Using Event Recorder for debugging a network performance issue

MDK Tutorial

AN320, Summer 2019, V 1.0

feedback@keil.com

The latest version of this document is here: www.keil.com/appnotes/docs/apnt_320.asp

Abstract

In complex embedded applications, it is often very difficult to find a reason for reduced performance or incorrect program operation.

This application note shows how Event Recorder and Keil MDK can be used for analyzing the program execution and locating the root cause for poor performance in a real network example.

Prerequisites

Event Recorder can be used on any Arm Cortex-M based device and with any MDK Edition or debug adapter. The concepts described in the application notes are universal.

The particular example used for analysis is based on MDK-Middleware that is available with MDK-Plus and MDK-Professional editions.

Note: there is an evaluation version available for MDK-Professional.

Following software packs are used:

- ARM.CMSIS.5.0.1.pack (or higher) for CMSIS and CMSIS-RTOS
- Keil.ARM_Compiler.1.6.1.pack (or higher) for Event Recorder
- Keil.MDK-Middleware.7.8.0.pack (or higher) for MDK Network library

Contents

Introduction

Event Recorder is a software component that provides an API for event annotation in application code. Events get triggered when the application is running, providing developers with valuable insights such as timing information and event-specific arguments.

Event Recorder is available as part of Keil Arm Compiler Extensions pack and can be used with any Arm Cortex-M based device and any debug adapter.

Keil MDK natively supports Event Recorder and allows users to visually observe the recorded events in real-time. A logging functionality for later analysis is also available.

MDK-Middleware and CMSIS-RTOS components are already annotated for Event Recorder support and allow developers to analyze the internal execution flow. This is important, as the MDK-Middleware is delivered as a library that does not disclose its contents.

This application is based on a real-life performance issue observed in a network example for an STM32H7 evaluation board. As the debugging concepts and the usage of Event Recorder are universal, the hardware-specific details are abstracted.

Problem Description

Symptoms

When testing examples of the MDK-Middleware, low network performance was observed. Loading a web page was slow and refreshing the page was significantly faster. Everything else operated correctly.

When pinging the IP address of the evaluation board, unexpected delays of 25 ms are observed:

```
C:\>ping 192.168.3.26

Pinging 192.168.3.26 with 32 bytes of data:
Reply from 192.168.3.26: bytes=32 time=25ms TTL=128
Reply from 192.168.3.26: bytes=32 time=25ms TTL=128
Reply from 192.168.3.26: bytes=32 time=25ms TTL=128
Reply from 192.168.3.26: bytes=32 time=25ms TTL=128

Ping statistics for 192.168.3.26:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 25ms, Maximum = 25ms, Average = 25ms

C:\>
```

Figure 1 Slow response on *ping* command to the evaluation board

Analyzing the issue

Following software components are used in the system:

- Network library MDK-Middleware
- Operating system CMSIS-RTOS RTX5
- CMSIS-Driver Ethernet for the target board

To locate the reason for the problem, the processing of the *ping* command was analyzed individually in each component using Event Recorder.

Debugging the network library

MDK-Middleware contains a network CORE library with debug support that is annotated with multiple events for Event Recorder and provides visibility to the operation of the network stack. It is added to the project using µVision RTE as shown on Figure 2:

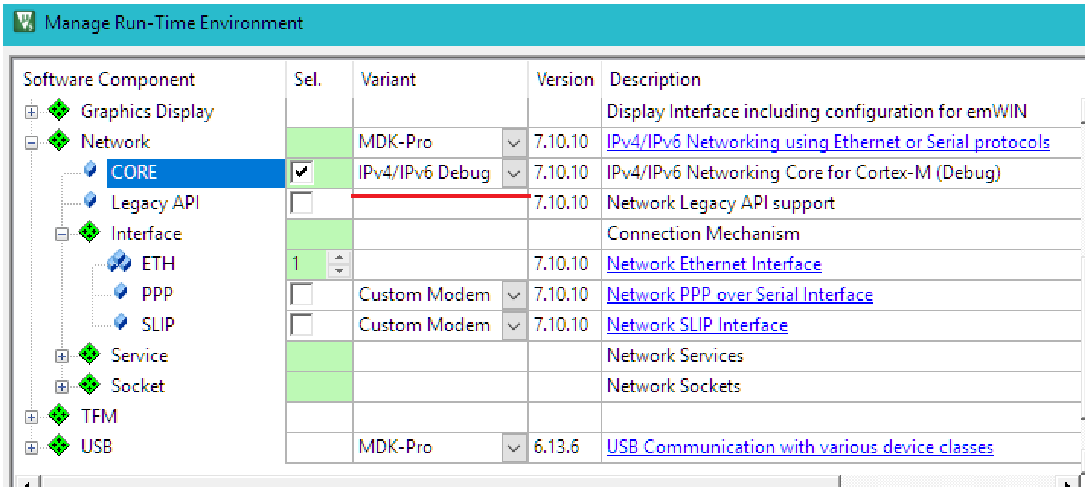


Figure 2 Selecting network library with debug support in µVision Manage Run-Time Environment window

In the *Net_Debug.c* file, full debug is enabled for *ETH Interface* and *ICMP Control* as shown on Figure 3.

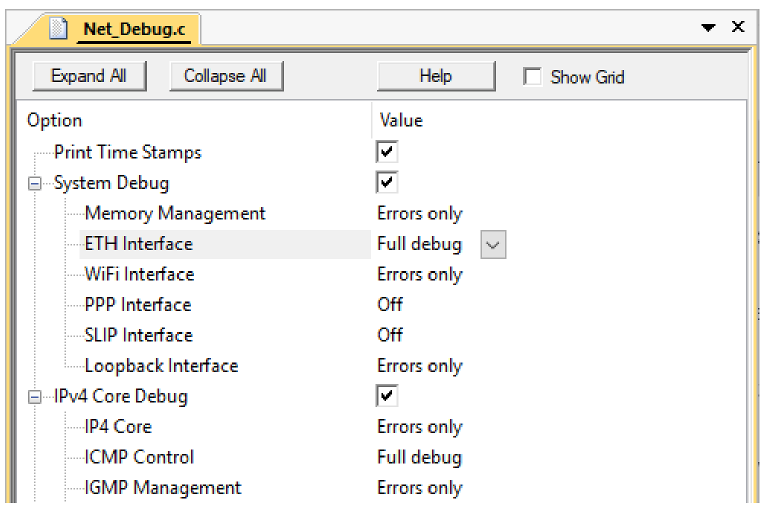
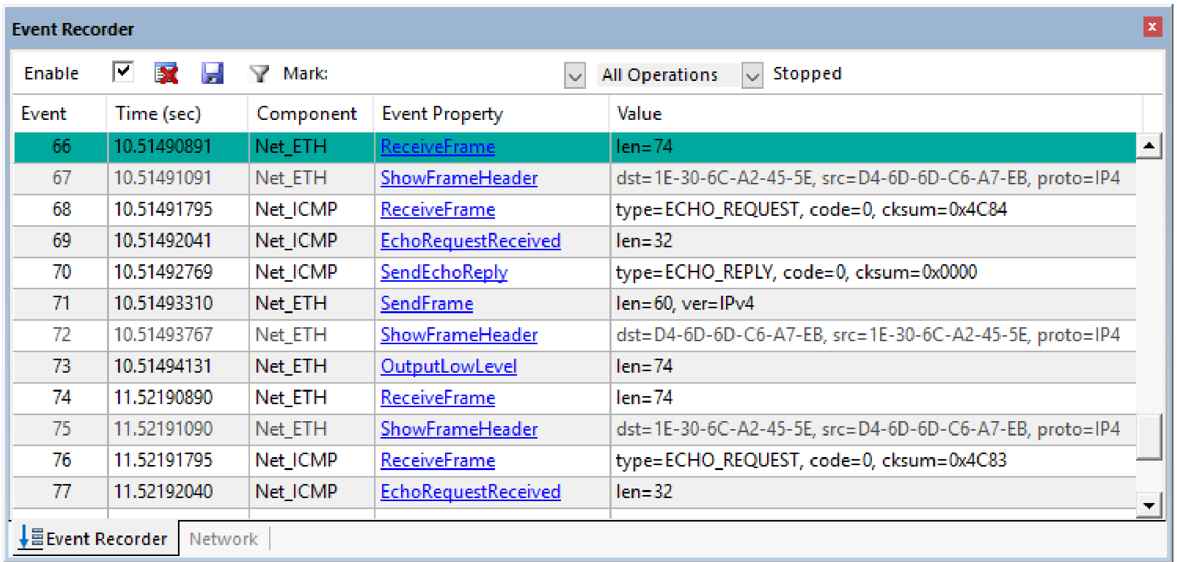


Figure 3 Enabling full debug for *ETH_Interface* and *ICMP Control*

The program needs rebuilding. When the debug operation is started, the network stack events get captured and are listed in the [µVision Event Recorder window](). When *ping* command is issued to the board from the PC following events were observed as shown on Figure 4:



Event	Time (sec)	Component	Event Property	Value
66	10.51490891	Net_ETH	ReceiveFrame	len=74
67	10.51491091	Net_ETH	ShowFrameHeader	dst=1E-30-6C-A2-45-5E, src=D4-6D-6D-C6-A7-EB, proto=IP4
68	10.51491795	Net_ICMP	ReceiveFrame	type=ECHO_REQUEST, code=0, cksum=0x4C84
69	10.51492041	Net_ICMP	EchoRequestReceived	len=32
70	10.51492769	Net_ICMP	SendEchoReply	type=ECHO_REPLY, code=0, cksum=0x0000
71	10.51493310	Net_ETH	SendFrame	len=60, ver=IPv4
72	10.51493767	Net_ETH	ShowFrameHeader	dst=D4-6D-6D-C6-A7-EB, src=1E-30-6C-A2-45-5E, proto=IP4
73	10.51494131	Net_ETH	OutputLowLevel	len=74
74	11.52190890	Net_ETH	ReceiveFrame	len=74
75	11.52191090	Net_ETH	ShowFrameHeader	dst=1E-30-6C-A2-45-5E, src=D4-6D-6D-C6-A7-EB, proto=IP4
76	11.52191795	Net_ICMP	ReceiveFrame	type=ECHO_REQUEST, code=0, cksum=0x4C83
77	11.52192040	Net_ICMP	EchoRequestReceived	len=32

Figure 4 Network events captured in Event Recorder window when receiving *ping* command

Table 1 provides description for the captured events:

Event	Description
66	Ethernet interface receives a frame.
67	Network library prints the details of the ethernet header of this frame.
68	ICMP process receives this frame.
69	ICMP recognizes an Echo Request while processing the frame.
70	ICMP generates an Echo Reply.
71	Ethernet interface sends a response.
72	Network library prints the details of the ethernet header of the response.
73	Ethernet interface passes the constructed ethernet frame to the network driver.

Table 1 Description of recorded network events

The total time difference can be calculated from the Event Recorder time stamps

- Event 66 time is *10.51490891* seconds.
- Event 73 time is *10.51494131* seconds.

The time difference between the two is *32.40* microseconds. Hence, it can be concluded that the Network library reacts to the input frame in just 32 microseconds and is not responsible for the delay of 25 milliseconds.

Debugging the RTX5 thread switches

In Keil RTX5, thread switches can potentially introduce delays. To analyze this, the source variant of Keil RTX5 is used as shown on Figure 5:

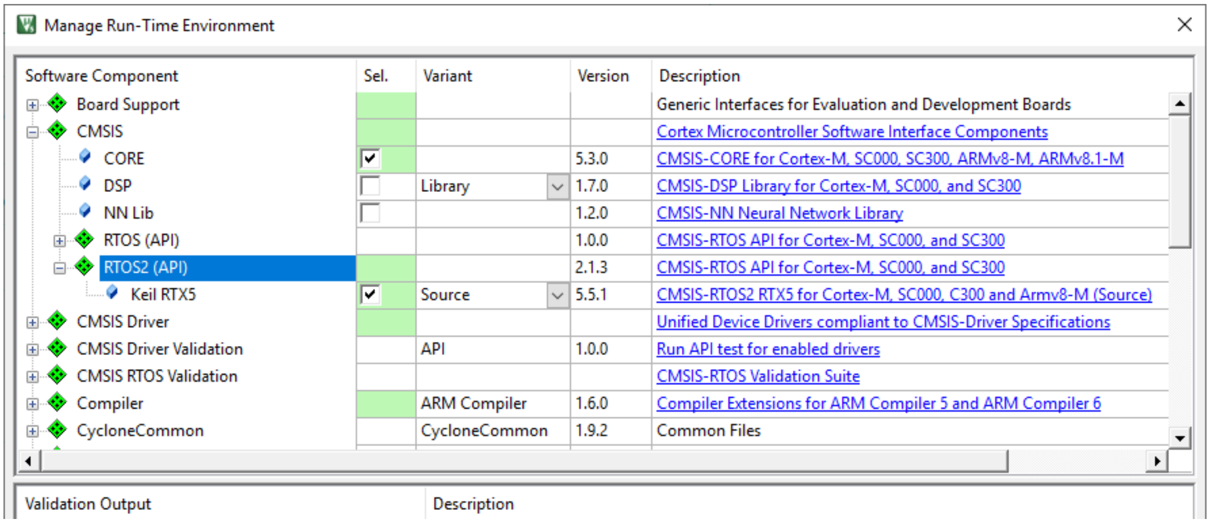


Figure 5 Selecting Keil RTX5 source code variant

In the *RTX_Config.h* configuration file, the *Global Initialization*, *Thread* and *Thread Flags* events are enabled under *Event Recorder Configuration* section as shown in Figure 6. The Network library uses Thread Flags to synchronize network threads.

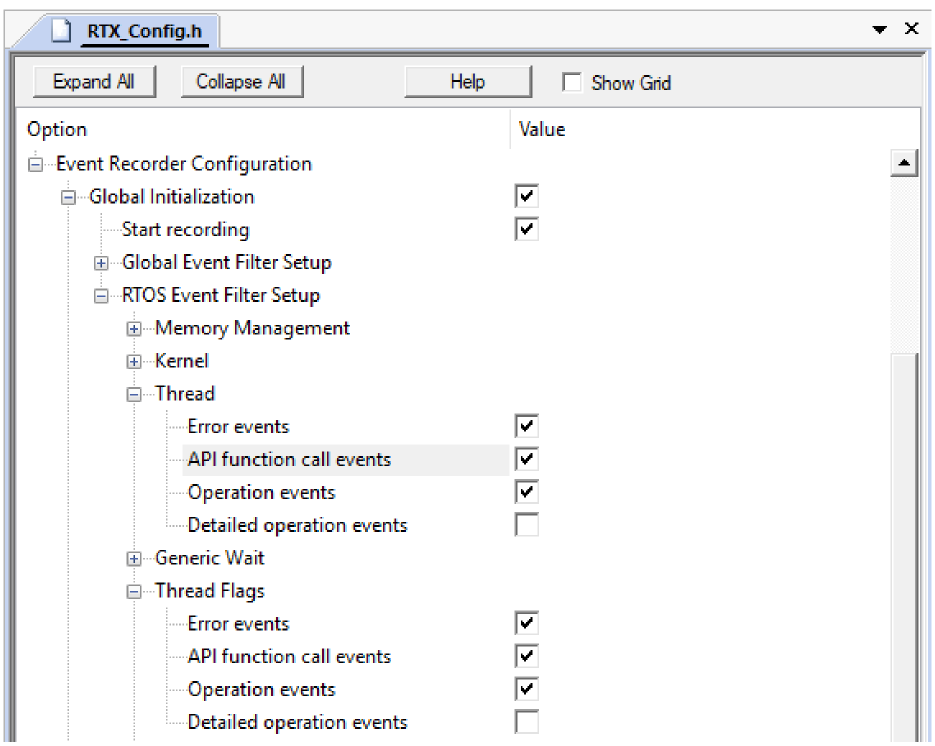
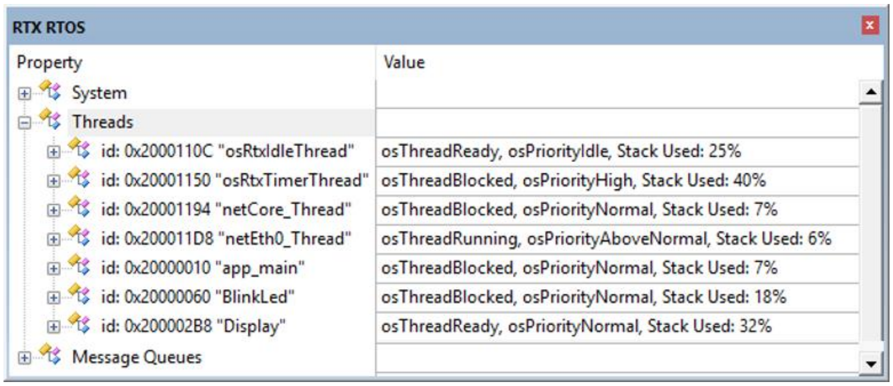


Figure 6 Event Recorder configuration for RTX RTOS

To track the thread switches, we must first find the thread identifiers. The *RTX RTOS* Component Viewer provides this information:



RTX RTOS

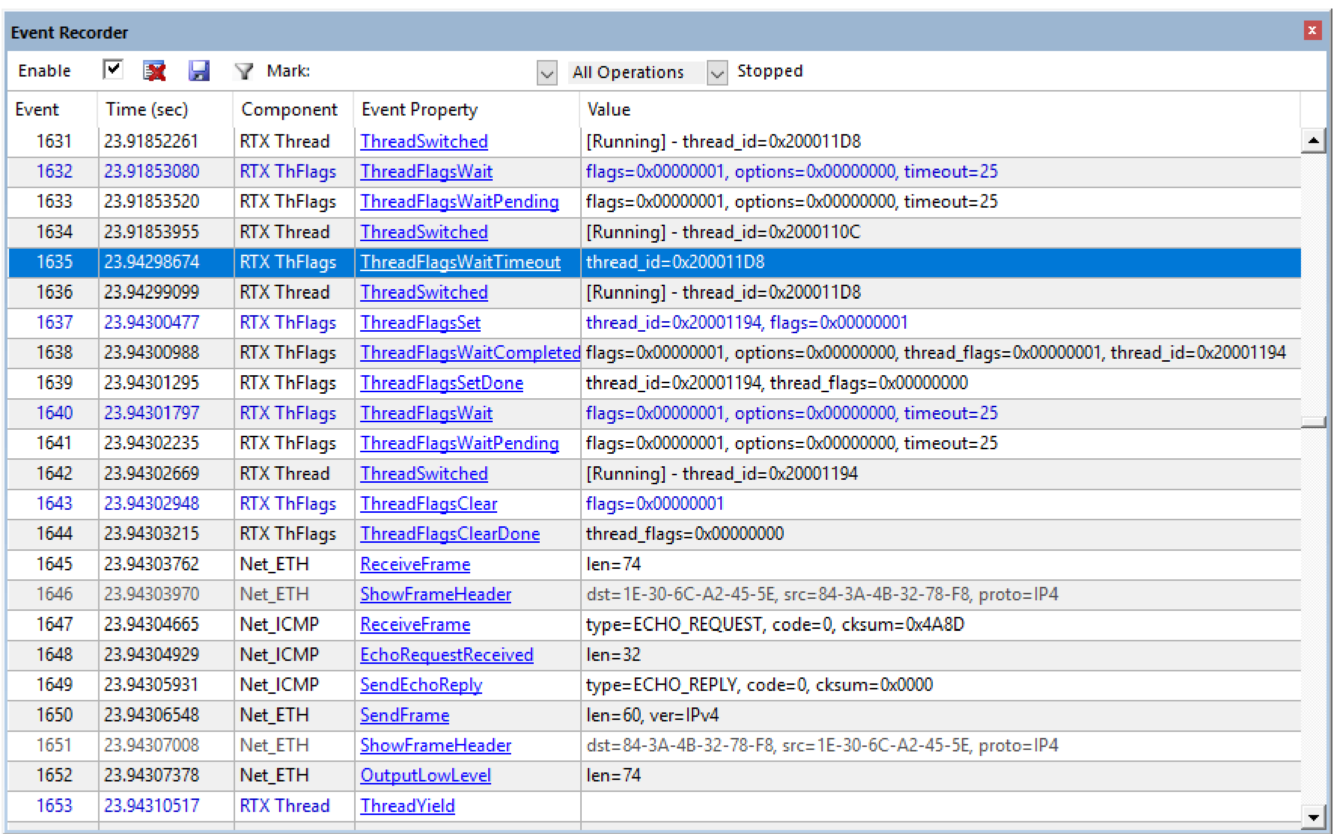
Property	Value
System	
Threads	
id: 0x2000110C "osRtxIdleThread"	osThreadReady, osPriorityIdle, Stack Used: 25%
id: 0x20001150 "osRtxTimerThread"	osThreadBlocked, osPriorityHigh, Stack Used: 40%
id: 0x20001194 "netCore_Thread"	osThreadBlocked, osPriorityNormal, Stack Used: 7%
id: 0x200011D8 "netEth0_Thread"	osThreadRunning, osPriorityAboveNormal, Stack Used: 6%
id: 0x20000010 "app_main"	osThreadBlocked, osPriorityNormal, Stack Used: 7%
id: 0x20000060 "BlinkLed"	osThreadBlocked, osPriorityNormal, Stack Used: 18%
id: 0x200002B8 "Display"	osThreadReady, osPriorityNormal, Stack Used: 32%
Message Queues	

Figure 7 RTX RTOS Watch window

The following threads are relevant for the analysis:

- ***netEth0_Thread*** with identifier ***0x200011D8:*** This thread handles the Ethernet interface. The thread waits to receive interrupt. When the thread wakes up, it calls the *GetRxFrameSize* function. If the function returns a positive number, the thread calls the *ReadFrame* to read the frame and release it from the driver.
- ***netCore_Thread*** with identifier ***0x20001194:*** This thread implements the Network Core function. The thread waits for the frame to be received, then wakes up and processes the frame.

Figure 8 shows the RTX and Network events captured when ping command is issued to the board from the PC:



Event Recorder

Enable | Mark: | All Operations | Stopped

Event	Time (sec)	Component	Event Property	Value
1631	23.91852261	RTX Thread	ThreadSwitched	[Running] - thread_id=0x200011D8
1632	23.91853080	RTX ThFlags	ThreadFlagsWait	flags=0x00000001, options=0x00000000, timeout=25
1633	23.91853520	RTX ThFlags	ThreadFlagsWaitPending	flags=0x00000001, options=0x00000000, timeout=25
1634	23.91853955	RTX Thread	ThreadSwitched	[Running] - thread_id=0x2000110C
1635	23.94298674	RTX ThFlags	ThreadFlagsWaitTimeout	thread_id=0x200011D8
1636	23.94299099	RTX Thread	ThreadSwitched	[Running] - thread_id=0x200011D8
1637	23.94300477	RTX ThFlags	ThreadFlagsSet	thread_id=0x20001194, flags=0x00000001
1638	23.94300988	RTX ThFlags	ThreadFlagsWaitCompleted	flags=0x00000001, options=0x00000000, thread_flags=0x00000001, thread_id=0x20001194
1639	23.94301295	RTX ThFlags	ThreadFlagsSetDone	thread_id=0x20001194, thread_flags=0x00000000
1640	23.94301797	RTX ThFlags	ThreadFlagsWait	flags=0x00000001, options=0x00000000, timeout=25
1641	23.94302235	RTX ThFlags	ThreadFlagsWaitPending	flags=0x00000001, options=0x00000000, timeout=25
1642	23.94302669	RTX Thread	ThreadSwitched	[Running] - thread_id=0x20001194
1643	23.94302948	RTX ThFlags	ThreadFlagsClear	flags=0x00000001
1644	23.94303215	RTX ThFlags	ThreadFlagsClearDone	thread_flags=0x00000000
1645	23.94303762	Net_ETH	ReceiveFrame	len=74
1646	23.94303970	Net_ETH	ShowFrameHeader	dst=1E-30-6C-A2-45-5E, src=84-3A-4B-32-78-F8, proto=IP4
1647	23.94304665	Net_ICMP	ReceiveFrame	type=ECHO_REQUEST, code=0, cksum=0x4A8D
1648	23.94304929	Net_ICMP	EchoRequestReceived	len=32
1649	23.94305931	Net_ICMP	SendEchoReply	type=ECHO_REPLY, code=0, cksum=0x0000
1650	23.94306548	Net_ETH	SendFrame	len=60, ver=IPv4
1651	23.94307008	Net_ETH	ShowFrameHeader	dst=84-3A-4B-32-78-F8, src=1E-30-6C-A2-45-5E, proto=IP4
1652	23.94307378	Net_ETH	OutputLowLevel	len=74
1653	23.94310517	RTX Thread	ThreadYield	

Figure 8 Thread and Network events for handling ping command

The column *Time (sec)* provides the time stamps of events. The echo frame is received in the event record *1635* when execution of *netEth0_Thread* with identifier *0x200011D8* is started. The processing of the frame is completed in the event record *1652*.

The *ThreadFlagsWaitTimeout* event *1635* is something that attracts attention in this case. The *netEth0_Thread* is waiting for a thread flag to be set from the Ethernet receive interrupt (as visible in event record *1632*) but receiving a timeout is not expected here.

Debugging the Ethernet driver

Adding custom events to the driver

Custom events can be added to record the time execution for following functions of interest:

- Ethernet receive interrupt.
- *GetRxFrameSize* function and its return value.

Using the *EventRecord2* function, a custom event with *id=1* is added in the *ETH_IRQHandler* interrupt handler:

```
/* Ethernet IRQ Handler */
void ETH_IRQHandler (void) {
   ..
  /* Callback event notification */
  EventRecord2 (1, 0, 0);
  Emac.cb_event (event);
}
```

An event with *id=2* is added in the *GetRxFrameSize* driver function. This event logs also the return value of the function:

```
static uint32_t GetRxFrameSize (void) {
  uint32_t len;
   ..
  EventRecord2 (2, len, 0);
  return (len);
}
```

Analyzing the complete flow

With the custom events added the Event Recorder gives now the complete picture showing the internal software operation when processing a ping request as shown on Figure 9:

Event Recorder

Enable Mark: All Operations Stopped

Event	Time (sec)	Component	Event Property	Value
2041	4.80003849		id=0x0001	0x00000000,0x00000000
2042	4.80004065	RTX ThFlags	ThreadFlagsSet	thread_id=0x200011D8, flags=0x00000001
2043	4.80004358	RTX ThFlags	ThreadFlagsSetDone	thread_id=0x200011D8, thread_flags=0x00000001
2044	4.80004816	RTX ThFlags	ThreadFlagsWaitCompleted	flags=0x00000001, options=0x00000000, thread_flags=0x00000001, thread_id=0x200011D8
2045	4.80005306	RTX Thread	ThreadSwitched	[Running] - thread_id=0x200011D8
2046	4.80005820		id=0x0002	0x00000000,0x00000000
2047	4.80006318	RTX ThFlags	ThreadFlagsWait	flags=0x00000001, options=0x00000000, timeout=25
2048	4.80006758	RTX ThFlags	ThreadFlagsWaitPending	flags=0x00000001, options=0x00000000, timeout=25
2049	4.80007190	RTX Thread	ThreadSwitched	[Running] - thread_id=0x2000110C
2050	4.80096957	RTX Thread	ThreadSwitched	[Running] - thread_id=0x20001150
2051	4.80097301	RTX ThFlags	ThreadFlagsSet	thread_id=0x20001194, flags=0x00000001
2052	4.80097783	RTX ThFlags	ThreadFlagsWaitCompleted	flags=0x00000001, options=0x00000000, thread_flags=0x00000001, thread_id=0x20001194
2053	4.80098095	RTX ThFlags	ThreadFlagsSetDone	thread_id=0x20001194, thread_flags=0x00000000
2054	4.80098724	RTX Thread	ThreadSwitched	[Running] - thread_id=0x20001194
2055	4.80099062	RTX ThFlags	ThreadFlagsClear	flags=0x00000001
2056	4.80099329	RTX ThFlags	ThreadFlagsClearDone	thread_flags=0x00000000
2057	4.80101540	RTX Thread	ThreadYield	
2058	4.80101879	RTX ThFlags	ThreadFlagsClear	flags=0x00000001
2059	4.80102136	RTX ThFlags	ThreadFlagsClearDone	thread_flags=0x00000000
2060	4.80103755	RTX ThFlags	ThreadFlagsWait	flags=0x00000001, options=0x00000000, timeout=-1
2061	4.80104198	RTX ThFlags	ThreadFlagsWaitPending	flags=0x00000001, options=0x00000000, timeout=-1
2062	4.80104635	RTX Thread	ThreadSwitched	[Running] - thread_id=0x2000110C
2063	4.80696061	RTX Thread	ThreadSwitched	[Running] - thread_id=0x20000060
2064	4.80696852	RTX Thread	ThreadSwitched	[Running] - thread_id=0x2000110C
2065	4.82495789	RTX ThFlags	ThreadFlagsWaitTimeout	thread_id=0x200011D8
2066	4.82496209	RTX Thread	ThreadSwitched	[Running] - thread_id=0x200011D8
2067	4.82496745		id=0x0002	0x0000004A,0x00000000
2068	4.82497730		id=0x0002	0x00000000,0x00000000
2069	4.82497968	RTX ThFlags	ThreadFlagsSet	thread_id=0x20001194, flags=0x00000001
2070	4.82498450	RTX ThFlags	ThreadFlagsWaitCompleted	flags=0x00000001, options=0x00000000, thread_flags=0x00000001, thread_id=0x20001194
2071	4.82498761	RTX ThFlags	ThreadFlagsSetDone	thread_id=0x20001194, thread_flags=0x00000000
2072	4.82503280	RTX ThFlags	ThreadFlagsWait	flags=0x00000001, options=0x00000000, timeout=25
2073	4.82503722	RTX ThFlags	ThreadFlagsWaitPending	flags=0x00000001, options=0x00000000, timeout=25
2074	4.82504154	RTX Thread	ThreadSwitched	[Running] - thread_id=0x20001194
2075	4.82504435	RTX ThFlags	ThreadFlagsClear	flags=0x00000001
2076	4.82504701	RTX ThFlags	ThreadFlagsClearDone	thread_flags=0x00000000
2077	4.82505237	Net_ETH	ReceiveFrame	len=74
2078	4.82505443	Net_ETH	ShowFrameHeader	dst=1E-30-6C-A2-45-5E, src=84-3A-4B-32-78-F8, proto=IP4
2079	4.82506123	Net_ICMP	ReceiveFrame	type=ECHO_REQUEST, code=0, cksum=0x4A68
2080	4.82506386	Net_ICMP	EchoRequestReceived	len=32
2081	4.82507370	Net_ICMP	SendEchoReply	type=ECHO_REPLY, code=0, cksum=0x0000
2082	4.82507928	Net_ETH	SendFrame	len=60, ver=IPv4
2083	4.82508385	Net_ETH	ShowFrameHeader	dst=84-3A-4B-32-78-F8, src=1E-30-6C-A2-45-5E, proto=IP4
2084	4.82508757	Net_ETH	OutputLowLevel	len=74
2085	4.82511874	RTX Thread	ThreadYield	

Figure 9 Event Recorder capture with custom events

Table 2 describes key events:

Event	Description
2041	Ethernet receive interrupt occurs.
2042	Interrupt function sets event for netEth0_Thread (id=0x200011D8).
2045	RTOS switches to netEth0_Thread.
2046	netEth0_Thread calls GetRxFrameSize which returns 0 (no frame available).
2049	netEth0_Thread suspends, no frame is processed.
2065	netEth0_Thread times out after 25 milliseconds and resumes execution.
2067	netEth0_Thread calls GetRxFrameSize which returns 74 (frame valid).
2069	netEth0_Thread sets event for netCore_Thread (id=0x20001194).
2074	RTOS switches to netCore_Thread, which processes the frame and generates the echo reply.

Table 2 Description of observed network events

This log shows that the *GetRxFrameSize* driver function is not working properly. When the frame is received, *netEth0_Thread* calls *GetRxFrameSize*, but the function returns *0* instead of the correct size of the received frame. Then, the *netEth0_Thread* switches to sleep mode and wakes up after a safety timeout of *25* milliseconds. The function *GetRxFrameSize* is then called again, but this time, the function returns the correct frame length.

Further review of the *GetRxFrameSize* function code showed a problem in processing received frames. After correcting the problem, the driver is running correctly as shown on Figure 1010.

```
C:\>ping 192.168.3.26

Pinging 192.168.3.26 with 32 bytes of data:
Reply from 192.168.3.26: bytes=32 time<1ms TTL=128
Reply from 192.168.3.26: bytes=32 time<1ms TTL=128
Reply from 192.168.3.26: bytes=32 time<1ms TTL=128
Reply from 192.168.3.26: bytes=32 time<1ms TTL=128

Ping statistics for 192.168.3.26:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

C:\>
```

Figure 10 Correct response on ping **command**

Summary

This application note demonstrated Event Recorder's powerful capabilities for locating a performance issue observed in a complex network application with multiple components:

- Events present in the MDK-Middleware network library provided visibility into the internal operation of the networking stack.
- Keil RTX5 events informed about thread switches and other kernel operations.
- User annotated events were used for custom events that provided additional details and helped to measure execution times.

Useful links

- Keil Arm Compiler Extensions pack contains Event Recorder component
- Event Recorder documentation provides details on configuring and using Event Recorder and Event Statistics in application.
- Event Recorder support in MDK explains how to use Event Recorder in µVision debugger
- Add Event Recorder visibility describes how to enable Event Recorder in CMSIS-RTOS2
- Troubleshooting a network application gives some recommendations on debugging network-related issues.