AN12437

i.MX RT Series Performance Optimization

Rev. 1 — Febuary 2020

Application Note

Contents

1 Introduction

i.MX RT series take advantage of the Arm® Cortex®-M7 core with 32K/32K L1 I/D-Cache, which operates at the speed up to 600 MHz to provide high CPU performance and best real-time response.

- i.MX RT1050 processor has 512 KB on-chip RAM, which can be flexibly configured as TCM or general-purpose on-chip RAM.
- i.MX RT1060 processor has extra 512 KB OCRAM, totally 1 MB on-chip RAM.

i.MX RT series provide various memory interfaces, including SDRAM, RAW NAND FLASH, NOR FLASH, SD/eMMC, and FlexSPI. These rich features help i.MX RT series to implement flexible applications and high performance. The system performance running in these memory devices depends on system and memory type.

This document intends to introduce how to optimize the system performance running on different memory device.

2 Overview

As integrated with high performance of the Cortex-M7 core, the i.MX RT can:

- Run up to 600 MHz.
- Enhance the performance with 32 K DCACHE and ICACHE.
- Partition 512 KB FlexRAM to DTCM/ITCM/OCRAM-based application with a flexible and configurable FlexRAM.

Refer to AN12077 for how to configure FlexRAM.

i.MX RT is flashless. However, it is embedded with the high performance internal SRAM and integrates the rich peripherals to interface with lots of memory devices, such as, SDRAM, RAW NAND FLASH, NOR FLASH, SD/eMMC, Quad SPI flash, and hyper flash.

According to the working mode, memory can be divided into two types.

- XIP memory: Executing codes in place.
- Non-XIP memory: Not supporting the codes executing in place but loading the code to executable memory.

The below lists the executable memory supported by i.MX RT series.

- ITCM/DTCM
- SDRAM
- OCRAM
- Hyper RAM
- Hyper/Octal NOR Flash (XIP support)
- QSPI NOR Flash (XIP support)
- Parallel NOR Flash (XIP support)

- Parallel SRAM

Based on the bus architecture and memory characteristics, difference memory present different performance. Figure 1 shows the bus architecture of RT series, taking the i.MX RT1060 system bus diagram as an example.

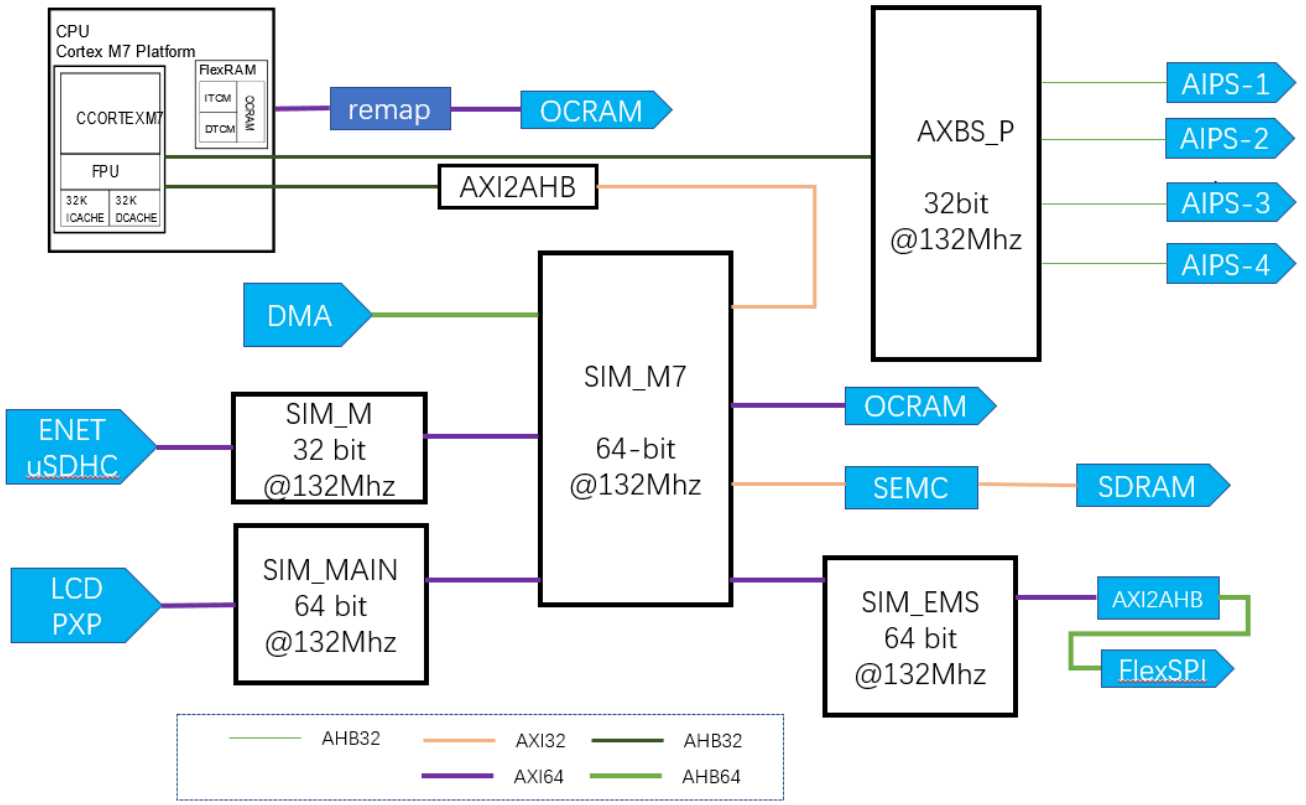

Figure 1. i.MX RT1060 system bus diagram

As shown in Figure 1, TCM is tightly coupled with M7 core and contains the same frequency with core. OCRAM and SEMC connect to SIM_M7 fabric, and FlexSPI connects to SIM_EMS. it show the different performance to different master accessing the same memory. For example,

- TCM shows high performance accessed by MCU core.
- OCRAM shows higher performance than TCM when accessed by DMA, while lower performance when accessed by MCU core. The reason is that OCRAM and DMA are in this same bus fabric, with less latency during the access.

Table 1 describes the bus fabric summary.

Table 1. Bus fabric summary

Name	Bus width	Typical frequency	Comments
SIM_M7	64	132 MHz	All the fabric runs at the same clock frequency and it is always m:1 synchronous to M7 core clock. This table is based on the core frequency of 528 MHz.
SIM_MAIN	64		
SIM_EMS	64		
SIM_AXBS_P	32		
SIM_M	32		

Table 2 describes the bus bandwidth of each memory supported by i.MX RT.

Table 2. Memory bus bandwidth

Memory	Bus width (:bit)	Max speed (:MHz)	Comments
ITCM	64	600	It has two DTCM controllers with 32 bit, available to access odd or even address by different controller.
DTCM	2*32	600	
OCRAM	64	132	
SDRAM	16	166	
Hyper RAM	8 (DDR)	166	
Hyper flash	8 (DDR)	166	
QSPI flash	4	133	

The bus bandwidth is the major element to impact the memory performance, but it is not all for device memory performance. There are some enhance features from the bus architecture to improve memory performance, such as ICACHE/DCACHE. FlexSPI IP supports extra 1 KB RX AHB prefetch buffer, and it may prefetch the flash data to dedicated buffer, which saves the access latency during the read access. However, the improvement depends on application. For example, it gets improvement more when the cache hit rate is high and accessing the QSPI flash is in sequence. The system performance is related to memory device and application case. It can get the similar performance in some application cases, as shown in Table 7. However, it has the big gap running on different memory in the other case. The below describes what is gap and how to improve it.

3 Memory performance test

Memory performance depends on memory characteristics, system architecture, and other facts, such as, cache, prefetch buffer and pipeline, and so on.

The same memory presents different performance when accessed by different masters (CPU Core, PXP, LCD, CSI, USB, eDMA, and others). For example, SDRAM can reach to high throughput when accessed by LCD and PXP, as these two masters support back-to-back access. It can get better performance comparing other master access, but drop more when accessed by CPU core. The following performance discussion is based on the access by CPU core.

3.1 SDRAM performance

i.MX RT series support to interface with 8/16-bit SDRAM device and can run up to 166 MHz. Table 3 shows the test result of transferring, by reading/writing 4096 bytes which measures the duration of SDRAM transferring by system tick.

Table 3. SDRAM performance

Items	Performance (:MB/s)		Comments
	DCache enabled	DCache disabled	
SDRAM read	111	25	SDRAM Working @ 166 MHz
SDRAM write	323	322	SDRAM Working @ 166 MHz

Table 3 shows the good performance on SDRAM write access. The benefits are from the pipeline and SEMC IP high performance, also cache improved more on reading performance.

To reproduce the test above, you can fetch the test code from the attached software package. The test steps are as follows.

- Unzip the **performance test** package and open `semc.eww` through `C:\Users\nxa18895\Desktop\New folder\AN12437SW\boards\evkbimxrt1050\demo_apps\performance_test\sdram_perforamnce_test\iar`. Please first install IAR version 8.40 or later.
- Build the **debug** sub-project to generate the `s-record` file. The macro **DCACHE_ENABLE** is used to disable or enable DCACHE. You can modify it based test requirements.

- Generate the `sb` file with the following commands.

```
elftosb.exe -f imx -V -c imx-itcm-unsigned.bd -o ivt_flexspi_nor_normal.bin semc.srec
```

```
elftosb.exe -f kinetis -V -c program_flexspinor_image_hyperflash.bd -o boot_image.sb
```

```
ivt_flexspi_nor_normal_nopadding.bin
```

- Program the flash by MFGTools based on the IMXRT1050-EVKB board.
- The code is running on the internal ITCM. Do not directly run the code by debugging, which may affect performance.

After downloading the code to flash, you can run and see the test results in serial terminals.

NOTE

The last test is for hyper flash. Configure the flash and enable it working at the target speed. For details, refer to FlexSPI performance.

```
DCACHE is enabled!

 SEMC SDRAM Performance test Start!

 Start test SDRAM write performance!
##sdram write perf###t1: 324695; t2: 317090; diff: 7605; ns: 12675,
datasize: 4096 byte; perf: 323MB/s; g_ms: 0

 Start test SDRAM read performance!
sdram read and write correctly!
##sdram read perf###t1: 201722; t2: 179718; diff: 22004; ns: 36673,
datasize: 4096 byte; perf: 111MB/s; g_ms: 0

 Start test Hyper Flash read performance!
##Hyper flash AHB read perf###t1: 445829; t2: 437546; diff: 8283; ns:
13805, datasize: 4096 byte; perf: 296MB/s; g_ms: 0

 SEMC SDRAM Performance test End.
```

Figure 2. Sdram performance test

3.2 FlexSPI performance

The i.MX RT supports the FlexSPI interface. It provides flexible configurations to interface the QSPI flash, OCTAL flash, hyper flash and hyper RAM. It supports AHB and IP command access. AHB access helps to achieve high performance, which is described as follows.

The FlexSPI supports the eXecute-In-the-Place (XIP) on that connected NOR flash. BEE module attached to FlexSPI decrypts images on the fly. The following enhanced features of FlexSPI help to improve the performance.

- System cache(32 k DCACHE and 32 K ICACHE)
- AHB buffer, 8*64 bit TX AHB buffer and 128*64bit RX AHB buffer

Table 4 shows the performance evaluation, taking the hyper/QSPI flash as an example.

Table 4. Hyper flash performance

Items	Performance (:MB/s)				Comments
	DCache enabled, prefetch buffer enabled	DCache disabled, prefetch buffer enabled	DCache disabled, prefetch buffer disabled	DCache enabled, prefetch buffer disabled	
Hyper flash	296	70	15	92	Continuously reading 4 KB bytes @166 MHz DDR mode
QSPI flash	58	58	8	35	Continuously reading 4 KB bytes @133 MHz SDR mode

The hyper flash contains higher performance than QSPI flash. It benefits from the bus bandwidth, working speed and working mode (DDR). The performance gets more improvement by enabling cache and prefetching buffer. The test results show that prefetching buffer improve performances more even it gets the similar performance on QSPI flash when prefetch buffer is enabled but no matter when the cache is enabled or disabled. The performance drops by about 77 % when the prefetch buffer and cache is disabled.

The prefetch provides the significant effects to flexSPI performance. It specifies different buffer size for different master. That means some master may have the dedicated prefetch buffer, which can optimize performance in some applications. For example, it can assign specified buffer size to eDMA. If it needs frequent data transfers from external QSPI flash to internal SRAM by eDMA, other master will not destroy prefetch buffer contents used for eDMA. Reduce the access latency if next access eDMA requests exactly hit buffer. In this way, it improves performance more.

The FlexSPI provides register as follows to set buffer size for different master.

- AHBRXBUF0CR0
- AHBRXBUF0CR1
- AHBRXBUF0CR2
- AHBRXBUF0CR3

User can modify these registers and assign dedicated buffer sizes to service in certain master, and master ID definition, as shown in Table 5.

Table 5. Master IDs

Module	Master ID
Core platform	000b
eDMA	001b
DCP	010b
All others	011b

As seen in Table 5, the independent master ID is assigned to core, eDMA and DCP. Other masters share one ID, say PXP, USB and so on.

Figure 3 shows the general prefetching scheme.

When the prefetch buffer is enabled, once receiving the request from the bus, it first checks whether the request matched the current AHB buffer address range. If yes, it directly returns the data. If not, it triggers to read new data to AHB buffer. After returning the required data to bus, it continues to prefetch the following flash data to AHB buffer until the buffer is full.

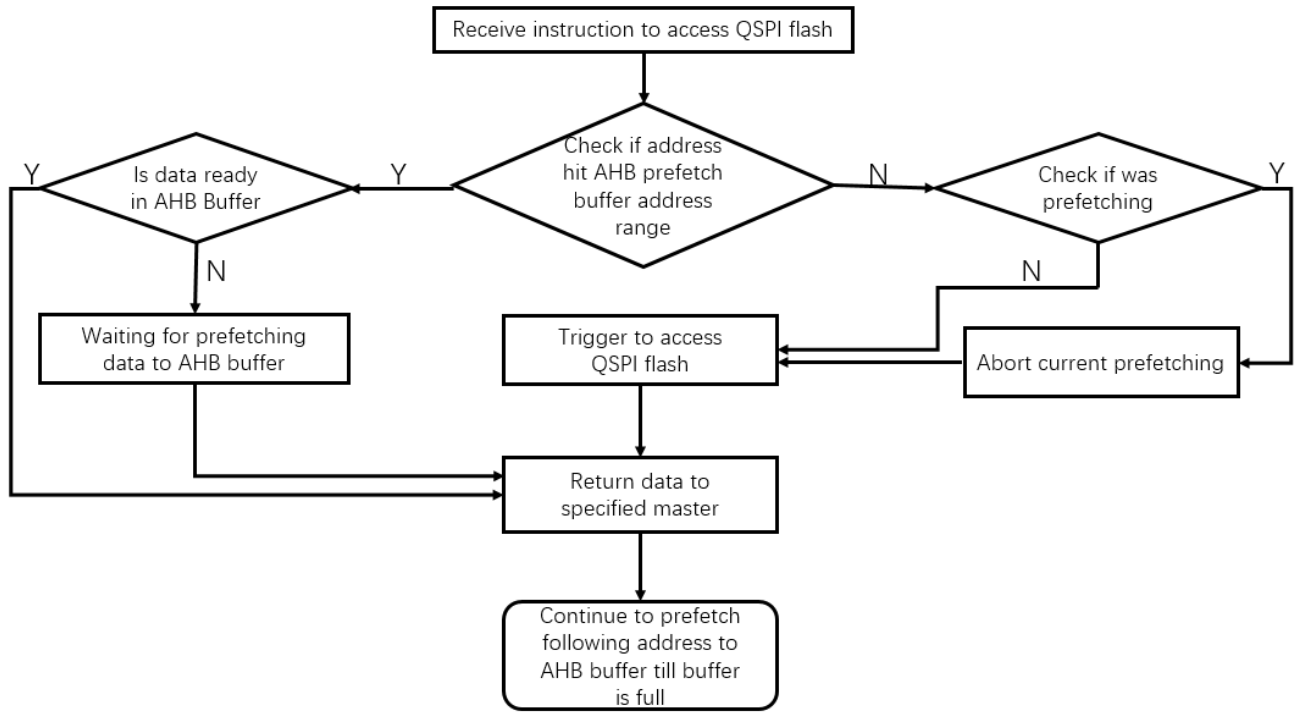

Figure 3. FlexSPI prefetching flow

3.3 Performance comparison on different memory

To evaluate the performance of executing code in different memory, it takes the test to run the same code in different memory and then calculates running time for comparison.

Taking one common audio encoding algorithm, OPUS, as an example, it encodes the same waveform file saved in SD card by software algorithm, and calculates the duration of encoding waveform to OPUS format. Table 6 shows the test results.

Table 6. Performance comparison

Test application	Code size (:Bytes)	Code location	Flash speed	Avergae speed (:μS)
Opus (encoder)	188 238	Hyper flash	166 MHz DDR	828364
	188 238	Hyper flash (encrypted image)	166 MHz DDR	847894
	188 238	QSPI flash	127 MHz SDR	1065541
	188 238	SDRAM	163 MHz SDR	826454
	188 238	ITCM	600 MHz	732964

Table 6 shows the performance comparison in different memory. It gets the best performance on running on ITCM, and the lowest performance on QSPI flash. QSPI flash is slower than TCM by about 45%, and than Hyper flash or SDRAM by 28%. It also has a slight drop on encrypted image, about 2.6%.

The performance depends on the application case. In some applications, it drops more on running code in some memory (QSPI flash), while the other application drops less. In some applications, it is possible to get the same performance. For example, in Table 7, the CoreMark nearly gets the same score in different memories.

Table 7. CoreMark score

Memory	Hyper flash	QSPI flash	SDRAM	ITCM
CoreMark Score(:/MHz)	5.059662	5.059659	5.059659	5.059659

Table 7 shows that the performance is determined by application and coding optimization. The coding optimization means to optimize the code to get the high cache hit rate and place the key code to ITCM for performance improvement. For details, refer to How to improve performance.

4 How to improve performance

The ways to improve the performance include:

- Keep high cache hit rate.
- Allocate the key code to internal RAM.

The facts impacting the cache hit rate include:

- Look-up table
- Branch

In most applications, the look-up table is used. When the look-up table is frequently accessed, it reads small size data, even one byte each time. Each read access possibly leads to cache miss hit. A new flash read operation is triggered and the perforamnce drops due to frequent triggering issues. The frequently branch also impacts the performance. it possibly leads to cache miss hit in this case. For these applications, the best way to improve the performance is to allocate the code to internal SRAM (ITCM/DTCM).

4.1 Allocating parts of code to specified memory

This section introduces how to allocate the codes to an internal RAM or other specified memory.

For how the SDK allocates the part of code to an internal SRAM, refer to the demo of `power_mode_switch`.

It defines the MACRO **QUICKACCESS_SECTION_CODE** in the `lpm.h` to allocate the function to a specified memory, as shown in Figure 4.

```
/*! @name Time sensitive region */
/* @{ */
#if defined(XIP_EXTERNAL_FLASH) && (XIP_EXTERNAL_FLASH == 1)
#if (defined(__ICCARM__))
#define QUICKACCESS_SECTION_CODE(func) __ramfunc func
#elif(defined(__ARMCC_VERSION))
#define QUICKACCESS_SECTION_CODE(func) __attribute__((section("RamFunction"))) func
#elif defined(__MCUXPRESSO)
#define QUICKACCESS_SECTION_CODE(func) __attribute__((section(".ramfunc.$SRAM_ITC"))) func
#elif(defined(__GNUC__))
#define QUICKACCESS_SECTION_CODE(func) __attribute__((section("RamFunction"))) func
#else
#error Toolchain not supported.
#endif /* defined(__ICCARM__) */
#else
#if (defined(__ICCARM__))
#define QUICKACCESS_SECTION_CODE(func) func
#elif(defined(__ARMCC_VERSION))
#define QUICKACCESS_SECTION_CODE(func) func
#elif(defined(__MCUXPRESSO))
#define QUICKACCESS_SECTION_CODE(func) func
#elif(defined(__GNUC__))
#define QUICKACCESS_SECTION_CODE(func) func
#else
#error Toolchain not supported.
#endif
#endif /* __FSL_SDK_DRIVER_QUICK_ACCESS_ENABLE */
```

Figure 4. Macro definition for function allocation

Accordingly, it defines one section, `RamFunction`, in the linker file, `evkmimxrt1060_power_mode_switch_ca.scf`, as shown in Figure 5.

```
  RW_m_data m_data_start m_data_size-Stack_Size-Heap_Size { ; RW data
    .ANY (+RW +ZI)
    * (NonCacheable.init)
    * (NonCacheable)
  }
  ARM_LIB_HEAP +0 EMPTY Heap_Size {    ; Heap region growing up
  }
  ARM_LIB_STACK m_data_start+m_data_size EMPTY -Stack_Size { ; Stack region growing down
  }
  RW_m_ram_text m_text2_start UNINIT m_text2_size { ; load address = execution address
    * (RamFunction)
  }
}
```

Figure 5. Linker file definition for "RamFunction"

To allocate one function to a specified RAM, refer to the code in Figure 6.

```
QUICKACCESS_SECTION_CODE(void LPM_SwitchBandgap(void));
QUICKACCESS_SECTION_CODE(void LPM_RestoreBandgap(void));
QUICKACCESS_SECTION_CODE(void LPM_SwitchToXtalOSC(void));
QUICKACCESS_SECTION_CODE(void LPM_SwitchToRcOSC(void));
QUICKACCESS_SECTION_CODE(void LPM_SwitchFlexspiClock(lpm_power_mode_t power_mode));
QUICKACCESS_SECTION_CODE(void LPM_RestoreFlexspiClock(void));
```

Figure 6. Examples of allocationg function to specified RAM

The `fsl_common.h` in `sdk` provides a similar MACRO definition for use.

5 How to identify key codes in one application

The internal RAM size is finite even RT has provided a big size for internals RAM (up to 2 M bytes in some part). It is still not enough to place all codes to internal RAM in some applications, and it is hard to know which function is critical to impact performance. A complex application may contain hundreds of functions, so the function analysis is difficult and more efforts are required. The below introduces one simple way to get the function profiling by IDE tools (IAR and MDK).

5.1 Function profiling

Many IDE tools support to get function profiling by Serial Wire Output (SWO) or Embedded Trace Macrocell (ETM) trace. The i.MX RT provides the full support by ETM and SWO TRACE. The below takes IAR and MDK as examples to show how to get the function profiling.

5.1.1 Hardware settings

The i.MX RT evaluation board reserves the 20-way J-Link connector. SWO trace can be implemented by JLINK and the board can be reworked by flying wire to connect ULink supported by ETM trace.

- SWO trace

  SWO supports a single pin output signal from the core. The i.MX RT series support SWD and JTAG debug. Table 8 describes J-link definition for JTAG and SWD.

Table 8. J-link connector definition

Pin number	Pinout for JTAG	Pinout for SWD	Pin number	Pinout for JTAG	Pinout for SWD
1	V_{tref}	V_{tref}	2	NC	NC
3	n_{TRST}	NC	4	GND	GND
5	TDI	NC	6	GND	GND
7	TMS	SWDIO	8	GND	GND
9	TCK	SWCLK	10	GND	GND
11	RTCK	NC	12	GND	GND
13	TDO	SWO	14	GND	GND
15	RESET	RESET	16	GND	GND
17	QBGRQ	NC	18	GND	GND
19	5 V - supply	5 V - supply	20	GND	GND

i.MX RT1050 and i.MX RT1020 remap the trace SWO signals to different pin, which is not multiplex with JTAG_TDO. The board needs to be reworked by flying wire TRACE_SWO to pin13 of J-link connector. For i.MX RT1060, it multiplex JTAG_TDO with TRACE_SWO to the pin connected to J-Link connector. SWO trace can work without any changes.

- ETM trace

  ETM is a hardware microcell. When connected to a core, ETM outputs instructions and data trace information on a trace port. The ETM provides core-driven trace through a trace port compliant to the ATB protocol.

  The ULINK supports ETM trace and appropriate connectors, as shown in Figure 7.

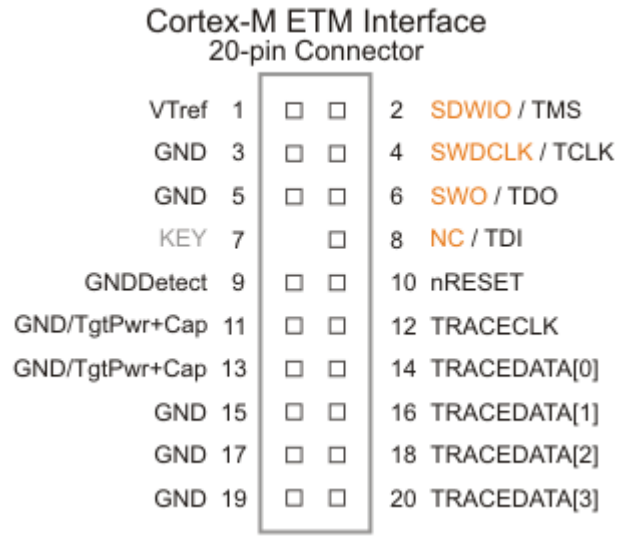

Figure 7. Cortex-M ETM interface

To implement ETM, fly a wire to connect the MCU and Cortex-M connector with the following signals.

  - TRACECLK
  - TRACEDATA[0]
  - TRACEDATA[1] (optional)
  - TRACEDATA[2] (optional)
  - TRACEDATA[3] (optional)

5.1.2 Software settings

To enable i.MX RT trace function, it is necessary to enable the TRACE clock and configure the appropriate pinmux.

- An example for configuring i.MX RT1060 to enable the SWO function:
  - Trace clock configurations

```
CLOCK_EnableClock(kCLOCK_Trace)
```

```
CLOCK_SetDiv(kCLOCK_TraceDiv, 2)
```

```
CLOCK_SetMux(kCLOCK_TraceMux, 2)
```

— PAD configurations

```
IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B0_10_ARM_TRACE_SWO, 0U)
```

- An example for configuring the i.MX RT1060 to enable the ETM function:
  — Trace clock configuration

```
CLOCK_EnableClock(kCLOCK_Trace);
CLOCK_SetDiv(kCLOCK_TraceDiv, 3);
CLOCK_SetMux(kCLOCK_TraceMux, 0);
```

— Pad configurations

```
IOMUXC_SetPinMux(IOMUXC_GPIO_B0_12_ARM_TRACE_CLK,  0U);
IOMUXC_SetPinMux(IOMUXC_GPIO_B0_04_ARM_TRACE0,  0U);
IOMUXC_SetPinMux(IOMUXC_GPIO_B0_05_ARM_TRACE1,  0U);
IOMUXC_SetPinMux(IOMUXC_GPIO_B0_06_ARM_TRACE2,  0U);
IOMUXC_SetPinMux(IOMUXC_GPIO_B0_07_ARM_TRACE3,  0U);
```

5.1.3 IDE settings

The below takes IAR as an example to introduce how to set tools for function profiling.

1. Connect the J-Link to the target board, MIMXRT1060-EVK, and click the **J-Link** to configure the SWO, as shown in Figure 8.

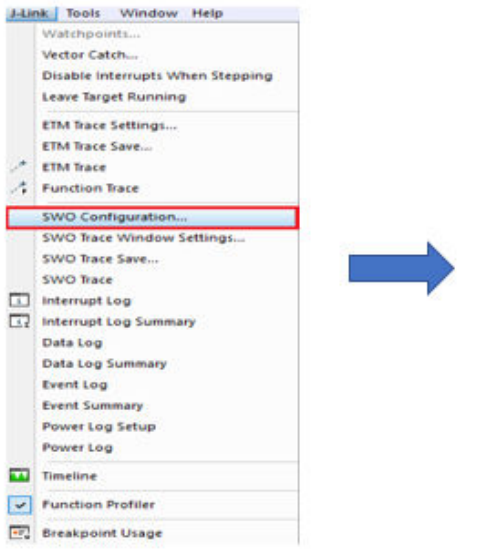

Figure 8. SWO configurations

2. Click **Function Profiler** to open the **Function Profiler** window. Figure 9 shows how to open the **J-link** window and **Function Profile** window.

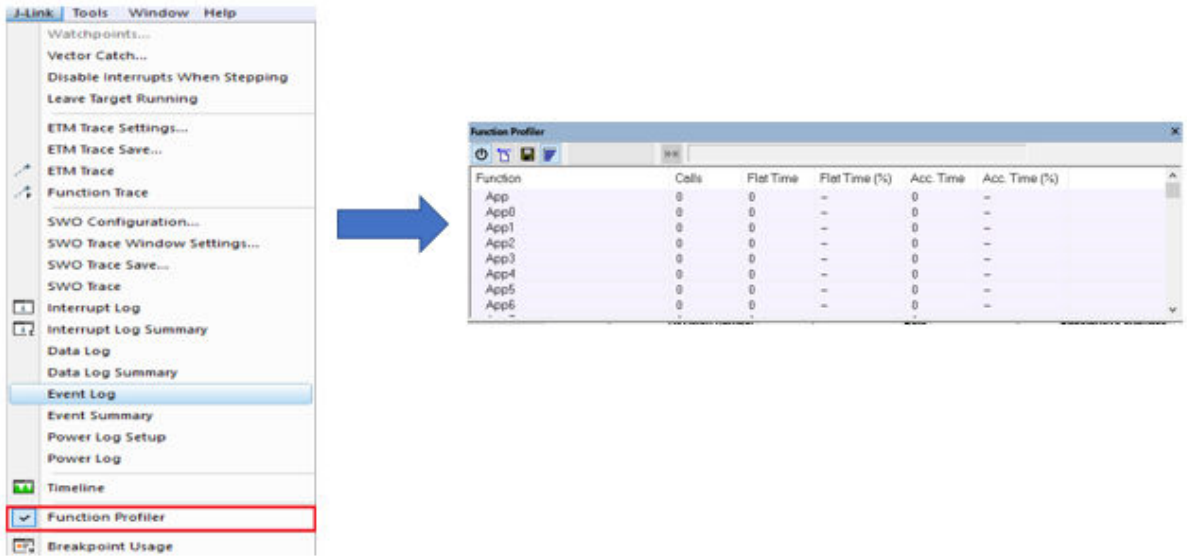

Figure 9. Function Profile window

3. Right click **Function** and select **source:Sampling**, as shown in Figure 10.

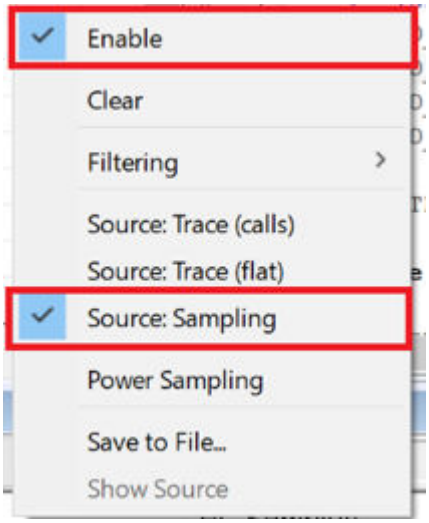

Figure 10. Function Profile settings

Run codes for some time and stop to check the functon profiling, as shown in Figure 11.

Function	PC Samples	PC Samples (%)	Address
App7	127637	25.05	0x6017c800–0x6...
App8	104876	20.58	0x6018c800–0x6...
App6	80883	15.88	0x60164400–0x6...
App5	76105	14.94	0x6015b000–0x6...
App	38493	7.56	0x60003e42–0x6...
App2	21685	4.26	0x60003d48–0x6...
App4	14601	2.87	0x60003df0–0x6...
App3	14516	2.85	0x60003d9c–0x6...
App1	14459	2.84	0x60003cf0–0x6...
App0	14458	2.84	0x60003c98–0x6...

Figure 11. Function Profile result

As seen in Figure 11, App7 account for high loading rate in this application and then optimize this code to ITCM for performance improvement.

After taking optimization of allocating App7 to ITCM, the performance improves by 18.5 %. Figure 12 shows the test result.

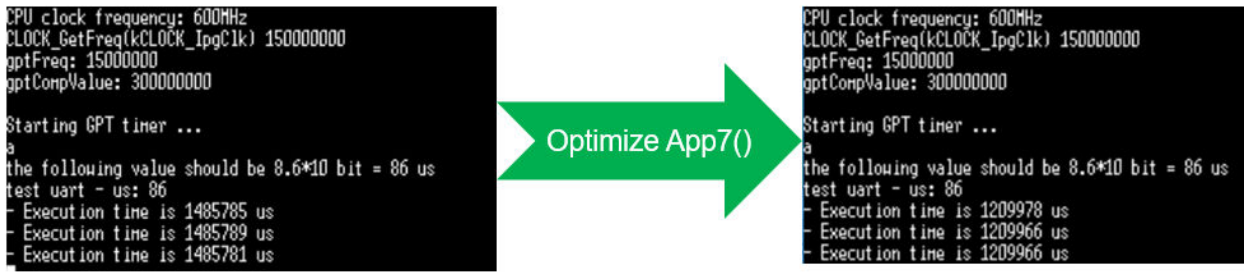

Figure 12. Test result of code optimization

The MDK IDE doesn't support the function profiling by SWO trace. The board needs to be reworked to support the ETM trace. For details, refer to Hardware settings.

The below shows how to perform the ULINK Pro settings.

1. Connect the ULINK Pro to the target board, the reworked MIMXRT1060-EVK. Select the correct debugger and click **setting**, as shown in Figure 13.

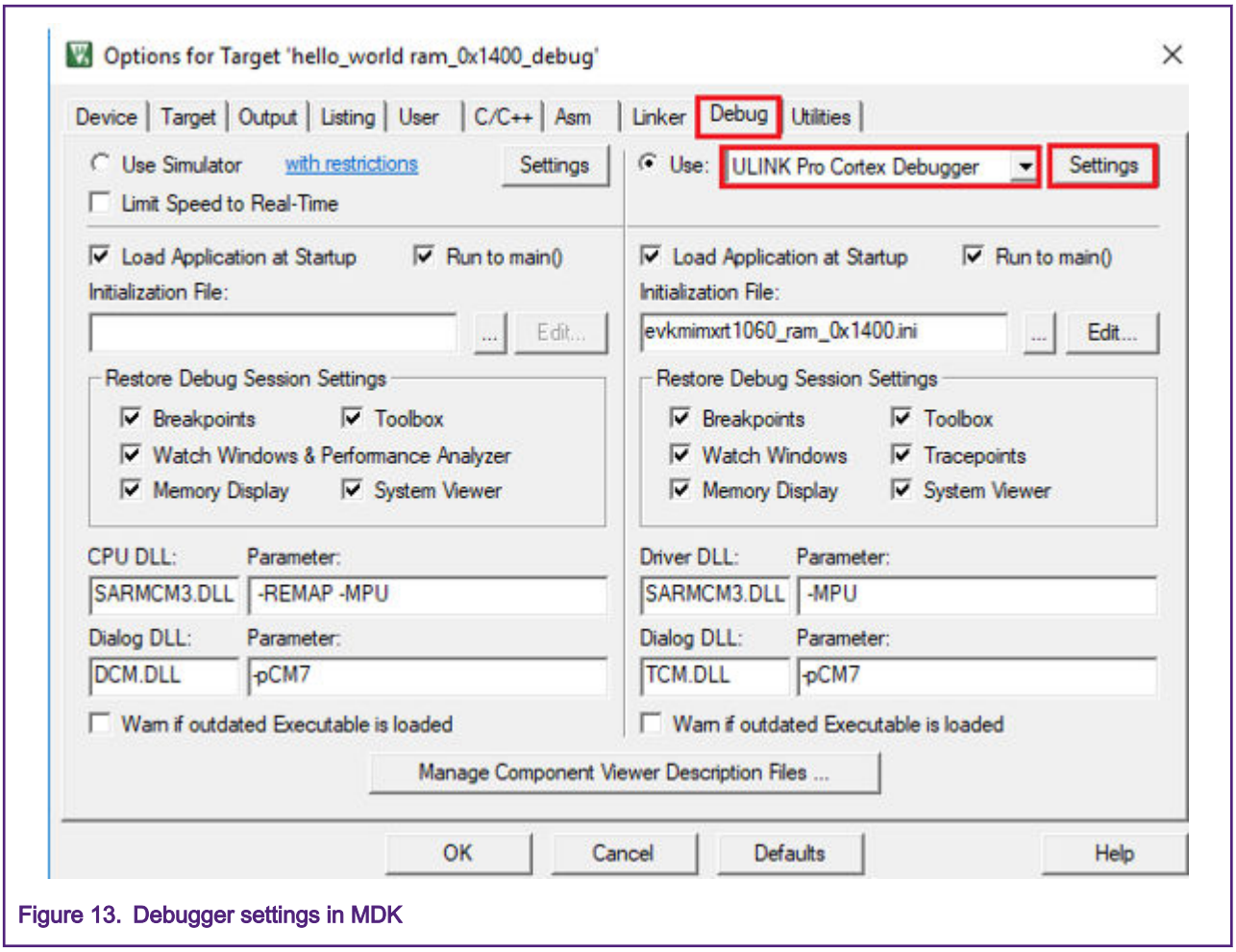

Figure 13. Debugger settings in MDK

2. Click **Trace** to set the ETM, as shown in Figure 14.

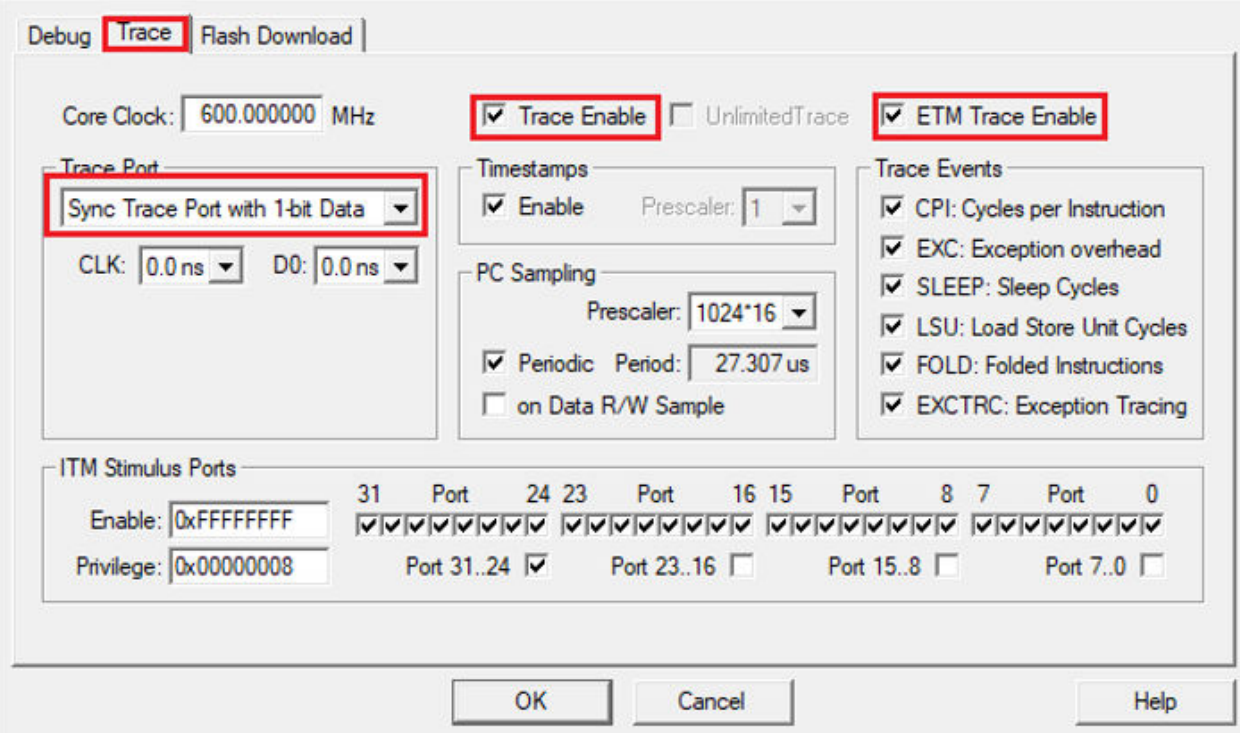

Figure 14. Trace settings in MDK

To select the Trace Port with 1-bit data or other options is determined by the hardware connection.

3. Figure 15 shows the **Performance Analyze** window.

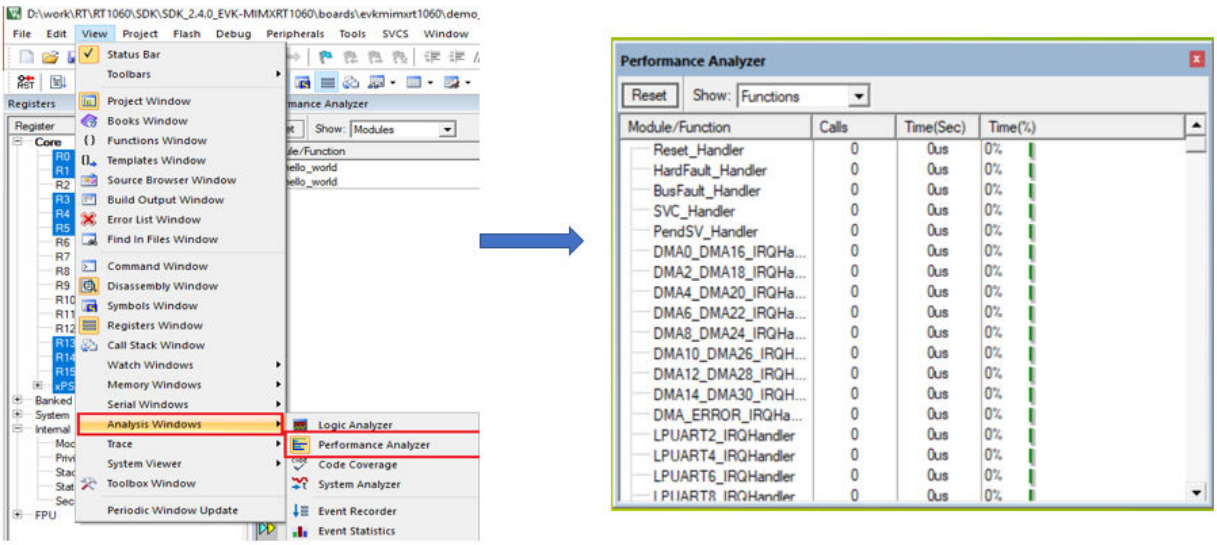

Figure 15. Performance Analyzer window

4. Run codes and stop to watch the performance in the **Performance analyzer** window.

6 Conclusion

This document introduces the system architecture and memory performance supported by i.MX RT10xx series, as well as how to improve performance and get the function profiling by IDE tools. The document helps customer to optimize code and get good performance when using the i.MX RT series.

7 Revision history

Table 9. Revision history

Revision number	Date	Substantive changes
0	05/2019	Initial release
1	02/2020	Update Memory performance test and SDRAM performance

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: Febuary 2020

Document identifier: AN12437

arm