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
Using Event Recorder for debugging a network performance issue
Abstract
Prerequisites
Introduction
Problem Description
Symptoms
Analyzing the issue
Debugging the network library
Debugging the RTX5 thread switches
Debugging the Ethernet driver
Adding custom events to the driver
Analyzing the complete flow
Summary
Useful links
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:

![Ping statistics](image)

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:

![Manage Run-Time Environment](image)

**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.

![Net_Debug.c](image)

**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 Recorder Window](image)

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

Table 1 provides description for the captured events:

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

**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.
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:

![Selecting Keil RTX5 source code variant](image)

**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.

![Event Recorder configuration for RTX RTOS](image)

**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 Watch window](image)

**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:

![Thread and Network events for handling ping command](image)

**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:

```c
/* 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:

```c
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 capture with custom events](image)

Figure 9 Event Recorder capture with custom events
Table 2 describes key events:

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

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 10.

![Figure 10 Correct response on ping command](image-url)
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.