 1 / (16.66 / 2)  = 120 ns
```

Figure 8: winIDEA Special Function Register Window listing the RST.MODEMR Register relevant for determining the STM Time Stamp Clock (Trace Cycle Duration)

3 Function Profiling using STM Trace Instrumentation

This chapter gives a generic description about function profiling by means of STM trace instrumentation.
Function profiling refers to the analysis of the temporal behavior of C-function execution. This analysis comprises both a statistical analysis as well as the reconstruction of the function call sequences over time.
Function profiling using STM trace instrumentation is based on marking the entry and exit(s) of a function. This means that the instrumentation code is added at the entry and at the exit of the function to be profiled. This instrumentation code writes to the STM Stimulus port and thus causes the generation of an STM trace message, including a time stamp. The trace message emitted at the function entry contains a unique function identifier (integer number assigned to this function). The trace message emitted upon function exit contains the common function exit indicator, i.e. the value ‘0’.

3.1 Code Instrumentation

The code in Listing 1 shows a sample of STM instrumentation for function profiling. The instrumentation code needs to assign a unique function ID for each function to be instrumented. This function ID mapping is used by the instrumentation code and also for the Profiler configuration.

```
isystem_stm_function_ids.h:

/* STM Trace Function IDs */
#define STM_FUNCTION_EXIT     (0x00)
#define STM_IPL_LOAD_R7_RTOS (0x50)
```

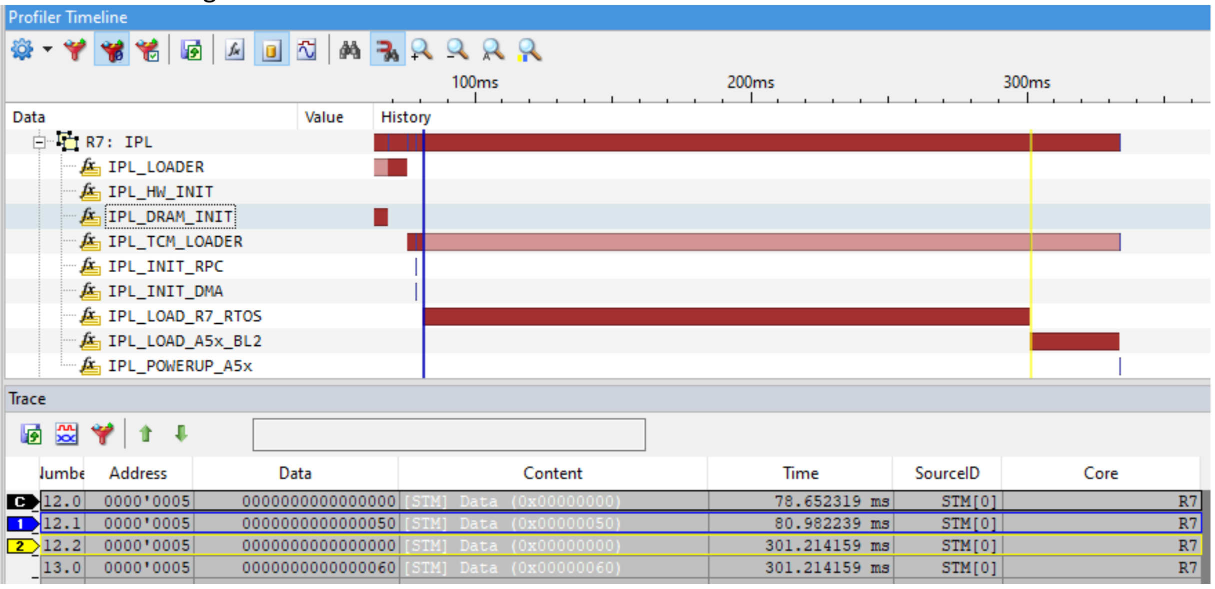
```
isystem_stm_trace_r7ipl.h:

/* SoC-specific Base Address of STM Extended Stimulus Port */
#define STM_ADDR   0xE9000000
/* Offset within Channel to generate Message with Data + Timestamp */
#define STM_OFFSET 0x00000010

/* STM Access Macro. Ch is channel number. */
#define STM32_DTS(ch) *(volatile int*)(STM_ADDR+STM_OFFSET+(ch*0x100))

/* STM Function Trace Instrumentation API (using STM channel 0x300) */
#define STM_TRACE_FUNCTION_ENTRY(value)   { STM32_DTS(5) = value; }
#define STM_TRACE_FUNCTION_EXIT()         { STM32_DTS(5) = 0; }
```

```
R7 IPL tcm_loader_main.c:

#include <isystem_stm_function_ids.h>
#include <isystem_stm_trace_r7ipl.h>

void ipl_load_r7_rtos(void)
{
  STM_TRACE_FUNCTION_ENTRY(STM_IPL_LOAD_R7_RTOS);

  /* … Load the R7 RTOS … */

  STM_FUNCTION_EXIT();
}
```

Listing 1: Sample Code Listing for Function Profiling by means of STM trace

3.2 Profiler Visualization

Figure 9 shows the STM trace recording corresponding to the sample code of Listing 1. Here, multiple functions have been instrumented for STM trace. Looking at the function IPL_LOAD_R7_RTOS, you can see that the function entry (blue cursor, #1) is signaled by an STM trace message using STM channel 5 (Trace Address column) with a payload data (Trace Data column) of 0x50. The function exit (yellow cursor, #2) is signaled with an STM trace message using also STM channel 5 and a payload data of 0x00. The trace timing (Time column) is derived from the time stamp value included in each STM trace message.



	Jumbe	Address	Data	Content	Time	SourceID	Core
C	12.0	0000'0005	0000000000000000	[STM] Data (0x00000000)	78.652319 ms	STM[0]	R7
1	12.1	0000'0005	0000000000000050	[STM] Data (0x00000050)	80.982239 ms	STM[0]	R7
2	12.2	0000'0005	0000000000000000	[STM] Data (0x00000000)	301.214159 ms	STM[0]	R7
	13.0	0000'0005	0000000000000060	[STM] Data (0x00000060)	301.214159 ms	STM[0]	R7

Figure 9: Sample STM Trace recording and Function Profiling Timeline

4 Bootloader Instrumentation for STM Trace

This chapter describes how the function profiling approach, explained in Chapter 3, can be applied for profiling the bootloaders of an R-Car boot-up sequence. The following section will use the R7 Initial Program Loader (IPL) and the A5x BL2 loader as an example.

4.1 Instrumentation of the R7 Initial Program Loader (IPL)

The R7 IPL source code is available at Renesas.

In the following section we explain a sample R7 IPL STM instrumentation for measuring the time required for loading the R7 RTOS image.
The header files *include\isystem_stm_Ids.h* and *include\isystem_stm.h* define the macros for the write access to the STM Stimulus port and the IDs for the functions to be profiled (see Listing 2 and Listing 3).

```
#ifndef __ISYSTEM_STM_IDS_H__
#define __ISYSTEM_STM_IDS_H__

#define STM_IPL_LOADER              0x0001
#define STM_IPL_HW_INIT             0x0010
#define STM_IPL_DRAM_INIT           0x0012
#define STM_IPL_TCM_LOADER          0x0020
#define STM_IPL_INIT_RPC            0x0030
#define STM_IPL_INIT_DMA            0x0040
#define STM_IPL_LOAD_R7_RTOS        0x0050
#define STM_IPL_LOAD_A5x_BL2        0x0060
#define STM_POWERUP_A5x             0x0070

#endif /* __ISYSTEM_STM_IDS_H__ */
```
Listing 2: R7 IPL C Header File with Function ID Macro Definitions

```
#ifndef __ISYSTEM_STM_H__
#define __ISYSTEM_STM_H__

#define STM_EXIT  0

#define STM32_DTS(ch) *(volatile unsigned int*)(0xE9000010 + (ch*0x100))

#define STM_TRACE_R7IPL(value)   do { STM32_DTS(0x5) = value; } while(0)

#endif /* __ISYSTEM_STM_H__ */
```
Listing 3: R7 IPL C Header File with Sample STM Trace Macro Definitions

The actual code for copying the R7 RTOS image from the boot memory to DRAM is located in the C source file *tcm_loader\tcm_loader_main.c*. In this particular case, we are not instrumenting one single function but rather a group of functions which are involved in checking and loading the RTOS image.

```
/**********************************************************************
 *    Load RTOS from HyperFlash
***********************************************************************/

      STM_IPL_LOAD_R7_RTOS();

      get_info_from_cert(RTOS_CERT_ADDR, &size, &rtos_load_addr);
      if (size == 0U) {
            ERROR("RTOS image size error\n");
            panic();
      } else if (size > RTOS_MAX_SIZE) {
            ERROR("RTOS image size error\n");
            panic();
      } else {
            /* No else processing QAC compliant */
      }
      check_load_area(rtos_load_addr, FLASH_RTOS_IMAGE_ADDR, size);
      execDMA(rtos_load_addr, FLASH_RTOS_IMAGE_ADDR, size);

      STM_IPL_RETURN ();
```

Listing 4: R7 IPL C Source Code Snippet (tcm_loader_main.c) including STM Trace Instrumentation

After building the instrumented R7 IPL, the new binary image can be programmed into the boot memory, e.g. QSPI FLASH, by means of winIDEA.

4.2 Instrumentation of an A5x Bootloader, e.g. BL2

The BL2 bootloader is part of the ARM Trusted Firmware (ATF) included in the R-Car Yocto project (see https://elinux.org/R-Car/Boards/Yocto-Gen3).

The bootloaders for the A5x core can be instrumented for STM trace basically the same way as for the R7 core. For illustration, we add STM trace instrumentation to the ATF BL2 bootloader.
Listing 5 shows a sample implementation of the header file *\include\common\isystem_stm.h*.

```
#ifndef __ISYSTEM_STM_H__
#define __ISYSTEM_STM_H__

#define STM32_DTS(ch) *(volatile unsigned int*)(0xE9000010 + (ch*0x100))

#define STM_TRACE_BL2(value)   do { STM32_DTS(0x7) = value; } while(0)

#define STM_EXIT        0x0000
#define STM_BL2         0x0070

#endif /* __ISYSTEM_STM_H__ */
```

Listing 5: Sample isystem_stm.h Header File for BL2 Instrumentation

Listing 6 shows the listing of the function *bl2_main()*. The macro STM_TRACE_BL2(STM_BL2) marks the entry in the function *bl2_main()*. Marking the function exit is a bit more tricky. The function *bl2_run_next_image()* called just before the end of the *bl_main()* function actually never returns as it invokes the next bootloader stage. Thus, the *bl2_main()* function “exit” needs to marked just before the function call *bl2_run_next_image()*.

```
#include <isystem_stm.h>

void bl2_main(void)
{
      STM_TRACE_BL2(STM_BL2);

      entry_point_info_t *next_bl_ep_info;

      NOTICE("BL2: %s\n", version_string);
      NOTICE("BL2: %s\n", build_message);

      /* Perform remaining generic architectural setup in S-EL1 */
```

```
	bl2_arch_setup();

	/* initialize boot source */
	bl2_plat_preload_setup();

	/* Load the subsequent bootloader images. */
	next_bl_ep_info = bl2_load_images();

	/* … */

	NOTICE("BL2: Booting " NEXT_IMAGE "\n");
	print_entry_point_info(next_bl_ep_info);
	console_flush();

	STM_TRACE_BL2(STM_EXIT);
	bl2_run_next_image(next_bl_ep_info);
}
```
Listing 6: Sample STM Trace Instrumentation of the BL2 Bootloader Function bl2_main()

After building the instrumented A5x BL2, the new binary image can be programmed into the boot memory, e.g. QSPI FLASH, by means of winIDEA.

5 winIDEA Configuration for STM Trace

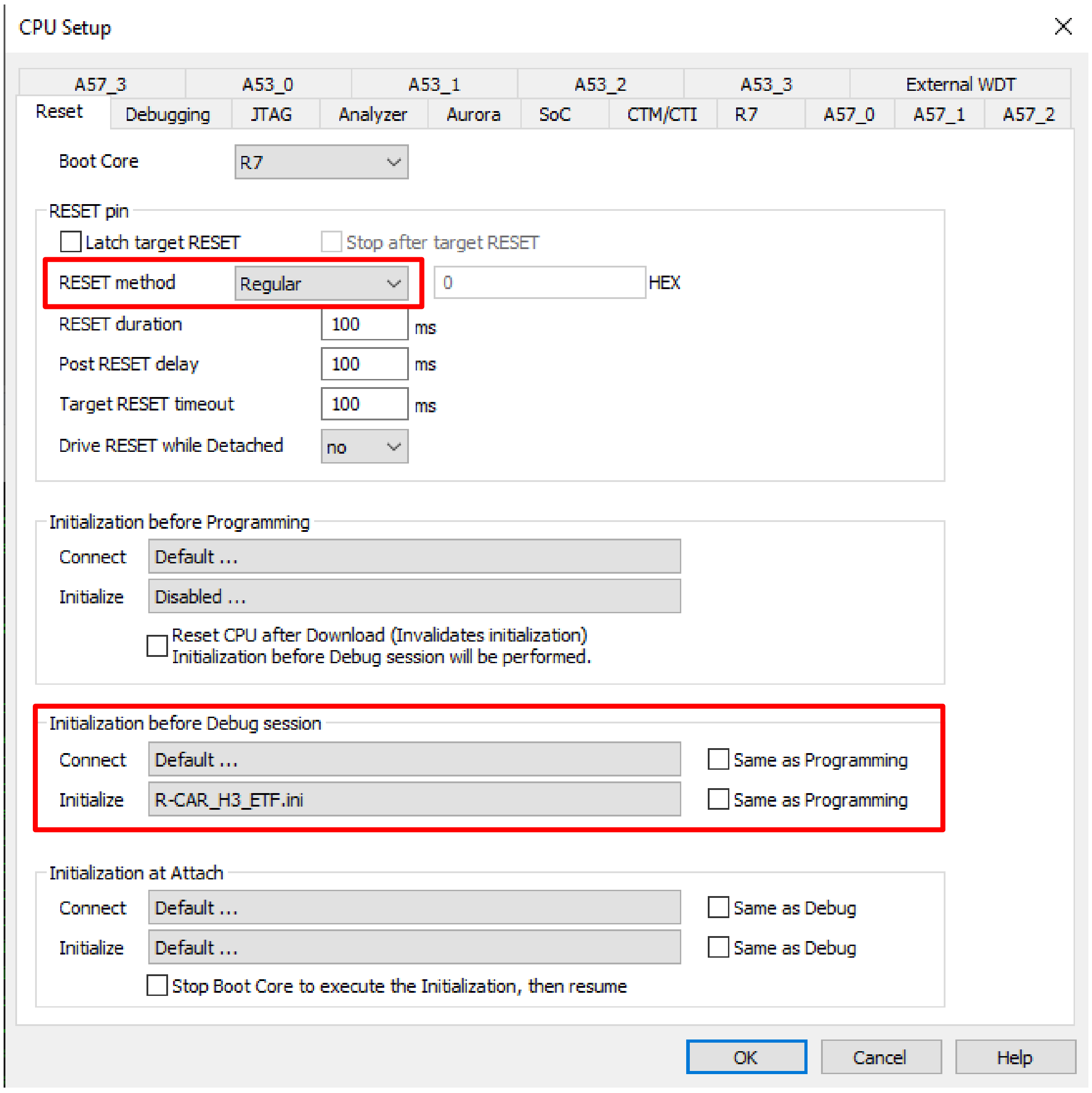
This chapter describes the SoC-specific configurations and initialization to be done in the winIDEA workspace to prepare the SoC for STM Trace.
Two trace options are explained with regards to storage of the trace messages. First we will discuss STM trace into an on-chip trace buffer, the Embedded Trace FIFO (ETF). The next section will explain how to stream out to a trace recording tool via a trace streaming interface such as the ARM High Speed Serial Trace Port (HSSTP).
Such a trace configuration may be sufficient for the timing analysis of only the bootloaders, excluding the (start-up) of the operating systems and applications.

5.1 STM Trace into ETF

This chapter explains how to use winIDEA to configure the R-Car SoC and the trace tool for STM trace buffering in the on-chip ETF.
The advantage of this approach is that no dedicated trace hardware is needed. This STM trace data is stored within the chip and subsequently the trace data can be read our via the standard JTAG debug interface. The disadvantage is the limited buffer size, thus the trace duration is limited.

5.1.1 Reset Selection & Device Initialization

On R-Car SoCs, the RESET method “Regular” can be used, i.e. the BlueBox asserts the Reset pin.
For tracing into the ETF, the device needs to be initialized before starting a debug/trace session. As shown in Figure 10 this can be accomplished by executing an iSYSTEM .ini file. The main device initialization performed in this case is enabling the global time stamp counter for generating the STM time stamp.


CPU Setup

A57_3 | A53_0 | A53_1 | A53_2 | A53_3 | External WDT
Reset | Debugging | JTAG | Analyzer | Aurora | SoC | CTM/CTI | R7 | A57_0 | A57_1 | A57_2

Boot Core: R7

RESET pin
- [ ] Latch target RESET
- [ ] Stop after target RESET

RESET method	Regular	0 HEX
RESET duration	100	ms
Post RESET delay	100	ms
Target RESET timeout	100	ms
Drive RESET while Detached	no	

Initialization before Programming

Connect	Default ...
Initialize	Disabled ...

- [ ] Reset CPU after Download (Invalidates initialization) Initialization before Debug session will be performed.

Initialization before Debug session

Connect	Default ...	[ ] Same as Programming
Initialize	R-CAR_H3_ETF.ini	[ ] Same as Programming

Initialization at Attach

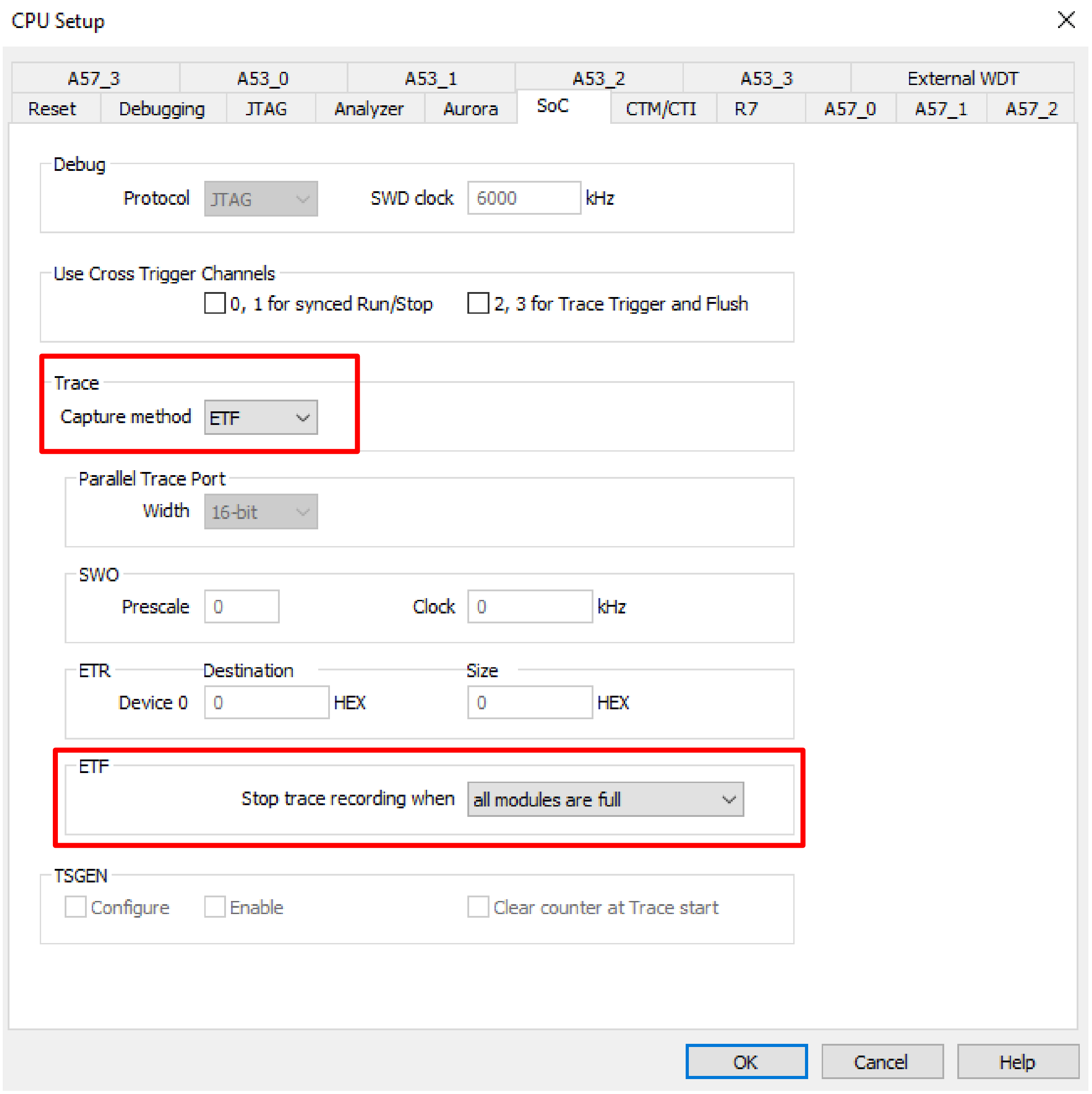
Connect	Default ...	[ ] Same as Debug
Initialize	Default ...	[ ] Same as Debug

- [ ] Stop Boot Core to execute the Initialization, then resume

Figure 10: winIDEA Hardware – CPU Options Dialog, Settings required for STM Trace into ETF

5.1.2 Trace Capture Method into ETF

In the dialog „Hardware – CPU Options… - SoC" the Trace Capture method needs to be set to ETF (= Embedded Trace FIFO).
The additional ETF trace settings, i.e. "ETF – Stop trace recording when: " and the TSGEN related settings are currently not supported and can be ignored.



Debug
Protocol: JTAG
SWD clock: 6000 kHz

Use Cross Trigger Channels
☐ 0, 1 for synced Run/Stop
☐ 2, 3 for Trace Trigger and Flush

Trace
Capture method: ETF

Parallel Trace Port
Width: 16-bit

SWO
Prescale: 0
Clock: 0 kHz

ETR
Destination: Device 0: 0 HEX
Size: 0 HEX

ETF
Stop trace recording when: all modules are full

TSGEN
☐ Configure
☐ Enable
☐ Clear counter at Trace start

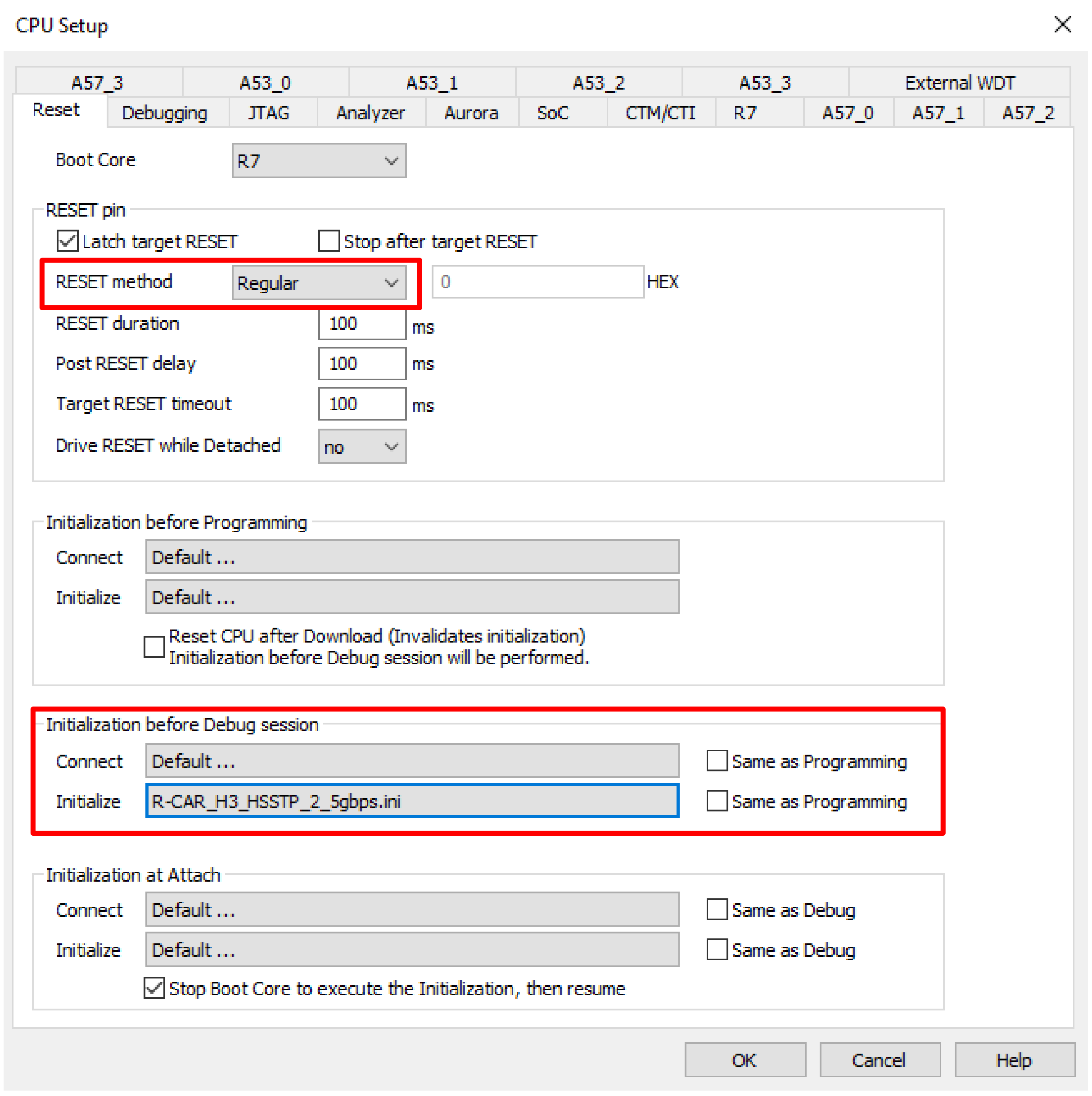
Figure 11: Trace into ETF Configuration in winIDEA Dialog "Hardware – CPU Options… - SoC"

5.2 STM Trace via HSSTP

This chapter explains how to use winIDEA to configure the R-Car SoC and the trace tool for STM trace via the HSSTP trace interface.
The advantage of this approach is that the trace duration is essentially unlimited, as the trace hardware allows for streaming the data from the HSSTP trace port of the SoC via the HSSTP Active Probe and the iC5700 BlueBox to the hard drive of the PC. The disadvantage is obviously that a dedicated HSSTP trace hardware setup is required.
Such a trace configuration is recommended in case the timing analysis (i.e. trace recording) shall not only cover the bootloaders, but also extend to the (start-up) of the operating systems and applications.

5.2.1 Reset Selection & Device Initialization

On R-Car SoCs, the RESET method “Regular” can be used, i.e. the BlueBox asserts the Reset pin.
For trace output via HSSTP, the device needs to be initialized before starting a debug/trace session. As shown in Figure 12 this can be accomplished by executing an iSYSTEM .ini file. The main device initializations performed in this case are the configuration of a PCIe channel to operate in HSSTP mode (with a bit-rate of 2.5Gbps) and enabling the global time stamp counter.



CPU Setup

Boot Core: R7

RESET pin

- [✓] Latch target RESET
- [ ] Stop after target RESET

RESET method: Regular | 0 HEX

RESET duration: 100 ms

Post RESET delay: 100 ms

Target RESET timeout: 100 ms

Drive RESET while Detached: no

Initialization before Programming

Connect: Default ...

Initialize: Default ...

- [ ] Reset CPU after Download (Invalidates initialization) Initialization before Debug session will be performed.

Initialization before Debug session

Connect: Default ... | [ ] Same as Programming

Initialize: R-CAR_H3_HSSTP_2_5gbps.ini | [ ] Same as Programming

Initialization at Attach

Connect: Default ... | [ ] Same as Debug

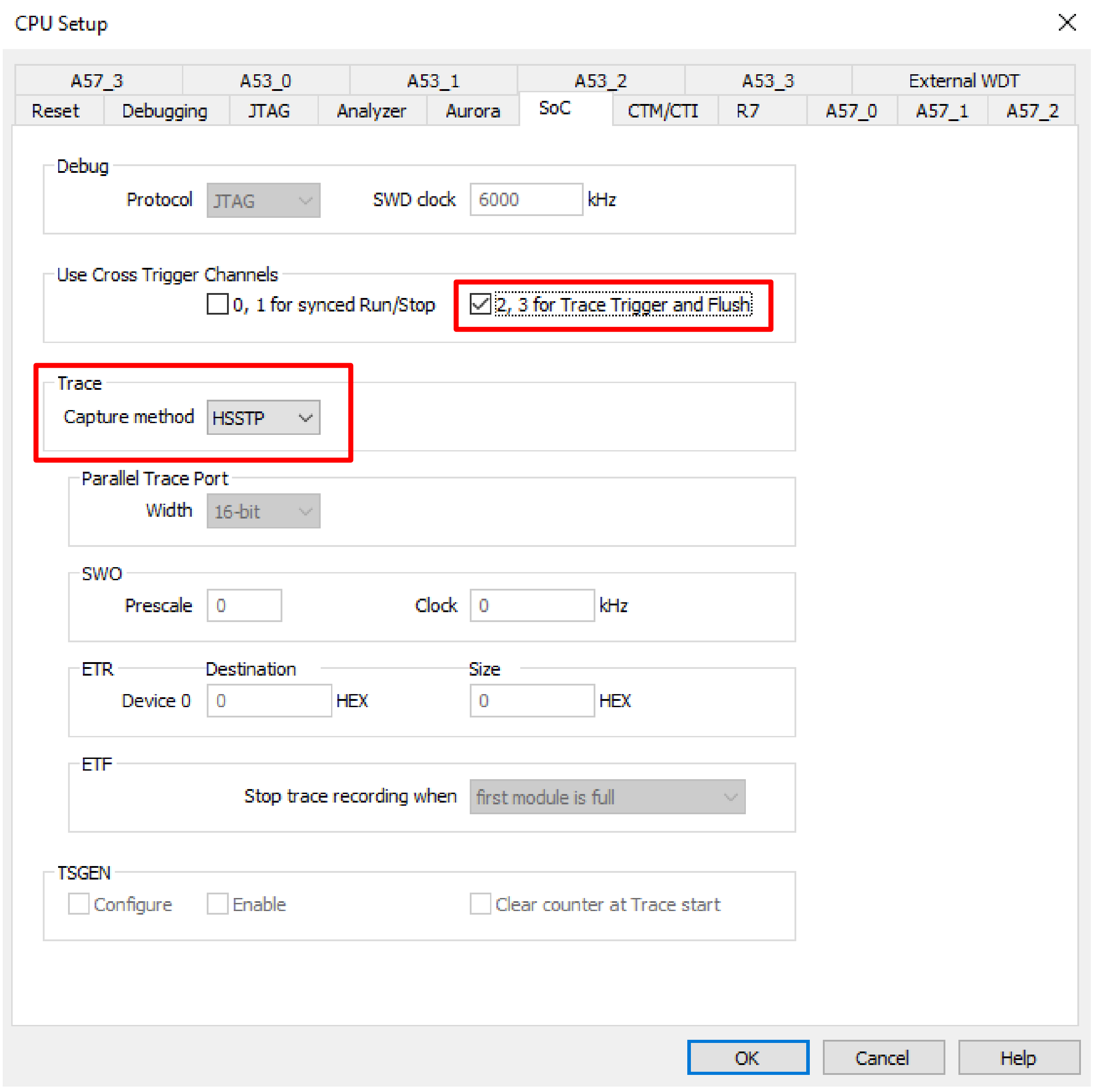
Initialize: Default ... | [ ] Same as Debug

- [✓] Stop Boot Core to execute the Initialization, then resume

Figure 12: winIDEA Hardware – CPU Options Dialog, Settings required for STM Trace via HSSTP

5.2.2 Trace Capture Method via HSSTP

In the dialog „Hardware – CPU Options… - SoC" the Trace Capture method needs to be set to HSSTP.


CPU Setup

Debug
Protocol JTAG SWD clock 6000 kHz

Use Cross Trigger Channels
☐ 0, 1 for synced Run/Stop ☑ 2, 3 for Trace Trigger and Flush

Trace
Capture method HSSTP

Parallel Trace Port
Width 16-bit

SWO
Prescale 0 Clock 0 kHz

ETR Destination Size
Device 0 0 HEX 0 HEX

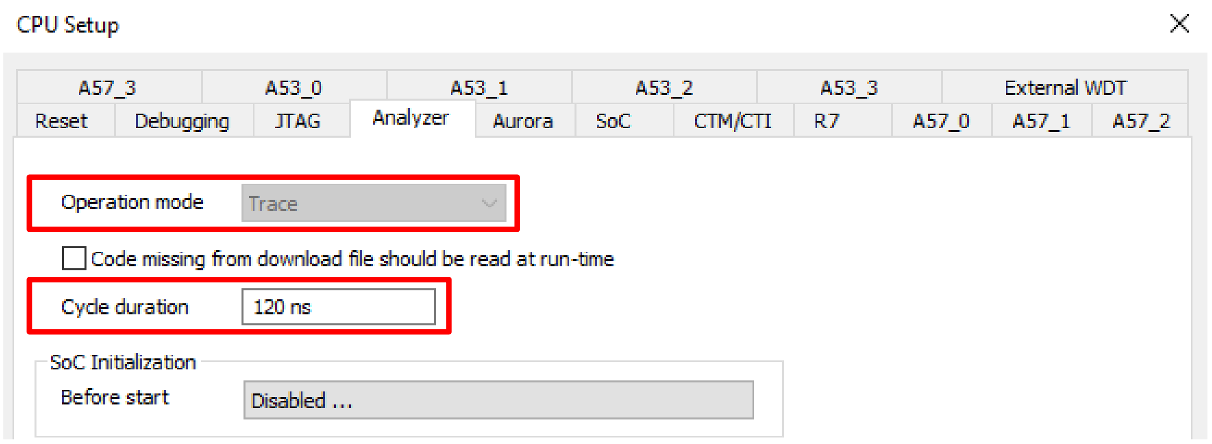
ETF
Stop trace recording when first module is full

TSGEN
☐ Configure ☐ Enable ☐ Clear counter at Trace start

Figure 13: Trace via HSSTP Configuration in winIDEA Dialog "Hardware – CPU Options… - SoC"

5.3 Analyzer Operation Mode and Trace Time Stamp Cycle Duration

For both trace methods described in the previous sections, the winIDEA Trace Analyzer needs to be aware of the global time stamp counter clock rate. This information is needed to allow the analyzer to convert the time stamp value, received from the SoC, into an absolute time value.
The “Cycle Duration” calculated according to the description in Section 2.3 needs to be entered in the dialog “Hardware – CPU Options… - Analyzer”.
In addition, the Analyzer Operation mode needs to be set to “Trace”.



Operation mode: Trace

☐ Code missing from download file should be read at run-time

Cycle duration: 120 ns

SoC Initialization
Before start: Disabled …

Figure 14: winIDEA Dialog “Hardware – CPU Options… - Analyzer”

6 winIDEA Analyzer Configuration

This chapter describes the required configurations in the winIDEA Trace Analyzer. This involves on one hand the setup of the SoC trace hardware, i.e. the STM module. On the other hand, the winIDEA Profiler needs to be configured to allow for a correct and user-friendly interpretation and visualization of the recorded STM trace data.

6.1 STM Trace Hardware Configuration

It is recommended not to modify the “Default” trace configuration, but instead create a new trace configuration, dedicated to each individual trace use-case. Figure 15 shows how to create a new trace configuration.

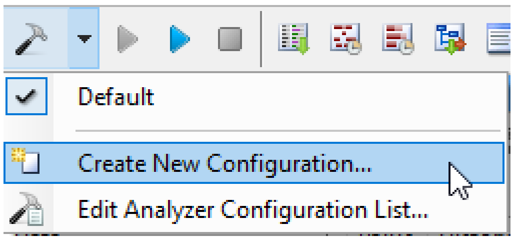
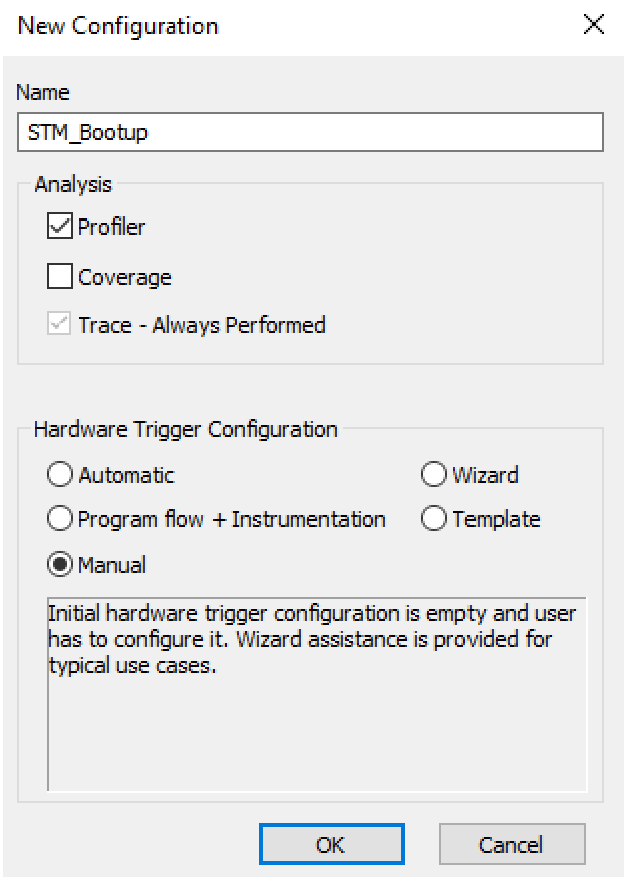

Default
Create New Configuration...
Edit Analyzer Configuration List...

Figure 15: Create a new Trace Configuration for STM Tracing

In our use-case of STM boot-up profiling, we only need a Profiler analysis of the recorded trace data, but we don’t need any Coverage analysis.
The “Hardware Trigger Configuration” will be done manually.


New Configuration

Name: STM_Bootup

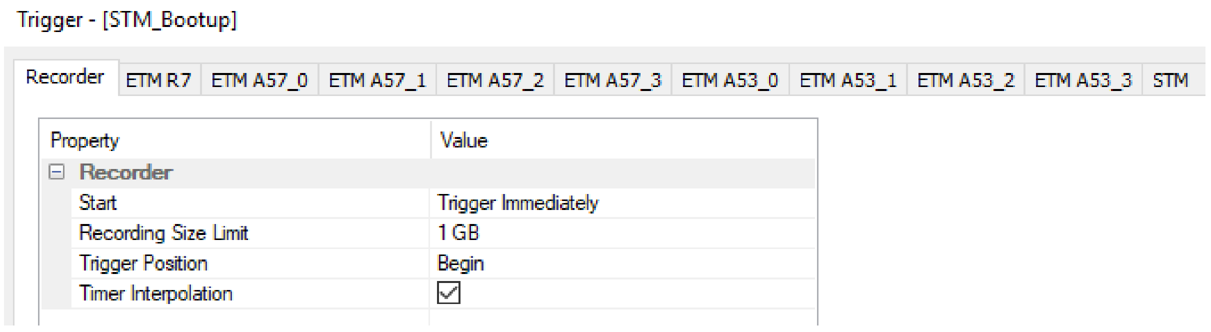
Analysis
- ☑ Profiler
- ☐ Coverage
- ☑ Trace - Always Performed

Hardware Trigger Configuration
- ○ Automatic
- ○ Wizard
- ○ Program flow + Instrumentation
- ○ Template
- ● Manual

Initial hardware trigger configuration is empty and user has to configure it. Wizard assistance is provided for typical use cases.

Figure 16: New Trace Configuration for manual Trace Hardware Configuration and STM based Profiling

Figure 17 shows the required Trace Recorder (i.e. iSYSTEM BlueBox) settings.

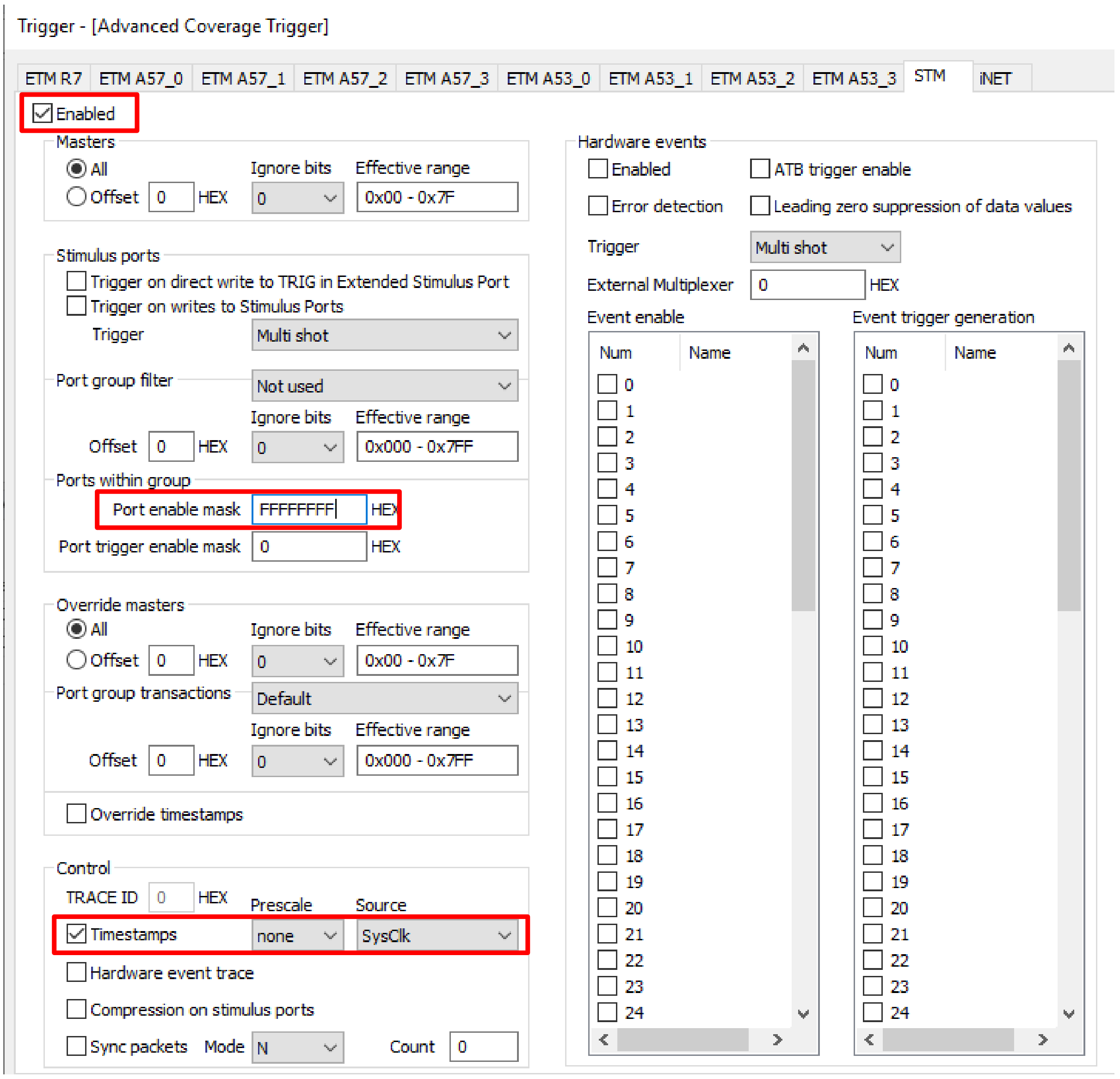

Trigger - [STM_Bootup]

Recorder | ETM R7 | ETM A57_0 | ETM A57_1 | ETM A57_2 | ETM A57_3 | ETM A53_0 | ETM A53_1 | ETM A53_2 | ETM A53_3 | STM

Property	Value
Recorder	
Start	Trigger Immediately
Recording Size Limit	1 GB
Trigger Position	Begin
Timer Interpolation	☑

Figure 17: Trace Recorder (BlueBox) Settings for STM Trace into

Figure 18 depicts the required hardware configuration of the STM module.
All other tabs (ETMs and iNET) are not used, i.e. the corresponding hardware modules (e.g. ETMs) are left disabled.


Trigger - [Advanced Coverage Trigger]

ETM R7 | ETM A57_0 | ETM A57_1 | ETM A57_2 | ETM A57_3 | ETM A53_0 | ETM A53_1 | ETM A53_2 | ETM A53_3 | STM | iNET

☑ Enabled

Masters
- ● All — Ignore bits: 0 — Effective range: 0x00 - 0x7F
- ○ Offset 0 HEX

Stimulus ports
- ☐ Trigger on direct write to TRIG in Extended Stimulus Port
- ☐ Trigger on writes to Stimulus Ports
- Trigger: Multi shot
- Port group filter: Not used
- Offset 0 HEX — Ignore bits: 0 — Effective range: 0x000 - 0x7FF

Ports within group
- Port enable mask: FFFFFFFF HEX
- Port trigger enable mask: 0 HEX

Override masters
- ● All — Ignore bits: 0 — Effective range: 0x00 - 0x7F
- ○ Offset 0 HEX
- Port group transactions: Default
- Offset 0 HEX — Ignore bits: 0 — Effective range: 0x000 - 0x7FF
- ☐ Override timestamps

Control
- TRACE ID 0 HEX
- ☑ Timestamps — Prescale: none — Source: SysClk
- ☐ Hardware event trace
- ☐ Compression on stimulus ports
- ☐ Sync packets — Mode: N — Count: 0

Hardware events
- ☐ Enabled
- ☐ ATB trigger enable
- ☐ Error detection
- ☐ Leading zero suppression of data values
- Trigger: Multi shot
- External Multiplexer: 0 HEX

Event enable

Num	Name
☐ 0	
☐ 1	
☐ 2	
☐ 3	
☐ 4	
☐ 5	
☐ 6	
☐ 7	
☐ 8	
☐ 9	
☐ 10	
☐ 11	
☐ 12	
☐ 13	
☐ 14	
☐ 15	
☐ 16	
☐ 17	
☐ 18	
☐ 19	
☐ 20	
☐ 21	
☐ 22	
☐ 23	
☐ 24	

Event trigger generation

Num	Name
☐ 0	
☐ 1	
☐ 2	
☐ 3	
☐ 4	
☐ 5	
☐ 6	
☐ 7	
☐ 8	
☐ 9	
☐ 10	
☐ 11	
☐ 12	
☐ 13	
☐ 14	
☐ 15	
☐ 16	
☐ 17	
☐ 18	
☐ 19	
☐ 20	
☐ 21	
☐ 22	
☐ 23	
☐ 24	

Figure 18: STM Trace Hardware Configuration

First, we need to enable the STM module. The STM module would allow to mask out certain STM ports, i.e. channels (for instance for filtering out certain channels from generating trace messages). However, on our case we enable all ports, by setting the mask to 0xFFFFFFFF.
Finally, we need to enable STM timestamps. The STM timestamps shall be derived directly from the global time stamp counter (i.e. SysClk), without any clock pre-scaling.

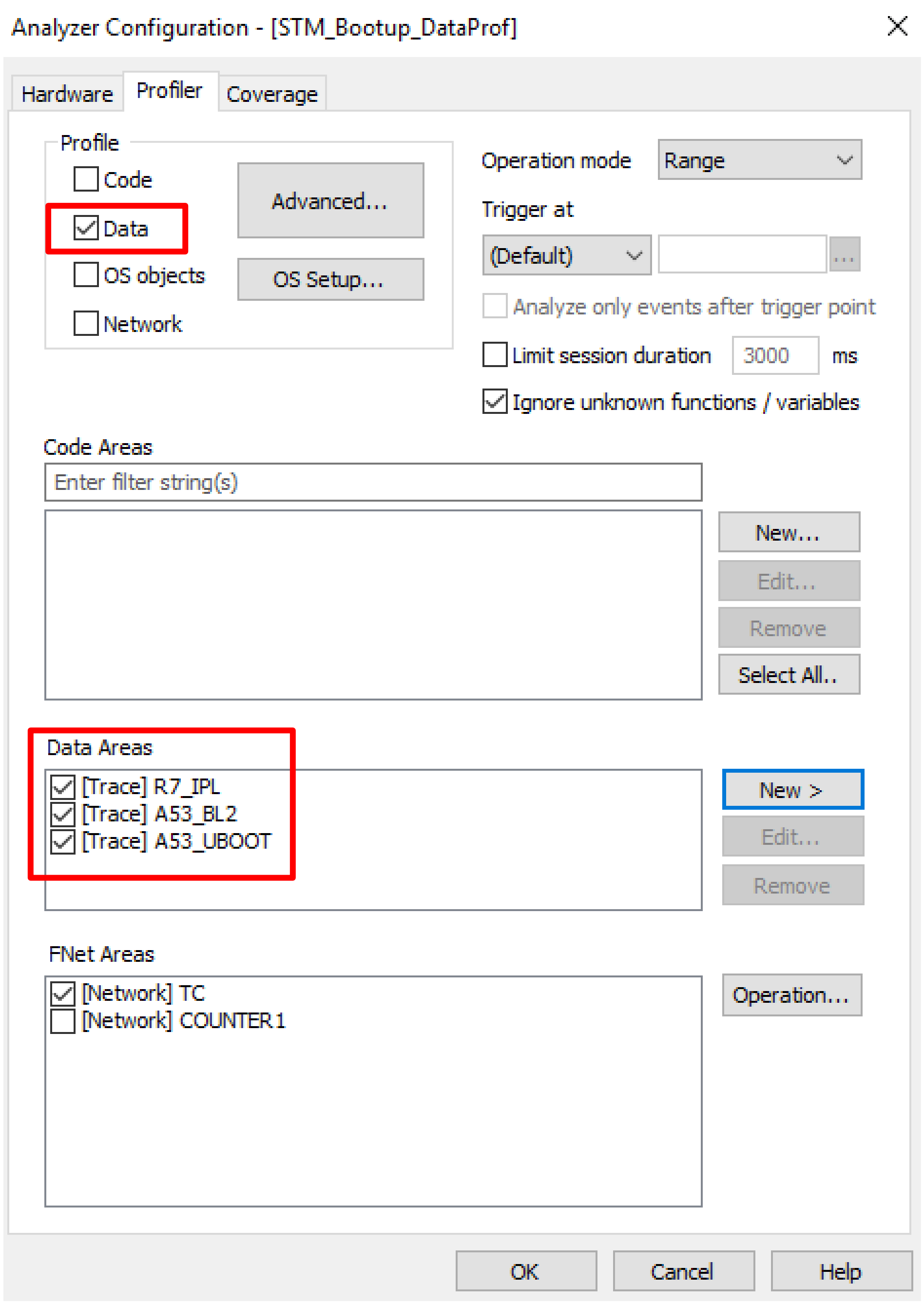
7 winIDEA Profiler Configuration & Visualization

As mentioned earlier, the Profiler needs to be able to interpret and visualize the recorded STM trace data in a correct and user-friendly way.
In terms of interpretation, this means, that the profiler needs to understand the allocation of the STM channels to specific event types, e.g. a specific STM channel is used for function profiling of a specific bootloader. In addition, the profiler needs to be able to map the function IDs to user-understandable function names. Finally, these function names need to appear in the Profiler Timeline view.

7.1 Function Profiling by means of Data Profiling

The most basic approach for function profiling is provided via data profiling.
In this case, the basic data profiling is extended for function profiling by instructing the profiler to interpret any recorded data value X which is not equal to 0, as an entry into a function X and to interpret a data value equal to 0 as an exit from a function.

For data profiling by means of STM trace, we first need to create a Profiler Data Area for each used STM Channel.


Analyzer Configuration - [STM_Bootup_DataProf]

Hardware | Profiler | Coverage

Profile
- [ ] Code
- [x] Data
- [ ] OS objects
- [ ] Network

Advanced... | OS Setup...

Operation mode: Range
Trigger at: (Default)
- [ ] Analyze only events after trigger point
- [ ] Limit session duration 3000 ms
- [x] Ignore unknown functions / variables

Code Areas
Enter filter string(s)

New... | Edit... | Remove | Select All..

Data Areas
- [x] [Trace] R7_IPL
- [x] [Trace] A53_BL2
- [x] [Trace] A53_UBOOT

New > | Edit... | Remove

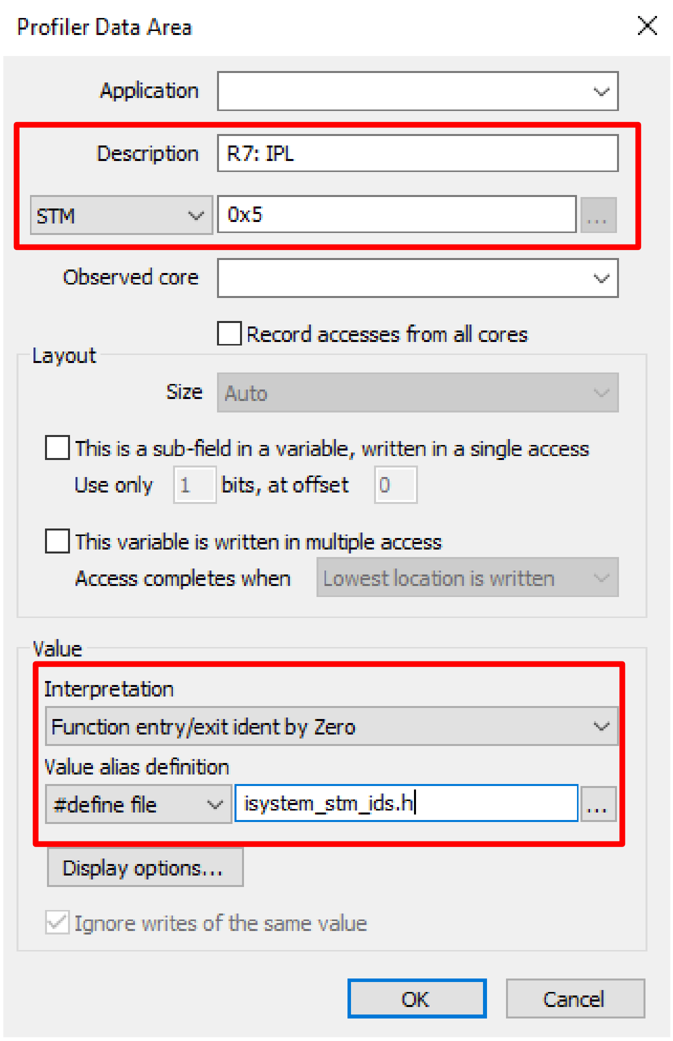
FNet Areas
- [x] [Network] TC
- [ ] [Network] COUNTER1

Operation...

OK | Cancel | Help

Figure 19: Overall Profiler Configuration for STM Data Profiling

In the sample profiler data area configuration shown in Figure 20, the data area is used to profile functions of the R7 IPL, using STM channel 5.
The Interpretation option “Function entry/exit ident by Zero” instructs the profiler to interpret trace data values equal to 0 as function exits.
The C header file selected as “#define file”, includes the ID-to-Function name mapping. The IDs are the STM data values signaled when entering the functions (see also Listing 7).


Profiler Data Area

Application:

Description: R7: IPL

STM: 0x5

Observed core:

☐ Record accesses from all cores

Layout

Size: Auto

☐ This is a sub-field in a variable, written in a single access

Use only 1 bits, at offset 0

☐ This variable is written in multiple access

Access completes when: Lowest location is written

Value

Interpretation: Function entry/exit ident by Zero

Value alias definition: #define file — isystem_stm_ids.h

Figure 20: Sample Profiler Data Area Settings for Function Profiling (R7 IPL Functions using STM Channel 5)

```
#ifndef __ISYSTEM_STM_IDS_H__
#define __ISYSTEM_STM_IDS_H__

#define STM_IPL_LOADER              0x0001
#define STM_IPL_HW_INIT             0x0010
#define STM_IPL_DRAM_INIT           0x0012
#define STM_IPL_TCM_LOADER          0x0020
#define STM_IPL_INIT_RPC            0x0030
#define STM_IPL_INIT_DMA            0x0040
#define STM_IPL_LOAD_R7_RTOS        0x0050
#define STM_IPL_LOAD_A5x_BL2        0x0060
#define STM_POWERUP_A5x             0x0070

#endif /* __ISYSTEM_STM_IDS_H__ */
```
Listing 7: Sample Data Profiler “#define file” for R7 IPL Function Profiling

In addition, the Data Profiler may be configured to strip-off certain prefixes of the function names when displaying the functions in the Profiler views. For instance, the prefix “STM_” can be omitted, so that the function name is displayed as, e.g. “IPL_LOADER” instead of “STM_IPL_LOADER”. Some specific prefixes may be needed to have unique macro names for the code instrumentation.

7.2 Function Profiling by means of OS Profiling

The more advanced concept for function profiling utilizes the OS profiling capabilities of winIDEA. The advantage of this approach is that it is not limited to just functions, but can be extended to also cover other types of events or objects such as OS task switches, user-specific data objects, etc... The awareness of the Profiler for all such events and objects is achieved by means of an iSYSTEM-proprietary XML file which is included into the winIDEA workspace.
Typically, such a Profiler XML file is used to describe the structure of an operating system, such as an AUTOSAR OS (i.e. the so-called OS Awareness). However, by means of this XML scheme is, the AUTOSAR-awareness can also extend to not only cover the OS but also other objects such as AUTOSAR RTE Runnables. This mix of OS and Runnable awareness can be utilized for our use-case of SoC boot-up profiling, meaning, the instrumented functions of the bootloaders can be treated as Runnables and the start-up of an operating system, e.g. an AUTOSAR Classic OS or a Linux kernel of an AUTOSAR Adaptive stack, can be handled by the OS awareness that come with the OS profiling approach.

7.2.1 winIDEA OS Awareness

In order to enable the OS awareness via a Profiler XML file in winIDEA, you have to add a new OS awareness file via the menu “Debug – Operating System”. Then you add a new OS and select the OS type “OSEK AUTOSAR”.

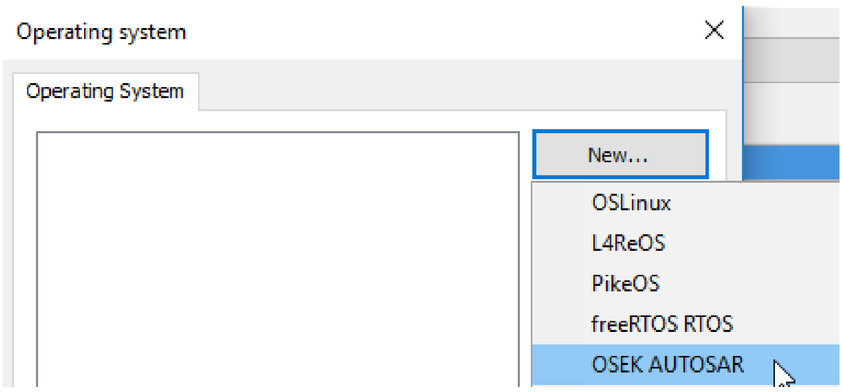
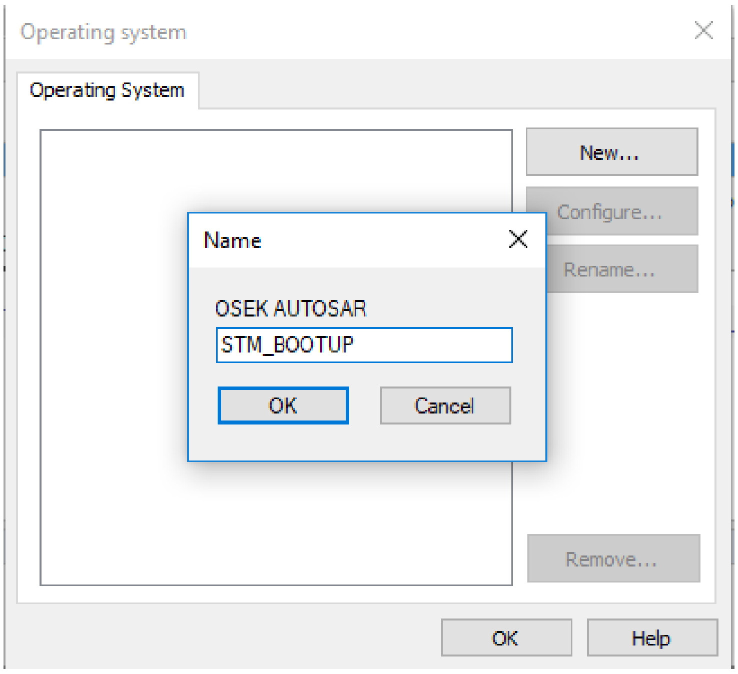


Figure 21: winIDEA Profiler Configuration via “OSEK AUTOSAR” Awareness

Optionally, you can give this OS awareness an arbitrary name, e.g. “STM_BOOTUP”.

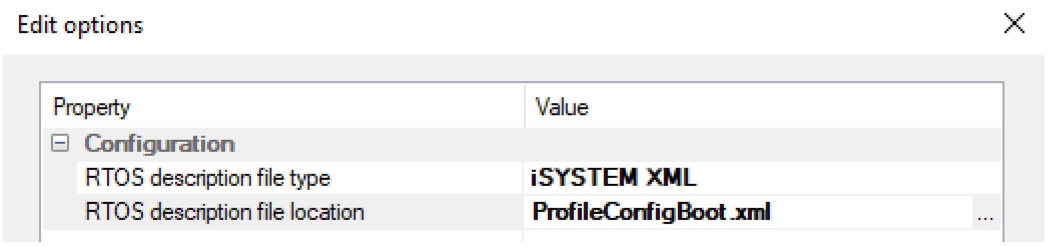


OSEK AUTOSAR

STM_BOOTUP

Figure 22: “OSEK AUTOSAR” awareness used for STM_BOOTUP profiling

Finally, you have to select the corresponding iSYSTEM XML file, as shown in Figure 23.

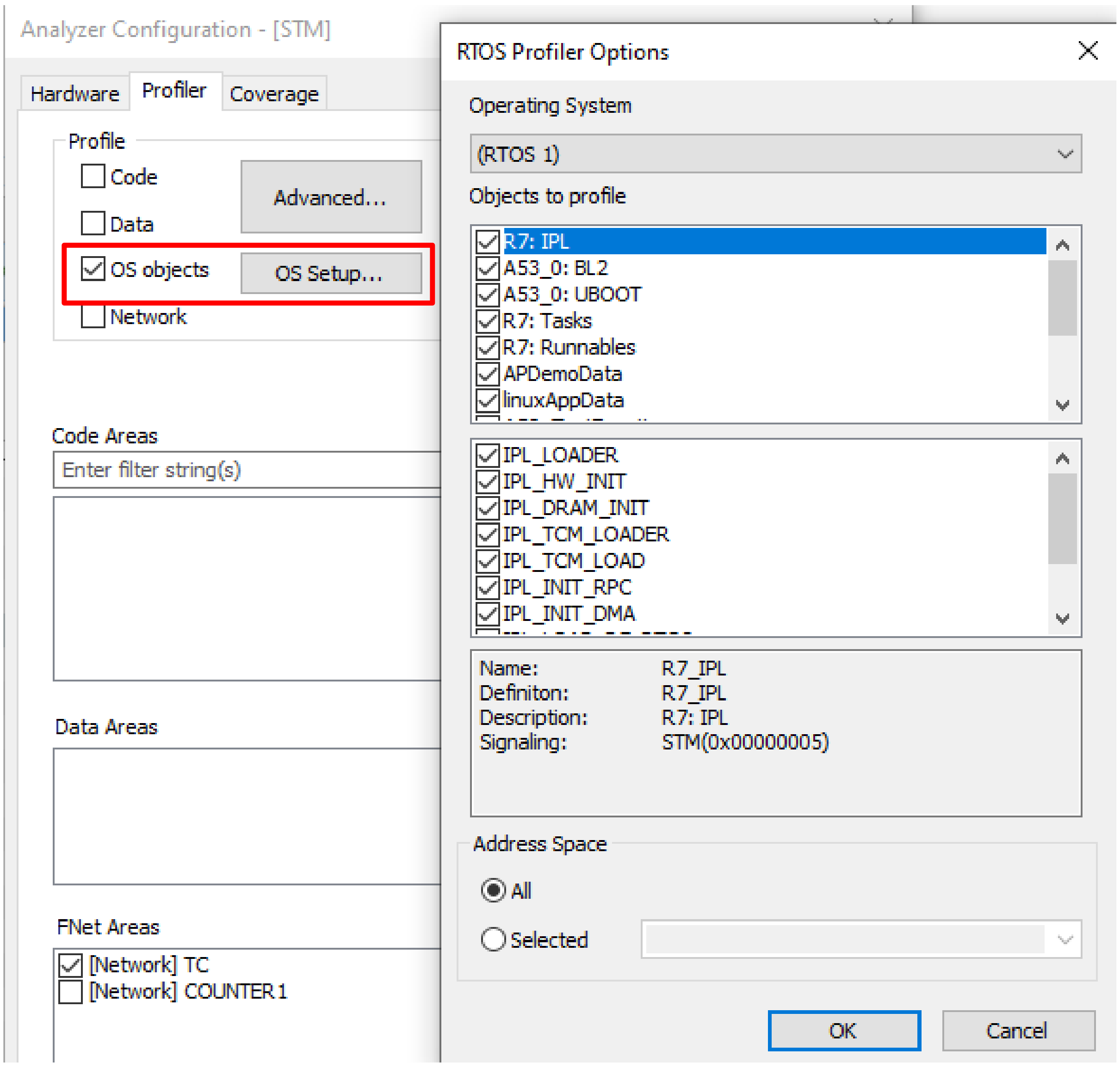


Edit options

Property	Value
⊟ Configuration	
RTOS description file type	iSYSTEM XML
RTOS description file location	ProfileConfigBoot.xml

Figure 23: Selection of the iSYSTEM Profiler XML File

The winIDEA Profiler needs to be configured for OS profiling as shown in Figure 24.



Analyzer Configuration - [STM]

Hardware | Profiler | Coverage

Profile
- ☐ Code
- ☐ Data
- ☑ OS objects
- ☐ Network

Advanced... | OS Setup...

Code Areas: Enter filter string(s)

Data Areas

FNet Areas
- ☑ [Network] TC
- ☐ [Network] COUNTER1

RTOS Profiler Options

Operating System: (RTOS 1)

Objects to profile
- ☑ R7: IPL
- ☑ A53_0: BL2
- ☑ A53_0: UBOOT
- ☑ R7: Tasks
- ☑ R7: Runnables
- ☑ APDemoData
- ☑ linuxAppData

- ☑ IPL_LOADER
- ☑ IPL_HW_INIT
- ☑ IPL_DRAM_INIT
- ☑ IPL_TCM_LOADER
- ☑ IPL_TCM_LOAD
- ☑ IPL_INIT_RPC
- ☑ IPL_INIT_DMA

Name:	R7_IPL
Definiton:	R7_IPL
Description:	R7: IPL
Signaling:	STM(0x00000005)

Address Space
- ● All
- ○ Selected

OK | Cancel

Figure 24: OS Profiler Configuration for Boot-Up Profiling using the iSYSTEM XML file.

7.2.2 iSYSTEM Profiler XML File

Listing 8 shows a sample iSYSTEM XML file. It contains all information relevant for the profiler to visualize the boot-up process as signaled by the STM trace instrumentation within the bootloaders.

```
<?xml version='1.0' encoding='UTF-8' ?>
<OperatingSystem>
  <Name>RCARH3_Boot</Name>
  <NumCores>6</NumCores>

  <Types>
    <TypeEnum><Name>TypeEnum_R7_IPL</Name>
      <Enum><Name>IPL_LOADER</Name><Value>0x01</Value></Enum>
      <Enum><Name>IPL_HW_INIT</Name><Value>0x10</Value></Enum>
      <Enum><Name>IPL_DRAM_INIT</Name><Value>0x12</Value></Enum>
      <Enum><Name>IPL_TCM_LOADER</Name><Value>0x20</Value></Enum>
      <Enum><Name>IPL_INIT_RPC</Name><Value>0x30</Value></Enum>
      <Enum><Name>IPL_INIT_DMA</Name><Value>0x40</Value></Enum>
      <Enum><Name>IPL_LOAD_R7_RTOS</Name><Value>0x50</Value></Enum>
      <Enum><Name>IPL_LOAD_A5x_BL2</Name><Value>0x60</Value></Enum>
      <Enum><Name>IPL_POWERUP_A5x</Name><Value>0x70</Value></Enum>
    </TypeEnum>

    <TypeEnum><Name>TypeEnum_A53_0_BL2_TYPE</Name>
      <Enum><Name>BL2</Name><Value>0x70</Value></Enum>
    </TypeEnum>

    <TypeEnum><Name>TypeEnum_UBOOT_RUN_TYPE</Name>
      <Enum><Name>UBOOT</Name><Value>0x60</Value></Enum>
    </TypeEnum>

    <TypeEnum>
      <Name>TypeEnum_R7_TaskMapping</Name>
      <Enum><Name>NoTask</Name><Value>0</Value></Enum>
      <Enum><Name>Task_A</Name><Value>1</Value></Enum>
      <Enum><Name>Task_B</Name><Value>2</Value></Enum>
        <Enum><Name>Task_IDLE</Name><Value>0xFF</Value></Enum>
    </TypeEnum>

  </Types>

  <Profiler>

    <Object>
      <Definition>R7_IPL</Definition>
      <Description>R7: IPL</Description>
      <Name>R7_IPL</Name>
      <Signaling>STM(0x00000005)</Signaling>
      <Level>Runnable</Level>
      <Type>TypeEnum_R7_IPL</Type>
      <Properties>
        <Runnable_MaskID>0xffffffff</Runnable_MaskID>
        <RunnableExitValue>0</RunnableExitValue>
      </Properties>
    </Object>

    <Object>
      <Definition>A53_0_BL2</Definition>
      <Description>A53_0: BL2</Description>
      <Name>A53_0_BL2</Name>
      <Level>Runnable</Level>
      <Signaling>STM(0x00000007)</Signaling>
      <Type>TypeEnum_A53_0_BL2_TYPE</Type>
      <Properties>
        <Runnable_MaskID>0xffffffff</Runnable_MaskID>
        <RunnableExitValue>0</RunnableExitValue>
      </Properties>
    </Object>

    <Object>
      <Definition>UBOOT</Definition>
      <Description>A53_0: UBOOT</Description>
      <Name>UBOOT</Name>
      <Level>Runnable</Level>
      <Signaling>STM(0x00000006)</Signaling>
      <Type>TypeEnum_UBOOT_RUN_TYPE</Type>
      <Properties>
        <Runnable_MaskID>0xffffffff</Runnable_MaskID>
        <RunnableExitValue>0</RunnableExitValue>
      </Properties>
    </Object>

    <Object>
      <Definition>AUTOSAR_TASK</Definition>
      <Description>R7: Tasks</Description>
      <Name>AUTOSAR_TASK</Name>
      <Type>TypeEnum_R7_TaskMapping</Type>
      <Expression>osRunningTask</Expression>
      <DefaultValue>Task_IDLE</DefaultValue>
      <Level>Task</Level>
      <Core>0</Core>
    </Object>

  </Profiler>

</OperatingSystem>
```
Listing 8: Sample Profiler XML File

The “OperatingSystem” root element of the XML file consists of four nodes, i.e. “Name”, “NumCores”, “Types” and “Profiler”.
The sub-node “Objects” of the “Profiler” node describes the objects which will be visible in the winIDEA Profiler, e.g. “R7 IPL:” (see also Figure 25). This object description comprises the name of the object, the method for tracing this object (e.g. via STM channel 5) and a link to an enumeration type of the “Types” node.
The sub-nodes “TypeEnum” define enumeration types which map integer numbers received via STM trace to meaningful strings. These stings are displayed in the profiler as areas of that profiler object (e.g. the “IPL_TCM_LOADER” area of the “R7: IPL:” object).

7.3 Visualization

Figure 25 shows a sample winIDEA Function Profiler Timeline, generated by means of STM trace and Function Profiling via OS Profiling as described in 7.2 “Function Profiling by means of OS Profiling”. The profiling includes the R7 IPL, A5x bootloaders BL2 and U-Boot as well as the AUTOSAR Classic OS tasks running on the R7 core.

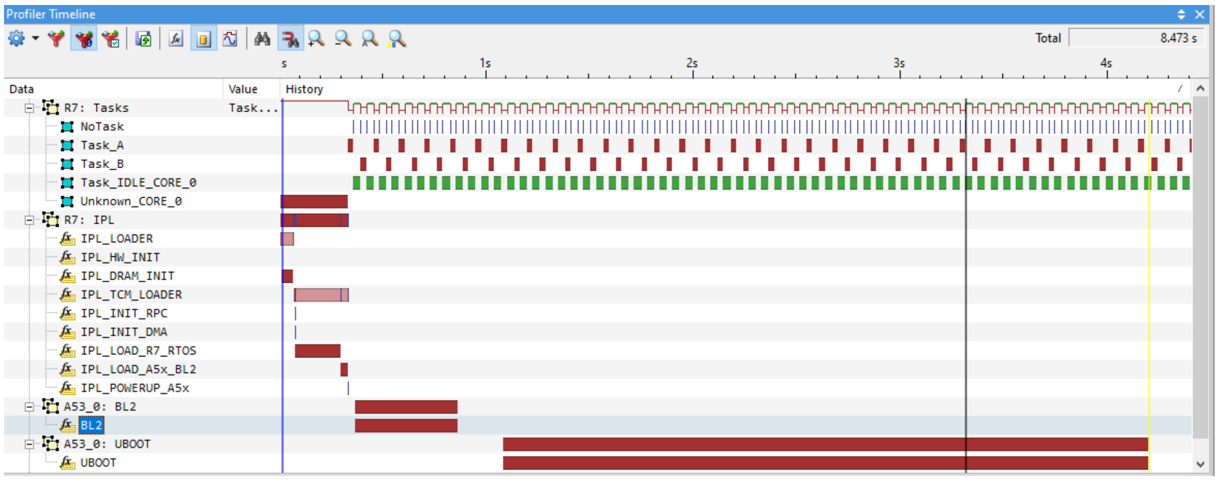


Figure 25: Sample winIDEA Profiler Timeline of an R-Car Boot-up Sequence

8 Technical support

8.1 Online resources

Online Help ▸ winIDEA and testIDEA online help	Knowledge Base ▸ Tips & tricks categorized by issue type and architecture	Tutorials ▸ From a beginner to an expert
Technical Notes ▸ How-tos for winIDEA functionalities with scripts	Application Notes ▸ How-to notes on advanced use-cases	Webinars ▸ Technical webinars about ISYSTEM tools with use cases

8.2 Contact

Please visit https://www.isystem.com/contact.html for contact details.