

# Understanding Embedded Trace

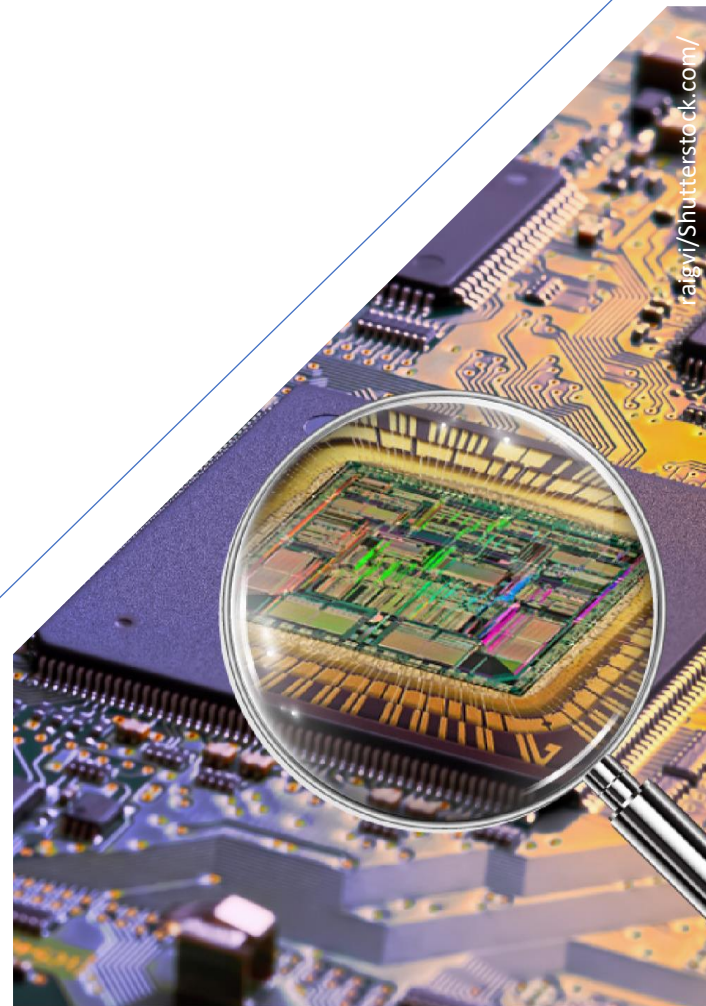
---

## WHITEPAPER

Rev. 1.0 / SEP 2020

Embedded Trace is an integral part of nearly all modern processors. This whitepaper summarizes the **essential facts** about this powerful but still far too seldom used functional unit that **application engineers, test engineers and project managers should know** in order to test, optimize and debug a system efficiently.

This paper briefly discusses the problems associated with software-based instrumentation and how non-intrusive electronic probing was defeated by advancing computer architecture and system integration. We then identify embedded trace as the solution to this observability conundrum laying out the techniques that enable efficient and economically reasonable implementations for this innovative technology. We describe the challenges in bandwidth and volume that are faced by hopeful observers and backend applications highlighting the benefits of modern innovative online analysis capabilities. Finally, we provide a short overview over common physical trace interfaces and simple guidelines that ensure that your next system design is capable of leveraging this cutting-edge technology.



## Something about Embedded Processor Observation

Monitoring embedded processors can be a difficult task. However, it is essential both during the development process and during deployment:

- Functional Testing: e.g., measure and validate timing behavior;
- Structural Testing: e.g., measure code coverage or data and control flow coupling;
- Debugging: pin down the root cause of anomalies.

## Software Instrumentation

One obvious method of observation, which has been in use for a long time, is software instrumentation. The application software is modified to produce the desired information, such as control flow indicators, via standard output channels. Unfortunately, this approach can have a massive impact on the timing behavior and the memory footprint of an application.

In the field of embedded systems, this method has serious disadvantages, including:

- Due to the changed timing behavior, the significance of functional tests is very limited.
- Unintended phantom synchronization may be introduced to concurrent programs by the access arbitration to shared output channels.
- Safety-critical software already deployed in the field cannot simply be modified to produce debug information as this would risk functional degradation or the triggering of Heisenbugs<sup>1</sup>.
- The common mitigation of observability constraints by first performing transparent coverage validation tests with software instrumentation before repeating them without software instrumentation leads to a doubled test effort. This is a major disadvantage, especially, when limited and expensive test resources, such as HIL test benches, are involved.

Even though the software instrumentation approach is archaic, it is still widely used.

The Embedded Trace presented on the next pages often also offers functions for hardware-supported software instrumentation. These will be presented in detail in one of our following whitepapers.

---

<sup>1</sup> Anomalies that seem to disappear or alter their behavior when looked into are called Heisenbugs (named after the uncertainty principle described by the physicist Werner Heisenberg, which is often informally conflated with the probe effect).

# Embedded Trace

## MOTIVATION

In the olden days, processors were slow (some 10 MHz) and only had external program memory. Eavesdropping on the memory bus was sufficient to observe the memory addresses of the instructions fetched by the CPU for execution. Following the program flow was, thus, rather trivial.

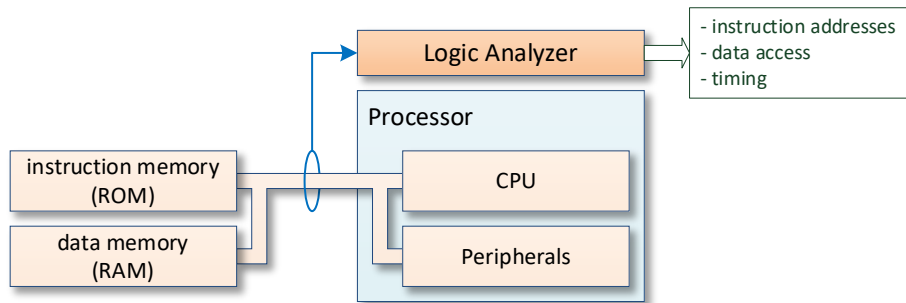


Figure 1: The olden days - all CPU activity is visible on the external bus.

CPUs have been becoming faster, and their processor architecture has been changing:

- Instructions and data are cached near the CPU. Individual accesses are, therefore, no longer visible on the external bus system.
- Particularly embedded devices integrate more and more memory (RAM or Flash) internally so as to achieve faster access times and lower the system cost. Accesses to such memory are not at all observable on the external bus system.
- To meet the demand for more and more integrated computing power and to reduce energy consumption, several CPUs are integrated into one processor. Whatever external effects remain observable now also pose the challenge of attributing them to the correct concurrent control flow among the ones scheduled across truly parallel cores.

All these advances in processor architecture require a radically new approach to observing what is happening inside the processor: **embedded trace**.

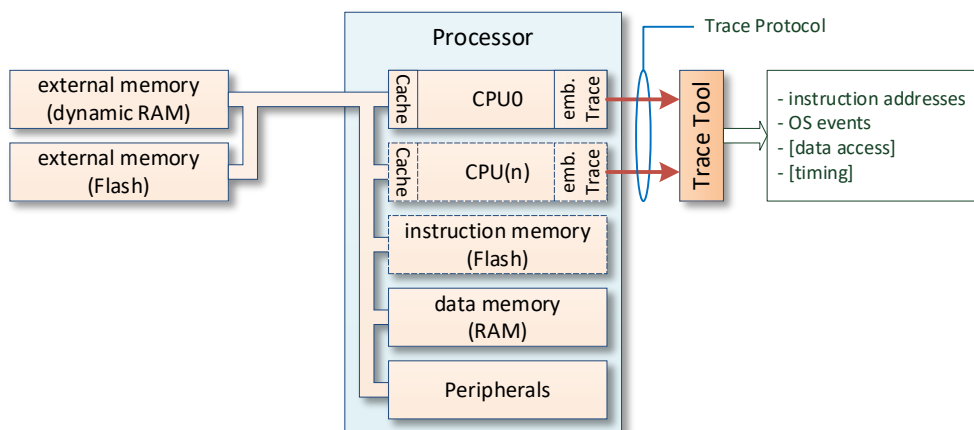


Figure 2: State-of-the-Art: Embedded Trace

Embedded trace is the integration of functional units that make the activity of the CPUs observable. However, there is a significant bandwidth problem: Monitoring a single CPU comprehensively, at least, requires information about the executed instructions and the changing CPU registers. For a 32-bit CPU, a naive encoding would amount to a bandwidth of roughly:

$$[\text{instruction address}] + [\text{changing CPU registers and flags}] = 32 + \sim 48 = \sim 80 \text{ bits / CPU clock cycle}$$

If the CPU is now clocked at 1 GHz, this will imply 80 Gbit/s of trace data. This is far beyond an economically reasonable solution. So, the trace data stream must be compressed.

## PROTOCOL OVERVIEW

Actual trace implementations such as Arm® CoreSight™ Program Flow Trace™ [1], Arm® CoreSight™ Embedded Trace Macrocell ETMv4.x [2], Intel® Processor Trace [3], and Nexus 5001 Forum™ [4] must compress the transmitted control flow trace rigorously. They inject relevant OS-related information, such as context switches, into their output to facilitate a highly efficient context-aware compression.

The system observability can be boosted by also tracing the data flow. However, the implied trace data is much harder to compress. This results in significantly higher bandwidth requirements and, hence, more costly trace interfaces. Therefore, most implementations actually refrain from implementing this capability.

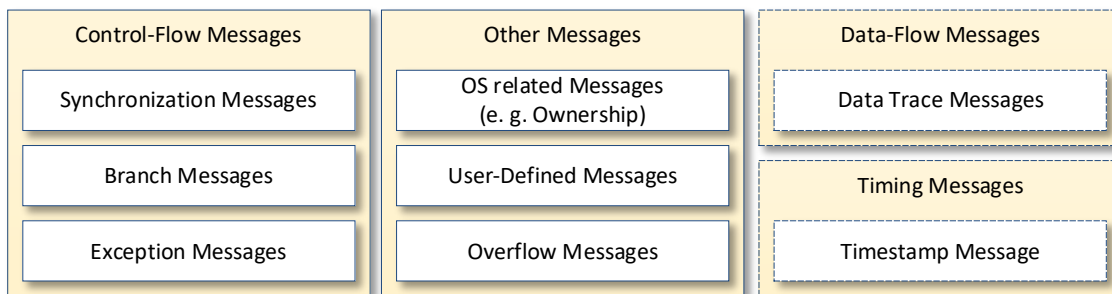


Figure 3: Most relevant trace message types

In the following, the most relevant trace message types are briefly introduced.

### Control-Flow Messages

As already mentioned, the continuous output of the program counter alone would consume significant trace bandwidth. This is overcome (a) by assuming that the executed application is known to the observer and (b) by exploiting the default sequential execution of instructions therein. This allows to use the following strategy for trace data compression:

**Synchronization messages** are generated in greater intervals, e.g., every 1000 messages. Only they establish a concrete value of the program counter that identifies the reference point for the further trace data interpretation. The execution of the sequential code following this point is implied.

**Branch messages** communicate actual control flow decisions. A single bit indicates whether or not a conditional branch instruction has been taken to leave the sequential execution path. Branches not taken imply the sequential continuation of the execution. They do not require any further trace data. Neither do taken *direct* branches as their fixed continuation target can be inferred from the executed application

binary. Only taken *indirect* branches trigger the generation of an alternate message that enables the observer to reconstruct the dynamically computed branch target address.

Unconditional branches are handled differently by the embedded trace architectures. Some choose to establish posts in the control flow and produce execution bits just as they do for executed conditional direct branches (e. g. Arm® CoreSight™ Program Flow Trace™ [1]). Others leave out this clearly implied control flow from the emitted trace altogether (e. g. Intel® Processor Trace [3]).

**Exception messages** are generated for externally induced control flow diversions such as by interrupts. They typically provide a hint on the nature of the exception and contain all information necessary to resume the control flow reconstruction.

This highly efficient compression results in an average of significantly less than one bit of trace data per executed instruction.

### Timing Messages

Depending on the embedded trace architecture, different message types conveying timing information may be available. In addition to wall-clock messages, **cycle-count messages** are of special importance. They indicate how many CPU clock cycles have elapsed since the last timing update. Observers can typically choose to receive cycle-count messages with each branch message, in programmable time intervals or not at all. This way, the significant trace bandwidth consumption by high-frequency cycle-count messages can be balanced against other desired trace quality properties.

### Data-Flow Messages

Data trace is difficult to compress. Depending on the trace architecture, the address of a data access, the transmitted value and the type of access can be communicated. In a 32-bit system, each access can result in a trace message of more than 60 bits in length.

Due to its high bandwidth requirements, data trace is not available in all architectures and otherwise limited regularly. E.g., it may be constrained to designated address regions or to producing partial information such as the addresses of write accesses only.

### Other Messages

There are a number of other trace message types. They establish the trace context as by allowing the OS to communicate context switches and convey trace-specific information as for signaling internal trace buffer overflows.

# Trace Data Processing

## OFFLINE CAPTURE

There are three main options for processing trace data (Figure 4 to Figure 6).

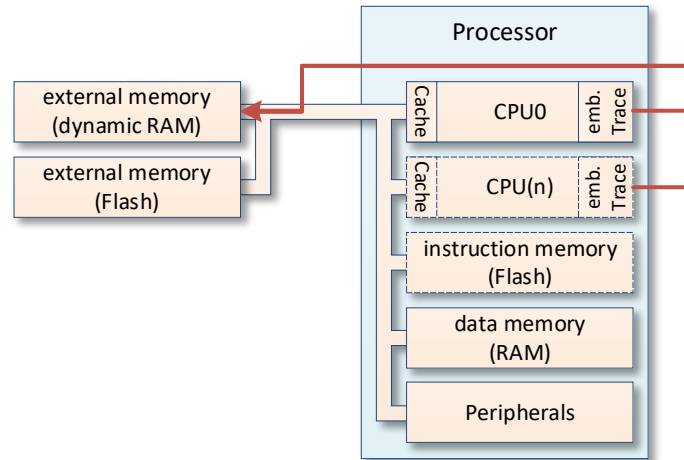


Figure 4: Buffering trace data within the processor system

Figure 4 shows a solution, which captures the trace data stream in system memory. This solution is often used in the desktop environment. However, it has a significant impact on system behavior and allows only short-term observations limited by memory capacity.

To prevent behavioral feedback to the system under observation, trace data can also be captured by an external trace tool via a designated trace interface. Traditionally, this trace tool is essentially a large memory buffer that collects the trace data for their later offline processing on a PC (Figure 5).

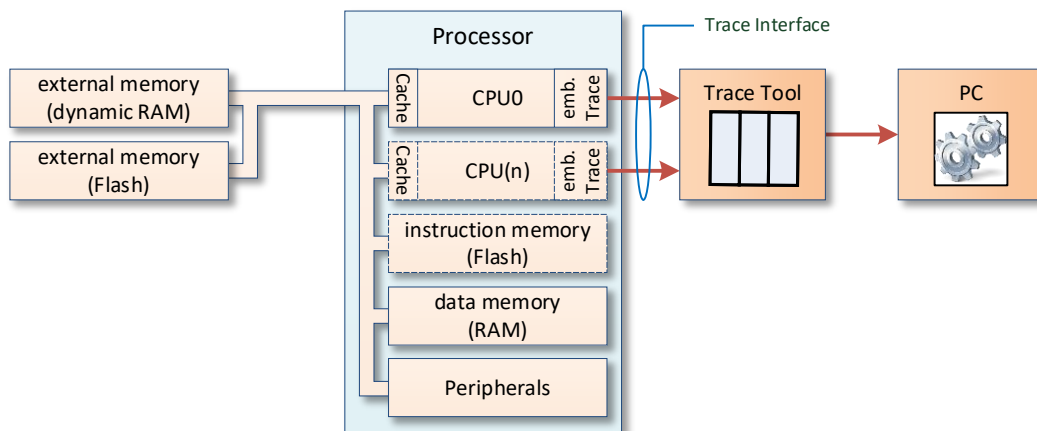


Figure 5: Buffering trace data in trace tool, offline processing in PC

The approach of buffering the trace data (Figure 4 and Figure 5) has two decisive disadvantages:

- (1) The observation time is always strictly limited by the buffer size. Depending on the trace bandwidth, this may allow trace snapshots of a few milliseconds or, at most, seconds. The analyses of long-running integration and system tests, the statistically significant measurement of worst-case execution times, or the search for complex non-deterministic errors cannot be scaled to longer time frames and are, thus, only possible to very a limited extent.
- (2) The later offline processing of the recorded trace data leads to a long latency between the observation and the availability of results. On the one hand, this is an inconvenience for the engineers involved. On the other hand, this precludes any innovative exploitation of the gained system observability that would require a prompt reaction.

### ONLINE ANALYSIS

Our latest innovations now enable the live processing of processor trace data. A large buffer memory decoupling the downstream trace processing is no longer necessary (Figure 7).

For their processing on the fly, the highly compressed trace data stream must be decompressed and the control flow executed by the CPU(s) must be reconstructed. This demanding computation must often cope with the execution traces from multiple fast CPUs that are running at nominal clock speeds above 1 GHz. This decoding may be further challenged by additional abstractions and indirections introduced by the used operating system.

The reconstructed control flow must be analyzed into an apt event stream abstraction that is suitable to drive the desired of various possible backend tasks. For example, (a) branch information for a coverage analysis may be recorded or (b) dynamic properties over the event stream may be computed and validated against a temporal logic specification.

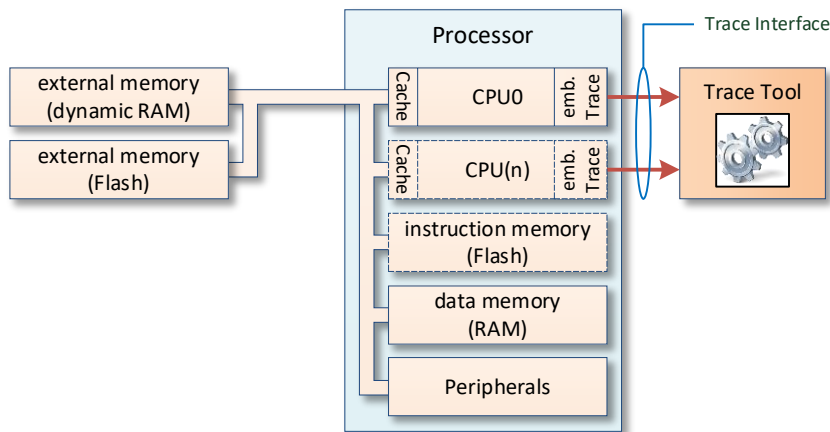


Figure 6: Trace data online processing by trace tool



Figure 7: [CEDARtools](#)® trace data online processing tool with high-speed serial trace connection to an Infineon AURIX™ processor

A comparison between the embedded trace monitoring techniques is given in Table 1. It is obvious that the fundamental ability to observe a system for an arbitrarily long timespan, combined with the extremely short latency of result availability, leads to a new quality of non-intrusive observability of processors.


	Buffering trace data within the processor system (Figure 4)	Buffering trace data in trace tool, offline processing in PC (Figure 5)	Trace data online processing by trace tool (Figure 6) 
Intrusiveness	High	Low	Low
Observation period	Limited by buffer size	Limited by buffer size	Arbitrary long
Latency for accessing observation results	High	High	Low
Trigger conditions	Limited complexity	Limited complexity	Complex (high-level language support)

Table 1: Comparison of embedded trace techniques (green: good).



## Physical Trace Interfaces

Trace data can be output either via standard interfaces (e. g. PCIe) or via dedicated interfaces. Several industry standards for dedicated trace interfaces exist.

	Parallel	high-speed serial
Signals	clock, strobe, data[]	differential TX line(s), automatic clock recovery
Bandwidth	~ 100 Mbit per data line	some Gbit/transceiver pair
Data Link Layer (OSI layer 2)	none	yes (with CRC)
Example	Nexus parallel [4] MIPI PTI <sup>SM</sup> [5]	Nexus Aurora [4] Arm CoreSight HSSTP [6] MIPI HTI <sup>SM</sup> [7]

Table 2: Comparison of parallel and high-speed serial trace interfaces.  
Short conclusion: Prefer the high-speed serial interface whenever possible.

## Q&A

**Q:** Does a JTAG interface on my system provide the trace-like functionality?

**A:** Unfortunately, not. The bandwidth of JTAG (some 10 Mbps) is orders of magnitude lower than the bandwidth required for embedded trace output (some Gbps).

**Q:** Should I really consider that faults still occur after the release of our product - despite the experience of our developers and extensive testing?

**A:** The statistics clearly say: YES. Even in a satellite software developed and extensively tested by NASA engineers, 520 faults were still found after the release [8]. The more code, the disproportionately more defects are initially present in the software and the worse is the efficiency of error correction. With the increasing use of multicore and networked multiprocessor systems as well as third party software components, more and more non-deterministic error patterns, which are difficult to reproduce, are added. Assuming an error correction ratio of about 95% during the development process (statistic is based on 12.000 software projects evaluated in [9]), a proud 5% of mistakes remain in the release code - in absolute numbers, this can mean several thousand defects.

**Q:** Where can I find support for planning and using Embedded Trace?

**A:** Our experts at Accemic Technologies are pleased to support you in the planning and concrete integration of embedded trace. We have close contact to the relevant processor suppliers, access to comprehensive documentation and a great deal of experience in integration.

We are an email away: [cedartools@accemic.com](mailto:cedartools@accemic.com)

## Summary

Embedded Trace, an integral part of almost all modern processors, is the key component for the non-intrusive and continuous monitoring of processors, especially in safety-critical embedded systems.

Embedded Trace enables instruction-accurate control flow reconstruction and non-intrusive monitoring of operating system activity. Optionally, exact timing information in nanoseconds resolution can be extracted and data accesses can be observed. Therefore, Embedded Trace is crucial for testing, performance optimization and efficient debugging of embedded systems.

For a responsible project planning, it is essential to ensure the accessibility of the trace interfaces already during the creation of the requirements specification.

## References

- [1] *ARM IHI 0035B - CoreSight™ Program Flow Trace™ PFTv1.0 and PFTv1.1 Architecture Specification*. ARM Limited, 2011.
- [2] *ARM IHI 0064D - ARM® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.2*. ARM Limited, 2016.
- [3] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2016.
- [4] IEEE-ISTO, 'The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface', *IEEE-ISTO 5001™-2012*, Jun. 2012.
- [5] 'MIPI PTI v2.0 Specification for Parallel Trace Interface'. MIPI Alliance, Inc., May 03, 2011.
- [6] *ARM PR106-PRDC-006159 - High Speed Serial Trace Port Architecture Specification*. ARM Limited, 2012.
- [7] 'MIPI HTI Specification (High-speed Trace Interface) Specification Version 1.0, Errata 01'. MIPI Alliance, Inc., Feb. 11, 2020.
- [8] M. Grottko, A. P. Nikora, and K. S. Trivedi, 'An empirical investigation of fault types in space mission system software', in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 447–456, doi: 10.1109/DSN.2010.5544284.
- [9] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley, 2011.

## About Accemic Technologies

Accemic Technologies, a German company with offices in Kiefersfelden (Munich Metropolitan Region) and Dresden (one of the most beautiful cities in the world), has developed [CEDARtools®](#) – a patented breakthrough technology for the dynamic analysis of dependable embedded systems.

We make software tests for embedded systems more effective and efficient. We simplify the debugging process and provide the necessary leverage to pin down the root causes of sporadic, non-deterministic anomalies.

Our new analysis method leverages the trace capabilities embedded into virtually all modern processors. Their trace units expose the details of the operation of the CPU and its peripherals to the outside. However, they easily produce a few GBit of trace data per second. This quickly renders approaches combining storage and offline analysis as infeasible options.

The live analysis of the execution trace at run time is a quantum leap enhancement over the offline analysis of recorded trace data as it effectively eliminates the bottlenecks imposed by the need for the intermediate buffering. CEDARtools® enables (a) the measurement of the control flow coverage during the execution of integration tests and system tests, as well as (b) the dynamic constraints monitoring.

We provide development and test engineers with the powerful tool that boosts their productivity by enabling them to monitor a system over large time frames and to pin down even sporadic errors quickly.

[Talk to one of our experts today.](#)

### **Accemic Technologies GmbH**

Franz-Huber-Str. 39  
83088 Kiefersfelden  
Germany

+49 8033 6039790

[cedartools@accemic.com](mailto:cedartools@accemic.com)

[www.accemic.com](http://www.accemic.com)

All rights reserved. Accemic Technologies and CEDARtools® are trademarks or registered trademarks of Accemic Technologies. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.