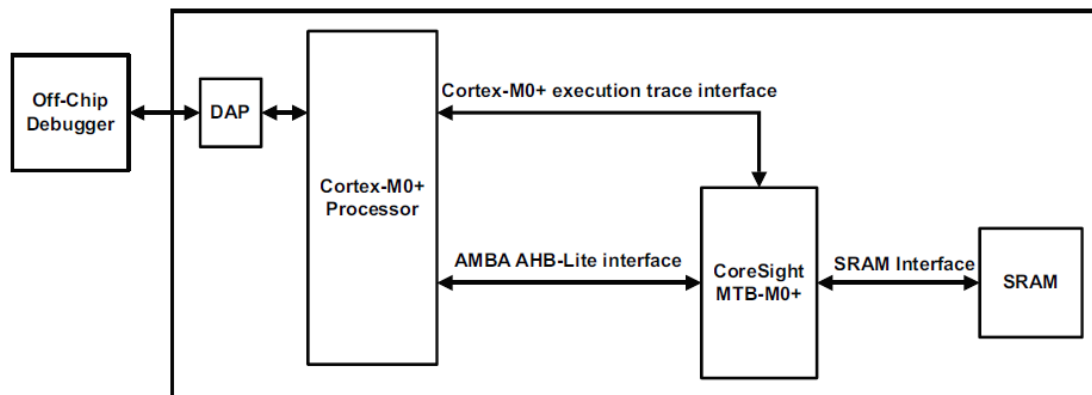# Micro trace buffer (MTB)

Overview

The micro trace buffer module provides a single execution trace capability to the Cortex-M0+ processor core.

The following figure shows the main interfaces on the MTB and how they are connected in a simple Cortex-M0+ based system.

When enabled, the MTB records changes in program flow, reported by the Cortex-M0+ processor over the execution trace interface. This information is stored as trace packets in the SRAM. An off-chip debugger can extract the trace information using the DAP to read the trace information from the SRAM, over the AHB-Lite interface. The debugger can then reconstruct the program flow from this information.

The processor accesses the SRAM using the AHB-Lite interface. The MTB simultaneously stores trace information into the SRAM, and gives the processor access to the SRAM. The MTB ensures that trace write accesses have priority over processor accesses.

Note: the SRAM here mentioned is the internal SRAM around address 0x2000_0000, so MTB will share the same address range as processor core. It is debugger responsibility to make sure the MTB trace buffer range will not be used by application code to store global variables, heap or stack.

Trace packet format

The execution trace packet consists of a pair of 32-bit words that the MTB generates when it detects the processor PC value changes non-sequentially. A non-sequential PC change can occur during branch instructions or during exception entry.

Following figure shows the MTB execution trace packet format when it's stored in internal SRAM. For information on how to interpret the trace, please refer to CoreSight™ MTB-M0+ TRM or KL27 Reference manual.
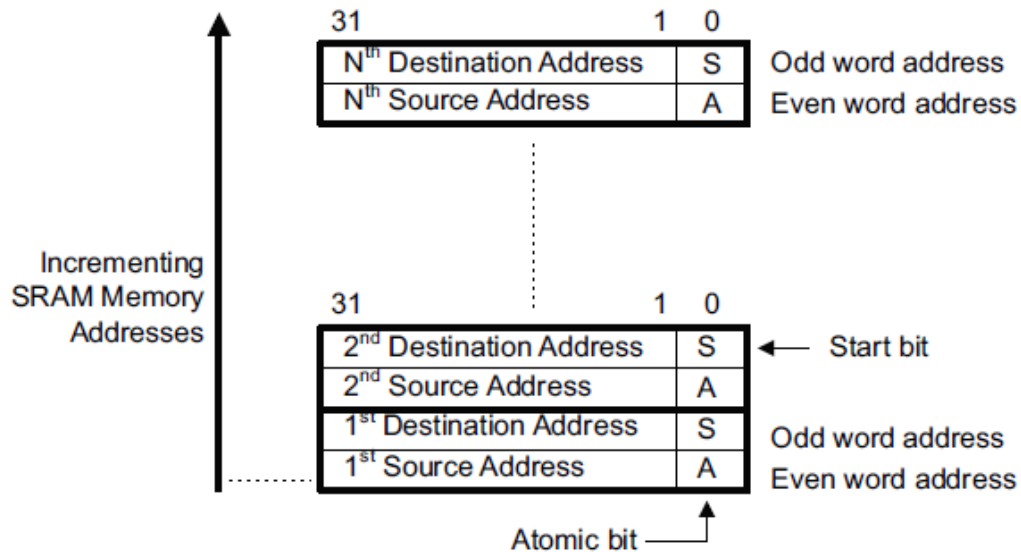
Register map for MTB

There are mainly four registers for MTB, though KL27 implements more. The four registers are as follows:

- POSITION register
- MASTER register
- FLOW register
- BASE register

POSITION register has information on where next trace packet will be written in SRAM, this address can be calculated with the following code on KL27.

if ((MTB_POSITION >> 13) == 0x3)

systemAddress = (0x1FFF << 16) + (0x1 << 15) + (MTB_POSITION & 0x7FF8);

else

systemAddress = (0x2000 << 16) + (0x0 << 15) + (MTB_POSITION & 0x7FF8);

BASE register points to the starting address for internal SRAM, for KL27, this address is at 0x1FFF_F000.

Using MTB with IAR

In order to discuss how to use MTB within IAR IDE, demos inside KSDK will be used to show how to enable MTB trace, what the trace packet look like in SRAM and what is changed with the MTB registers.
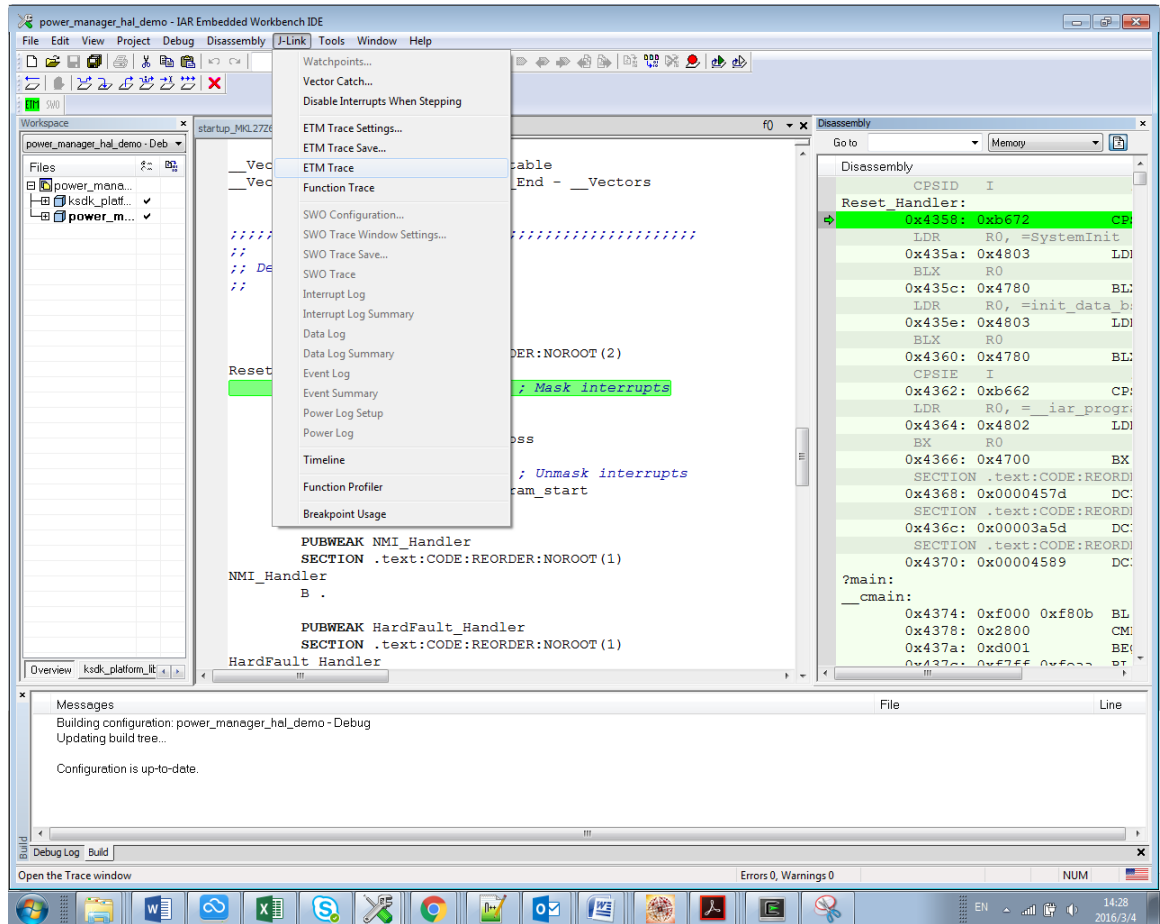
Major steps

1. Open "power_manager_hal_demo" demo under KSDK1.3 for KL27 Freedom board, KSDK_1.3.0\examples\frdmkl27z\demo_apps\power_manager_hal_demo\iar, make sure in debugger setting, J-Link and ETM/ETB is selected. Also deselect "run to main".
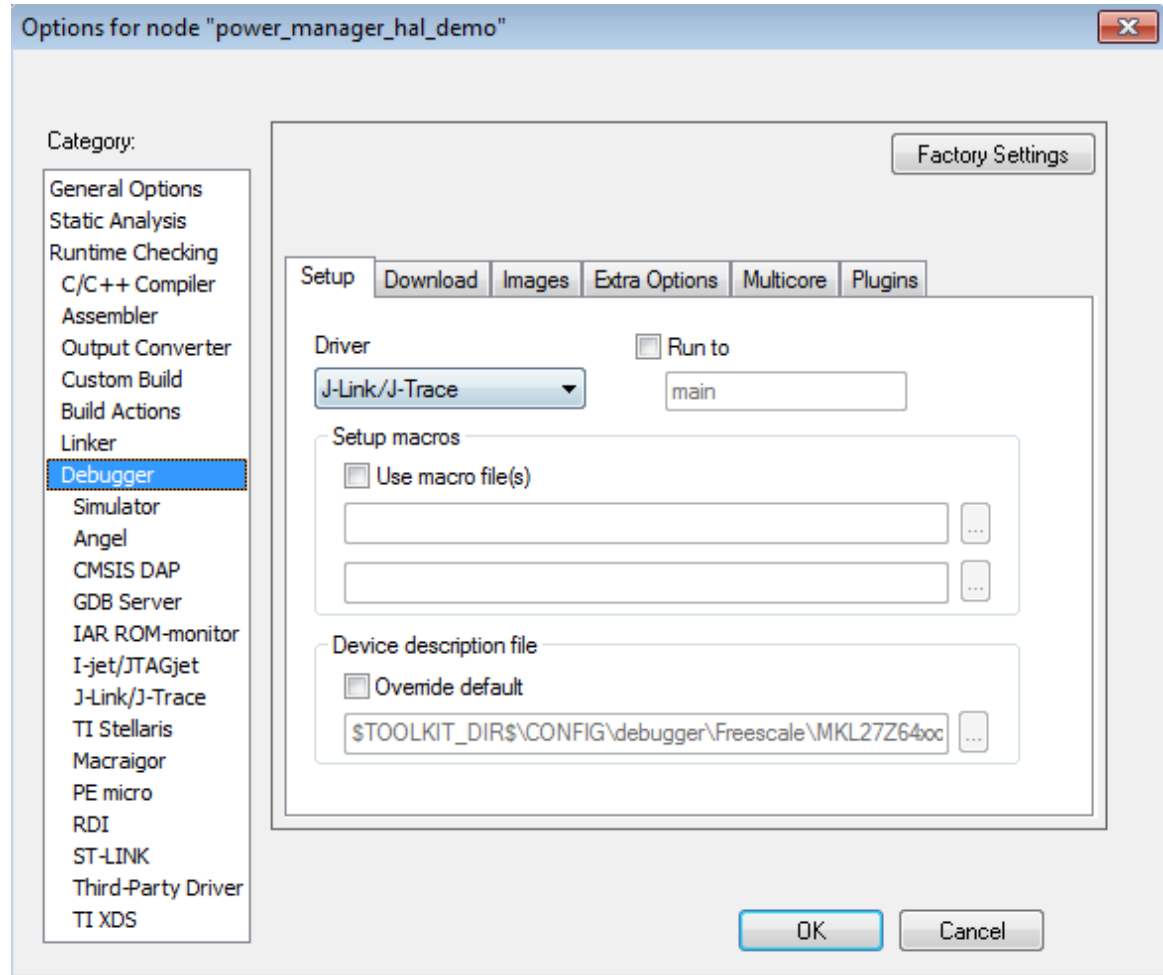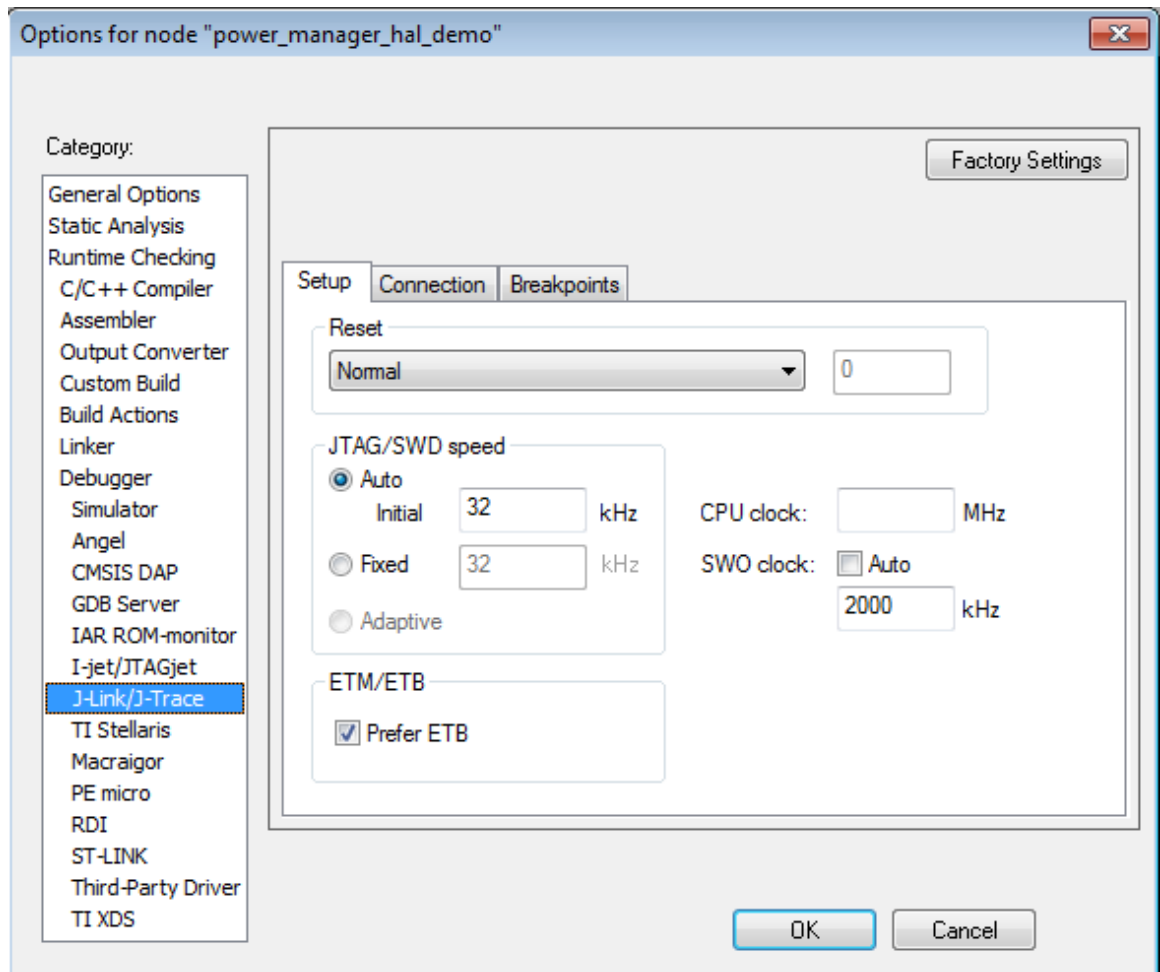
2. Compile the demo and download the image to KL27 flash, then select ETM trace under J-Link menu.

Then under the following window, you can click the leftmost button to enable trace.

3. Open debug log window and register window, place mouse pointer in disassembly window and single step, so this will single step in instruction level, and you will see from debug log window that the trace buffer is reserved at 0x2000_0000.
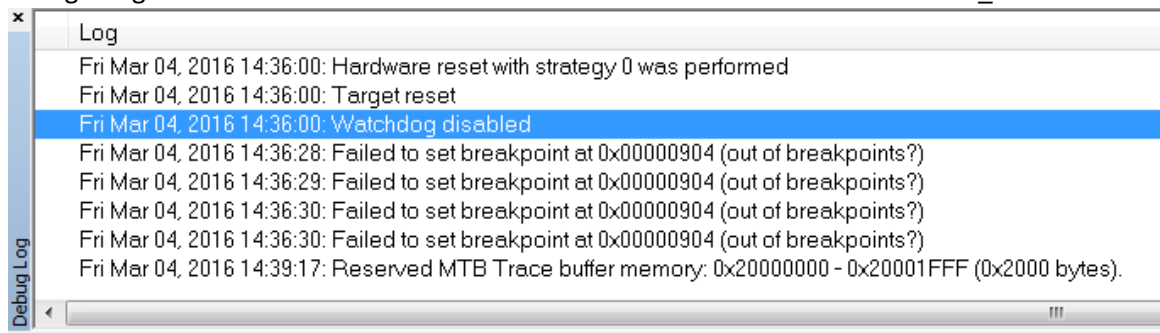
This can be verified with MTB_POSITION register value.

Register   ✕

MTB   ▾   <find register>   ▾

⊞MTB_POSITION	= 0x00000000
⊞MTB_MASTER	= 0x00000080
⊞MTB_FLOW	= 0x00000000
⊞MTB_BASE	= 0x1FFFF000
⊞MTB_MODECTRL	= 0x00000000
⊞MTB_TAGSET	= 0x00000000
⊞MTB_TAGCLEAR	= 0x00000000
⊞MTB_LOCKACCESS	= 0x00000000
⊞MTB_LOCKSTAT	= 0x00000000
⊞MTB_AUTHSTAT	= 0x0000000F
⊞MTB_DEVICEARCH	= 0x47700A31
⊞MTB_DEVICECFG	= 0x00000000
⊞MTB_DEVICETYPID	= 0x00000031
⊞MTB_PERIPHID4	= 0x00000004
⊞MTB_PERIPHID5	= 0x00000000
⊞MTB_PERIPHID6	= 0x00000000
⊞MTB_PERIPHID7	= 0x00000000
⊞MTB_PERIPHID0	= 0x00000032
⊞MTB_PERIPHID1	= 0x000000B9
⊞MTB_PERIPHID2	= 0x0000001B
⊞MTB_PERIPHID3	= 0x00000000
⊞MTB_COMPID0	= 0x0000000D
⊞MTB_COMPID1	= 0x00000090
⊞MTB_COMPID2	= 0x00000005
⊞MTB_COMPID3	= 0x000000B1

4. Set a breakpoint at main() function entry, so it will stop tracing when code hit this breakpoint. Open a memory window and point to 0x2000_0000 so we can see what will be written into the trace buffer when we start running code from reset handler to main() function.

5. In the ETM trace window, right click mouse button to show menu

And in the pop-up dialog, click "save" button to save the trace log file.

Now let's get back to entry point of the code at Reset Handler and see how to interpret the trace logs written in SRAM.

When you single-step code from Reset Handler to the instruction at address 0x435c, there is the branch instruction "BLX R0".
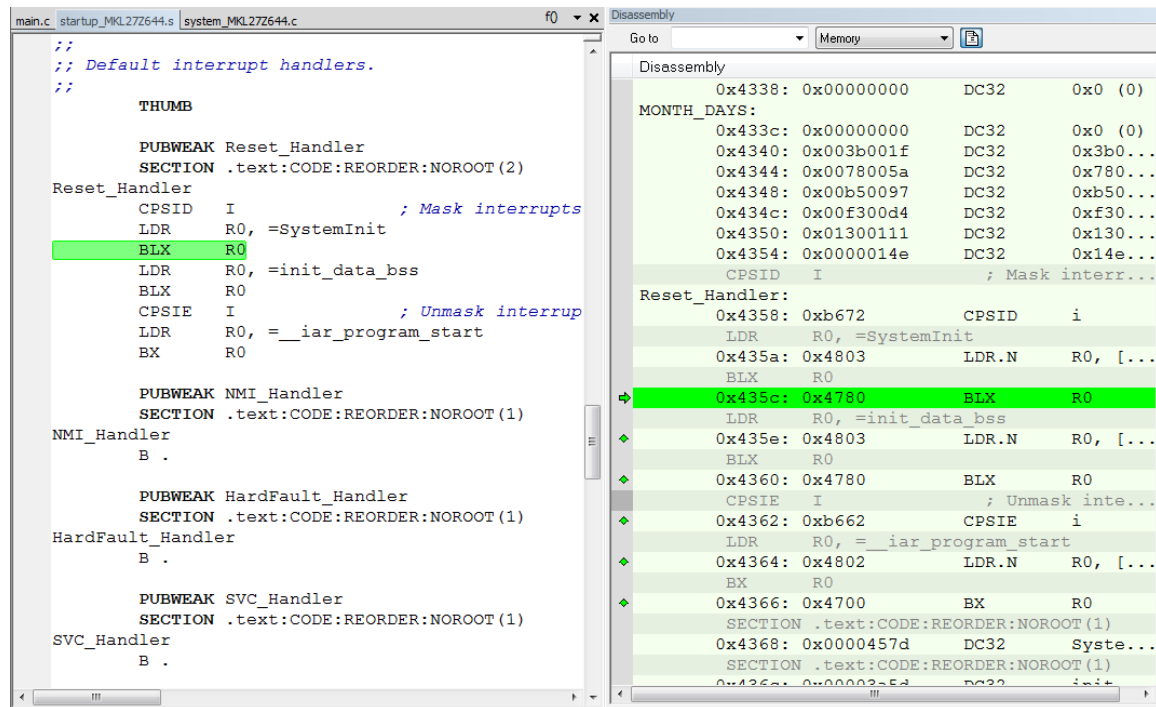
After single-stepping this instruction, you will see one trace logged and this is written at SRAM address 0x2000_0000.
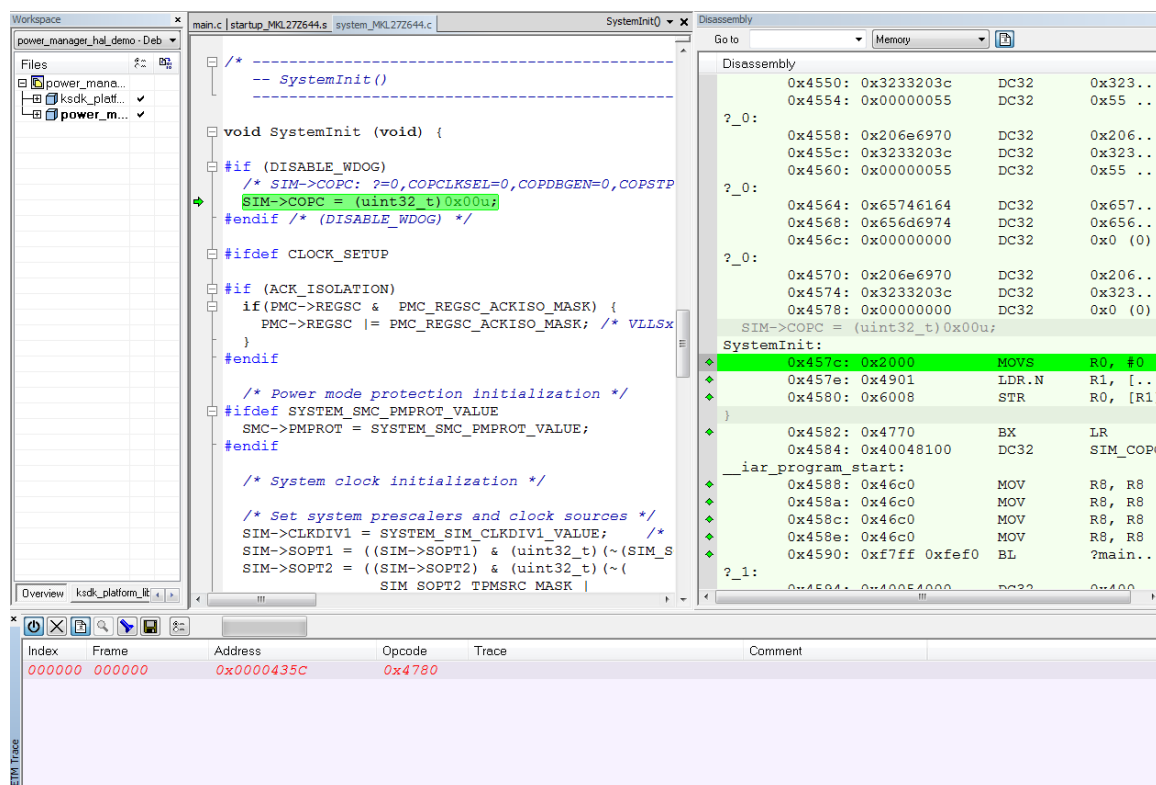
Following is the first few trace logs captured in the log file we saved.

ARM ETM Trace log Fri Mar 04 15:06:39 2016

Index	Frame	Address	Opcode	Mnemonic	Operands	Comment
000000	000000	0x0000435C	0x4780	BLX	R0	
000001	000001	0x0000457C	0x2000	MOVS	R0, #0	
000002	000002	0x0000457E	0x4901	LDR.N	R1, [PC, #0x4]	; SIM_COPC ;
000003	000003	0x00004580	0x6008	STR	R0, [R1]	
000004	000004	0x00004582	0x4770	BX	LR	
000005	000005	0x0000435E	0x4803	LDR.N	R0, [PC, #0xc]	; init_data_bss ;
000006	000006	0x00004360	0x4780	BLX	R0	
000007	000007	0x00003A5C	0xb500	PUSH	{LR}	
000008	000008	0x00003A5E	0x480d	LDR.N	R0, [PC, #0x34]	; 0x0 (0) ;
000009	000009	0x00003A60	0x490d	LDR.N	R1, [PC, #0x34]	; 0x0 (0) ;
000010	000010	0x00003A62	0x4288	CMP	R0, R1	
000011	000011	0x00003A64	0xd012	BEQ.N	0x3a8c	
000012	000012	0x00003A8C	0x4802	LDR.N	R0, [PC, #0x8]	; 0x0 (0) ;
000013	000013	0x00003A8E	0x4904	LDR.N	R1, [PC, #0x10]	; VTOR ;
000014	000014	0x00003A90	0x6008	STR	R0, [R1]	
000015	000015	0x00003A92	0xbd00	POP	{PC}	
000016	000016	0x00004362	0xb662	CPSIE	i	
000017	000017	0x00004364	0x4802	LDR.N	R0, [PC, #0x8]	; __iar_program_start ;
000018	000018	0x00004366	0x4700	BX	R0	

As the trace buffer is inside internal SRAM and global variables, stack and heap used by application code is also inside internal SRAM. Then how does debugger tool make sure trace log will not overwritten variables used by application code?

When looking at the memory map file generated for this sample project. You can see end of BSS section is at 0x1fff_f078, while the initial trace buffer allocated when we start debugging the code is at 0x2000_0000, so debugger leaves enough space to avoid trace data to overwrite global variables.

If you look at other sample code in KSDK, you may find the initial trace buffer allocated will be different based on how much space used by global variables.

```
"P3":                                0x38
  RW                    0x1ffff000    0x38  <Block>
    RW-1                0x1ffff000    0x38  <Init block>
      .data     inited  0x1ffff000    0x18  gpio_pins.o [1]
      .data     inited  0x1ffff018     0xc  main.o [1]
      .data     inited  0x1ffff024     0xc  main.o [1]
      .data     inited  0x1ffff030     0x4  main.o [1]
      .data     inited  0x1ffff034     0x4  system_MKL27Z644.o [1]
                      - 0x1ffff038    0x38

"P4":                                0x40
  ZI                    0x1ffff038    0x40  <Block>
    .bss      zero      0x1ffff038    0x10  fsl_debug_console.o [1]
    .bss      zero      0x1ffff048    0x10  fsl_clock_manager.o [4]
    .bss      zero      0x1ffff058     0x4  fsl_interrupt_manager.o [4]
    .bss      zero      0x1ffff05c    0x10  fsl_power_manager.o [4]
    .bss      zero      0x1ffff06c     0x4  fsl_mcglite_hal.o [4]
    .bss      zero      0x1ffff070     0x4  fsl_mcglite_hal.o [4]
    .bss      zero      0x1ffff074     0x1  main.o [1]
                      - 0x1ffff078    0x40

"P6":                               0x300
  CSTACK                0x20002d00   0x300  <Block>
    CSTACK    uninit    0x20002d00   0x300  <Block tail>
                      - 0x20003000   0x300
```

References

CoreSight™ MTB-M0+ TRM

KL27 Sub-Family Reference Manual