Reference CoreSight Trace Decoder: API Specifications and Component Design

Debug and Trace

Development Solutions Group

Document number:	ARM-ECM-0398495 Version 0.3
Date of Issue:	27/05/15
Author:	Mike Leach
Authorised by:	

Abstract

Description of key RCTDL API elements and infrastructure.

Keywords

Distribution list

Name	Company	Name	Company
John Connell	ARM Ltd	Michael Darling	ARM Ltd
Mike Leach	ARM Ltd	Al Grant	ARM Ltd
Paul Neuhardt	ARM Ltd	Mathieu Porier	Linaro
Bill Fletcher	Linaro	Serge Broslavsky	Linaro
David Griego	Linaro	Viswanath Puttagunta	Linaro
William Mills	T.I.	Tor Jeremaissen	T.I.

Contents

1 ABOUT THIS DOCUMENT

1.1 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
PE	Processor Element
Trace Decode Block	Decodes trace from multiple sources in a single capture buffer into generic trace elements.
Protocol Decode Block	Decodes trace from a single source into generic trace elements.
Packet Processor	Decodes raw byte data into individual trace protocol packets
Packet Trace Decoder	Decodes trace protocol packets into generic trace elements.
Trace Fame De-formatter	Decodes multiplexed trace sources in the CoreSight frame format into individual trace data streams associated with a trace ID.
Trace Decode Tree	The hierarchical structure of the trace decode block. A single capture buffer source fans out into multiple individual trace sources.

2 COMPONENTS AND ARCHITECTURE

2.1 Architecture and Component Design Summary

2.1.1 Protocol Decode Block

The principle component in the trace decode architecture is the Protocol decode block. This block takes in a stream of CoreSight trace bytes from a given single source ID generated by some hardware protocol generator (e.g. ETM), and converts these into a protocol agnostic set of generic trace elements.

The client application of the RCTDL uses these generic trace elements to perform the required analysis function, for example create a display the trace flow in the program.

The protocol decode block consists of two elements:-

- A **packet processor** – this takes the incoming byte stream and converts this into individual protocol packets.
- A **packet trace decoder** – this further interprets the combinations of packets to generate the set of generic trace elements.

  For example, when the trace source is a core then the decoder will generate generic trace elements that describe the instructions executed and the core state and flow.

Both the packet processor and the packet trace decoder within the protocol decode block will require configuration information from the CoreSight system to determine the operating mode of the protocol generator.

The packet trace decoder for a core trace source will further require access to the memory image and some basic level of instruction decode to determine instruction execution flow for the trace.

2.1.2 Trace Frame De-formatter

Multiple trace sources are combined in hardware by the CoreSight formatter into 16 byte trace frames. The trace frame de-formatter is provided to split the CoreSight formatted trace frames into individual trace streams.

A protocol decode block can be attached to the trace de-formatter for each trace source ID in use.

When processing trace, data from source IDs that do not have an attached decode block will be ignored.

2.1.3 Trace Decode Component Design

The trace decode components follow a common design structure.

- Components implement defined virtual interfaces as data input points.
- Components provide attachment points for specified interfaces as data output points.
- Additional attachment points are provided for additional functionality required for the correct decode of trace, such as target memory access and decode, alongside ancillary functionality such as indexing and error logging.

This structure allows the construction of a ***trace decode tree***, with components connected to decode all desired trace sources from a single trace buffer.

2.1.4 Trace decode block – trace data path

The diagram below shows the arrangement of trace decode components and principle data path through a single branch of a decode tree. The diagram shows the key interfaces and attachment points used by the components.

The analysis program takes raw data from a trace source and passes it through the decoder, resulting in generic trace elements appearing at the decode output.

The trace data path also passes the trace index for any data to allow indexing and relating output trace elements to their generating trace input data. The trace index is the byte position in the trace buffer of any packet used to generate trace elements.

A protocol decode block can be attached to the trace de-formatter for all trace source IDs in use by the system.

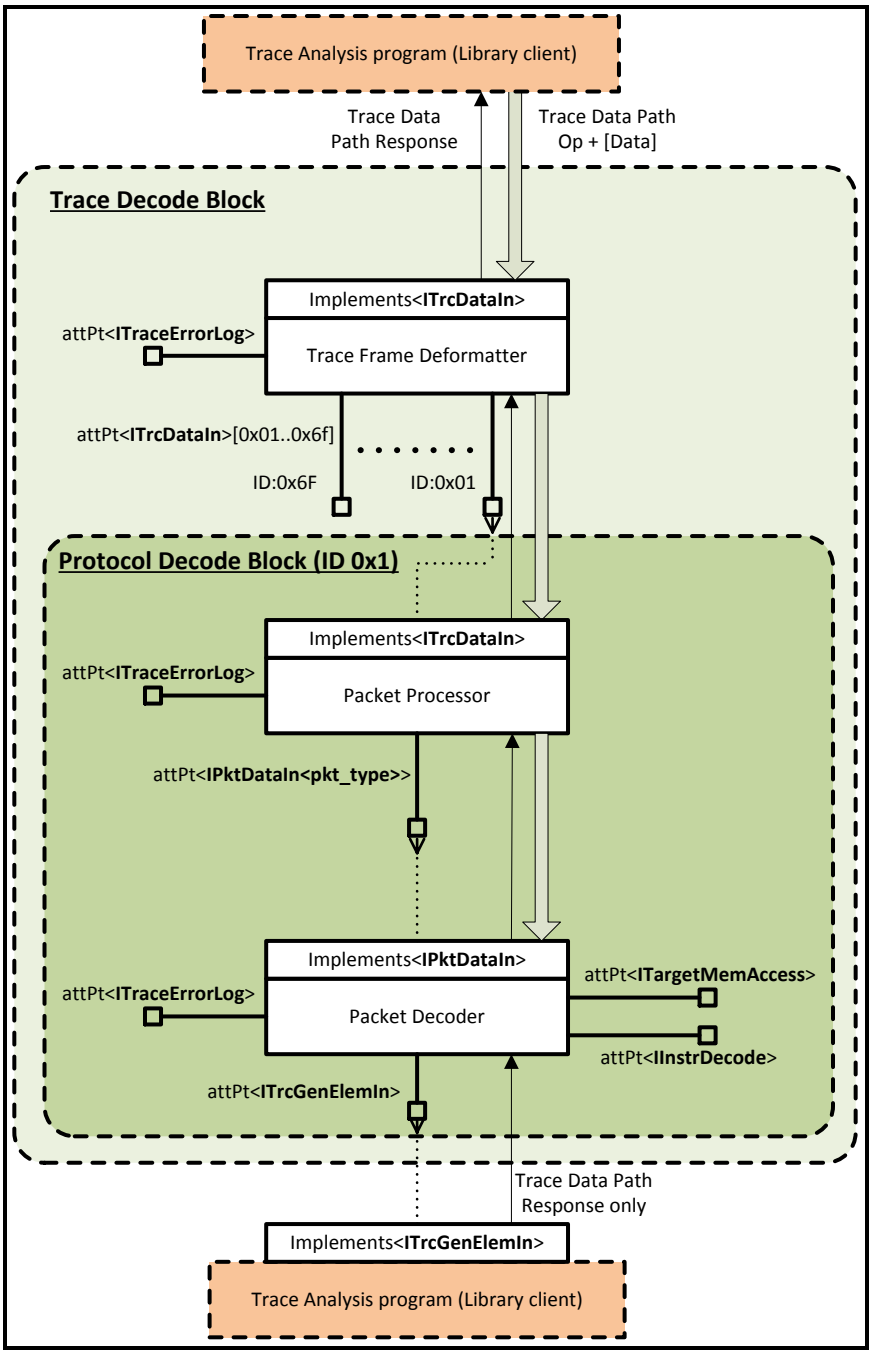


Figure 1 : Trace Decode Block API structure

Flow control through the decoder is achieved by the trace data path input operation defined at the input to the decoder. This can be a normal data operation, a flush operation or decoder reset.

The system will process the data and provide a data path response which can continue processing, pause processing or signal a fatal error. This response will determine the next operation that should be used.

This flow control is necessary as a single packet from the packet processor can result in multiple generic trace elements at the decoder output.

These operations and responses are shown in the tables below.

Table 1 : Trace Datapath Operations

Data Path Operation Code	Description
RCTDL_OP_DATA	This is the normal operation. Data is passed through the decoder
RCTDL_OP_EOT	This call marks the end of available trace from the source buffer.
RCTDL_OP_FLUSH	This call tells the system to flush data after a previous response required a pause. Flush operations continue decode from the paused state.
RCTDL_OP_RESET	Resets the decoder to initial state. All current state data is lost.

The response codes are arranged in groups.

Table 2 : Trace Datapath Response Codes

Data Path Response Code Group	Description
RCTDL_RESP_CONT	Processing can continue with data operations.
RCTDL_RESP_WAIT	Call decoder with flush operation until a RCTDL_RESP_CONT code is returned.
RCTDL_RESP_FATAL	Decode fatal error. No further processing is possible.

The interface functions on the trace data path take an operation code as a parameter and return the response code.

2.1.5 Generic Trace Elements

The output of the trace decoder will be generic trace elements. These will be protocol agnostic, allowing the trace analysis program to determine the instruction flow for a PE trace source, or look at the software stimulus payloads for an STM/ITM trace source.

Custom trace protocols will be expected to generate these same set of trace elements allowing tools to work irrespective of the trace source hardware.

The precise structure of these elements is still to be determined but the following types have been identified as required.

Table 3 : Decoder Output - Generic Trace Elements

Trace Element Type	Description
RCTDL_GEN_TRC_ELEM_INSTR_RANGE	Traced N consecutive instructions from address. (No intervening events or data elements.) ISA / PE state as previous update.
RCTDL_GEN_TRC_ELEM_PE_STATUS	Update of PE status – ISA, context ID, VMID etc.

Trace Element Type	Description
RCTDL_GEN_TRC_ELEM_ADDR_NACC	PE trace went out of available target address range.
RCTDL_GEN_TRC_ELEM_DATA_VAL	Data value - associated with prev instr (if same stream) + daddr, or data assoc key if supplied.
RCTDL_GEN_TRC_ELEM_DATA_ADDR	Data address - associated with prev instr (if same stream), or data assoc key if supplied.
RCTDL_GEN_TRC_ELEM_TIMESTAMP	A timestamp.
RCTDL_GEN_TRC_ELEM_CYCLE_COUNT	A cycle count if PE is being traced cycle accurate.
RCTDL_GEN_TRC_ELEM_EVENT	Event - trace on, reti, trigger, (TBC - perhaps have a set of event types - cut down additional processing?)
RCTDL_GEN_TRC_ELEM_SWCHAN_DATA	Data out on a SW channel (master, ID, data payload, type/size).
RCTDL_GEN_TRC_ELEM_BUS_TRANSFER	Bus transfer event from a bus trace module (HTM)
RCTDL_GEN_TRC_ELEM_EO_TRACE	End of trace buffer.

2.2 Full Architecture and Component Infrastructure.

The complete library infrastructure is shown in the diagram below.

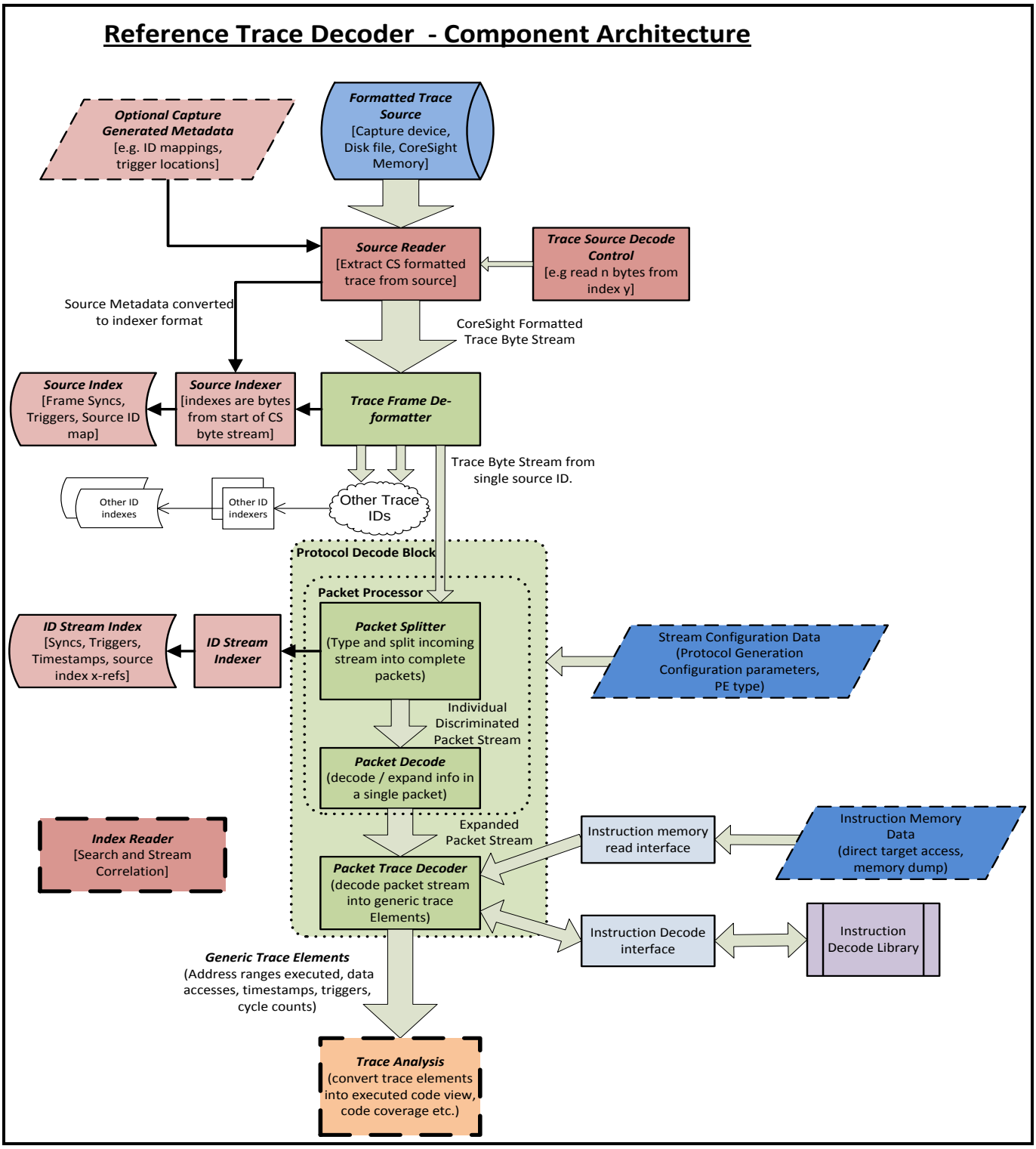


Figure 2 : Reference Trace Decoder Component Architecture

The principle component in the trace decode architecture is the Protocol Decode block as previously described. This block takes in a stream of CoreSight trace bytes from a given source ID generated by some hardware protocol generator (e.g. ETM), and converts these into a protocol agnostic set of generic trace elements.

The client application of the RCTDL uses these trace elements to perform the required analysis function, for example, to create a display the trace flow in the program.

The protocol decode block will require configuration information from the CoreSight system to determine the operating mode of the protocol generator, and further will require access to the memory image and some basic level of instruction decode to determine instruction execution flow for the trace.

Where trace data consists of multiple trace sources having been passed through a CoreSight trace formatter, the trace frame de-formatter is provided to split the CoreSight formatted trace frames into individual trace streams.

The library provides some ancillary functionality to standardise trace source readers, provide indexing of trace IDs and sources allowing the client application to efficiently search and identify points of interest within the captured trace data - for example trace trigger points / events, or correlate trace sources according to timestamps.

2.3 Component Descriptions

2.3.1 Protocol Decode Block

This decodes the protocol from the trace protocol generator such as and ETM or STM.

The protocol decode is split into two components, packet interpreter and trace decoder.

2.3.1.1 Packet Processor

This takes in a trace data byte stream for a single ID from the frame decoder, or alternatively a known single ID source byte stream, and builds complete packets from the incoming stream.

The packet processor then converts the complete raw packet into a fully decoded packet.

This will decompress any information contained in the packet (e.g. address values etc.) and present the decoded packet to the next decode stage.

The packet processor will maintain any intra packet data necessary to ensure correct packet decode. For example the ETM may compress address data to only bits that have changed since the last address output.

The packet processor will require protocol generator configuration information in order to decompress packets.

The packet processor may have a stream ID indexer attached. This will record the source start index of significant protocol generator packet types (e.g. Synchronisation packets, Timestamp packets, etc.).

An incomplete packet may be present in the processor any point during the decode operation. This will be dropped if the decode is flushed or restarted.

The interpreter will require protocol generator configuration information in order to determine packet types and boundaries.

The packet interpreter will provide attachment points for:-

- Appropriate trace decoder for the complete expanded packet (PE or SW Stimulus decoder).
- Packet type indexer – provides index for later stream correlation.
- Raw packet monitor – outputting just type, size and packet data.

2.3.1.2 Packet Trace Decoder – PE trace

A trace generator (ETM, PTM) attached to a PE will require a further stage of decode to allow the packets to be decompressed into a set of executed instructions and when present, associated data events. This decompression will be presented to the analysis block as a series of generic trace elements. This removes any generator protocol from the trace stream.

Instruction decompression will require access to the target memory either directly or in the form of a memory dump file. This will allow the decompressor to follow the instruction path and determine the number of instructions executed for a given trace packet or packets.

Instruction decompression will also require access to at least partial instruction decode, sufficient to allow the PE decode to follow the instruction stream and determine subsequent instruction addresses. This decode will be provided through an instruction decode library.

2.3.1.3 Packet Trace Decoder - Software Stimulus

Software stimulus trace generators (STM, ITM) have an associated decoder that will output SW stimulus generic primitives. This will decode the packet stream into generic trace objects combining information regarding the source ids (e.g. STM master and channel IDs), along with the data payload. This will remove any packet protocol specific information from the SW stimulus source, providing a common output format.

Timestamp objects and stream event and error objects will also be generated.

2.3.1.4 Instruction Memory Read interface.

The instruction memory read interface will allow the P.E. decode component to access areas of target memory required for reading the instruction opcodes for decode.

2.3.1.5 Instruction Decode Interface and Library.

The PE decoder will require an instruction decode service to establish the flow of instructions being traced.

Full decode is not required, a sub-set of information regarding the instruction type and address of the next instruction is sufficient to follow the instruction trace flow.

The instruction decode could be provided by an external library or component, with the interface specified to pass back only the information required for decode.

2.3.2 Trace Frame De-formatter

This component takes a stream of CoreSight trace bytes that are formatted into frames according to the CoreSight formatter specification.

This will then split this data into a stream of byte for each single trace ID.

Attachment points are provided for the next stage decode components as follows:-

- Each valid trace source ID.
- An attachment point for unpacked trace frame elements. This is to allow the analysis or display of the raw packed trace stream.

Additionally the component will have an attachment point for a trace source indexer.

The decoder will require information on the trace capture component – if the source has frames aligned to memory locations, with no frame sync packets (an ETB or other on target memory capture device), or contains frame synchronization packets (a CoreSight TPIU).

2.3.3 Ancillary Support Components.

2.3.3.1 Source Indexer.

The source indexer will generate an index of key events in the source trace along with a mapping of the trace IDs present in the source. This is the main source index.

Index values are byte references from the start of the trace source data.

Indexes are arranged hierarchically. The main source index is a parent index for Trace ID stream indexes.

The ID mapping granularity will be configurable with smaller granularity generating a larger index database. Smaller granularity allows faster location of significant entries.

The source indexer will note the location of any events/triggers marked by special trace IDs in the formatted trace frame.

Formatted trace sources may contain frame synchronisation packets – if the source is a CoreSight TPIU. These will be periodically noted in the index. Where the formatter source is memory based, (ETB / ETR), then there are no frame syncs as frames are memory aligned.

The source index will contain metadata describing the total amount of trace data present and presence of synchronisation frames.

The source index format will support partial indexing, and allow the trace reader components to determine indexed and none index portions of the source trace.

The source index may be created in memory and / or serialised to data file on disk.

2.3.3.2 Trace ID stream Indexer.

The trace ID stream index will contain significant elements common to all trace streams (timestamps, events/triggers, synchronisation points) plus optionally protocol specific elements where indexing would be beneficial.

Index values are byte references from the start of the trace source data, not the beginning of data for this ID. This allows the elements to be located within the main trace source index.

The trace ID stream index will contain metadata indicating the trace ID and protocol that generating the index.

Where the trace data source is an unformatted single stream then the trace ID will be unused. The indexes will be references to the start of the source data, but there will be no main source index generated.

The trace ID stream index can either be a full index, where significant element instances are recorded; or a sparse index, where numbers of elements and key values are noted for a block of trace.

e.g. for a block of 64k source trace, the number of timestamps, plus the first and last values in the block could be recorded. This will allow an analysis program to quickly find a block then start decode from the start of the block.

The trace ID stream index may be created in memory and / or serialised to data file on disk.

2.3.3.3 Index Reader.

The index reader will be able to load the indexes for a given trace source.

An API will be provided to allow the trace analysis applications to search indexes for desired events or significant trace elements.

2.3.3.4 Error Reporting.

All components will be able to connect to an error reporting component via a standard interface.

A component implementation can save only the last error, save all errors to disk or print to screen.

A standard decode error / warning format will be defined, that will contain the source index, stream ID (if present in the component) and further error information. This will allow any decode issues to be associated with a specific part of the incoming decode stream.

The verbosity of the warnings and errors will be configurable.

Components will be designed to ensure that comprehensive error and warning output is available when required.

2.3.3.5 Decode Tree Configuration.

An arrangement of components to perform a decode operation is referred to as a decode tree, An API will be provided to allow analysis programs to build decode trees according to their requirements.

The API will allow the configuration of the inter component connections, configuration of the decode components and attachment to any additional components required for the decode task.

3 API AND CLASS DEFINITIONS AND DESCRIPTIONS

3.1 Key Data Path Interface APIs

Latest information on these APIs can be obtained by generating the Doxygen documentation from the source code.

3.1.1 ITrcDataIn

```
rctdl_datapath_resp_t ITrcDataIn::TraceDataIn( const rctdl_datapath_op_t op,
                                               const rctdl_trc_index_t index,
                                               const uint32_t dataBlockSize,
                                               const uint8_t *pDataBlock,
                                               uint32_t *numBytesProcessed)
```

op : Data path operation.
index : Byte index of start of pDataBlock data as offset from start of captured data. May be zero for none-data operation
dataBlockSize : Size of data block. Zero for none-data operation.
***pDataBlock** : pointer to data block. Null for none-data operation
***numBytesProcessed** : Pointer to count of data used by processor. Set by processor on data operation. Null for none-data operation

return **rctdl_datapath_resp_t** : Standard data path response code.

Data input method for a component on the Trace decode data path. Data path operations passed to the component, which responds with data path response codes.

This API is for raw trace data, which can be:-

- CoreSight formatted frame data for input to the frame deformatter.
- Single binary source data for input to a packet decoder.

3.1.2 IPktDataIn

```
rctdl_datapath_resp_t IPktDataIn::PacketDataIn( const rctdl_datapath_op_t op,
                                                const rctdl_trc_index_t index_sop,
                                                const P *p_packet_in)
```

op : Data path operation.
index_sop : Trace index for the start of the packet, 0 if not RCTDL_OP_DATA.
***p_packet_in** : Protocol Packet - when data path operation is RCTDL_OP_DATA. Null otherwise. Parameterised in template base classes.

return **rctdl_datapath_resp_**t : Standard data path response.

Interface function to process a single protocol packet. Output attachment point for trace packet processors.

Pass a trace index for the start of packet and a pointer to a packet when the datapath operation is RCTDL_OP_DATA.

3.1.3 ITrcGenElemIn

```
rctdl_datapath_resp_t ITrcGenElemIn::TraceElemIn(const rctdl_trc_index_t index_sop,
                                                 const RctdlTraceElement &elem) = 0;
```

index_sop : Trace index for start of packet generating this element.
&elem : Generic trace element generated from the deocde data path

return **rctdl_datapath_resp_t** : Standard data path response.

Interface for analysis blocks that take generic trace elements as their input. Output attachment point for trace packet decoders.

Final interface on the decode data path. The index provided is that for the generating trace packet. Multiple generic elements may be produced from a single packet so they will all have the same start index.

3.1.4 ITargetMemAccess

```
rctdl_err_t ITargetMemAccess::ReadTargetMemory(const rctdl_vaddr_t address,
                                               uint32_t *num_bytes,
                                               uint8_t *p_buffer);
```

address : Address to access.
num_bytes : [in] Number of bytes required. [out] Number of bytes actually read.
***p_buffer** : Buffer to fill with the bytes.

return **rctdl_err_t** : RCTDL_OK on successful access (including memory not available)

Read a block of target memory into supplied buffer. Bytes read set to 0, along with a success return code indicates memory location not accessible.

Read Target memory call is used by the decoder to access the memory location in the target memory space for the next instruction(s) to be traced.

Memory data returned is to be little-endian.

The implementor of this interface could either use file(s) containing dumps of memory locations from the target (trace capture snapshots), be an elf file reader extracting code, or a live target connection, depending on the tool execution context.

3.1.5 IInstrDecode

```
rctdl_err_t IInstrDecode::DecodeInstruction(rctdl_instr_info *instr_info)
```

***instr_info** : Structure to pass current opcode, and receive required decode information.

return **rctdl_err_t** : RCTDL_OK if successful.

Instruction opcode decode for the packet trace decoder to follow the instruction execution flow. The **`rctdl_instr_info`** structure has an input section containing PE ISA information and opcode, filled in by the caller, and an output section containing an instruction type and next address filled in by the callee.

The opcode decoder implementing this interface needs to be capable of limited decode required for trace execution flow determination.

3.2 Key Class Descriptions.

Brief class descriptions and usage for key classes in the library.

Class	Operation and Usage
TraceComponent	Common base for all decode components. Provides the **ITraceErrorLog** attachment point and functionality for error logging. Also provides a standard interface for setting component operational flags.
TrcPktProcBase	Template Base class for all trace packet processors. Implements the **ITrcDataIn** interface. Provides attachment points for **IPktDataIn** interface and an attachment for an optional packet indexer. Contains the interface to set the protocol configuration for the decoder. Implements basic processing logic based on the incoming data path operation, calling packet processing implementation interface functions to execute the processing operations. Derived packet processors provide the specific packet data classes and configuration classes, and override the implementation packet processing interface functions.
TrcPktDecodeBase	Template base class for the packet decoders. Implements the **IPktDataIn** interface. Provides attachment points and access to the output **ITrcGenElemIn** interface as well as the **ITargetMemAccess** and **IInstrDecode** interfaces as required. Derived packet decoders provide the specific packet data class.
TrcPktProcPtm	Packet processor for PTM. Overrides **TrcPktProcBase** providing **PtmTrcPacket, rctdl_ptm_pkt_type**, and **PtmConfig** as the packet class, packet type class and protocol configuration class.
TrcPktProcEtmV3	Packet processor for ETM v3. Overrides **TrcPktProcBase** providing **EtmV3TrcPacket, rctdl_etmv3_pkt_type**, and **EtmV3Config** as the packet class, packet type class and protocol configuration class.
TrcPktProcEtmV4I	Packet processor for ETMv4 instruction trace stream. Overrides **TrcPktProcBase** providing **EtmV4ITrcPacket, rctdl_etmv4_i_pkt_type**, and **EtmV4Config** as the packet class, packet type class and protocol configuration class.
TrcPktProcEtmV4D	Packet processor for ETMv4 data trace stream. Overrides **TrcPktProcBase** providing **EtmV4DTrcPacket**, **rctdl_etmv4_d_pkt_type**, and **EtmV4Config** as the packet class, packet type class and protocol configuration class.

4 THE ARM SNAPSHOT FORMAT

4.1 Summary.

To ease testing and re-use of the RCTDL it is proposed that a trace “snapshot” format is used for interchange of captured trace data for decode offline. This will allow testing during the development phase and provide a library of standard test examples for regression testing.

The format allows the description of the trace component hierarchy and the relationship between trace sources, trace data buffers, and captured memory dumps in a more comprehensive and accurate manner than the previous format.

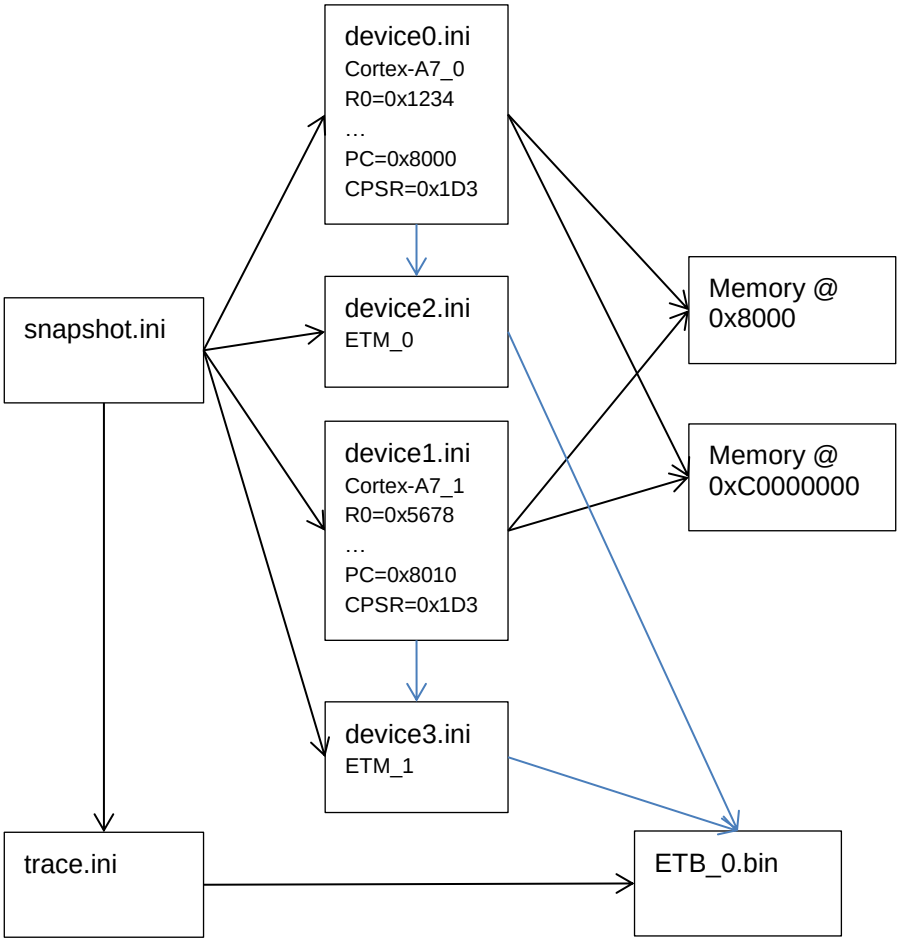
The following description outlines the current format used by the DS-5 debugger. This format is used for both trace decode and to display in DS-5 the current state of a core. As such there are elements of the format that will not be required for trace decode, but the overall structure is relevant.
For example, the device files describing cores have a minimum requirement to provide certain core register values. These are not normally required for trace decode, but are required to load a snapshot into DS-5.
The device file for a trace source, such as an ETM, will require the specified registers to be present as part of the trace decode.

Additionally, the ARM CoreSight Access library, which can be used to program trace sources as part of runtime Linux applications, already contains functionality to dump trace in this format.

4.2 Snapshot Format Descriptions.

The snapshot consists of the following files:

- snapshot.ini: Provides the index of available debug devices and topology
- Device files: Provides register and memory information for each debug device
- trace.ini: Trace metadata. Provides information on the trace topology and buffers
- Trace buffer contents: Files containing trace data.
- Memory region contents: Files containing device memory data.



4.2.1 snapshot.ini

This file uses the ini file format and consists of the following sections:

- "snapshot" (required)
- "device_list" (required)
- "clusters" (optional)
- "trace" (optional)

4.2.1.1 "snapshot" section

This provides information about the snapshot. This section can contain the following keys:

- "version" (required): The version of the snapshot. The interpretation of the rest of the snapshot depends on the value of this field: readers of the snapshot must check this field first. The only valid value for this is currently "1.0".
- "description" (optional): Description of the contents of this snapshot.

4.2.1.2 "device_list" section

Each value in this section is a reference to an INI file that defines the state (registers and memory) of a device. The key names aren't used, but must be unique. It is recommended that they are "device0", "device1", ..."deviceN". The path to the ini file is relative to the snapshot.ini file.

4.2.1.3 "clusters" section

Each entry in this section defines a cluster. The key is the name of the cluster, the value is a comma separated list of devices in that cluster. If this section is not present, all cores are placed in one cluster.

4.2.1.4 "trace" section

May be omitted if snapshot does not contain trace data. This section can contain the following keys:

- "metadata" (required): relative path to another ini file that defines the metadata describing the trace buffers and trace source associations

4.2.1.5 Example

```
[snapshot]
version=1.0
description=Example snapshot

[device_list]
device1=cortex-a15_0.ini
device2=cortex-a15_1.ini
device3=PTM_0.ini
device4=PTM_1.ini
device5=ITM_0.ini
device6=cortex-a7_0.ini
device7=cortex-a7_1.ini
device8=cortex-a7_2.ini
device9=ETM_0.ini
device10=ETM_1.ini
device11=ETM_2.ini

[clusters]
Cluster 0=cortex-a15_0,cortex-a15_1
Cluster 1=cortex-a7_0,cortex-a7_1,cortex-a7_2

[trace]
metadata=trace.ini
```

4.2.2 Device files

These are .ini files with the following sections:

- "device": Provides information about the device
- "regs": Provides register values for the device
- "dump": 0 or more sections, each describing a memory region

4.2.2.1 "device" section

This section contains the following keys:

- "name" (required): the value of this is the name that will be reported from RDDI and used to identify the device. Each device in the snapshot must have a unique name. There is no requirement for this to match a core type (e.g. Cortex-A15) - Core_0, Core_1, ... will be acceptable.
- "class" (optional): the general type of the device. Recognised values for this are:
  - "core": Indicates that this device represents a core

    - “trace_source”: Indicates that this device represents a trace source
    - “memory_space”: Indicates that this device represents a memory space, e.g. a view of an AHB or AXI bus. A debugger may present the device as an additional address space.
    - Other values are permitted for information purposes
- “type” (optional): the specific type of the device.
    - For cores this is the name of the core, e.g. “Cortex-A9”, “Cortex-A57”, “ARM1136JF-S”. If the specific core type is not known, the architecture version may be used, e.g. “ARMv7-A”, “ARMv8-A”. This is not case sensitive. A debugger can use this value to determine which register set to load, what architecture features to support etc.
    - For trace sources this is the name of the trace protocol, e.g. “ETM”, “PFT”, “ITM”, “STM” concatenated with the version of the protocol used, e.g.: “ETM3.3”, “ETM4.0”, “PFT1.1”
- “location” (optional): This describes how the agent that produced the snapshot locates the device. It consists of a comma separated sequence of key:value entries that define a unique path to the device. For an external debugger, this could consist of the DAP number, AP number and AP base address (e.g. dap:2,ap:1,address:0x80010000) of the device. For self-hosted debug, this could consist of the base address of the device in the core’s memory space (e.g. address:0x1200010000). If specified, the location must be unique within the snapshot.

Any other keys in this section are ignored – this allows extra client specific data to be stored.

4.2.2.2 “regs” section

Each key in this section consists of the register name, with optional extra information in parentheses. The extra information is a comma separated list of keys and values, with ‘:’ separating keys from values. Supported extra information is:

- “size”: The size in bits of the register
- “id” or unkeyed: The ID number of the register. May be hex with 0x prefix or decimal

For example:

```
REG_A(77) = 0x1234
REG_B(id:78) = 0x1234
REG_C(id:0x80,size:64) = 0x1234000012340000
REG_D(size:64,0x82) = 0x12340000123400000
```

The values are integers of the appropriate size (32-bits if no size is specified).

For CoreSight components, the register IDs can be converted to address by adding (ID * 4) to the base address of the component.

4.2.2.3 “dump” sections

0 or more dump sections may occur in the device file. Each defines a region of memory visible from that device. Each section can contain the following keys:

- “file”: relative path to the file containing the memory contents
- “space” (optional): address space of the region. Supported values are: “N”, “S”, “H”, “EL1N”, "EL1S", "EL2", “EL3”
- “address”: address of the region
- “length” (optional): length of the region – must be less than or equal to the file size
- “offset” (optional): offset into the file

4.2.2.4 Example

This example represents a core with the code and data sections of an image visible in the memory

```
[device]
name=cpu_0
class=core
```

```
type=Cortex-A7
location=address:0x1200013000

[dump]
file=ER_RW.bin
space=N
address=0x8000DD90
length=0x00000020

[dump]
file=ER_RO.bin
space=N
address=0x80001000
length=0x0000CD90

[dump]
file=STACK.bin
space=N
address=0x8001D748
length=0x00000800

[dump]
file=HEAP.bin
space=N
address=0x8000DF48
length=0x0000F7FC

[regs]
R0=0x00000000
R1=0x00000000
R2=0x10001060
R3=0x00000000
R4=0x80011990
R5=0x80011990
R6=0x00000028
R7=0x80010120
R8=0x00000000
R9=0xFFFFFFFF
R10=0x8000D53C
R11=0x00000000
R12=0x00000000
SP=0x8001DF20
LR=0x80001133
PC=0x8000A5F8
CPSR=0x600001D3
D0(id:90,size:64)=0x123456700000000
D1(92,size:64)=0x123456700000000
```

4.2.3 Trace metadata

The trace metadata file, typically “trace.ini”, will contain data required to describe the topology of the system and interpret the trace buffers. This file contains the following sections:

- “trace_buffers” (required): Describes the trace buffers present in the snapshot
- Trace buffer metadata: Metadata about each trace buffer
- “core_trace_sources” (optional): Defines the associations between cores and trace sources
- “source_buffers” (required): Defines which trace buffer contains data for each trace source

4.2.3.1 “trace_buffers” section

The “trace_buffers” section describes the trace buffers present in the snapshot. This section has the following keys:

- “buffers” (required): A comma separated list of buffer IDs. Each ID should be unique and must not conflict with other sections in this file (“core_trace_sources”, “source_buffers”). Typically these will be “buffer0”, “buffer1”, .., “bufferN”. The metadata for each buffer will be in a section name with this ID.

4.2.3.2 Trace buffer metadata sections

Each buffer section can contain the following keys:

- “name” (required): The unique name of this buffer. This may be displayed to the user.
- “file” (required): This entry is a comma separated list of file paths relative to the snapshot.ini that contain the trace data. Multiple files allow the buffer size to exceed the practical file size. Where multiple files are used, the buffer data is the concatenation of the files in the order given, and the buffer size is the sum of the file sizes.
- “format” (required): The format of the data. This will be one of:
    - “coresight”: the data is formatted in 16 byte CoreSight frames
    - “source_data”: the data is unformatted (and only contains data for a single source)
    - Other values for format (e.g. DSTREAM’s raw contents) are supported, but these are not expected to be used.

4.2.3.3 “core_trace_sources” section

The associations between cores and their trace sources are defined in a section called “core_trace_sources”. Each entry in this section defines the trace source for a core. The key is the name of the core, the value is either:

- the name of the trace source.
- “@” followed by the value of “location” of the trace source, e.g.

```
[core_trace_sources]
cortex-a7_0=@address:0x12308000
```

4.2.3.4 “source_buffers” section

Where multiple trace buffers are present, the “source_buffers” section defines which buffers the data for each trace source may be found in. Each entry defines the buffers that may contain data for a trace source. The key is the name of the trace source, the value is a comma separated list of buffer names. Where data for a given source may be present in more than one buffer (e.g. a snapshot from system that replicates trace data into an ETB and TPIU), a debugger may present the user with a choice or select one buffer (DS-5 5.21 will select the first).

This section may be omitted if there is only one trace buffer, in which case all trace sources shall use that buffer.

Where a trace source has multiple streams (e.g. ETMv4 has separate instruction and data streams), a snapshot may have the data each stream in different buffers. To specify the buffers, the source may be suffixed with “(stream:N)”, where N is the stream number (i.e. to offset to the base ATB ID). For example with an ETMv4 with name “ETM_0”, this will be “ETM_0(stream:0)” (instruction) and “ETM_0(stream:1)” (data). If a specific buffer is not specified for a stream, then the unqualified source name should be used.

4.2.3.5 Example

This example shows trace of a two cluster system (2x Cortex-A15 and 3x Cortex-A7) where each cluster traces into a different ETB. There is also an ITM (not associated to any core) that traces into ETB_0

```
[trace_buffers]
buffers=buffer0,buffer1

[buffer0]
name=ETB_0
file=ETB_0.bin
format=coresight

[buffer1]
name=ETB_1
file=ETB_1.bin
format=coresight

[core_trace_sources]
cortex-a7_0=etm_0
cortex-a7_1=etm_1
cortex-a7_2=etm_2
cortex-a15_0=ptm_0
cortex-a15_1=ptm_1

[source_buffers]
etm_0=ETB_0
etm_1=ETB_0
etm_2=ETB_0
ptm_0=ETB_1
ptm_1=ETB_1
itm_0=ETB_0
```

4.3 Required contents of device snapshot files.

Device.ini snapshot files have different requirements according to whether they represent a core, or a trace protocol source such as an ETM. The descriptions for the trace protocol sources provide the required register set needed for full trace decode. The core registers specified are optional in respect of trace decode – it should be sufficient to declare just the originating core type / architecture.

4.3.1 ETMv3

The following registers are required for trace decode

- ETMCR
- ETMCCER
- ETMIDR
- ETMTRACEIDR

4.3.2 PTM

The following registers are required for trace decode

- ETMCR
- ETMCCER

- ETMIDR
- ETMTRACEIDR

4.3.3 ETMv4

The following registers are required for trace decode

- TRCIDR0
- TRCIDR1
- TRCIDR2
- TRCIDR8
- TRCIDR9
- TRCIDR10
- TRCIDR11
- TRCIDR12
- TRCIDR13
- TRCTRACEIDR
- TRCCONFIGR
- TRCAUTHSTATUS

4.3.4 ITM CONTROL_REGISTER

The following registers are required for trace decode

- CONTROL_REGISTER: trace stream ID is taken from bits [22:16]

4.3.5 STM

The following registers are required for trace decode

- STMTCSR: trace stream ID is taken from bits [22:16]

4.3.6 ARMv7-A/R

- The core registers are named R0-R15, CPSR. SP, LR, PC may be used for R13-R15.
- SP, PC and CPSR are required.
- If security extensions supported, SCR is used to determine Secure / Non-secure state. A debugger will assume NS if not SCR is not present.
- If security extensions supported, valid memory spaces are “S” and “N”. If virtualization is supported, “H” is also valid. If no security or virtualization extensions are supported, no memory space prefix is used.

4.3.7 ARMv8

4.3.7.1 AArch64

- The core registers are named X0-X30, SP, PC. LR may be used for X30.
- CPSR, PC, SP are required to determine core state
- If security extensions supported, SCR is used to determine Secure / Non-secure state. A debugger will assume NS if not SCR is not present.
- Valid memory spaces are: “EL3”, “EL2”, “EL1S”, “EL1N”

4.3.7.2 AArch32

- The core registers are named R0-R15, CPSR. SP, LR, PC may be used for R13-R15.
- SP, PC and CPSR are required.
- If security extensions supported, SCR is used to determine Secure / Non-secure state. A debugger will assume NS if not SCR is not present.
- If security extensions supported, valid memory spaces are “S” and “N”. If virtualization is supported, “H” is also valid.

4.3.8 ARMv6-M/ARMv7-M

- The core registers are named R0-R15, xPSR. SP, LR, PC may be used for R13-R15.
- xPSR, PC, SP are required to determine core state
- Other NVIC registers may be present
- No memory space prefix is used.