Software Analysis with Event Annotations

High-level program trace via the standard Cortex-M debug access port

May, 2020

White Paper

Run/stop debug combined with instruction trace was the technology for analyzing embedded software. Today finding bugs at the instruction level is unmanageable as devices lack trace pins, operate at high-speed or have multi-core processors. Instead, event annotations can be used to analyze the dynamic operation and the timing of complex software stacks on Arm Cortex-M systems during execution. It is available for all Cortex-M devices, does not require trace and offers detailed insight into "black-box" software components. Multiple views to run-time behavior, including task scheduling and timing provide high-level information about the operation of communication stacks or inter-process communication. Statistic view show infrequent program anomalies or worst-case timing. Events are easy to add to user code and many RTOS systems or IoT communication stacks are already annotated. Other benefits include post-mortem system analysis or remote diagnosis of IoT devices.

Introduction

Flexible and easy-to-use middleware components are essential in modern microcontroller applications. These software components are often a "black box" for the application programmer. Even when comprehensive documentation and source code are provided, analyzing potential issues is challenging.

Therefore, modern software development tools offer viewers for software components and event recording facilities to help to understand and analyze the internal operation. Some RTOS systems (for example FreeRTOS or Keil RTX) and middleware components already include annotations that provide high-level operation of the software components. However, it is easy to add event annotations to any user application or software component.

Traditional Debug Technology

Below is a review of various traditional debug techniques that are used with Cortex-M based microcontrollers in embedded systems.

JTAG/SWD – Run/Stop Debugging

Run/stop debugging enables the programmer to monitor the execution of a program, stop it, restart it, set breakpoints and change values in memory. But obviously stopping program execution changes the overall system behavior – timing cannot be analyzed. Therefore, for time-critical embedded systems, it imposes several practical problems, for example:

- Power conversion algorithms drive electromagnetic coils (i.e. in motors). Stopping the algorithm may cause high current state when the voltage of a dynamic system is consistently on. It may result in overheating and potential damage to the system.
- A communication protocol typically has timeouts. Stopping a system that uses for example an internet protocol may quickly cause timeouts and therefore the system cannot be analyzed this way.

SWO: ISR events and ITM annotations

Many Cortex-M microcontrollers offer a SWO (Serial Wire Output) that is single, serial data trace pin. SWO enables the Serial Wire View (SWV) capabilities and provides real-time data trace information from various sources within a Cortex-M device. This information is transmitted while your system processor continues to run at full speed.
The SWV provides you with features such as:

- PC (Program Counter) sampling
- Event counters that show CPU cycle statistics
- Exception and Interrupt execution with timing statistics
- Trace data - data reads and writes used for timing analysis
- ITM (Instrumented Trace Macrocell) information that can be used for printf-style outputs or other simple program annotations.

In practice SWV gives you great insight into ISR (interrupt service routine) timing but utilizing to many features (such as extensively using ITM for user code annotations) may result in bandwidth limitations that are caused by the small data buffers in the SWV hardware. As the data output rate depends also on the processor clock, embedded systems that change dynamically the clock rate are hard to analyze. The SWO is not available in JTAG mode or with devices that are based on Cortex-M0/M0+/M23 processor or implement a heterogenous Cortex-A/Cortex-M systems impose further restrictions.

ETM: Instruction trace analysis

Instruction trace can be used for execution profiling, performance optimization, and code coverage. However, recording all instructions on microcontrollers that executed 100 million instructions per second can be ooverwhelming for high-level analysis as it is not easy to identify the relevant program points. Also just recording the instructions is typically insufficient as the data that are processed are equally important.

The ETM output requires additional I/O pins and an expensive debugger that can capture the trace information. Frequently ETM is not implemented in a cost constrain microcontroller and therefore this technique is not consistently available in microcontroller devices.

printf-style debugging via USART

Annotations give insights for application execution.

Practical problems with printf:

- Test and production not identical
- Intrusive in execution timing
- Not suitable for interrupts
- Multi-threading requires additional provisions

Event Recording

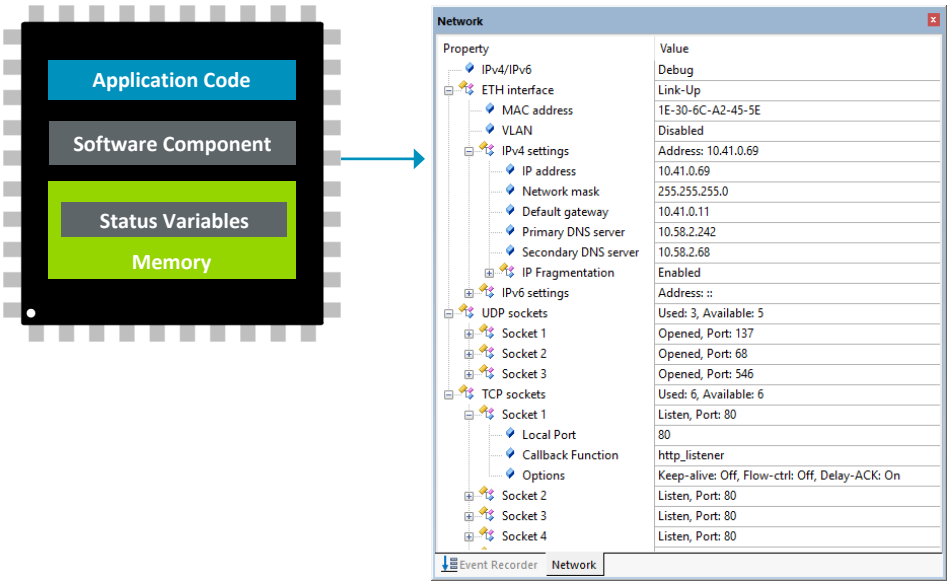
Today's applications are composed of ready-made software components and users need an effective way to view the operation and ensure proper usage. Current status information and dynamic operation views are required to analyze a software component. While both can be obtained using printf-style debugging, it becomes soon to complicated with complex software applications.

For microcontrollers the printf annotations may also impose significant memory overhead. Therefore, offloading information to debug tools that are hosted on the development computer are required to cope with the resource restrictions. The following two components are available in the MDK debugger that work via the standard Cortex-M JTAG/SWD debug interface and only require memory reads to the target system to capture data. No additional trace port is needed, and it is possible to use low-cost debug adapters that are frequently integrated on evaluation kits or professional debug adapters that provide a higher data rates on the debug port.

Component Viewer

The **Component Viewer** reads specific memory locations from the target hardware using a standard debug unit (for example a ST-Link or ULINK) that is connected via JTAG or SWD to Debug Access Port (DAP) of the device. The address of these memory locations is typically defined by symbols that represent variables, arrays or linked lists.

Figure 1:
Component Viewer



A System Component View Description (*.SCVD) file (XML format) defines the symbols and the formatting in the Component Viewer window of the debugger. Such *.SCVD files are already available for several software components like FreeRTOS, Keil RTX5 and the MDK middleware that includes TCP/IP stack, file system and various USB interfaces.

The overall benefits of the **Component Viewer** are:

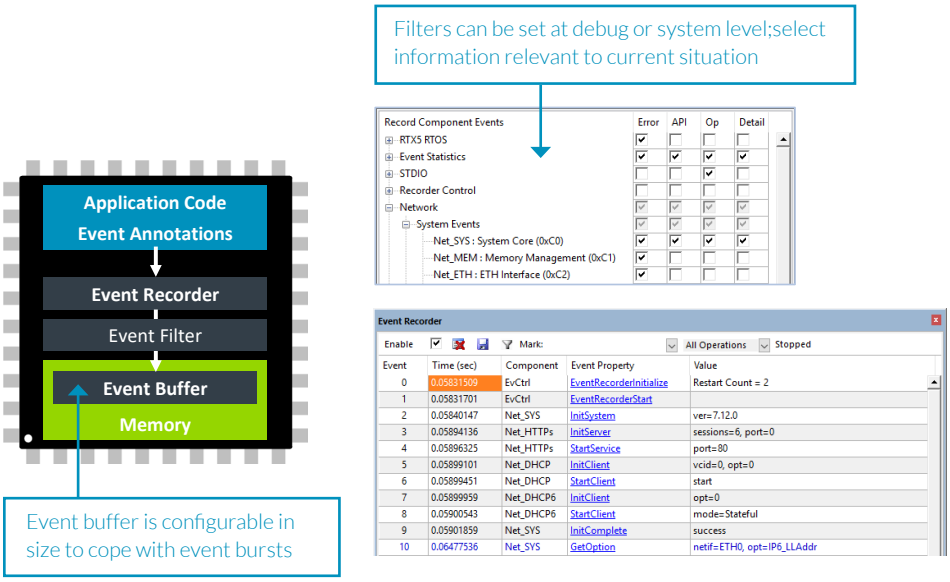
- Visibility of a relevant software component or application program for the static user with no additional software overhead in the target application.
- The information is obtained via debug symbols from target memory using simple read commands via JTAG or SWD connectivity to a debug adapter.
- Debug adapters, that support hot plugging, allow to show the current status of the application software in case of failures.

Event Recorder

The software component Event Recorder provides an API (function calls) for event annotations in the application code or software component libraries. These API functions record event timing and data information while the program is executing.

The **Event Filter** is controlled from the target application or the debugger and allows to specify the event IDs, that need to be stored in the **Event Buffer**, located in the memory of the target system.

Figure 2:
Event Recorder



During program execution, the debugger reads the content of the Event Buffer using a standard debug unit (for example a ULINK debug adapter) that is connected via JTAG or SWD to the CoreSight Debug Access Port (DAP).

The **Event Recorder** requires no trace hardware and therefore can be used on any Cortex-M processor with the following functionalities:

- Visibility of the dynamic execution of an application with little memory overhead and no extra target hardware requirements
- On Arm Cortex-M3/M4/M7/M33 based devices, the Event Recorder functions do not disable interrupts. There is no impact or thread execution or interrupt behaviour.
- Fast time-deterministic execution of event recorder functions with minimal code and timing overhead
- No need for debug or release build as the event annotations can remain in the production code
- Saving the event data in local memory ensures fast recording.
- Collecting the data from the on-chip memory is done using simple read commands. These commands work on all Cortex-M processor based devices and require only JTAG or SWD connectivity to the debug adapter.
- Flexible time stamp generation, for example using the Debug Watch Timer (DWT does not require any extra peripherals. The DWT is available on Arm Cortex-M3/M4/M7/M33.

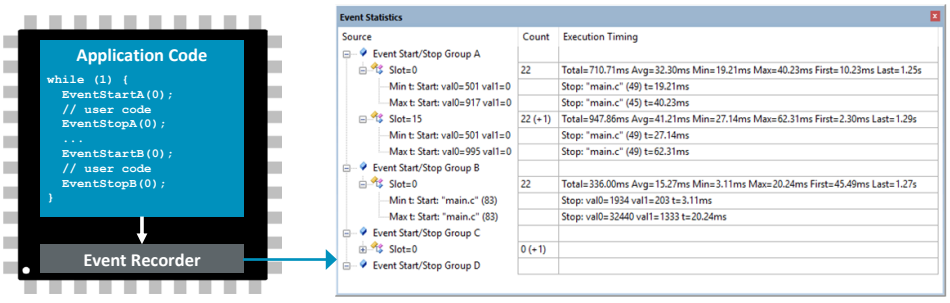
Event Statistics

Port of the Event Recorder interfaces are also Event Statistics functions that allow statistical data collection about the code execution. Any debug adapter can be used to record execution timing and number of calls for annotated code sections.

Using the power measurement capabilities of ULINKplus debug adapter, the **Event Statistics** can collect power consumption data (with min/man/average values) when the annotated code section is being executed. Such energy profiling is especially relevant for battery-driven applications.

Log files enable comparisons between different build runs in continuous integration (CI) environments.

Figure 3:
Event Statistics



Using Event Recorder

For using Event Recorder in an embedded application, the related header and source files are included in the project. A configuration file allows the setup of the event buffer size and the time stamp source clock.

The following Event Recorder functions are used for capturing program information:

```
EventRecord2 (int id, int val1, int val2)
EventRecord4 (int id, int val1, …, int val4)
EventRecordData (int32 id, void *data, int len)
```

A complete reference to all Event Recorder functions is provided in the related documentation [1]. During program testing, the debugger retrieves the Event Recorder data from the target and visualizes the information.
An application note explains how custom events can be used to analyze a timing anomaly in a peripheral driver of a networking application [2]. Using run-stop debug on such applications is typical not feasible as it would result in communication timeout.

Comparison Event Recorder vs. Printf

We have compared the capabilities and required overhead of using Event Recorder vs. printf-style output for analyzing the dynamic operation of a software component.

- The Event Recorder provides time stamps on events based on a selected clock source. For printf-style output a custom framework needs to be developed.
- Data management with event ID assignment, filter options and protocol output are available with Event Recorder. It requires custom implementation for parsing ASCII text when printf-style output is used.

Comparison of Resource Requirements

Table 1: Compares the resources and execution time required for printf-style or Event Recorder output on a typical embedded system platform[1].

Resource	printf- style output	Event Recorder
Communication peripheral	UART	Debug port
ROM	7684 Bytes	1180 Bytes
RAM	368 Bytes	Configurable RAM Buffer
Execution time	1.5 ms	2.8 µs

[1]: Reference platform: Cortex-M4 @ 120MHz (Infineon XMC4500) using CMSIS-UART driver @ 115200 baud

Visual Trace Diagnostics

Event recording has a significant potential for improving software development, assuming there are adequate analysis tools. Event trace analysis can be divided into two main classes, bottom-up and top-down, with respect to the abstraction level.

Bottom-up analysis implies a direct inspection of the detailed sequence of events and is suitable for verifying specific test cases in the analyzed system.

To use event recording for debugging, a top-down analysis is often needed in order to identify abnormal patterns in the event trace. This is greatly facilitated by tool support for Visual Trace Diagnostics with Exploratory Analysis. This implies a tool that not only visualizes the data but also facilitates spotting anomalies in large event traces, in cases where the problematic code is unknown.

Percepio Tracealyzer

A prime example of a visual trace diagnostics tool with support for exploratory analysis is Percepio Tracealyzer. It supports event recording for several embedded software platforms, including the Event Recorder for Cortex-M microcontrollers, as shown in this application note and webinar.

System-Level Debugging using Visual Trace Diagnostics

This section presents an example of how system-level debugging can be facilitated by a visual top-down exploratory analysis using Percepio Tracealyzer.

Fig. 4 shows UART output from an embedded system, where an error is highlighted. The UART is used as a convenient example of a shared hardware resource, but the same type of problem can occur for any type of shared resource, such as an SPI bus, and with less obvious symptoms.

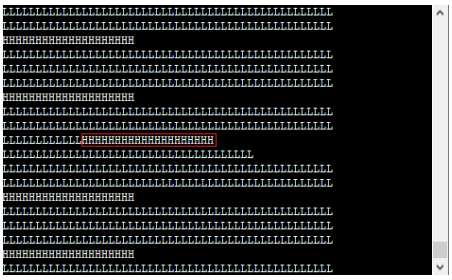
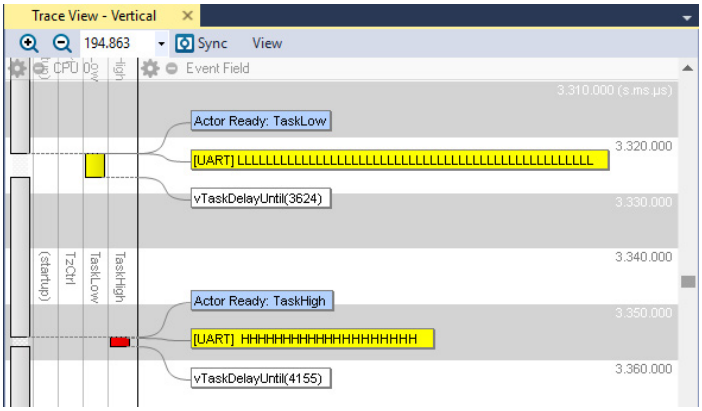


Figure 4: An example error in UART output

Fig. 4 shows two kinds of output symbolized by “L” and “H” for clarity. Sometimes the “L” data is accidentally mixed with the “H” data, resulting in incorrect output.

To analyze this issue, we record an event trace and get a visual display in Tracealyzer, as seen in below. Custom logging calls have been added to capture the UART transmissions, while RTOS kernel events are recorded automatically by pre-existing annotations in the RTOS kernel. This kind of custom logging gives the flexibility of classic printf debugging, but without the limitations and drawbacks.

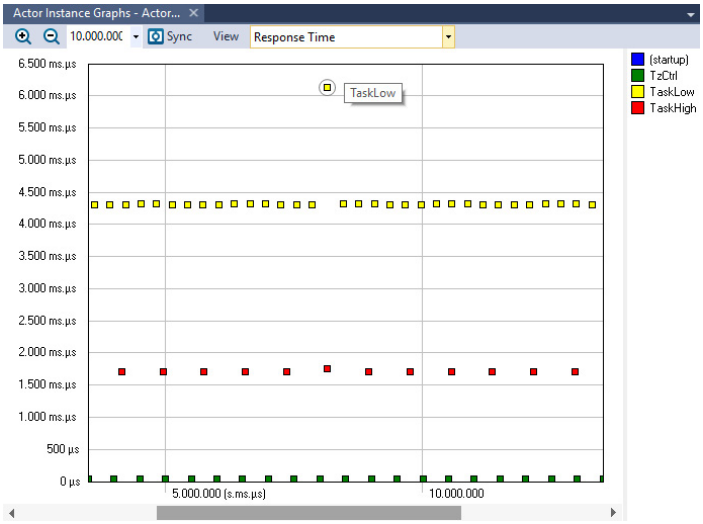
Figure 5:
A subset of the recorded event data, shown in Tracealyzer



We can see that the UART is used by two different RTOS tasks, but before finding the cause of the error, first the error in the trace should be found. An event trace may contain thousands or even millions of events, so a manual search is not practical, especially since we don't know what the error looks like on this detailed level.

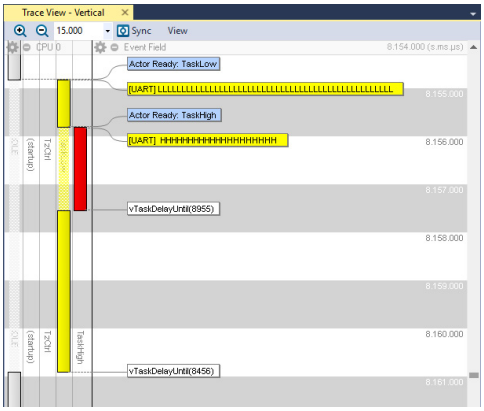
For this purpose, we can use one of the overviews available in Tracealyzer, such as the Actor Instance Graphs (Fig 6) showing a plot over timing metrics for each execution of the tasks, in this case the Response Time. This reveals any disturbances in the task execution, such as preemptions by higher priority tasks.

Figure 6:
An anomaly in task response times can be a sign of the error.



In Fig 6 we can immediately spot an anomaly where the response time is a lot higher than normal, which can be a sign of the error. To find out, we inspect the corresponding event sequence (Fig 7). This is easy to find as the views are linked.

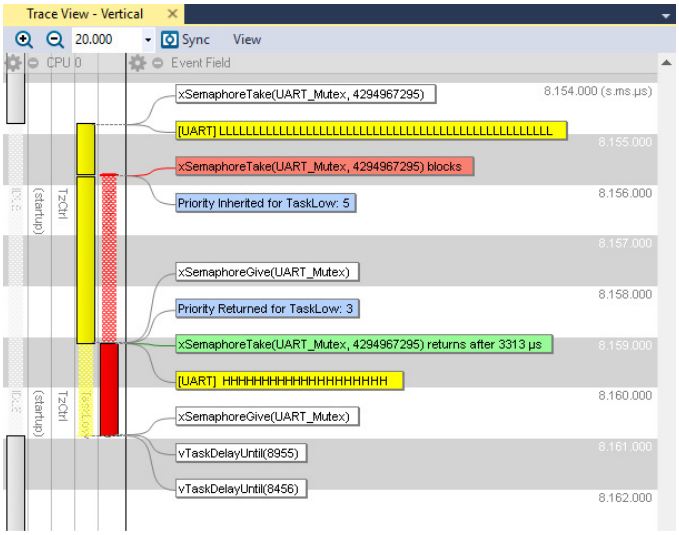
Figure 7:
The corresponding event sequence reveals the problem.



In Fig. 7 it is shown, how TaskHigh preempts TaskLow during the UART transmission. It seems as the UART driver is not suitable for use in multi-threaded RTOS applications, but this can be fixed by adding Mutex calls in the UART driver to prevent such simultaneous access.

Once the solution is implemented, we can also use event recording and visual trace diagnostics to verify the solution. The result can be seen in Fig.8. Both tasks must now attempt to take the “UART_Mutex” object before they can write to the UART. TaskHigh still preempt TaskLow but is now blocked by the Mutex before it begins the UART transmission. As a result, the UART output is no longer corrupted by the preemption.

Figure 8:
The event sequence after adding Mutex calls.



Summary

In complex embedded applications, it is often very difficult to find the root cause of reduced performance or incorrect program operation. Event annotations provide serious benefits in such situations. Several software stacks and RTOS kernels have already hooks for event annotations. Applications that use such software components are easier to develop as incorrect usage can be identified faster.

The Event Recorder works via the standard debug access port of all Cortex-M microcontrollers with a wide range of debug adapters. The program annotations can remain in the production code, which lets to verify an application before final release. Filters allow the retrieval of specific information for analyzing a problem and a protocol can be used to verify systems after code modifications.

Combined with Visual Trace Diagnostic, which gives you a feature-rich toolbox, developers can identify sporadic anomalies before resulting reduced system performance or functional product defects.

References

[1] Arm, "Event Recorder and Component Viewer" www.keil.com/pack/doc/compiler/EventRecorder/html/

[2] Arm, "Application Note 320: Using Event Recorder for debugging a network performance issue" www.keil.com/appnotes/files/apnt_320.pdf

[3] Arm, "Application Note 321 & Example Code: Event Recorder Debugging with STM32G Cortex-M0/M0+" www.keil.com/appnotes/docs/apnt_321.asp

[4] Arm and Percepio, 'Webinar: Software analysis of complex Cortex-M applications www.brighttalk.com/webcast/17792/408140