# CoreSight trace decode metadata requirements

**SDD Business Unit**

**APD Group**

Document number:
Date of Issue:
Author: Al Grant
Authorised by:




## Abstract

Captured CoreSight trace can generally not be decoded without additional information. This document describes the additional information and suggests how it can be packaged for conveying trace from a target to a trace decoding tool.



## Keywords

CoreSight, trace, ETM, STM


## Distribution list

| Name | Function | Name | Function |
|---|---|---|---|

# Contents

# 1 ABOUT THIS DOCUMENT

## 1.1 Change control

### 1.1.1 Current status and anticipated changes

### 1.1.2 Change history

| Issue | Date | By | Change |
|---|---|---|---|
| A01 | 2/6/13 | AG | First draft |
| A02 | 8/10/13 | AG | Concrete metadata format; updated userspace trace decode |
| A03 | 4/9/14 | AG | Mostly editorial changes for external release |

## 1.2 References

This document refers to the following documents.

| Ref | Doc No | Author(s) | Title |
|---|---|---|---|
| [CS] | ARM IHI 0029B | ARM | CoreSight v1.0 Architecture |
| [ETMv3] | ARM IHI 0014Q | ARM | Embedded Trace Macrocell Architecture ETMv1.0 to ETMv3.5 |
| [ETMv4] | ARM IHI 0064B | ARM | Embedded Trace Macrocell Architecture ETMv4 |
| [ITM] | ARM DDI 0314H | ARM | CoreSight Components TRM section 12.1.1 |
| [PFT] | ARM IHI 0035B | ARM | CoreSight PFT v1.1 Architecture |
| [RVT] | | ARM | RVT trace file specification |
| [STP] | | MIPI | STP Specification v2 |

## 1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

| Term | Meaning |
|---|---|
| ASLR | Address space layout randomization (see Wikipedia) |
| ATB | CoreSight trace bus (see [CS]) |
| CCN | ARM Cache Coherent Network interconnect |
| ELF | Executable and Linking Format, standard format for program image files |
| ETB | ARM CoreSight Embedded Trace Buffer |
| ETM | ARM CoreSight Embedded Trace Macrocell, see [ETMv3] or [ETMv4] |

| | |
|---|---|
| ITM | ARM Instrumentation Trace Macrocell |
| PFT | Program Flow Trace, the format used by PTM |
| PTM | ARM CoreSight Processor Trace Macrocell (essentially the version of processor trace that followed ETMv3.5 and preceded ETMv4) |
| SoC | System-on-chip |
| STM | ARM CoreSight System Trace Macrocell |
| STP | MIPI System Trace Protocol v2, see [STP] |
| TPIU | ARM CoreSight Trace Port Interface Unit |

# 2 SCOPE

This document gives an overview of ARM's CoreSight on-chip trace solution, and describes the out-of-band metadata needed to successfully demultiplex, decode and decompress CoreSight trace. It pays special attention to trace of Linux systems, including trace of userspace processes. It does not cover in detail how the trace sources are configured or how the trace data is captured or used.

The document currently assumes the target runs a single OS such as Linux. Virtualization is not covered in this document.

# 3 INTRODUCTION

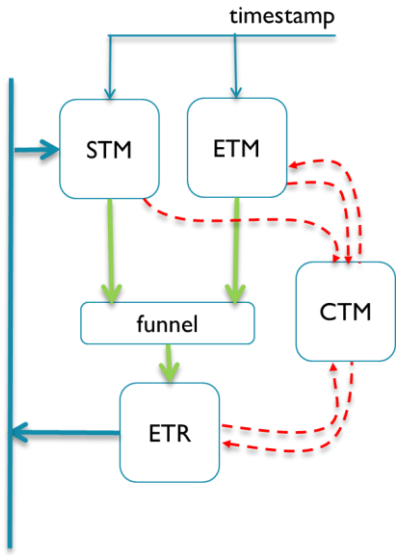
CoreSight is ARM's on-chip debug/trace solution. It is both an architecture (specifications of various register-level and bus interfaces and data formats) and a set of on-chip peripherals implementing that architecture, licensed as Verilog source to silicon partners for integration into their SoCs. CoreSight provides debug, trace and triggering features, but in this document we'll be concerned with trace.

CoreSight trace is generally non-invasive. IP blocks on the SoC produce trace data in various stream formats, and this is captured and routed on dedicated ATB trace buses. Trace streams may be funnelled (multiplexed) together. Trace data can be collected in an embedded trace buffer (ETB), written to memory via a bus-mastering trace component (ETR), or routed out via a physical trace port (either a dedicated interface such as TPIU or a reused functional interface such as PCIe or USB) to an external trace capture device such as ARM's DSTREAM.

Generally, for software debugging and performance analysis we're interested in two types of trace source:

- ETM: processor trace, tracing processor execution. Some timing information may be present. Trace is highly compressed but bit rates can still range up to 10 Gbits/s. Each core has its own ETM. Some versions are referred to as PTM. Essentially, the sequence of versions is ETMv3.x -> PTMv1.x -> ETMv4.
- STM: software trace. Trace packets are produced when software writes messages to memory-mapped trace ports. Individual pages of the port space may be mapped into address spaces, allowing firmware, hypervisor, kernel, middleware and user applications to share the STM "spectrum". STM essentially plays the role of a trace ring buffer but with much lower overhead.

The following diagram shows a simple CoreSight subsystem with an ETM (attached to a CPU) and an STM (attached to the main bus). The trace streams from the ETM and STM are funnelled together and the resulting trace stream is written out to memory.




**A simple CoreSight subsystem**

CoreSight trace data can provide a variety of information to help in system debugging and performance analysis. Generally it will need decoding and post-processing.

Trace streams are timestamped using the CoreSight global timestamp – a constant-rate, always-on time source. The timestamp will generally run at a lower frequency than the cores.

When, and how, each trace stream indicates timestamps, is specific to the trace source and its stream format. Generally, timestamps are output for certain kinds of packet, and also periodically.

To demultiplex the trace data, and decode and interpret individual trace streams, extra information is needed that is not present in the trace streams. This document explains what that information is, why it is needed and how it can be supplied.

There are essentially two reasons that extra information is needed.

Firstly, the trace data is not self-describing. The streams are identified by 7-bit trace source identifiers, which are configured when programming the trace sources. There's nothing either in each trace stream, or alongside it in the multiplexed trace stream, to tell the decoder that, say, stream 10 is CPU #1's ETM, while stream 20 is the system STM. In order to decode each trace stream using the appropriate decode algorithm, the decoder needs to know what trace sources are present and how they are identified. Decoders may also need to know precisely what trace protocols are in use and how the trace sources have been configured – an ETM, for example, may use alternative packet formats depending on its version and how it is configured.

In the past, trace tools have generally been interactive, in the sense that the user the trace tool to configure the target, and then collect trace, within one debugging session. So most of this knowledge about how the trace was identified and configured would be already known by the tool and could be used by its built-in decoder – so there is no provision for embedding it in the trace stream. However, this has several limitations: it is limited to configurations supported by the tool; it does not handle situations where the trace needs to be captured in one place and analyzed in another, and it does not handle self-hosted trace capture. So, to support interchange of trace data, the trace identification and configuration needs to be captured. Generally this information, although initially unknown to the decoder and possibly configurable at runtime, will be fixed for the duration of a trace file, so it can be considered as static metadata.

The second reason why extra information is needed for decode, relates specifically to processor instruction trace (ETM and PTM): because of its very high bitrate, this trace is heavily compressed. It may for example indicate only a sequence of instruction steps and conditional branch choices, or sometimes only the latter. In order to be

decoded even to a sequence of addresses of executed instructions (let alone disassembly of the instructions themselves), the decoder needs to have access, by some other means, to the opcodes at each address. So at least the decoder will need a mapping and a set of program images. Some of this (e.g. the kernel) will be unchanging and could be provided statically. But if the contents of the address space are changing (either because it is being switched in a multitasking OS or because of program loading or JIT) the decoder will have to keep track of what is where at any given time.

If a suitable instrumentation trace source (e.g. STM) is present, this dynamic metadata might itself be carried on a separate CoreSight trace stream – i.e. the STM stream would provide the metadata needed to decode the ETM stream(s). Alternatively the metadata could be captured by software trace.

Metadata, both static and dynamic, may also be used to communicate how the STM channels are being allocated to different users. The STM stream itself would carry this metadata – as a kind of "station id". A separate proposal is being drafted on STM channel management.

This document outlines the static and dynamic metadata needed at each stage of trace processing. It supports improvement on current tools functionality in two respects:

- it provides for exporting trace and metadata from one environment to another in a robust way
- it supports tracing environments where the program code and configuration is changing dynamically

Currently this document does not specify a concrete representation of the metadata.

# 4 TRACE DECODING STAGES

## 4.1 Overview

Trace decoding is the process of taking the raw trace data (a binary blob, which could be anything from a few Kb to many Gb in size) and producing a stream of meaningful events.

The raw trace data is assumed to consist of the contents of one or more ETBs or memory areas written by ETR or equivalent, or data captured by an external trace capture device from a trace port. In the discussion following we will deal with a single trace buffer. Where multiple trace buffers are used concurrently, the trace streams from each buffer would need to be correlated in the same way as the trace streams within a single buffer.

The trace decode "pipeline" has several stages:

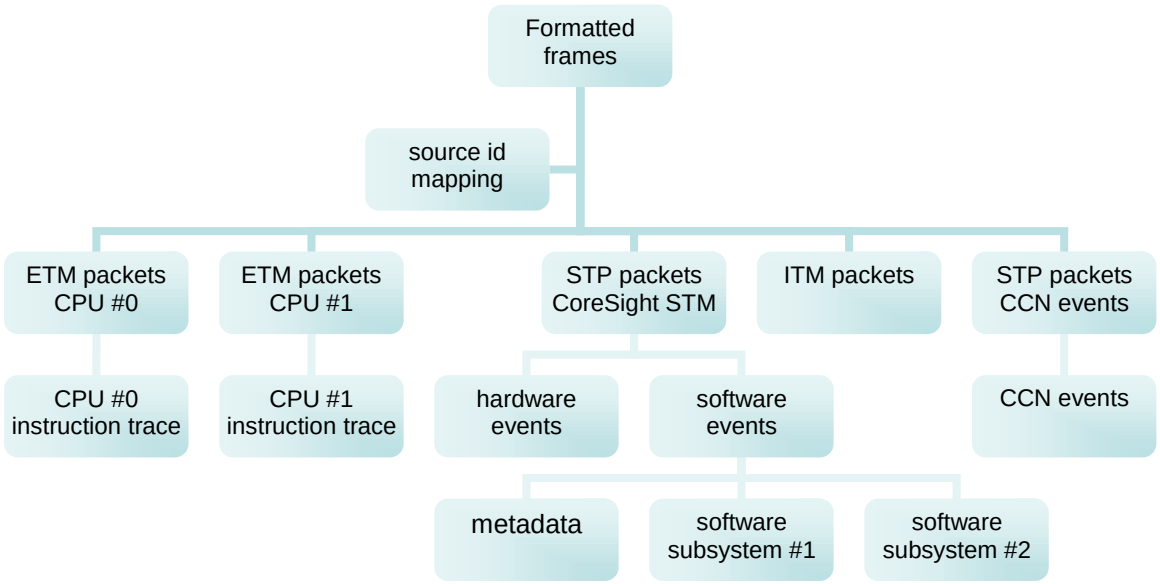
1 Preconditioning the data so that it appears as a sequence of 16-byte formatted frames

2 Demultiplexing: recovery of trace streams for individual trace sources, from the multiplexed stream

3 Stream decoding: recovery of packet boundaries etc.

4 Stream interpretation: association of stream data with actual events on the target

The following diagram shows the stages of trace decode from formatted frames onwards.




## 4.2 Trace preconditioning

Trace output from an external trace port (e.g. TPIU) may have extra padding bytes or other formatting information. Trace captured in a buffer may have the earliest data somewhere in the middle of the buffer (if it has wrapped round).

For the rest of this document we'll assume that some capture-specific process has been done to precondition this trace data into a standard form, i.e.

- the trace data should start at the earliest data
- if the data is formatted it should consist of a sequence of whole 16-byte frames with no padding
- if the data is unformatted it should start on a byte boundary

### 4.2.1 Wrapped trace with unknown starting point

There may be situations where the trace is retrieved from a buffer but the current write pointer has been lost, e.g. due to hardware reset of the trace sink. In that case the buffer might be exported with a flag to indicate that it's got wrapped data, in the hope that a tool could use some kind of heuristic to work out whereabouts in the buffer the trace really started. Handling this situation is outside the scope of this document.

## 4.3 Phase 1: Stream demultiplexing

The trace stream may combine data from several trace sources, using the CoreSight formatting (multiplexing) protocol which takes several streams and produces a sequence of 16-byte formatted frames with the stream identifiers embedded in the frames. This process needs to be reversed in order to extract the streams and pass them to their decoders.

Note that stream multiplexing is done at the byte level not the (stream-specific) packet level. In general, when each decoder starts decoding its respective stream, it should not expect to start on a packet boundary – it will first have to find a packet boundary, e.g. by looking for a special (stream-specific) synchronization packet.

(For unformatted trace this phase is bypassed. Unformatted trace might be found in a trace buffer that was guaranteed to be dedicated to a single trace source, perhaps because the buffer was physically only connected to

one trace source. In this case the demultiplexing stage can be skipped and the stream can be passed straight to the stream decoder.)

The trace data must be presented to the demultiplexer as a sequence of 16-byte formatting frames as defined in [CS]. The output is a set of trace streams, each labelled by trace source id. (Formatted frames may also indicate when triggers occurred.) The frame format, and hence the demultiplexing, is defined by the CoreSight architecture.

For further processing each stream needs to be handed over to a stream-specific decoder. In order to do this we need a mapping from trace source id to

- the type of source (ETMv3.x, PTM, STM etc.) to allow the stream to be assigned to a stream-specific decoder
- an identifier for the specific trace source (e.g. “PTM for CPU #1”, “STM #2”), so that the stream, and trace source, can be identified to its consumer

Note that, in CoreSight, some types of information that might in principle apply to all streams, e.g. timestamps and overflow indicators, is encoded in a stream-specific way. As a result, retrieving any of this information from trace data requires streams to be decoded, though not necessarily fully interpreted. A trace decoder framework might of course implement a generic way for stream-specific decoders to report timestamp and overflow information.

## 4.4 Phase 2: Stream decoding

This phase deals with decoding the stream according to the packet format specified for that trace source. Depending on the kind of trace source, it may need some information about both the static configuration (architecture version, RTL configuration) and dynamic configuration (as programmed through writeable registers) of the trace source.

In some cases the distinction between two different kinds of trace source, and one kind in two different static configurations, is not clear-cut, and the line has to be drawn somewhere. STM and ETM are clearly different, producing different sets of events. ETMv3.5, PTM and ETMv4 convey similar information but formatted in different ways – the decoding algorithm is completely different, but the consumer interface (the stream of decoded information) would likely be the same.

The first step in stream decode is likely to be synchronizing to a packet boundary. This is specific to each stream format.

### 4.4.1 ETMv3 and PTM

The ETM trace formats are specified by various CoreSight architecture documents:

- [ETMv3] for ETMv1.0 to ETMv3.5
- [PFT] for PTM
- [ETMv4] for ETMv4

For the purposes of this document, PTM can be seen as a kind of ETM, fitting between ETMv3.x and ETMv4.

The ETMv4 trace format and programmer’s model is significantly different and is described in the next section.

Trace source information is a combination of

- fixed information, designed into the trace source or configured by the silicon integrator: indicated in ETMIDR and ETMCCER
- runtime configuration, which controls the amount of information output (and bit rate needed): indicated in ETMCR

Runtime configuration would not generally change in the lifetime of a trace session (and the trace source would have to be disabled while this change was being made), but in principle, it could. So it would be useful to provide some way to trace a configuration change, as dynamic metadata.

Trace source information needed for decode to packet boundaries includes

- ETM version (from ETMIDR)
- size of timestamp (bit 29 of ETMCCER)
- whether cycle-accurate trace is enabled (bit 12 of ETMCR)
- presence and size of CONTEXTID information (bits 15:14 of ETMCR)
- presence and size of VMID information (currently always 8 bits)

Also, to correctly decode timestamps and verify they are monotonically increasing, the decoder needs to know

- whether the timestamp is Gray-coded (bit 28 of ETMCCER)

Some additional information about the ETM configuration may be needed for decompression/interpretation – see 4.5.1.

For robustness, the information might include the values of the configuration registers (e.g. ETMIDR, ETMCR and ETMCCER) from the ETM/PTM block, though a decoder should avoid depending on this information, because an artificial trace generator might not be able to synthesize these register contents to match a physical one.

Once the stream is decoded, it then needs to be passed on to the decompression phase. This needs additional data, e.g. for PTM it needs the image of the program to be decoded.

For ETMv3.x, the trace may also include data trace. In this case the metadata will need to indicate

- whether data address tracing is enabled (bit 3 of ETMCR)
- whether data value tracing is enabled (bit 2 of ETMCR)

ETM provides a rich set of programmable features for enabling/disabling trace on events, address ranges, processor modes etc. None of this is needed to decode trace.

### 4.4.2 ETMv4

For ETMv4, ETM data trace, if present, is on a separate stream from the flow trace, and may even be collected in a separate trace buffer.

Generally the trace formats are more self-describing, e.g. the presence of information fields is indicated by packet header bits.

Instruction trace source information needed for decode to packet boundaries includes

- ETM version
- size of CONTEXTID information (TRCIDR2.CIDSIZE)
- size of VMID information (TRCIDR2.VMIDSIZE)
- suppression of COMMIT elements (TRCIDR0.COMMOPT)

No extra metadata information is needed to decode ETMv4 data trace.

### 4.4.3 ITM

Synchronization and packet decode details are covered in [ITM]. No configuration information is needed for decode.

The output of decode is a stream of events and timestamps. The interpretation phase may assign meanings to events.

### 4.4.4 STP

The STPv2 format is defined by [STP]. As this specification is not publicly available, one approach that could be taken is to demultiplex the trace and decode the STP stream(s) to packet boundaries at an early stage, and add the decoded packet stream(s) in a publicly defined format, for later interpretation. Defining such a format is outside the scope of this document.

There is no configuration information needed for decoding an STP stream into packets. Interpreting what the packets mean is covered in the next phase.

The STP protocol might be used by other trace sources. Each trace source will have its own trace stream and the trace source id will be used to route the stream to the appropriate decoder (which will use a generic STP packet decoder but then apply its own interpretation to the resulting packets). In CoreSight, multiple hardware trace sources are not multiplexed together at the STP (packet) level.

## 4.5 Phase 3: Stream interpretation

### 4.5.1 ETM/PTM flow trace decompression

In this step the trace is decompressed. The input is the sequence of packets from the decoder, and metadata to allow the decompressor to locate program images and fetch opcodes. The output is an ordered sequence of addresses, together with exception indications, timestamps etc.

For example, the input trace packets may contain a sequence of E and N atoms indicating that branch instructions passed or failed their condition checks. The decompressor would have to find the instruction boundaries; it would have to identify branch instructions and consume an E or N atom, and for a taken branch it would need to follow the branch and update the current instruction address.

All this implies that the decompressor must be provided with a program image or some other way to look up opcodes by address. If opcodes are unavailable for part of the address space, there may be gaps in the output trace, resynchronizing when address packets or synchronization packets are seen. If the wrong opcodes are provided to the decoder (e.g. because code has been patched at runtime) the output trace may be corrupt.

In an embedded environment where all code is run out of ROM it is sufficient to have just one image. In a rich OS environment there may be multiple address spaces, and the content of the image may change over time. Even in some embedded environments, code may be loaded dynamically. The complexities of trace decompression in dynamic environments are described in more detail below.

For PFT, some trace source configuration information is needed for decompression:

- whether DMB/DSB are treated as waypoints (bit 24 of ETMCCER)

ETMv4 trace may include event packets, and to interpret these we need

- configuration of the events

### 4.5.2 ETMv4 data trace

No extra metadata needed.

### 4.5.3 ITM

The input to this phase is a sequence of events with channel number (0..31) and payload (up to 32-bit), along with timestamps. Any interpretation of the events is likely to be specific to a specific use case, e.g. instrumented RTOS on an MCU, or instrumented firmware running on the system control processor on a SoC.

Currently there are no standardized meanings to any ITM messages.

### 4.5.4 STP from STM

For software instrumentation, a further level of demultiplexing of the STM trace stream would be done, based on a channel allocation scheme. For example, firmware messages might be routed to a viewer for the firmware, Linux kernel messages to a viewer for that etc. STM channel usage is expected to be covered in a separate proposal.

## 4.6 Trace metadata summary

The following abstract notation describes the trace data and metadata:

```
trace = set of {
    trace buffer data,
    trace buffer identifier,
    trace buffer metadata }

trace buffer data = (binary blob)

trace buffer identifier = (some unique way to distinguish the trace buffer when
multiple buffers are present; e.g. the base address of an ETB, the base address of
a trace buffer in memory, or the unique identifier of an external trace capture
device)

trace buffer metadata = formatted ? (trace source id -> trace source metadata) :
trace source metadata

trace source id = 0x01..0x6F

trace source metadata =      // tagged union
    etm metadata |
    itm metadata |
    stp metadata |
    other

etm metadata = { etm static metadata, etm dynamic metadata }

etm static metadata = {
    etm architecture version (string),
    timestamp size (48 or 64),
    timestamp is Gray-coded (flag),
    dmb_dsb_are_waypoints (flag) }

etm dynamic metadata = {
    cycle-accurate (flag),
    context id size (0/8/16/32),
    vmid size (0/8/16) }

itm metadata = none

stp metadata = {
    stm metadata |
    third-party metadata...
```

```
stm metadata = {
    mux register setting (optional)
```

We'd expect other identification to be attached to a trace – e.g. the machine identifier, time etc. – but our focus is on the specific data needed by a trace decoder.

# 5 DYNAMIC PROGRAM TRACE DECOMPRESSION

## 5.1 Overview

As mentioned above, decompression of ETM/PTM trace formats in general needs access to the program images, in order to follow a stream of instructions, identify branches, and produce a sequence of instruction addresses. In a simple embedded system this may be as simple as pointing the decoder to a ROM image.

There are several reasons why this gets more complex:

- code may have been patched or moved at startup. Even if the code is not changing during the trace session, this may mean the code being executed differs from or is not present in the image.
- application code may have been loaded at unpredictable addresses due to address space layout randomization (ASLR).
- there may be multiple address spaces (in a VMSA system), each associated with their own image(s); the addresses appearing in the trace stream will be virtual, and to decompress them we need to know the active address space
- the code may change dynamically over time, perhaps even during the trace session, due to processes starting and terminating, modules being dynamically loaded or unloaded, or self-modifying and dynamically generated code

To deal with dynamically changing mappings we need a "side channel" to notify the trace decoder so it can update its mapping of addresses to opcodes concurrently with program flow trace decode. The notification could take the form of (timestamped) snapshots of system state, and events which indicate changes to system state.

For example, in a crash dump situation, a system state snapshot might be obtained by looking at data structures in system memory; an event log could be found in a circular buffer in system memory, and trace decode could use the event log to "wind back" the system state.

A program-trace decoder should be able to handle the case of an incomplete event stream, e.g. where a trace buffer has wrapped and a process creation event has been lost.

## 5.2 Simple use cases on Linux

The following discussion is going to get rather tedious and complicated. It's probably worth thinking about some simple trace use cases to see how much of this complication is really necessary.

### 5.2.1 Simple kernel tracing

Generally, it's possible to trace the kernel given only the static metadata about how the trace is configured, plus knowledge of where the kernel is loaded, and the kernel image itself.

If you try this you'll probably see some trace missing, around first-level exception handling and dynamically loaded kernel modules. Generally a decoder can recover from this situation.

Worse, you might also see corrupt trace around function entries due to code patching, especially if NOPs have been replaced by branches or vice versa (as in dynamic tracepoints).

These problems can be avoided if instead of using the built kernel image, an image is created from the actual kernel address space, i.e. essentially a core dump. Of course, a decoder might want to merge in the symbolic and debugging information from the original image.

### 5.2.2 Full kernel tracing

Tracing into kernel loadable modules, would need to know where they are loaded.

Some kernel code may be either moved into position after load (currently on ARM, first-level exception handler seems to be like this) or may be patched after load. This code could be described in the form of a delta.

The overall “package” then consists of

- a map locating the kernel, loadable kernel modules, and patched code
- the kernel image (e.g. vmlinux)
- images for kernel loadable modules
- details of the code that has been patched in
- ETM configuration metadata

With this information the decoder should be able to trace all execution in the kernel, assuming that there are no changes during the trace session.

### 5.2.3 Single user address space

If a single address space is being traced, the decoder will need the address space map and the image files loaded into it.

For a long-running workload this might be relatively simple:

- start the workload (perhaps pinning it to one core) and remember its PID
- give it enough time to load its shared libraries
- read `/proc/<PID>/maps`
- capture some trace

The address space map and images (together, as always, with the ETM configuration metadata) would be enough to decode the userspace trace. Userspace might also execute code mapped in by the kernel, e.g. VDSO, and this should be mapped and provided too.

The problem would be with other processes being run on the same core, which might use the same userspace virtual addresses (but have different program contents at those addresses). If the OS is using the hardware CONTEXTIDR register as described below, then processes can be tracked by observing CONTEXTIDR changes in the ETM stream – or the ETM could be programmed to filter only the trace from one CONTEXTIDR. If CONTEXTIDR is not used, some other way is needed.

These problems don’t arise if the OS has been set up to prevent all other processes from running on the core being traced.

For workloads that load and unload dynamic libraries, the address space contents will be much more fluid and might change during a tracing session. In this case the address space change details will need to be supplied as dynamic metadata and correlated with the ETM stream. The same applies to JIT. This dynamically changing information is the focus of the following sections.

### 5.2.4 Kernel plus single user address space

Assuming the kernel and userspace address maps don't overlap, decoding both is simply a matter of providing the combined map and set of images.

### 5.2.5 Multiple address spaces

ETM includes a mechanism, the CONTEXTIDR register, that is designed to allow decoding of multiple address spaces. Use of CONTEXTIDR is described in [ETM] section 9, "Tracing Dynamically Loaded Images". Basically, the OS sets a value in CONTEXTIDR to indicate the address space. ETM outputs changes to CONTEXTIDR in the ETM trace stream.

If the OS uses CONTEXTIDR then the decoder needs a map for each address space, and an indication of how the CONTEXTIDR value indicates the address space. Alternatively, the ETM could be configured to select trace only from the specified CONTEXTIDR. (Note that if the OS puts the thread id in CONTEXTIDR, and it's desired to trace all threads in an address space, this filtering on CONTEXTIDR won't work.)

If CONTEXTIDR is not being used, then context switches need to be indicated some other way, or the problem needs to be worked around some other way e.g. by pinning each address space to a different core.

### 5.2.6 Summary

Many simple trace use cases can be handled without the need for dynamic metadata, when when the workloads are simple, steady-state and long-running. Dynamic metadata comes into play when the address space being traced is changing, or when process switch is not indicated by CONTEXTIDR. The rest of this section outlines how dynamic metadata can be used to provide full trace decode in these situations.

## 5.3 Multiple address spaces and use of CONTEXTIDR

To deal with multiple address spaces, a program-trace decompressor will need to know

- whether, and if so how, the current address space is indicated in the program trace – this may be from the ARM core's CONTEXTIDR (Context ID Register), but not all OSes are configured to use this.
- a mapping of address space and address, to program image(s) – i.e. a set of tuples of the form (asid, address, image).

A snapshot would list the current address spaces and their images.

Where only one address space is in use, this section can be ignored.

### 5.3.1 Interpretation of CONTEXTIDR

The idea is that the OS sets CONTEXTIDR to indicate the current address space and ETM is configured to output CONTEXTIDR change events. Then a decoder needs only an address space map for each address space. It does not need any dynamic metadata to indicate when address spaces are switched, since this is indicated by CONTEXTIDR in the ETM trace stream. (Dynamic metadata might be useful for other purposes e.g. if the address space maps are changed during the trace session – but those events may be much less frequent than process switch.)

In the ARM architecture [ARM ARM B4.1.36], CONTEXTIDR is a 32-bit register accessible from privileged state. It can be used by debug logic and trace logic to identify and match against the current process. It might or might not also hold an 8-bit address space identifier (ASID) used for virtual address translation.

In an implementation using the short-descriptor format, the low 8 bits of CONTEXTIDR hold the process's current ASID on the core being traced. The top 24 bits, if used, typically hold a process or thread identifier determined by the OS. The mapping of process identifier to ASID is not necessarily fixed.

In an implementation using long-descriptor format, the entire register is free to be programmed by the OS.

The tracing of CONTEXIDR is configured through the ETM main control register. Trace may include the low 8 bits, 16 bits or 32 bits of CONTEXTIDR, or might not include it at all. As described earlier (4.4.1), to decode to packet-boundary level the decoder will need to know the size of the traced CONTEXTIDR.

To recover the address space from CONTEXTIDR (if it is present) the decoder will need to know the format of the information in CONTEXTIDR. This could take various forms, e.g.:

- an ASID in bits 0..7
- an ASID in bits 0..7 and a process (i.e. thread group / OS address space) identifier in bits 8..31
- an ASID in bits 0..7 and a thread identifier in bits 8..31
- a thread identifier in bits 0..31

### 5.3.2 Use of CONTEXTIDR on Linux

In Linux for ARM, the use of CONTEXTIDR is controlled by CONFIG_PID_IN_CONTEXTIDR. This can be checked from userspace by e.g.

```
root@ve2:~# grep CONTEXTIDR /boot/config-`uname -r`
# CONFIG_PID_IN_CONTEXTIDR is not set
```

If CONFIG_PID_IN_CONTEXTIDR is unset, then CONTEXTIDR contains only the ASID (short descriptors) or nothing at all (long descriptors).

If CONFIG_PID_IN_CONTEXTIDR is set, then CONTEXTIDR contains the thread identifier in bits 8..31. It may also contain the ASID. Note that the thread identifier here is the "lightweight" process identifier – the number returned by the `gettid()` system call, not the Unix process identifier returned by `getpid()`, which Linux views internally as the thread group identifier.

If CONTEXTIDR contains only the ASID (e.g. short descriptors used, and CONFIG_PID_IN_CONTEXTIDR defaulted off), then userspace trace decompression can detect process switch by observing an ASID change, but will need to maintain a dynamic mapping of ASIDs to PIDs.

## 5.4 Dynamic metadata overview

Dynamic metadata is timestamped information that is used to help decode ETM trace. The decoder will unpack the raw CoreSight trace data into separate ETM streams. When decoding each ETM stream it will keep track of the timestamp in the stream and look for dynamic metadata events occurring at that time.

If the dynamic metadata is being collected via CoreSight STM, then it will already be timestamped with the CoreSight global timestamp. Otherwise, if dynamic metadata is being collected via other trace mechanisms, the trace will have to be correlated. We assume that people would often want to use CoreSight trace alongside existing kernel trace events (of all kinds), so the time correlation problem has to be solved somehow.

## 5.5 Dynamic changes to code

The ETM/PTM decoder will in general need to know the state of any traced address space at the time it decodes program trace within that address space (for this purpose, the kernel counts as an address space). So it will need to be told about changes to the address space.

If full information about code-changing events is not available, a decoder could use information collected either before or after the trace. E.g. in a post-mortem situation the address space mapping at the end of the trace could be obtained by inspecting kernel data structures. A decoder might decode the trace on the basis of this mapping – this would handle processes, modules etc. that were active at trace termination but not ones that terminated during the trace capture.

We now suggest a set of events that should be traced (either via STM or in some other trace buffer). This would apply to any operating system that had a concept of “process” and “image file”.

An STM usage standard would likely define a standard STM-based protocol for these messages.

### 5.5.1 General rules for messages

All the code-change messages described below are assumed to be timestamped. If traced using STM, these can be correlated with program flow trace using the CoreSight global timestamp. If traced some other way, some time correlation mechanism is needed.

A message generally indicates the existence (at some time) of some entity such as an OS process, or a relationship between two entities. A STATE parameter will be one of START, END or CURRENT indicating whether the entity (or relationship) is being created, is being terminated, or is just asserted to exist.

Other parameters indicate properties of the entity. All parameters are present even in termination messages, to allow state to be reconstructed by processing events in reverse order, where the process creation message might not be present.

### 5.5.2 Process creation and termination

The process state message:

```
declare_process(STATE, pid, base_address, image_file, uuid)
```

notifies the creation or removal of an address space, with an image file located at a given base address. The image file is assumed to be in some appropriate format e.g. ELF. The base address is the base address of the image, to which addresses in the image are relative.

If the image file is non-relocatable, the file will indicate the load address, and the address parameter on this event is redundant.

The image can be specified by name and/or UUID, where UUID is some kind of unique identifier or hash that can be assumed to change if the image contents change. UUID allows more accurate identification of the right version of the image, which is critical for correct trace decode.

### 5.5.3 Thread creation and termination

The thread state message:

```
declare_thread(STATE, tid, pid)
```

notifies about a secondary thread within a process. This allows the decoder to identify the process (i.e. address space) when CONTEXTIDR contains thread identifiers rather than process identifiers.

### 5.5.4 Dynamic library loading and unloading

The library state message:

```
declare_library(STATE, pid, base_address, image_file, uuid)
```

indicates that a shared object was loaded, unloaded or is present at a given address. STATE is as for `declare_process`.

This event can also be used for modules in the kernel address space, e.g. loadable kernel modules in Linux.

### 5.5.5 JIT and code patching

This message copes with cases where the code being traced is not present in an image file. If the software being traced is a JIT engine (e.g. Java) this is likely to be a significant proportion of executed code.

On ARM, most code modification will need a system call for cache unification, and this may be a suitable point to generate a message. Some JIT engines will provide tracing mechanisms or use the GDB JIT registration mechanism.

```
patch_code(pid, start_address, length, id|code|None)
```

indicates that new code was installed at a given address. The new code can be supplied along with the event. Alternatively, to save trace bandwidth, the code might be stored in an on-target buffer, and an identifier (e.g. a unique sequence counter, or hash) used to match the code with the event. Or the code could be omitted on the assumption that when decoded, the code will still be present at its original location.

### 5.5.6 ASID-to-PID mapping

This message is needed when CONTEXTIDR contains the ASID but not the PID, and allows the decompressor to maintain an ASID-to-PID mapping.

```
declare_asid(STATE, asid, pid)
```

Note that [ARM ARM] guarantees that the ASID-to-PID mapping is consistent across all cores used by the OS:

> "For a symmetric multiprocessor cluster where a single operating system is running on the set of processing elements, ARMv7 requires all ASID values to be assigned uniquely within any single Inner Shareable domain. In other words, each ASID value must have the same meaning to all processing elements in the system."

## 5.6 System snapshots

A system snapshot can be represented as a sequence of events of type CURRENT which accumulate the represented state. For example a list of processes could be represented as

```
declare_process(CURRENT, 0, 0x0000, "kernerl.exe")
declare_process(CURRENT, 1, 0x8000, "/bin/ls")
```

This allows a trace decoder to populate and cross-check its information about system state, whether the snapshot is taken at the start or end of trace or at intermediate times.

So, in principle, the event log is sufficient – it may start and/or end with a series of CURRENT events representing the current state at that time.

## 5.7 Trace decode procedure

### 5.7.1 When CONTEXTIDR contains process id

Given the above code-change messages the following program flow trace decompression procedure can be used. Code-change events are assumed to be interleaved into the program flow trace for each core.

The code mapping for the decoder is conceptually a table that maps a (pid, address) pair to an instruction. In practice the entries in the table are likely to be address ranges pointing to executable files.

The decode procedure is as follows:

```
current_pc = UNKNOWN
current_pid = UNKNOWN
current_id = UNKNOWN
current_address_map = {}
current_id_aliases = {}
while <more trace events>:
  case <event>:
    ETM/PTM program flow packet:
      decompress packet using mapping for (current_pid, current_pc)
      update current_pc
    ETM/PTM CONTEXTID packet:
      set current_id to pid from packet
      if current_id in current_id_aliases:
        current_pid = current_id_aliases[current_id]
      else:
        current_pid = current_id
    declare_process(START|CURRENT, pid, base_address, image, UUID),
    declare_library(START|CURRENT, pid, base_address, image, UUID):
      add (pid, start_address) to current_address_map
    declare_process(END, pid, base_address, image, UUID),
    declare_library(END, pid, base_address, image, UUID):
      remove (pid, base_address) from current_address_map
    declare_thread(START|CURRENT, pid, tid)
      add (tid, pid) to current_id_aliases
    declare_thread(END, pid, tid)
      remove (tid, pid) from current_id_aliases
    patch_code(pid, start_address, length):
      add (pid, base_address) and code to current_address_map
```

### 5.7.2 When CONTEXTIDR contains only ASID

The decompression procedure when CONTEXTIDR contains only the ASID is more complex, as the decoder must maintain an ASID-to-PID mapping for the core:

```
current_pc = UNKNOWN
current_pid = UNKNOWN
current_asid = UNKNOWN
current_asid_to_pid_map = {}
current_address_map = {}
current_id_aliases = {}
while <more trace events>:
  case <event>:
    ETM/PTM program flow packet:
      decompress packet using mapping for (current_pid, current_pc)
      update current_pc
    ETM/PTM CONTEXTID packet:
      set current_asid to asid from packet
      if current_asid in current_asid_map:
        set current_pid = current_asid_map[current_asid]
      else:
        set current_pid = UNKNOWN
    declare_process(START|CURRENT, pid, base_address, image, UUID),
```

```
declare_library(START|CURRENT, pid, base_address, image, UUID):
  add (pid, base_address) to current_address_map
declare_process(END, pid, base_address, image, UUID),
declare_library(END, pid, base_address, image, UUID):
  remove (pid, base_address) from current_address_map
declare_asid(START|CURRENT, asid, pid):
  add asid to current_asid_to_pid_map
declare_asid(END, asid, pid):
  remove asid from current_asid_to_pid_map
declare_thread(START|CURRENT, pid, tid)
  add (tid, pid) to current_id_aliases
declare_thread(END, pid, tid)
  remove (tid, pid) from current_id_aliases
patch_code(pid, start_address, length):
  add (pid, start_address) and code to current_address_map
```

## 5.8 Trace configuration changes

As mentioned earlier, an ETM trace source might be reconfigured in such a way as to change its packet format. For example, the user might collect some non-cycle-accurate trace, then switch to cycle-accurate trace. One option would be to say that the entire trace subsystem must be drained when changing configuration. In this section we explore how to avoid that.

To allow reconfiguration without draining the trace data, the new ETM trace configuration should be output as dynamic metadata. The ETM decoder must process this metadata (along with the address-space metadata described above) and update its configuration. If metadata is output using STM into the same buffer, the buffer contents will look something like:

<ETM id>: … end of ETM trace with old configuration
<STM id>: STM message indicating new configuration for this ETM
<ETM id>: ETM I-Sync packet
<ETM id>: ETM trace with new configuration …

However, the metadata might be output into some other trace buffer or into software trace. Using timestamps for correlation is problematic as ETM might not output a timestamp packet when needed.

Instead, ETM reconfiguration can be marked by changing the ETM's trace source id (ATB id) at the same time, to an unallocated id. The ETM buffer will then look like:

<ETM old id>: … end of ETM trace with old configuration
<ETM new id>: ETM I-Sync packet
<ETM new id>: ETM trace with new configuration …

The dynamic metadata should indicate the new trace source id and new configuration:

```
define_trace(END, <old ATB ID>, <trace source identifier>)
define_trace(START, <new ATB ID>, <trace source identifier>, ETMCR)
```

Assuming ATB ids aren't reused too frequently, this method will allow the boundary between the old and new trace formats to be associated with the metadata that defines the new format.

(Note that this requires the trace to be collected with CoreSight formatting enabled.)

# 6 CONCRETE TRACE FORMATS

## 6.1 Trace package format overview

This section defines specific file formats for interchange between trace producers and trace consumers. In general a “trace capture package” will be an archive containing one or more trace buffer contents, program images, and metadata. A trace interchange format could be defined in its own right (as sketched below) or it could be defined as an extension of a system snapshot / dump format.

## 6.2 Using an existing format

Existing snapshot or trace formats could be extended to include trace and metadata. For example:

- an ELF core dump file (or equivalent) could have custom sections containing the CoreSight trace data, the configuration metadata describing it, and possibly a log of dynamic metadata events. The core dump would describe the program images needed for decode, but would not necessarily contain them.
- a general-purpose trace format such as perf.data or CTF could carry the dynamic metadata events, and also indicate when trace was captured. The trace file might carry the configuration metadata within the file, and indicate how to find the CoreSight trace data and program images.

## 6.3 Trace metadata format

The metadata needed for each trace source was previously defined in 4.6. This section sketches a simple, OS-neutral way for encoding it concretely, e.g. for export to a debugger.

### 6.3.1 INI format

The metadata can be encoded in an INI file as described in http://en.wikipedia.org/wiki/INI_file.

The first section describes the trace buffer format:

```
[buffer]
vendor: ARM
component: ETB
format: coresight
```

Remaining sections describe the trace streams, indexed by trace source identifier:

```
[trace_1]
format: ETM3.5
ETMCR: 0x12345678
cyclecount: 1
timestamp_bits: 48
timestamp_gray: 0
context_id: 32
```

The items are described in more detail in this table:

| Item | Values | Meaning |
|---|---|---|
| format | “ETM3.5”, “PFT1.1” etc. | ETM/PFT architecture version |
| ETMxx | 32-bit hex value | contents of an ETM register |

| | | |
|---|---|---|
| cyclecount | boolean | cycle counting is enabled |
| timestamp_bits | 0, 48, 64 | size of timestamp value |
| timestamp_gray | boolean | timestamp is Gray-coded |
| context_id | 0, 8, 16, 32 | size of CONTEXID value |

Data may be provided either in the form of ETM register contents, or as individual options, as long as sufficient information is provided to decode the trace. It is strongly recommended that individual options are supplied so that decoders do not have to be aware of the validity of individual bits within configuration registers for different ETM versions.

There are no default values. Items may be omitted only if not needed for decoding the trace.

Where a trace capture includes the contents of multiple buffers, and the trace source identifiers are unique and consistent across the system, all buffers can be described by one metadata file.

Where systems have so many trace sources that trace source identifiers need to be reused, multiple metadata files will be needed.

### 6.3.2 Other formats

Other formats (JSON, XML etc.) may be defined.

## 6.4 Trace package

The package should be supplied to the decoder as a directory, or some packaged version of a directory (e.g. ZIP). This should contain:

- A contents file `contents.ini`. This will locate the trace buffers and trace metadata in the package.
- A (binary) file for each trace buffer, preconditioned as described in 4.2
- System event log
- Image files for program trace decompression (where not otherwise available)
- Memory contents
- Other data as needed by custom trace plugins (e.g. STP viewers)

Example:

```
trace.zip:
  contents.ini
  trace.ini
  cluster0.data
  cluster1.data
  ...
  system-event-log

contents.ini:
  [trace]
  metadata=trace.ini
  buffer0=cluster0.data
  buffer1=cluster1.data
  ...
```

```
[...]
description of memory contents etc.
```

# 7 FURTHER WORK

- Define STM channel management overall, including how STM is shared between the many different software entities that might use it.
- Define how Linux kernel trace can use STM as a transport mechanism
- Define how dynamic metadata (as described in this document) is output by Linux trace events
- Consider defining a neutral (non Linux specific) way of communicating dynamic metadata over STM
- UUIDs – what do they look like and when are they present? Probably platform-specific
- Extend this document to virtualization (either hypervisor or KVM)
- Define a more generic system-state format of which trace is a special case, allowing trace to be packaged in core dump files (for example)