RAPITA Systems
A DANLAW Company

# Requirements for zero-footprint R**VS** analysis

## Introduction

The zero-footprint R**VS** tools Rapi**Cover**$^{Zero}$, Rapi**Time**$^{Zero}$ and Rapi**Task**$^{Zero}$ produce coverage, timing and scheduling information for uninstrumented code running on target devices.

By automatically producing results from branch traces produced by the board on which software executes, these tools reduce the effort needed to verify code coverage and timing and scheduling behavior. Along with branch traces produced by the system hardware, they only need a disassembly of the application, meaning there is no need for source code or instrumentation. In addition to reducing target overheads, this lets you analyze third-party libraries or other pieces of software for which you have no access to source code.

This document explores requirements for compatibility with zero-footprint tools, including requirements from the board on which software executes, from branch traces by those boards, and from data capture etc. This information can be used to assess specific software execution environments to see if they are likely to be compatible with Rapita's zero-footprint tools.

As an approximate minimum (you may be able to miss one of these if you have other hardware doing a similar role):

- The target processor must have the ability to produce execution or branch traces.
- The target board must be able to capture the trace from the processor without gaps (either real-time or with the ability to pause the processor whilst extraction happens).
- The board must have a connection to attach debugger hardware or an **RTB**x datalogger, and the debugger or **RTB**x must be able to get the trace off-board without gaps.
- Either the code under test must run without threads, or the RTOS managing threads must expose enough information about cores, process and thread IDs that they can be identified in the trace. It may be necessary to patch the RTOS to get this information.

## Platform Support Packages

Rapita Systems Platform Support Packages (PSP) convert native branch traces into traces understood by zero-footprint R**VS** tools (*standard traces*).

There are many different execution environments for code under test, both native on-hardware, and simulated in software, and these produce traces of different formats, which contain different events with varying levels of detail. Each PSP is designed to convert a single specific style of native trace to a standard trace format that can be analyzed by zero-footprint R**VS** tools.

## What must standard traces include and how are they generated?

Standard traces produced by PSPs must include:

- A single branch trace per thread in the executing code
- Information and timestamps for the following events for each branch trace:
  - Thread activated
  - Branch taken from address X to address Y
  - Branch not taken at address X
  - Thread suspended

PSPs convert native traces into standard traces understood by zero-footprint R**VS** tools in 4 stages (Figure 1):

1. First, the branch trace produced by the hardware or simulator (native trace) is captured and saved to a file.
2. Second, this trace is parsed.
3. Next, if necessary, certain elements in the trace are expanded using information taken from the disassembly of the executable (decompression of the trace).
4. Finally, the trace is split into separate traces per executing thread (demultiplexing of the trace).

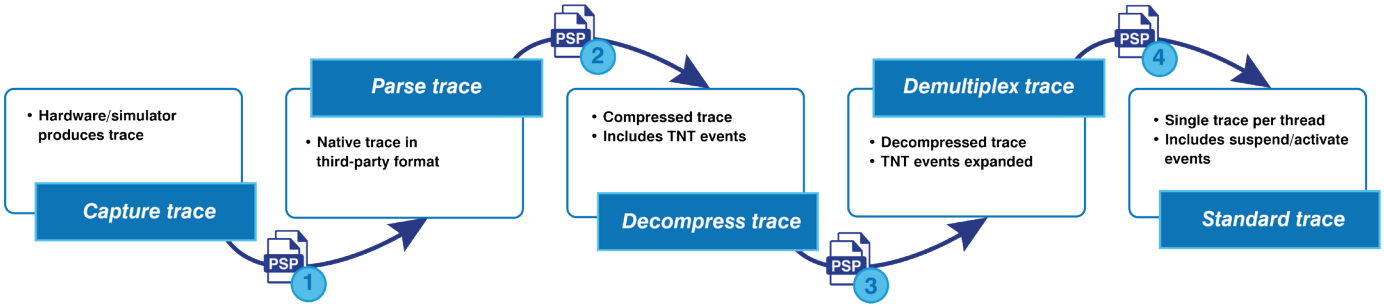


**Figure 1.** The process by which native traces are transformed into branch traces usable for analysis by zero-footprint R**VS** tools

## Requirements from the board

For zero-footprint analysis to be possible, the board on which software is executing must be capable and configured to produce a branch trace.

Examples of boards capable of producing branch traces include:

- Boards that have been developed to meet at least Class 2 of the Nexus™ message-based trace protocol
- ARM® boards including an ARM ETM component
- Custom boards developed to produce branch traces

## Requirements from native branch traces

For it to be possible to create a PSP that can convert a native branch trace into a standard trace analyzable by zero-footprint tools, the branch trace must contain information from which a standard trace can be generated. For a list of the elements standard traces need to have, see *What must standard traces include and how are they generated?*

Things to look out for when assessing hardware, which increase the likelihood that a PSP can be developed, include:

- The hardware produces a log of the branches in the executable code that were taken and not taken during execution
- This log includes origin and destination addresses through which the execution occurred

To investigate whether branch traces produced by a specific hardware or simulator environment are compatible with zero-footprint R**VS** tools, contact us at support@rapitasystems.com.

## Requirements when capturing traces

If the native trace originates from hardware, then it will need to be extracted from that hardware. This may be achieved using external devices including the following:

- A debugger – if a debugger is used to collect traces for analysis, your hardware must include a trace port that can connect to the debugger. Manufacturers that produce debuggers compatible with zero-footprint tracing include Lauterbach™ (PowerDebug), ARM (DSTREAM), iSYSTEM® (IC series) and Kyoto Microcomputer Co., Ltd. (PARTNER-Jet2). Other debuggers may be compatible with zero-footprint tracing. The requirements for connection depend on the hardware you are using. For example, if you're using an ARM ETM interface, you could connect a Lauterbach debugger to it in any of the following ways:
  - By using a 20-Pin Cortex® Debug and an ETM Connector
  - By using a 38-Pin ARM ETM Mictor Connector
  - By using a Samtec® 40-Pin Connector
  - By using a MIPI 34 or MIPI 60 Connector

  For more information on the requirements to connect a debugger to the board you are using, contact the debugger manufacturer. For example, you can view requirements for connection to Lauterbach debuggers on the Lauterbach website's chip selection page.
- A Rapita Systems **RTB**x data logger – **RTB**x data loggers are capable of capturing branch traces. If this strategy is used, however, the board must be configured to send information to a port that the **RTB**x can read from. This may be achieved, for example, by programming a free FPGA on the board.

Regardless of which external device is being used to capture traces, the device being used must be able to capture the data from the target fast enough so that there are no gaps or buffer overflows during data capture. If there are overflows, zero-footprint R**VS** tools will attempt to make sense of the remaining trace, but this may not be possible if information on which branches have been taken or not taken is missing from the trace. As the **RTB**x has a high bandwidth, data capture rate tends not to be an issue when using an **RTB**x.

For simulated execution, the simulator will usually be modified to output the native trace directly to a file. Because the trace is written directly to a file, there are no issues with bandwidth.

## Requirements for parsing

For Rapita Systems to be able to produce a PSP for new hardware, the trace format produced by the hardware must be documented, so that Rapita can develop a PSP that converts it to the standard trace format.

## Requirements for decompression

Because software execution tracing mechanisms must be fast and not miss information, they often use optimizations and provide only minimal details of what has happened. The decompression stage expands on the information available from the native trace by combining it with other known information such as a disassembly of the executable. It must be possible to expand on the information present in the native trace to produce a standard trace containing the following information:

- Thread activated
- Branch taken from address X to address Y
- Branch not taken at address X
- Thread suspended

For example, some platforms compress sequences of "Branch taken" and "Branch not taken" events into a single bit per decision, omitting the addresses involved. PSPs must be able to expand these "TNT" events to add the origin address (for branch not taken) or origin and target addresses (for branch taken). PSPs must be able to unambiguously determine these addresses, which is done for example by nearby addresses in the trace and information from a disassembly of the executable. As such, if decompression is required, then the addresses surrounding the compressed TNT events in the trace must correspond with the expected addresses from the assembly.

## Requirements for demultiplexing

For zero-footprint R**VS** tools to be able to analyze a multi-threaded application, there must be a way of detecting the context switch from one task to another so that the trace can be successfully demultiplexed.

For this to be possible, the native trace must contain enough information about interrupts, exceptions and context switches between threads and the RTOS kernel code so that the PSP can determine when each context switch occurs and which thread has become active.

There are various ways to detect context switches depending on the execution environment and information available, so different PSPs often determine context switches in different ways. Some hardware has a register that contains the currently running thread ID, so PSPs for these environments scan this register for demultiplexing. Sometimes, the context switch routine is easily identifiable, and PSPs for these environments may read unknown jumps from the context switch routine as a context switch.

If there is not quite enough information to be sure of the switch, or of the identity of the new task, then it may be possible to patch the RTOS to output the information required.

## Requirements of the executable file and disassembler

For zero-footprint R**VS** tools to be able to analyze the executable file, a disassembler must be available to disassemble the object code into a format that is analyzable by zero-footprint tools. PSPs generally use some variant of nm, objdump or similar software to disassemble object code.