|  |  |
| --- | --- |
|  | **Reference CoreSight Trace Decoder: API Specifications and Component Design**  **Debug and Trace**  **Development Solutions Group** |
| Document number: | ARM-ECM-0398495 Version 0.2 |
| Date of Issue: | 27/05/15 |
| Author: | Mike Leach |
| Authorised by: |  |

© Copyright ARM Limited 2015. All rights reserved.

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 3

1.1 References 3

1.2 Terms and abbreviations 3

2 Components and Architecture 4

2.1 Architecture and Component Design Summary 4

2.1.1 Protocol Decode Block 4

2.1.2 Trace Frame Deformatter 4

2.1.3 Trace Decode Component Design 4

2.1.4 Trace decode data path 4

2.1.5 Generic Trace Elements 6

2.2 Full Architecture and Component Infrastructure. 8

2.3 Component Descriptions 9

2.3.1 Stream Protocol Decode Block 9

2.3.1.1 Packet Processor 9

2.3.1.2 Packet Trace Decoder – PE trace 9

2.3.1.3 Packet Trace Decoder - Software Stimulus 10

2.3.1.4 Instruction Memory Read interface. 10

2.3.1.5 Instruction Decode Interface and Library. 10

2.3.2 Trace Formatted Frame Decoder 10

2.3.3 Ancillary Support Components. 10

2.3.3.1 Source Indexer. 10

2.3.3.2 Trace ID stream Indexer. 11

2.3.3.3 Index Reader. 11

2.3.3.4 Error Reporting. 11

2.3.3.5 Decode Tree Configuration. 11

3 API and Class Definitions and Descriptions 12

3.1 Key Data Path Interface APIs 12

3.1.1 ITrcDataIn 12

3.1.2 IPktDataIn 12

3.1.3 ITrcGenElemIn 13

3.1.4 ITargetMemAccess 13

3.1.5 IInstrDecode 13

3.2 Key Class Descriptions. 14

# About this document

## References

This document refers to the following documents.

|  |  |  |  |
| --- | --- | --- | --- |
| **Ref** | **Doc No** | **Author(s)** | **Title** |
|  |  |  |  |

## Terms and abbreviations

This document uses the following terms and abbreviations.

|  |  |
| --- | --- |
| **Term** | **Meaning** |
|  |  |
|  |  |
|  |  |
|  |  |

# Components and Architecture

## Architecture and Component Design Summary

### 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 primitives.

The client application of the RCTDL uses these trace primitives 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 primitives. Where the source is a core the decoder will be a PE decoder generating primitives that describe the instructions executed and the core state and flow.

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

### Trace Frame Deformatter

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

A decode block can be attached to the trace deformatter 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.

### 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 data tree, with components connected to decode all desired trace sources from a single trace buffer.

### Trace decode 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.

Packet processors and decoders can be attached to the trace deformatter for all trace source IDs in use by the system.



Figure : 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 : 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 : 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.

### 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 stream, or look at the software stimulus payloads for an STM/ITM trace stream.

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 : 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. |
| 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. |

## Full Architecture and Component Infrastructure.

The complete library infrastructure is shown in the diagram below.



Figure : Reference Trace Decoder Component Architecture

The principle component in the trace decode architecture is the Stream 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 primitives.

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

The protocol 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 formatted frame decoder 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.

## Component Descriptions

### Stream 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.

#### Packet Processor

This takes in a single trace ID data byte stream 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.

#### 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 primitives. 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.

#### 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.

#### 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.

#### 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.

### Trace Formatted Frame Decoder

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).

### Ancillary Support Components.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

# API and Class Definitions and Descriptions

## Key Data Path Interface APIs

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

### 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.

### 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.

### 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.

### 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.

### 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.

## 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. |