Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V

Authors: Iain Robertson

Version v1.0, 2024-07-05: Ratified

Table of Contents

Change Log . . . 1
Copyright and license information . . . 2
Contributors . . . 3
1. Introduction . . . 4
  1.1. Glossary . . . 4
2. Packet Encapsulation . . . 5
  2.1. Normal Encapsulation Structure . . . 5
    2.1.1. Header . . . 5
    2.1.2. srcID . . . 6
    2.1.3. Timestamp . . . 6
    2.1.4. Payload . . . 6
    2.1.5. Packet Length . . . 7
  2.2. Null Encapsulation . . . 7
  2.3. Synchronization . . . 8
3. Packet Encapsulation for E-Trace . . . 9
  3.1. **srcID** . . . 9
  3.2. **timestamp** . . . 9
  3.3. **type** . . . 9
  3.4. Synchronization . . . 10

Change Log

PDF generated on: 2024-07-05 15:21:36 UTC

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

- Iain Robertson (Siemens) <iain.robertson@siemens.com> - author
- Paul Donahue (Ventana) - reviews
- Michael Schleinkofer (Lauterbach) - reviews
- Beeman Strong (Rivos) - reviews
- Robert Chyla (MIPS) - reviews
- Ved Shanbhogue (Rivos) - reviews

Chapter 1. Introduction

The Efficient Trace for RISC-V (E-trace) standard defines packet payloads for instruction and data trace but does not fully define how this should be encapsulated into fully formed packets for transport, nor how instruction and data trace should be differentiated. Chapter 7 gives some illustrative examples but this is insufficiently detailed and informative only.

Although the primary motivation for developing this standard was to define an encapsulation format for E-Trace packets that allows tools to parse and decode them in a standard manner, the encapsulation format defined in this document is agnostic to the packet payload structure and meaning and so can be used for any kind of unformatted data. In addition to E-trace, it could also be used for a wide variety of other uses, for example: performance counter metrics, trace or other diagnostic data from a bus fabric monitor or on-chip logic analyser.

It is not suitable for data that has already been formatted into packets, such as N-Trace, which inserts a 2-bit MSEO formatting code into each byte.

This specification defines an encapsulation format suitable for use with a variety of transport mechanisms, including but not limited to AMBA Advanced Trace Bus (ATB) and Siemens' Messaging Infrastructure.

Examples of how trace packets can be routed for transport is given in the 'Trace Components' subsection of the RISC-V Trace Control Interface Specification.

1.1. Glossary

- **ATB** - Advanced Trace Bus, a protocol described in ARM document IHI0032B;
- **E-Trace** - Abbreviation for Efficient Trace for RISC-V;
- **PIB** - Pin Interface Block, a parallel or serial off-chip trace port feeding into a trace probe, as defined in the RISC-V Trace Control Interface Specification;
- **N-Trace** - Abbreviation for RISC-V N-Trace (Nexus-based trace) Specification
- **Trace Encoder** - Hardware module that accepts execution information from a hart and generates a stream of trace packets;
- **TFP** - Trace Formatter protocol, a trace framing protocol described in ARM document IHI0029E. Also adopted by MIPI as Trace Wrapper Protocol (TWP);
- **TWP** - See **TFP**.

Chapter 2. Packet Encapsulation

Two types of encapsulation are defined: *normal* and *null*. A transmitted stream of encapsulated packets comprises a mixture of *normal* and *null* packets. Each packet is atomic, and must be transmitted in its entirety before another packet can be sent.

2.1. Normal Encapsulation Structure

The normal encapsulation structure is comprised of four field-groups as shown in Table 1. In this and the following tables, the field-groups and fields are listed in transmission order: the uppermost field-group or field in a table is transmitted first, and multi-bit fields are transmitted least significant bit first.

Table 1. Encapsulation Field Groups

Group Name	# Bits	Description
header	8	Encapsulation header. See Section 2.1.1.
srcID	0 - 16	Source ID. See Section 2.1.2.
timestamp	T*8	Time stamp. See Section 2.1.3.
payload	1-248	Packet payload. See Section 2.1.4.

The groups are defined in the following sections:

2.1.1. Header

The header is a single byte comprising the fields defined in Table 2.

Table 2. Header Fields

Field Name	# Bits	Description
length	5	Encapsulated payload length. A value of *L* indicates an *L* byte payload. Must be > 0 - see Section 2.2.
flow	2	Flow indicator. This can be used to direct packets to a particular sink in systems where multiple sinks exist, and those sinks include the ability to accept or discard packets based on the flow value.
extend	1	Indicates presence of timestamp when 1. Must be 0 if timestamp width is 0.

2.1.2. srcID

The **srcID** field identifies the source of the packet. It can be between 0 and 16 bits in length. This length must be fixed and discoverable for a given system.

The **srcID** may be omitted (i.e. zero bits in length) if there is only one source in the system, or if the transport scheme includes a sideband bus for the source ID (for example, ATB).

When present, an 8-bit **srcID** will be sufficient for most use cases, and is simplest in terms of determining the packet length, keeping all field groups aligned to byte boundaries. However, the length of the field can be reduced to improve efficiency for small systems, or increased if required for larger systems. This is explained in more detail in Section 2.1.5.

2.1.3. Timestamp

The **timestamp** field provides a means to include time information with every packet. It is included in the encapsulation if **header.extend** is 1. When included, the timestamp must be T bytes in length. The length must be discoverable, and fixed for a given system.

Timestamps may be omitted either because time is not of interest to the user, or if time information is already included within the encapsulated payload.

2.1.4. Payload

The encapsulation payload can be up to 248 bits (31 bytes) in length, and comprises the fields shown in table Table 3.

Table 3. Payload Fields

Field Name	# Bits	Description
type	≥ 0	Packet type. May be eliminated (i.e. width set to 0) for sources with only one packet type. Length must be fixed for a given **srcID**, and discoverable if > 0.
trace_payload	≤ R	Packet payloads such as those defined for E-Trace. Maximum value of R is defined as 248 - Y - **srcID**%8, where Y is the length of the **type** field. See Section 2.1.5 for details of the relationship between **srcID** and payload length.

2.1.5. Packet Length

Encapsulated packets are a number of whole bytes in length, the exact number depending on the sizes of the **srcID**, **timestamp** (if present) and **header.length**:

Packet length = 1 + S + (T * **header.extend**) + **header.length**

S and T are discoverable constants; S is the number of whole bytes of **srcID**: int(#bits(**srcID**)/8).

For the case where the size of **srcID** is a multiple of 8 bits, **header.length** is simply the number of bits of payload rounded up to the nearest multiple of 8 and expressed in bytes:

header.length = ceiling(#bits(**payload**)/8)

However, if the **srcID** is not a multiple of 8 bits the remaining **srcID** bits not accounted for by 'S' are instead included when determining the value of **header.length**. Thus the more general definition for any **srcID** size is:

header.length = ceiling((#bits(**payload**) + #bits(**srcID**)%8)/8)

In this way, the maximum payload length is reduced by up to 7 when the **srcID** is not a multiple of 8 bits.

In cases where the number of bits of **payload** + **srcID** is not a multiple of 8, some padding bits are required. These must be placed in the most significant bits of the final byte of the packet. Their value is "don’t care", but they must not "leak" information (for example, the previous contents of an intermediate buffer that may relate to a different trace session which the current recipient of trace is not authorised to receive).

2.2. Null Encapsulation

For *normal* encapsulation, the **header.length** field is at least 1, and the overall length of the encapsulated packet will be at least 2 bytes (**header** plus 1 byte of payload).

A *null* packet is specified as consisting exclusively of one **header** byte with its **length** field set to zero, explicitly indicating the packet’s total size as one byte. The **extend** field is used to distinguish 2 different types of null packet, which are defined as follows:

- **extend** = 0: *null.idle*
- **extend** = 1: *null.alignment*

Usually, *null.idle* will be used. *null.alignment* is used for synchronization, as described in the following section.

Insertion of *null* packets typically occurs at a trace sink where there are no sideband signals accompanying the data stream to identify valid data. Packets emitted from a trace source are generally transported over some form of on-chip transport (e.g. ATB) that includes sideband signalling to indicate when data is valid. In this situation when there is no data to send, valid is simply deasserted. That said, the **flow** field definition in *null* packets is unchanged, so *null* packets can be routed from a trace source to a specific sink if required. When generated at a sink, the **flow** field value is unimportant and is typically 0. If the sink is generating a bit stream (i.e. the byte boundaries are not inherently known to the recipient) then the **flow** field must be zero in all *null* packets within the generated bitstream.

2.3. Synchronization

In a data stream comprised of packets, it’s a requirement to be able to determine where packets start and end, when starting from an arbitrary point, without knowledge of the full packet history. This can be achieved by inserting a synchronization sequence into the packet stream. This sequence is comprised of a sufficiently long sequence of *null* packets.

A 'null' byte is defined as a byte with the 5LSBs all zero, which may be a *null* packet, or may be part of a *normal* packet. The longest run N of ‘null’ bytes possible within a *normal* packet is:

N = 31 + T + S (see Section 2.1.3 and Section 2.1.5 for definitions of T and S respectively)

Therefore, in a sequence of N or more ‘null’ bytes, the first N 'null' bytes may actually be part of a packet. However, any 'null' bytes after this must be *null* packets, and the 1st non-null byte seen after this must therefore be the 1st byte of a *normal* packet.

For unframed data streams such as PIB, a *null.alignment* packet must be transmitted as the final *null* before a *normal* packet. Strictly speaking this is necessary only if the data stream is sent via an interface less than 8 bits wide, but for simplicity this is mandatory for any width. The single 1 at the end of this sequence uniquely identifies the byte boundary, and what follows as the start of a packet. For example, for two *normal* packets with M *nulls* between them, this would comprise M-1 *null.idles* and 1 *null.alignment* (M > 0).

For framed data streams which incorporate synchronization information in their own framing such as MIPI TWP (aka ARM Trace Formatter Protocol) or USB there is no requirement to include *null.alignment* packets.

The synchronization requirements are summarized in the following rules:

- A synchronization sequence must have a length of N+1 bytes (N defined above), comprising:
  - For unframed data streams, N consecutive *null.idle* packets, directly followed by one *null.alignment* packet;
  - For framed data streams, N consecutive *null.idle* packets, directly followed by one *null.idle* or *null.alignment* packet.

Synchronization sequences are typically inserted periodically. In addition, a sufficiently long run of *null* packets (due to a lack of *normal* packets to send) may also serve as an 'opportunistic' synchronization sequence. For unframed data streams, this requires *null.alignment* packets to be included, either as every (N+1)th *null*, or as the final *null*.

For writing unframed data to memory, alternative synchronisation mechanisms may also be employed. For example, by dividing memory into blocks of known size, and requiring that packets do not straddle block boundaries. The first byte of every block will therefore be the start of a packet. Details of such schemes are out of scope of this specification.

Chapter 3. Packet Encapsulation for E-Trace

Chapter 2 describes the packet encapsulation in general terms. This chapter describes how that applies in the context of E-Trace.

The RISC-V Trace Control Interface Specification includes several fields relevant for the construction and synchronization of packets, as described in the following sections.

3.1. srcID

- **trTeInhibitSrc** in the *trTeControl* register of a trace encoder indicates whether a **srcID** is present or absent in the encapsulated packets;
- **trTeSrcBits** in the *trTeInstFeatures* register indicates the length of the **srcID**;
- **trTeSrcID** in the *trTeInstFeatures* register indicates the value of the **srcID** associated with that particular source.

3.2. timestamp

- When **trTsEnable** in the *trTsControl* register is 1, a timestamp may be optionally included, via the **timestamp** field in the encapsulated packets. When present, **header.extend** will be 1 as described in Section 2.1.3;
- **trTsWidth** in the *trTsControl* register indicates the number of bits in **timestamp** fields.

Note that some E-Trace packets may optionally include a **time** field, as an alternative method of providing time information. Presence or absence of this is fixed for a given system based on a discoverable parameter, but is not run-time configurable.

3.3. type

The capabilities of the trace source will determine the minimum width of the **type** field in the **payload** field-group. For E-Trace:

- If only instruction trace is supported, the minimum width is 0 (i.e. the field is omitted)
- If both instruction and data trace are supported, the minimum width is 1, encoded as
  - 0: Instruction trace packet
  - 1: Data trace packet
- A width of 2 or more is permited for applications where packet types other than instruction and data trace are required. Encoding for this case is application specific and not mandated by this standard.

3.4. Synchronization

- **trPibAsyncFreq**, **trRamAsyncFreq** and **trAtbBridgeAsyncFreq** in the *trPibControl*, *trRamControl* and *trAtbBridgeControl* registers respectively are used to determine the interval between insertion of synchronization sequences.