Beyond Hardware-based Trace: Useful Alternatives for Run-Time Analysis

2026-01-26 - Comments (0) - Trace

Trace is often required for use cases such as function and task run-time analysis. Off-chip trace is generally the preferred approach for such use cases, as it provides detailed and reliable results without requiring any modification to the target application. When off-chip trace is not possible, for example due to limited processor or board capabilities, then on-chip trace can be an alternative. However, on-chip trace allows only very short recording times because of its limited on-chip trace buffer.

Background Information

Off-chip trace means that the trace data is transferred during recording to an external trace tool such as PowerTrace, CombiProbe, or µTrace®. On-chip trace, in contrast, stores the trace information in a dedicated internal trace buffer.

But what options remain if neither off-chip nor on-chip trace can be used? This article addresses that question by presenting and comparing alternative approaches for run-time analysis.

We begin with instrumentation-based trace, followed by sample-based profiling, highlighting the differences between the two methods.

Note

This article explains how the different trace methods work and compares the results obtained using these methods. It does not, however, provide detailed instructions on how to set up each method. For setup information, please refer to the chapter **“Related Documentation & Files.”**

Note

On some target processors, run-times can also be measured using BenchMark Counters (BMC). For more details, refer to Measuring function run-times with BenchMark Counters (BMC).

Used Target Platform and TRACE32 Hardware:

All measurements in this article were performed on a NUCLEO-H563ZI development board featuring an Arm Cortex-M33F core running FreeRTOS, connected to a µTrace®. The approaches described are, however, generic and applicable to other processor architectures as well.

Instrumentation-based Trace

TRACE32 provides two primary features for instrumentation-based trace: **LOGGER** and **FDX trace**.

Both LOGGER and FDX are software-based trace methods that require modifications to the target application, which must be instrumented to write trace information into reserved memory. Consequently, sufficient spare RAM on the target system is essential.

LOGGER trace records trace information into a reserved RAM buffer during program execution. Once recording stops, TRACE32 retrieves the data from target memory for display and analysis. The recording length is limited by the buffer size. If the buffer becomes full, the application can either overwrite older entries (FIFO mode) or stop writing new data (Stack mode).

FDX trace, by contrast, streams trace information to the TRACE32 PowerView software on the host, typically via memory access during execution. The reserved buffer on the target acts as a temporary FIFO, enabling longer recording sessions. If the communication channel cannot transfer data quickly enough, the FDX target

code stalls execution to prevent data loss.

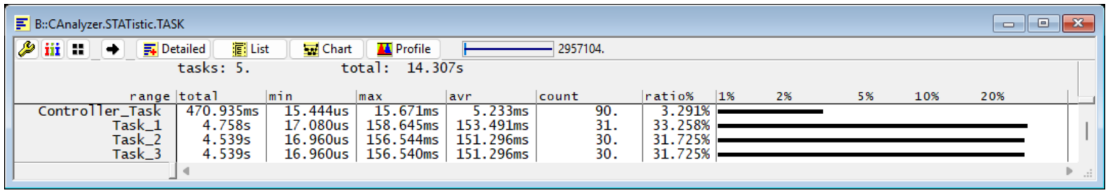
To support these methods, Lauterbach provides dedicated source file sets for LOGGER and FDX trace, which must be integrated into the target application.

Note

Lauterbach provides dedicated support for tracing AUTOSAR Classic Platform systems through instrumentation trace based on the AUTOSAR Run-Time Interface (ARTI) trace hooks. For further details, refer to the Application Note Profiling on AUTOSAR CP with ARTI.

In the following, we conduct a test measurement for task run-time analysis, first using LOGGER and then FDX trace. The results are subsequently compared with those obtained from off-chip trace. In all three cases, the execution recorded is identical.

The off-chip trace results for the original, unmodified target application are presented below and serve as the reference baseline for the subsequent comparisons.



Note

The displayed *min*, *max*, and *average* runtimes in the **Trace.STATistic.TASK** windows are derived from task-switch traces. They therefore represent the periods during which a task was executed without interruption by the scheduler, rather than the complete runtime of the task from start to finish.

For more advanced runtime analysis, TRACE32 PowerView allows tracing of task state changes, which provides a more detailed view of task execution behavior.

The runtimes shown in the statistic windows are sufficient for comparing the different trace methods.

LOGGER Task Trace

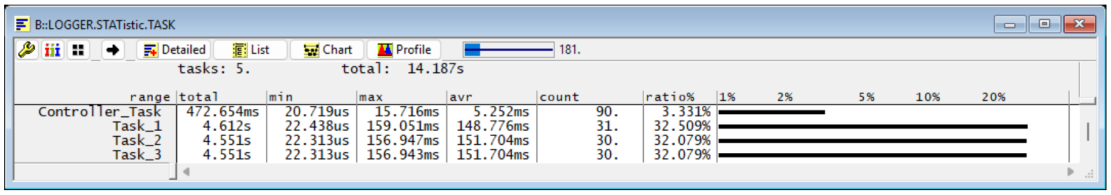
For FreeRTOS, task-switch information can be captured with LOGGER by defining the macro `traceTASK_SWITCHED_IN()` in `FreeRTOS.h`:

```
#define traceTASK_SWITCHED_IN() T32_LoggerData (T32_DATA_WRITE|T32_LONG, (void
*)&pxCurrentTCB, (unsigned long)pxCurrentTCB)
```

To obtain timing information when using LOGGER, the user must implement the `T32_TimerGet` function. In our case, we utilize the **TIM2** counter of the STM32H563ZI:

```
uint64_t T32_TimerGet(void)
{
    return (*(volatile uint64_t *)0x50000024u);
}
```

The following screenshot shows the task run-time results obtained with LOGGER trace.



By comparing the two results, we observe only minor deviations in the total, average, and maximum task run-times between LOGGER and off-chip trace. A more pronounced deviation appears in the minimum run-time values, as the logging overhead tends to mask very short execution times. The table below summarizes the differences for `Controller_Task` as an example.

Task Run-Time Comparison (Off-Chip vs. LOGGER Trace)

Metric	Off-chip Trace	LOGGER Trace	Deviation %
Controller_Task: total	470.935 ms	472.654 ms	+0.36%
Controller_Task: average	5.233 ms	5.252 ms	+0.36%
Controller_Task: min	15.444 µs	20.719 µs	+34.2%
Controller_Task: max	15.671 ms	15.716 ms	+0.29%

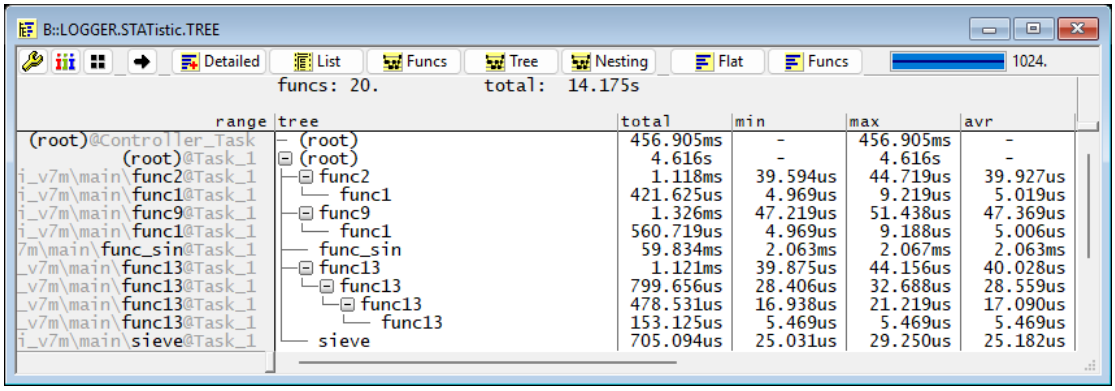
LOGGER Function Trace

Task tracing with instrumentation-based methods is relatively straightforward, as it requires instrumentation at only a single code location. However, when tracing function runtimes, both the entry and exit points of each function must be instrumented. This inevitably introduces higher run-time and memory overheads compared to task-level tracing.

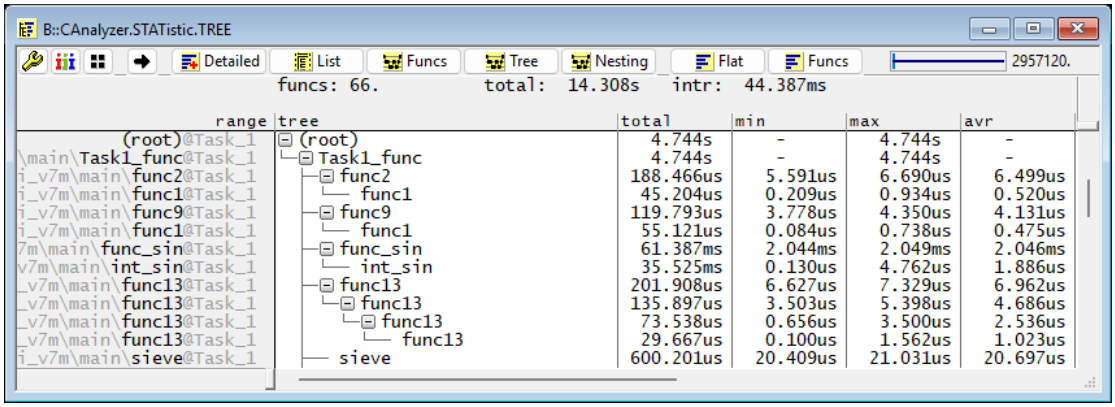
Example: Instrumentation of the function func_sin

```
static void func_sin()
{
        int x;
        T32_LoggerData (T32_FETCH, (void*)(((uint32_t)func_sin) & ~1U), 0 /* unused
*/);
        for (x = 0; x < 628; x++)
                sinewave[x] = int_sin(x)/(x/32+1);
        T32_LoggerData (T32_FETCH, NULL, 0 /* unused */);
}
```

In the following example, we trace a set of selected functions. The results are displayed below in form of a function call tree



The next screenshot shows the runtime results of the non-instrumented code, obtained though off-chip trace



Two key observations can be made from the comparison:

- Functions such as `func2` exhibit significantly higher execution times in LOGGER due to instrumentation overhead (maximum runtime: **44.718 µs** with LOGGER versus only **6.690 µs** with off-chip trace). In practice, `func2` calls the LOGGER function `T32_LoggerData` twice and invokes `func1` three times. Each call to `func1` in turn triggers `T32_LoggerData` twice, resulting in a total overhead of eight calls to `T32_LoggerData`. In our demo, a single execution of this function takes in average **4.3 µs** .

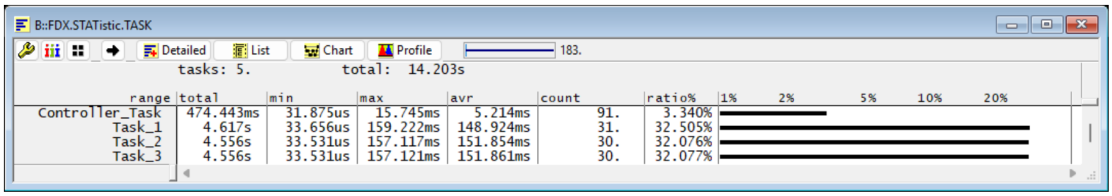
- For functions with higher runtimes, such as `func_sin`, LOGGER produces results very close to off-chip trace (maximum runtime: **2.067 ms** with LOGGER versus **2.049 ms** with off-chip trace). This indicates that LOGGER is more suitable for tracing larger functions. The small deviation observed is partly due to the fact that `int_sin`, which is called multiple times by `func_sin`, is not instrumented.

- The LOGGER trace only includes functions explicitly instrumented in the code. Any function not tagged (e.g., `int_sin`) will be absent from the LOGGER call tree, while off-chip trace captures all executed functions. The runtimes of these functions are however included in the LOGGER results for caller functions.

FDX Task Trace

We now repeat the same task trace example, this time using FDX instead of LOGGER. As previously explained, the key advantage of FDX over LOGGER is its ability to achieve longer recording times by streaming trace information directly to the host during execution.

The corresponding results are shown below.



The observed results are similar to those obtained with LOGGER, with a larger deviation in the minimum run-times.

Task Run-Time Comparison (Off-Chip vs. FDX Trace)

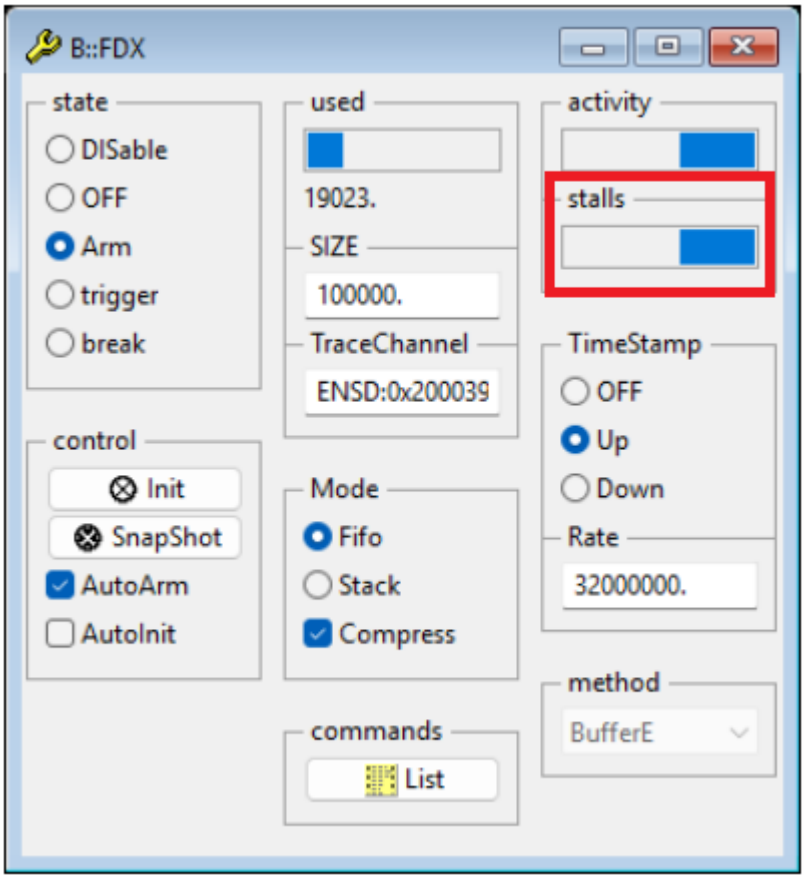
Metric	Off-chip Trace	FDX Trace	Deviation %
Controller_Task: total	470.935 ms	474.443 ms	+0.75%
Controller_Task: average	5.233 ms	5.214 ms	–0.36%
Controller_Task: min	15.444 µs	31.875 µs	+106.4%
Controller_Task: max	15.670 ms	15.745 ms	+0.48%

FDX Stalls

In addition to recording task switches, we now also capture write accesses to an array that is updated within a loop. This example illustrates the scenario in which the volume of generated trace data exceeds the available streaming bandwidth.

```
for (x = 0; x < 628; x++) {
    sinewave[x] = int_sin(x)/(x/32+1);
    T32_Fdx_TraceData (0x32, (void *)&sinewave[x], (unsigned long)sinewave[x]);
}
```

During recording, stalls can be observed in the **FDX** window, as shown below.



At certain times, the volume of generated trace information exceeds the capacity of the trace channel. Consequently, the FDX target code must wait until the channel becomes available. This behavior intentionally introduces stalls and affects the run-time performance of the target application in order to avoid trace data loss. The likelihood of stalls strongly depends on both the amount of trace data being exported and the characteristics of the target processor.

TRACE32 PowerView additionally provides the PRACTICE function `FDX.TargetSTALLS()` to monitor stalls from a script.

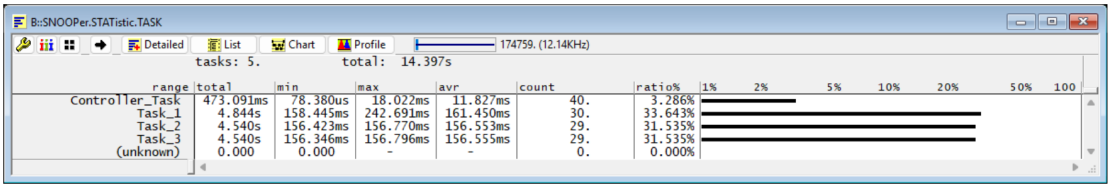
Sample-based Profiling

In some cases, instrumentation-based trace is not feasible; for example, when the target code cannot be modified. In such situations, the only available option to gain an overview of run-time behavior is sample-based profiling. TRACE32 supports this approach through its **SNOOPer** feature.

Sample-based profiling works by periodically collecting snapshots of the program counter and/or data values. For task tracing, this can be achieved by sampling the variable that stores the current task identifier. Because the trace relies on periodic sampling rather than continuous recording, the results are approximate and inherently include a statistical margin of error. Sample-based profiling is therefore useful for obtaining a broad overview of runtimes, but it is not suitable in scenarios where precise accuracy is required, such as verifying function or task run-time constraints. Moreover, the accuracy of sample-based profiling strongly depends on the maximum achievable sampling frequency, which is determined by the target processor, as well as on whether sampling can be performed during run-time or requires stopping program execution to collect samples.

We now test this by sampling tasks using the SNOOPer trace, again with exactly the same code execution as in the examples above.

The results are displayed below.



While the total function run-times show only a relatively small deviation, the accuracy is noticeably lower. This reduced precision is particularly evident in the longer average and maximum values.

Task Run-Time Comparison (Off-Chip vs. SNOOPer Trace)

Metric	Off-chip Trace	SNOOPer Trace	Deviation %
Controller_Task: total	470.935 ms	473.091 ms	+0.46%
Controller_Task: average	5.233 ms	11.827 ms	+126%
Controller_Task: min	15.444 µs	78.380 µs	+507.5%
Controller_Task: max	15.671 ms	18.022 ms	+15%

Note

Alongside **SNOOPer**, TRACE32 also supports sample-based profiling through the **PERF** command group. This provides functionality comparable to **SNOOPer**, but with alternative analysis views. For further details, refer to the General Commands Reference Guide P.

Summary

This article has compared several approaches for run-time analysis when off-chip or on-chip trace is not available:

- LOGGER trace provides good results for larger functions and tasks but is limited by buffer size and introduces overhead that distorts very short executions.
- FDX trace extends recording time by streaming data to the host, but stalls may occur if the trace channel is saturated.
- Sample-based profiling (SNOOPer) offers a lightweight alternative when instrumentation is not feasible, delivering a broad overview of runtimes but with reduced accuracy, especially for peak values.
- BenchMark Counters (BMC) can be used on some processors to measure run-times without instrumentation overhead.

In practice, the choice of method depends on the target system's capabilities and the required level of accuracy. LOGGER and FDX are suitable when precise timing is essential, while sample-based profiling provides a useful fallback for high-level performance insights when instrumentation is not feasible.

Related Documentation & Files

For further details, please refer to the following documents:

- Application Note for the LOGGER Trace
- Application Note for FDX

- Application Note for the SNOOPer Trace
- Application Note Profiling on AUTOSAR CP with ARTI

Additionally, the demos used in this article are provided as attachments.

Attachments

- demos-alternative-trace-solutions.zip [142.64 KB]