INSTRUMENTATION TRACE

Andrea Martin, Ingo Rohloff

Lauterbach GmbH

ABSTRACT

More and more chips, mainly for the mobile industry, provide *Instrumentation Traces* for output of software generated information on chip-internal events.

This paper explains how this information is currently processed by a trace port analyzer and how this processing could be optimized if standards would exist for transport protocol and information formats.

1. INTRODUCTION

Most developers are familiar with test scenarios where a simple printf() is the most efficient implementation. A typical example is the output of diagnostic information i.e., test scenarios where the application is already working as intended, and now operational tests have to be conducted. Usually, code instrumentation is used for these tests.

Commonly code instrumentation generates output via RS-232 or Ethernet. Today chips provide dedicated *Instrumentation Traces* for this purpose.

2. OPERATION OF INSTRUMENTATION TRACE

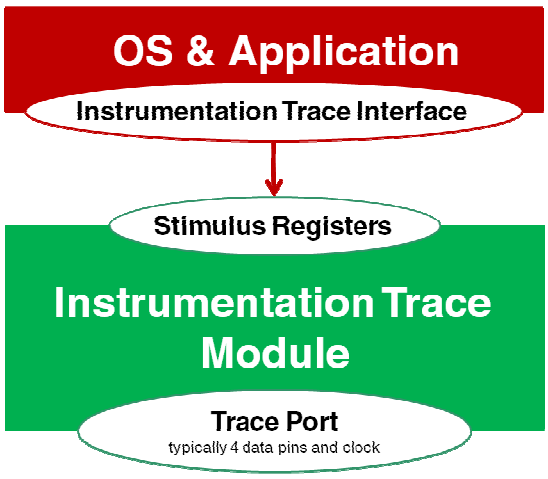

Figure 1: Instrumentation Trace with dedicated off-chip Trace Port

An *Instrumentation Trace Module* operates like this: Whenever data is written to one of the associated stimulus registers, trace packets are generated and emitted at the trace port.

Figure 1 illustrates the operation of an *Instrumentation Trace* with dedicated off-chip trace port. Other implementation options are e.g. a shared trace port or an on-chip trace buffer.

In order to use the *Instrumentation Trace Module*, the OS and the application software need an *Instrumentation Trace Interface* that converts their printf() information into a sequence of write accesses to the stimulus registers. Ideally this is done with only small overhead.

Apart from classical printf(), a variety of other information (PAYLOAD) can be emitted via an *Instrumentation Trace*:

- Value information (like variable values)
- Application events (e.g. function entries and exists)
- Target OS events (like task switches)
- System states (e.g. power saving modes)
- System event counters (e.g. to count interrupts)

For our discussion we assume the payload is wrapped by a message-based transport protocol.

3. CURRENT SOLUTIONS FOR DISPLAY AND ANALYSIS

Currently no standards for payload and message formats exist i.e. the *Instrumentation Trace Interface* is always user-specific. A *Trace Port Analyzer* which records the emitted trace packets can reconstruct the stream of stimulus register accesses. The payload itself however can't be reconstructed without detailed knowledge of the user-specific implementation of the *Instrumentation Trace Interface*.

To provide this knowledge to their trace port analyzer CombiProbe (128-MB of trace memory, bandwidth of 200 MBit/s per trace pin) Lauterbach offers an open API. This API makes it possible to load a user-implemented *Custom Trace DLL* to the *Trace Port Analyzer Software*, which has to perform the following tasks:

1. Reassemble messages from the stimulus register accesses.
2. Extract payload from messages.

3.a Prepare the payload for analysis and display within the trace port analyzer GUI.

3.b Pass the payload to external software for analysis and display.

Instrumentation tracing with CombiProbe can be performed in two operation modes: conventional tracing or real-time streaming.

Conventional tracing is done in two steps:

1. The trace packets emitted by the *Instrumentation Trace* are sampled at the trace port, reassembled to stimulus register accesses and placed in trace memory.
2. After sampling is stopped, the stimulus register accesses are transferred to the host. A *Custom Trace DLL* extracts the payload from the stimulus register accesses and performs its analysis and display within the trace port analyzer GUI.

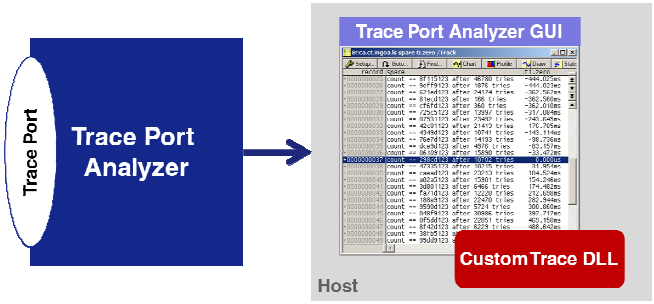

Figure 2: A *Custom Trace DLL* performs the task to analyze and display the payload within the trace port analyzer GUI when conventional tracing is used.

Real-time streaming, in contrast to conventional tracing, concurrently samples and analyzes the data: The trace packets emitted by the *Instrumentation Trace* are sampled at the trace port, reassembled to stimulus register accesses and placed in trace memory. From there they are directly transferred to the host. A *Custom Trace DLL* extracts the payload and passes it e.g. via a named pipe to an external *Custom Trace Analysis and Display Software*.

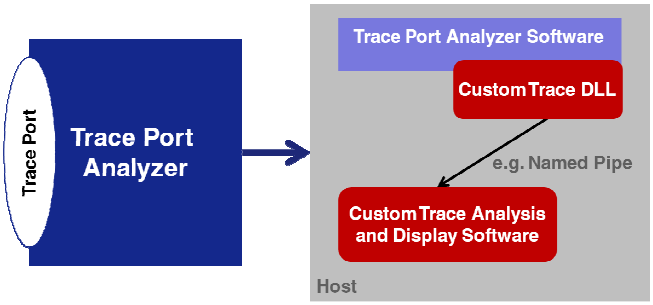

Figure 3: The *Custom Trace DLL* extracts the payload and passes it to the *Custom Trace Analysis and Display Software* when the trace port analyzer works in real-time streaming mode.

Each operation mode has its specific strengths.

Conventional Tracing:

- Can cope with high loads on the trace port.
- Provides a reasonable amount of trace information.
- Allows effective troubleshooting by examine the local history of an error condition.

Real-time Streaming:

- Trace information can be inspected while recording/analyzing.
- If trace data is stored to file (long-time trace) trace memory size is limited only by mass storage device.
- Unlimited sampling time if trace data is analyzed and displayed (real-time profiling) and then discarded.

The current solution requires that the user has to design and implement his own *Instrumentation Trace Interface* and *Custom Trace DLL*. This is likely a major hurdle to begin with instrumentation tracing. Standards for payload and transport protocol would make the usage of *Instrumentation Traces* much simpler and would likely make it more popular.

4. PROPOSAL FOR A PAYLOAD STANDARD

As a starting point for a standardization discussion, Lauterbach would like to suggest two exemplary payload formats.

Task Information Messages

Task Information Messages make a task run-time analysis or, to be more precise, a task state analysis possible. The presented message format is based on the assumption that tasks are dynamically created and ended by the target OS (e.g. Linux).

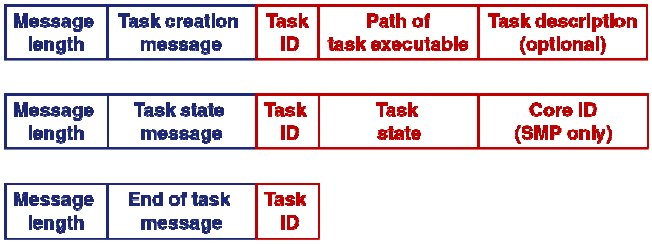

Figure 4: Three types of messages are required for task state analysis.

1. **Task Creation Messages** are emitted whenever a new task is created. For conventional tracing, with a cyclic trace buffer of limited size, they can also be used as synchronization messages. Synchronization messages are sent periodically to guarantee that full information for all existing tasks is always in the trace buffer.

2. **Task State Messages** are emitted whenever the state of a task is changed. Examples for task states are: running, ready, waiting, suspended.

3. **End of Task Messages** are emitted whenever a task ends or is terminated otherwise.

Instruction Watchpoint Messages

Instruction Watchpoint Messages are emitted whenever selected high-level language code lines are executed. They can be used, e.g. to analyze how frequently various interrupt handlers have been started. The *Watchpoint Group* provides a mechanism to add structure to the analysis of *Instruction Watchpoint Messages*. They enable e.g. the analysis software to analyze the *Instruction Watchpoint Messages* by groups.

Figure 5: The format of *Instruction Watchpoint Messages*.

If an *Instruction Watchpoint Database* is generated and added, the *Trace Analysis and Display Software* on the host can display more verbose information on the source and the meaning of a watchpoints.

An *Instruction Watchpoint Database* could contain this information:

- Source file name and source line number for the executed instruction.
- Attributes, e.g. a string that describes the meaning of the watchpoint more clearly.

...				
Watchpoint group	Watchpoint ID	Source file	Source line	Attributes
...				

Figure 6: Format example for the information included in the *Instruction Watchpoint Database*.

The major advantage of the *Watchpoint Message* approach is its bandwidth efficiency; Meta data is stored in the *Watchpoint Database* on the host and doesn't need to be emitted by the target.

5. CONCLUSION

Standards for common use cases would certainly encourage many users to start with instrumentation tracing. They would also allow preconfigured trace tools and third-party software for analysis and display.

6. REFERENCES

[1] About the Instrumentation Trace Macrocell, http://infocenter.arm.com

[2] Ingo Rohloff, Presentation "TRACE32 and CoreSight for Cortex-M3", May 2010