

Trace Configuration and Decoding using Command Line

Contents

1 Introduction

This chapter explains how to use the ARM v8 standalone configurators and decoders to process raw data trace. The objective of the component is to encapsulate all the functionality into one application, including the unformatted, decode, and export functionality. This component will be delivered as an independent product without any other libraries.

It provides details on the list of configurators and decoders required to collect and upload trace data.

- Configurators
- Decoders

2 Configurators

2.1 Introduction to Configurators

This chapter describes the following ARM v8 standalone configurators:

1	Introduction.....	1
2	Configurators.....	1
3	Decoders.....	19

Configurators

- [ARM v8 DDDI configurator](#)
- [ARM v8 PXDI configurator](#)
- [ARMv8 NoC configurator](#)
- [ARMv8 STM configurator](#)
- [Trace Complex 2 UI Configurator](#)

2.2 ARM v8 DDDI configurator

This section describes how to use the *DDDIConfigurator* tool to control the behavior of the DDDI Debug Trace Interface (DDDI). DDDI Configurator reads the user settings from the input XML file and transforms them into target access memory writes to the DDDI configuration registers.

2.2.1 Overview

The following image shows the DDDI trace path within the ARM v8 Architecture. The DDDI Configurator is responsible with configuring the modules highlighted by the red rectangle, DDDI1 and DDDI2 modules and NXC. NTC is a module that does not need any configuration. This tool does not configure all the trace modules that are required to output the DDDI trace to the debug probe or the trace buffer (Central Funnel and C-DTC). This task is performed by another tool that manages all trace sources: *Trace complex 2 Configurator*.

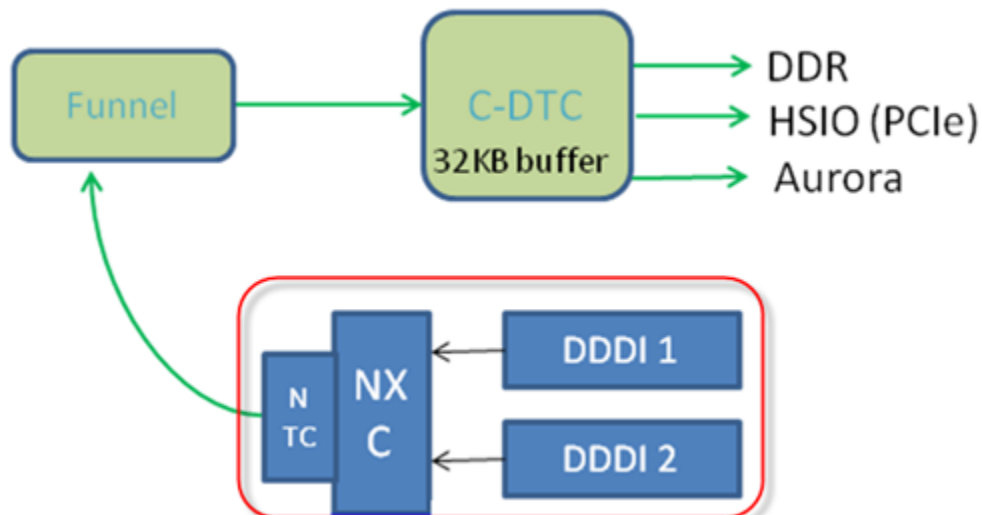


Figure 1. DDR trace path

The following settings can be configured for these entities:

- [DDDI1 and DDDI2](#)
- [NXC](#)

2.2.1.1 DDDI1 and DDDI2

- Enable/Disable Trace Generation
- Trace Mode: DDDI messages can be Terse or Verbose. *Terse* mode sacrifices visibility for reducing bandwidth and *Verbose* mode maximizes visibility at the cost of increased bandwidth. The following settings are available:
 - Verbose (highest bandwidth)

- Terse1
- Terse2
- Terse3 (lowest bandwidth)

2.2.1.2 NXC

- Enable/Disable Trace Generation
- Timestamp Enable with the following options:
 - 0 - Timestamp Disabled
 - 1 - DDDI 1 Timestamp Enabled, DDDI2 Timestamp Disabled
 - 2 - DDDI 1 Timestamp Disabled, DDDI2 Timestamp Enabled
 - 3 - DDDI 1 Timestamp Enabled, DDDI2 Timestamp Enabled
- Arbitration Mode of the ingress trace streams
 - Fixed Priority uses predefined priority: DDDI1 has a higher priority than DDDI2
 - Round Robin in an 8 state barrel shifter configuration.

2.2.2 Tool description

This section explains:

- [Usage](#)
- [XML configuration file](#)

2.2.2.1 Usage

```
dddi.config -h | -v | -c <xml_config_file>
```

Table 1. Usage Options

Option	Description
-h [--help]	Displays help message.
-v [--version]	Displays the product version.
-c [--config-file]	Displays the path off the xml configuration file

Example:

```
dddi.config -c LSDDRConfiguratorConfig.xml
```

2.2.2.2 XML configuration file

The configuration xml file contains all the settings described here for DDDI modules and NXC packed in configuration blocks.

Example of DDDI module configuration block

Configurators

```
<ConfigBlock>
  <Name>DDR Trace Config</Name>
  <BoolAttribute>
    <Name>DDDI1 Trace Enable</Name>
    <DefaultValue>true</DefaultValue>
  </BoolAttribute>
  <EnumAttribute>
    <Name>DDDI1 Trace Mode</Name>
    <DefaultValue>Verbose</DefaultValue>
    <EnumValues>Terse1, Verbose, Terse2, Terse3</EnumValues>
    <AllowMultiple>>false</AllowMultiple>
  </EnumAttribute>
  <BoolAttribute>
    <Name>DDDI2 Trace Enable</Name>
    <DefaultValue>true</DefaultValue>
  </BoolAttribute>
  <EnumAttribute>
    <Name>DDDI2 Trace Mode</Name>
    <DefaultValue>Terse2</DefaultValue>
    <EnumValues>Terse1, Verbose, Terse2, Terse3</EnumValues>
    <AllowMultiple>>false</AllowMultiple>
  </EnumAttribute>
</ConfigBlock> <!-- DDR Trace Config -->
```

Example of the NXC Configuration block

```
<ConfigBlock>
  <Name>NXC Trace Config</Name>
  <BoolAttribute>
    <Name>Trace Enable</Name>
    <DefaultValue>true</DefaultValue>
  </BoolAttribute>
  <Addr32Attribute>
    <Name>Timestamp Enable Mask</Name>
    <DefaultValue>0x0003</DefaultValue>
    <Addr32Range>
      <MinValue>0x0</MinValue>
      <MaxValue>0x1FFF</MaxValue>
    </Addr32Range>
  </Addr32Attribute>
  <EnumAttribute>
    <Name>Arbitration Mode</Name>
    <DefaultValue>Fixed Priority</DefaultValue>
    <EnumValues>Fixed Priority, Round Robin</EnumValues>
    <AllowMultiple>>false</AllowMultiple>
  </EnumAttribute>
</ConfigBlock>
```

Additionally there is a configuration block, *Target Access (TA)*, that specifies the way in which the tool connects to the target. Currently there is no target protocol available and the tool will only display the target accesses at **stdout** console. This is a stub TA protocol named *Logger*.

```
<ConfigBlock>
  <Name>Target Access</Name>
  <EnumAttribute>
    <Name>Target Access Method</Name>
    <DefaultValue>Logger</DefaultValue>
    <EnumValues>Logger</EnumValues>
    <AllowMultiple>>false</AllowMultiple>
  </EnumAttribute>
</ConfigBlock>
```

If a communication protocol is available (other than Logger), the DDDI Configurator will access the target and will perform the writes to the configuration registers.

Example stdout for Logger TA

```
write_mem 0x10012000 data=0x00000011 length=0x4
write_mem 0x10013000 data=0x00000021 length=0x4
write_mem 0x10012f48 data=0x00200000 length=0x4
write_mem 0x10013f48 data=0x00200000 length=0x4
write_mem 0x10002000 data=0x00000031 length=0x4
```

Each line describes a target access that the DDDI Configurator is performing including the Physical Address, the register size, how many registers are written, and the actual data to be written as an unsigned integer value.

2.3 ARM v8 PXDI configurator

This topic describes how to use the PCI-Express Debug Interface (PXDI) Configurator tool to control the behavior of the PXDI trace generator. PXDI Configurator reads the user settings from the input XML file and transforms them into target access memory writes to the PXDI configuration registers.

2.3.1 Overview

The following figure shows the PXDI trace path within the ARM v8 Architecture. The PXDI Configurator is responsible with configuring the modules highlighted by the red rectangle, PXDI_n (n form 1 to 3) modules and PXDI Funnel. This tool does not configure all the trace modules that are required to output the PXDI trace to the debug probe or the trace buffer (Central Funnel and C-DTC). This task is performed by another tool that manages all trace sources: *Trace complex 2 Configurator*.

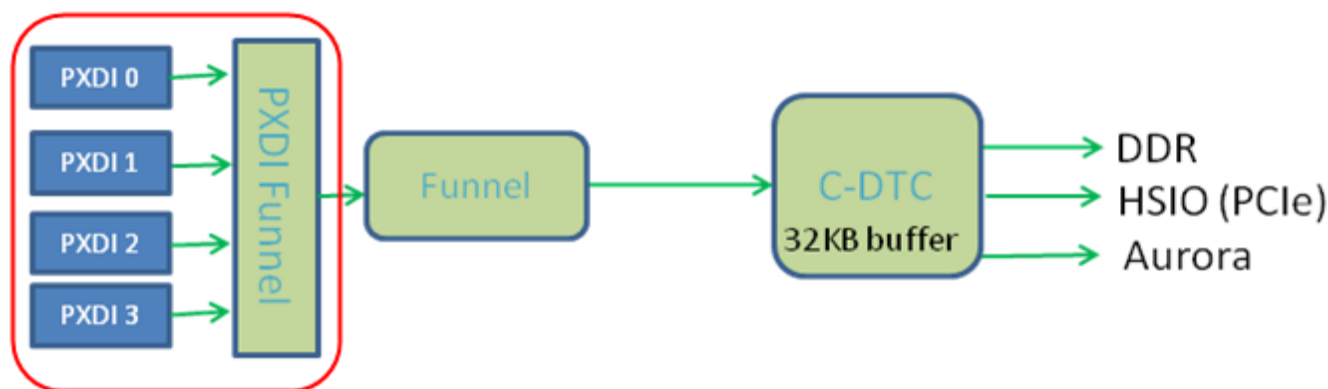


Figure 2. PXDI trace path

The following settings can be configured for these entities:

- [PXDI_n](#)
- [PXDI funnel](#)

2.3.1.1 PXDI_n

- *Enable/Disable* Trace Generation.
- *Debug Probe Select* is a number from 0 to 15, which specifies the set of signals that will be generated as trace data. The description of these sets can be found in the documents that depict the PXDI hardware modules.
- *Trace On Change* is an optimization specifying to generate trace only if incoming *Debug Probe* entry is different than previous entry.

2.3.1.2 PXDI funnel

- *Priority* of each PXDI ATB stream.
- *Minimum Hold Time* – the minimum number of cycles a trace source can transmit when a higher priority source becomes active during the transfer. The higher priority source will wait for the current source to generate this amount of cycles.

2.3.2 Tool description

This section explains:

- [Usage](#)
- [XML configuration file](#)

2.3.2.1 Usage

```
pxdi.configurator -h | -v | -c <xml_config_file>
```

Table 2. Usage options

Option	Description
-h [--help]	Displays help message.
-v [--version]	Displays the product version.
-c [--config-file]	Displays the path of the xml configuration file

Example

```
pxdi.configurator -c PXDI_config.xml
```

2.3.2.2 XML configuration file

The configuration xml file contains all the settings described here for PXDI modules and PXDI Funnel packed in configuration blocks.

Example of PXDI module configuration block

```
<ConfigBlock>
  <Name>PXDI0</Name>
  <BoolAttribute>
    <Name>Enable Trace</Name>
    <DefaultValue>true</DefaultValue>
  </BoolAttribute>
  <UInt32Attribute>
    <Name>Funnel Priority</Name>
    <DefaultValue>0</DefaultValue>
    <UInt32Range>
      <MinValue>0</MinValue>
      <MaxValue>7</MaxValue>
    </UInt32Range>
  </UInt32Attribute>
  <UInt32Attribute>
```

```

    <Name>Debug Probe Select</Name>
    <DefaultValue>0</DefaultValue>
    <UInt32Range>
        <MinValue>0</MinValue>
        <MaxValue>15</MaxValue>
    </UInt32Range>
</UInt32Attribute>
<BoolAttribute>
    <Name>Trace on Change</Name>
    <DefaultValue>true</DefaultValue>
</BoolAttribute>
</ConfigBlock> <!-- PXDI0 -->

```

Example of the PXDI Funnel Configuration block

```

<ConfigBlock>
    <Name>PXDI Funnel</Name>
    <UInt32Attribute>
        <Name>Minimum Hold Time</Name>
        <DefaultValue>3</DefaultValue>
        <UInt32Range>
            <MinValue>0</MinValue>
            <MaxValue>7</MaxValue>
        </UInt32Range>
    </UInt32Attribute>
</ConfigBlock>

```

Additionally there is a configuration block, *Target Access (TA)*, that specifies the way in which the tool connects to the target. Currently there is no target protocol available and the tool will only display the target accesses at **stdout** console. This is a stub TA protocol named *Logger*.

```

<ConfigBlock>
    <Name>Target Access</Name>
    <EnumAttribute>
        <Name>Target Access Method</Name>
        <DefaultValue>Logger</DefaultValue>
        <EnumValues>Logger</EnumValues>
        <AllowMultiple>false</AllowMultiple>
    </EnumAttribute>
</ConfigBlock>

```

If a communication protocol is available (other than Logger), the PXDI Configurator will access the target and will perform the writes to the configuration registers.

Example stdout for Logger TA

```

write_mem 0x10028000 data=0x00000000 length=0x4
write_mem 0x10029000 data=0x00000000 length=0x4
write_mem 0x1002a000 data=0x00000000 length=0x4
write_mem 0x1002b000 data=0x00000000 length=0x4
write_mem 0x10024000 data=0x00000305 length=0x4
write_mem 0x10024004 data=0x00fac688 length=0x4
write_mem 0x10028000 data=0x00000000 length=0x4
write_mem 0x10028004 data=0x00000001 length=0x4
write_mem 0x10028000 data=0x00000001 length=0x4
write_mem 0x10029000 data=0x00010000 length=0x4
write_mem 0x10029004 data=0x00000001 length=0x4
write_mem 0x10029000 data=0x00010000 length=0x4
write_mem 0x1002a000 data=0x000a0000 length=0x4
write_mem 0x1002a004 data=0x00000001 length=0x4
write_mem 0x1002a000 data=0x000a0001 length=0x4
write_mem 0x1002b000 data=0x000f0000 length=0x4
write_mem 0x1002b004 data=0x00000000 length=0x4
write_mem 0x1002b000 data=0x000f0000 length=0x4

```

Each line describes a target access that the PXDI Configurator is performing including the Physical Address, the register size, how many registers are written, and the actual data to be written as an unsigned integer value.

2.4 ARMv8 NoC configurator

This topic describes how to use the **Network-on-Chip (NoC) Configurator** tool to control the behavior of the NoC trace generator. NoC Configurator reads the settings from the input XML file and transforms them into target access memory and writes to the NoC configuration registers.

This section explains:

- [Overview](#)
- [Tool description](#)
- [Conclusions](#)

2.4.1 Overview

NoC technology uses task specific probes, placed on the user-specified probe points located on appropriate units of the datapath topology. Traced packets and observed events are gathered by observers, which can issue interrupts and forward detailed data to the supported third-party interfaces.

A **Probe** point is an intrusive or non-intrusive data collection device that provides observation subsystem units with information.

An **Observer** is a unit used to either log errors, or encapsulate traced packets of statistics packets in order to forward them to third-party interface.

The following figure illustrates the NoC trace generation and routing architecture in a LS2085A device. The Main NoC has 9 trace generators, called **Probe**. HSIO NoC has 3 trace generators, also called **Probe**. Each module has an **Observer**, **Main NoC** and **HSIO NoC**.

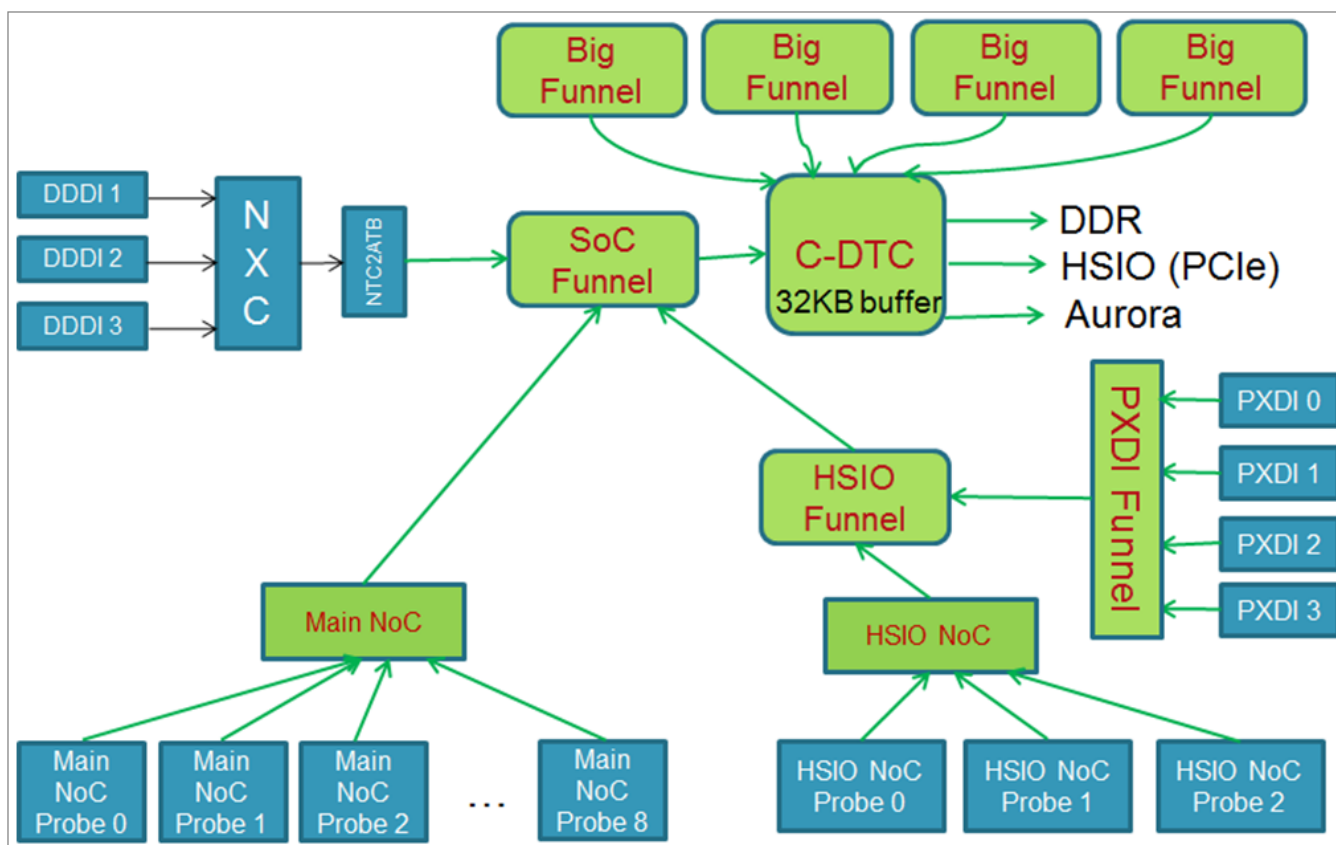


Figure 3. LS2085A Trace Generators

The NoC Configurator configures the following modules:

- Main and HSIO Probes
- Main and HSIO Observers
- SoC Funnel
- HSIO Funnel
- C-DTC (Central Debug Trace Controller)

The C-DTC stores and forwards trace data using a dedicated RAM buffer. This reduces trace loss by absorbing spikes in trace data. The destination of the raw trace is either the internal trace buffer of the C-DTC module or a user defined buffer in DDR.

The following settings can be configured per NoC Probe:

- Enable/Disable Trace Generation
- Enable/Disable Payload
- Enable/Disable Intrusive Mode

The following settings can be configured per data stream:

- Trace collection location - The options available are DTC and DDR.
- Trace Collection Mode - The options available are OneBuffer and Overwrite. Only one is selected at a time. If OneBuffer is selected, trace can only be collected only until buffer is filled. In this case only the first buffer size part of the trace is kept. If Overwrite is selected, in case buffer is filled the pointer returns to the beginning of the buffer and overwrites older trace. In this case only the last buffer size part of the trace is kept.

In case trace is collected in DDR buffer, the user has to set two more settings in the configuration file:

Configurators

- Trace Buffer Base Address - sets the address in DDR where the trace buffer starts
- Trace Buffer Size - sets the size of the DDR trace buffer

The DTC internal trace buffer is a DDR zone dedicated to this purpose.

2.4.2 Tool description

This section explains the following:

- [Usage](#)
- [XML configuration file](#)

2.4.2.1 Usage

The following command shows how to use the NoC configurator:

```
noc_tracetargetconfig -h | -v | -c <xml_config_file>
```

The following table lists the details of the command options:

Table 3. Usage options details

Options	Details
-h [--help]	Displays the help
-c [--config-file] path	Path to the xml configuration file.
-v [--version]	Displays the version information.

For example:

```
noc_tracetargetconfig -c NoCPlatformConfig.xml
```

2.4.2.2 XML configuration file

The configuration xml file contains all the settings described for NoC packed in configuration blocks. The configurator is following the multicore design. There is a single platform configuration file used by all the components to configure trace IP blocks. This file can be used for both configuring and collecting trace.

We identify the following components that need to be considered while configuring and collecting trace from a multicore system:

- Trace generator: It is an entity that can produce trace, for example NoC Probe.
- DataStream: It is the place where trace is collected in hardware, for example DTC and DDR.

The configuration xml file will have all the settings described for NoC module packed in configuration blocks.

The Trace Generator configuration block includes the configuration block for each Probe from NoC modules.

Below is the example of trace generators NoC MAIN Probe module configuration block:

Example of NoC Probe module

```
<ConfigBlock>
  <Name>acl</Name>
  <BoolAttribute>
    <Name>Enable Trace</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <BoolAttribute>
    <Name>Enable Payload</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <BoolAttribute>
    <Name>Enable Statistics</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <BoolAttribute>
    <Name>Enable Intrusive Mode</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
</ConfigBlock>
```

For LS2085A there will be 12 trace generators NoC probe, 9 Main NoC Probes and 3 HSIO NoC Probes.

For the NoC observers there will be a configuration block <NoC Modules> with two identical config block, <Main NoC Observer> and <HSIO NoC Observer>.

Below is the example of NoC Modules: Main NoC Observer module configuration block:

Example of NoC Observer module

```
<ConfigBlock>
  <Name>NoC Modules</Name>
  <ConfigBlock>
    <Name> MAIN_NO</Name>
    <BoolAttribute>
      <Name>Trace Enable</Name>
      <DefaultValue>>false</DefaultValue>
    </BoolAttribute>
    <UInt32Attribute>
      <Name>AtbID </Name>
      <DefaultValue>50</DefaultValue>
    </UInt32Attribute>
    <UInt32Attribute>
      <Name>SyncPeriod</Name>
      <DefaultValue>5</DefaultValue>
      <UInt32Range>
        <MinValue>0</MinValue>
        <MaxValue>22</MaxValue>
      </UInt32Range>
    </UInt32Attribute>
  </ConfigBlock>
</ConfigBlock>
```

The ATB ids for NoC observers will be NoC Main ATBid = 0x50 and NoC HSIO ATBid = 0x51.

Figure 3 will have a config block for HSIO Funnel, under SoC Modules config block.

Example of HSIO Funnel

```
<ConfigBlock>
  <Name>SoC Modules</Name>
  <!-- HSIO Funnel -->
  <ConfigBlock>
    <Name>HSIO Funnel</Name>
    <BoolAttribute>
      <Name>Enable Trace</Name>
      <DefaultValue>>false</DefaultValue>
    </BoolAttribute>
    <UInt32Attribute>
      <Name>Minimum Hold Time</Name>
      <DefaultValue>4</DefaultValue>
      <UInt32Range>
        <MinValue>1</MinValue>
        <MaxValue>15</MaxValue>
      </UInt32Range>
    </UInt32Attribute>
    <UInt32Attribute>
      <Name>Funnel Priority 0</Name>
      <DefaultValue>0</DefaultValue>
      <UInt32Range>
        <MinValue>0</MinValue>
        <MaxValue>1</MaxValue>
      </UInt32Range>
    </UInt32Attribute>
    <UInt32Attribute>
      <Name>Funnel Priority 1</Name>
      <DefaultValue>0</DefaultValue>
      <UInt32Range>
        <MinValue>0</MinValue>
        <MaxValue>1</MaxValue>
      </UInt32Range>
    </UInt32Attribute>
  </ConfigBlock>
</ConfigBlock>
```

If a communication protocol is available, the NoC Configurator will access the target and will perform the writes to the configuration registers.

Example stdout for Linux Target Agent

```
write_mem 0x1200008 data=0x00000000 length=0x4
write_mem 0x1200008 data=0x00000046 length=0x4
write_mem 0x120000c data=0x00000001 length=0x4
write_mem 0x1200014 data=0x0000000f length=0x4
write_mem 0x1200048 data=0x00000000 length=0x4
write_mem 0x1200054 data=0x0000003f length=0x4
write_mem 0x1200060 data=0x0000000f length=0x4
write_mem 0x1200064 data=0x00000003 length=0x4
write_mem 0x1200068 data=0x0000000f length=0x4
write_mem 0x120006c data=0x00000000 length=0x4
write_mem 0x1200074 data=0x00000000 length=0x4
write_mem 0x120007c data=0x00000000 length=0x4
write_mem 0x1200008 data=0x00000000 length=0x4
write_mem 0x1200008 data=0x00000046 length=0x4
write_mem 0x120000c data=0x00000001 length=0x4
write_mem 0x1200014 data=0x0000000f length=0x4
write_mem 0x1200048 data=0x00000000 length=0x4
write_mem 0x1200054 data=0x0000003f length=0x4
write_mem 0x1200060 data=0x0000000f length=0x4
write_mem 0x1200064 data=0x00000003 length=0x4
write_mem 0x1200068 data=0x0000000f length=0x4
write_mem 0x120006c data=0x00000000 length=0x4
write_mem 0x1200074 data=0x00000000 length=0x4
write_mem 0x120007c data=0x00000000 length=0x4
```

```

write_mem 0x700084fb0 data=0xc5acce55 length=0x4
write_mem 0x700004000 data=0x00000001 length=0x4
write_mem 0x70000cfb0 data=0xc5acce55 length=0x4
write_mem 0x70000c000 data=0x00000407 length=0x4
write_mem 0x70000c004 data=0x00000007 length=0x4
write_mem 0xdfb0 data=0xc5acce55 length=0x4
write_mem 0xd000 data=0x00000403 length=0x4
write_mem 0xd004 data=0x00000003 length=0x4
write_mem 0x120240c data=0x00000000 length=0x4
write_mem 0x1202408 data=0x00000050 length=0x4
write_mem 0x1202410 data=0x00000005 length=0x4
write_mem 0x120240c data=0x00000001 length=0x4
write_mem 0x122300c data=0x00000000 length=0x4
write_mem 0x1223008 data=0x00000051 length=0x4
write_mem 0x1223010 data=0x00000005 length=0x4
write_mem 0x122300c data=0x00000001 length=0x4
write_mem 0x1220808 data=0x00000000 length=0x4
write_mem 0x1220808 data=0x00000046 length=0x4
write_mem 0x122080c data=0x00000001 length=0x4
write_mem 0x1220814 data=0x0000000f length=0x4
write_mem 0x1220848 data=0x00000000 length=0x4
write_mem 0x1220854 data=0x0000003f length=0x4
write_mem 0x1220860 data=0x0000000f length=0x4
write_mem 0x1220864 data=0x00000003 length=0x4
write_mem 0x1220868 data=0x0000000f length=0x4
write_mem 0x122086c data=0x00000000 length=0x4
write_mem 0x1220874 data=0x00000000 length=0x4
write_mem 0x122087c data=0x00000000 length=0x4
write_mem 0x700084fb0 data=0xc5acce55 length=0x4
write_mem 0x700004000 data=0x00000001 length=0x4
write_mem 0x70000cfb0 data=0xc5acce55 length=0x4
write_mem 0x70000c000 data=0x00000407 length=0x4
write_mem 0x70000c004 data=0x00000007 length=0x4
write_mem 0xdfb0 data=0xc5acce55 length=0x4
write_mem 0xd000 data=0x00000403 length=0x4
write_mem 0xd004 data=0x00000003 length=0x4
write_mem 0x120240c data=0x00000000 length=0x4
write_mem 0x1202408 data=0x00000050 length=0x4
write_mem 0x1202410 data=0x00000005 length=0x4
write_mem 0x120240c data=0x00000001 length=0x4
write_mem 0x122300c data=0x00000000 length=0x4
write_mem 0x1223008 data=0x00000051 length=0x4
write_mem 0x1223010 data=0x00000005 length=0x4
write_mem 0x122300c data=0x00000001 length=0x4
Configuration successful.

```

Each line describes a target access that the NoC Configurator is performing including the Physical Address, the actual data to be written as an unsigned integer value and the register size (length).

2.5 ARMv8 STM configurator

This document describes how to use the **System Trace Macrocell (STM)** Configurator tool to control the behavior of the STM trace generator. STM Configurator reads the settings from the input XML file and transforms them into target access memory that writes to the STM configuration registers.

This section explains:

- [Overview](#)
- [Tool description](#)
- [Conclusions](#)

2.5.1 STM configurator overview

The STM is a trace source that is integrated into a CoreSight system, and that is designed primarily for high-bandwidth trace of instrumentation embedded into software. This instrumentation is made up of memory-mapped writes to the STM Advanced eXtensible Interface (AXI) slaves, which carry information about the behavior of the software.

In addition to the AXI slave, the STM provides a hardware event interface. The STM generates trace when signals are asserted on this interface.

The STM AXI slave is connected to a system interconnect that enables all system masters, such as processors and Direct Memory Access (DMA) controllers, to generate trace by writing to the STM stimulus ports.

The following figure illustrates all trace generators from LS2085A architecture: Core, PXDI, DDDI, NoC and STM. The STM module is dependent on SoC Funnel in order to transfer the trace to collector.

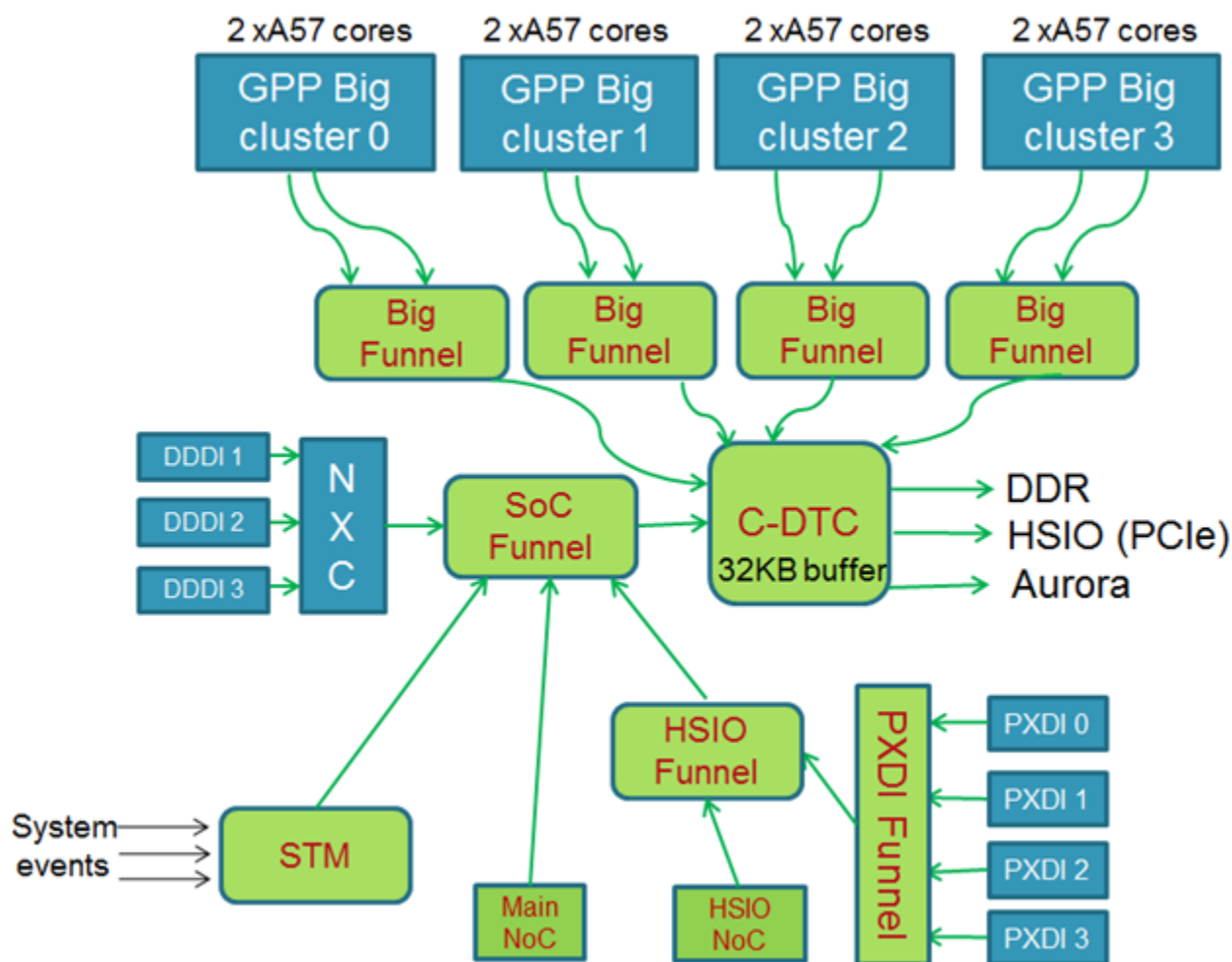


Figure 4. LS2085A Trace Generators

The STM Configurator configures the following modules:

- STM trace generator
- SoC Funnel
- C-DTC (Central Debug Trace Controller)

The C-DTC stores and forwards trace data using a dedicated RAM buffer. This reduces trace loss by absorbing spikes in trace data. The destination of the raw trace is either the internal trace buffer of the C-DTC module or a user defined buffer in DDR.

The following settings can be configured for STM:

- Enable/Disable Trace Generation
- Enable/Disable Timestamp

The following settings can be configured per data stream:

- Trace collection location - The options available are DTC and DDR.
- Trace Collection Mode - The options available are OneBuffer and Overwrite. If OneBuffer is selected, trace can only be collected only until buffer is filled. In this case only the first buffer size part of the trace is kept. If Overwrite is selected, in case buffer is filled the pointer returns to the beginning of the buffer and overwrites older trace. In this case only the last buffer size part of the trace is kept.

In case trace is collected in DDR buffer, the user has to set two more settings in the configuration file:

- Trace Buffer Base Address - sets the address in DDR where the trace buffer starts
- Trace Buffer Size - sets the size of the DDR trace buffer

The DTC internal trace buffer is a DDR zone dedicated to this purpose.

2.5.2 Tool description

This section explains the following:

- [Usage](#)
- [XML Configuration File](#)

2.5.2.1 Usage

The following command shows how to use the STM Configurator:

```
stm_tracetargetconfig -c STMPlatformConfig<xml_config_file>
```

The following table lists the details of the command options:

Table 4. Usage options details

Options	Details
-h [--help]	Displays the help
-c [--config-file] path	Path to the xml configuration file.
-v [--version]	Displays the version information.

For example:

```
stm_tracetargetconfig -c STMPlatformConfig.xml
```

2.5.2.2 XML configuration file

The configuration xml file contains all the settings described for STM packed in a configuration block. The configurator is following the multicore design. There is a single platform configuration file used by all the components to configure trace IP blocks. This file can be used for both configuring and collecting trace.

We identify the following components that need to be considered while configuring and collecting trace from a multicore system:

- Trace generator: It is an entity that can produce trace, for example STM.
- DataStream: It is the place where trace is collected in hardware, for example DTC and DDR.

The configuration xml file will have all the settings described for STM module packed in a configuration blocks.

The Trace Generator configuration block includes the configuration block for each STM module.

Below is the example of trace generators STM module configuration block:

Example of STM configuration module

```
<ConfigBlock>
  <Name>STM</Name>
  <BoolAttribute>
    <!-- Enables/disables trace generation -->
    <Name>Enable Trace</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <StringAttribute>
    <Name>Dependency</Name>
    <DefaultValue>SoC Funnel</DefaultValue>
  </StringAttribute>
  <BoolAttribute>
    <Name>StartTrace</Name>
    <DefaultValue>true</DefaultValue>
  </BoolAttribute>
  <BoolAttribute>
    <!-- Enable timestamping. If stimulus writes select timestamping, the STM outputs
a timestamp -->
    <Name>Timestamp</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
</ConfigBlock> <!-- STM -->
```

The ATB id for STM is **ATBid = 0x40**.

If a communication protocol is available, the STM Configurator will access the target and will perform the writes to the configuration registers.

Example stdout for Linux Target Agent

```
write_mem 0x701040fb0 data=0xc5acce55 length=0x4
write_mem 0x701040300 data=0x00000000 length=0x4
write_mem 0x701040004 data=0x00000000 length=0x4
write_mem 0x700012000 data=0x00000003 length=0x4
write_mem 0x701040010 data=0x00000840 length=0x4
write_mem 0x701040020 data=0x00000000 length=0x4
write_mem 0x701040024 data=0x00000000 length=0x4
write_mem 0x701040040 data=0x00000001 length=0x4
write_mem 0x701040034 data=0x0000000a length=0x4
write_mem 0x701040030 data=0x00000000 length=0x4
write_mem 0x701040080 data=0x00000201 length=0x4
write_mem 0x701040084 data=0x00000000 length=0x4
write_mem 0x701040088 data=0x00000000 length=0x4
write_mem 0x701040004 data=0x00000001 length=0x4
write_mem 0x701140fb0 data=0xc5acce55 length=0x4
```

```

write_mem 0x701140300 data=0x00000000 length=0x4
write_mem 0x701140004 data=0x00000000 length=0x4
write_mem 0x700012000 data=0x00000003 length=0x4
write_mem 0x701140010 data=0x00000840 length=0x4
write_mem 0x701140020 data=0x00000000 length=0x4
write_mem 0x701140024 data=0x00000000 length=0x4
write_mem 0x701140040 data=0x00000002 length=0x4
write_mem 0x701140034 data=0x0000000a length=0x4
write_mem 0x701140030 data=0x00000000 length=0x4
write_mem 0x701140080 data=0x00000201 length=0x4
write_mem 0x701140084 data=0x00000000 length=0x4
write_mem 0x701140088 data=0x00000000 length=0x4
write_mem 0x701140004 data=0x00000001 length=0x4
write_mem 0x1220808 data=0x00000000 length=0x4
write_mem 0x1220808 data=0x00000006 length=0x4
write_mem 0x122080c data=0x00000001 length=0x4
write_mem 0x700010fb0 data=0xc5acce55 length=0x4
write_mem 0x700012000 data=0x00000003 length=0x4
write_mem 0x700010e80 data=0x00400003 length=0x4
write_mem 0x700010e94 data=0x00000001 length=0x4
write_mem 0x700010e00 data=0xffffffff length=0x4
write_mem 0x700084fb0 data=0xc5acce55 length=0x4
write_mem 0x700004000 data=0x00000001 length=0x4
write_mem 0x700004fb0 data=0xc5acce55 length=0x4
write_mem 0x700004000 data=0x800000c2 length=0x4
read_mem 0x700004020 data=0x800000c2 length=0x4
write_mem 0x700004020 data=0x00000000 length=0x4
write_mem 0x70000cfb0 data=0xc5acce55 length=0x4
write_mem 0x70000c000 data=0x00000407 length=0x4
write_mem 0x70000c004 data=0x00000007 length=0x4
Configuration finished.

```

Each line describes a target access that the STM Configurator is performing including the Physical Address, the actual data to be written as an unsigned integer value and the register size (length).

2.6 Trace Complex 2 UI Configurator

This chapter describes how to use the ARMv8 Trace Complex 2 Configurator tool. This tool reads the user settings from the input XML file and transforms them into target access memory, then writes to the configuration registers.

NOTE

For more details, see "*CodeWarrior Development Studio for QorIQ LS series - ARM V8 ISA, Tracing and Analysis User Guide*".

2.6.1 Overview

Figure 5 shows the core trace path within the ARMv8 Architecture.

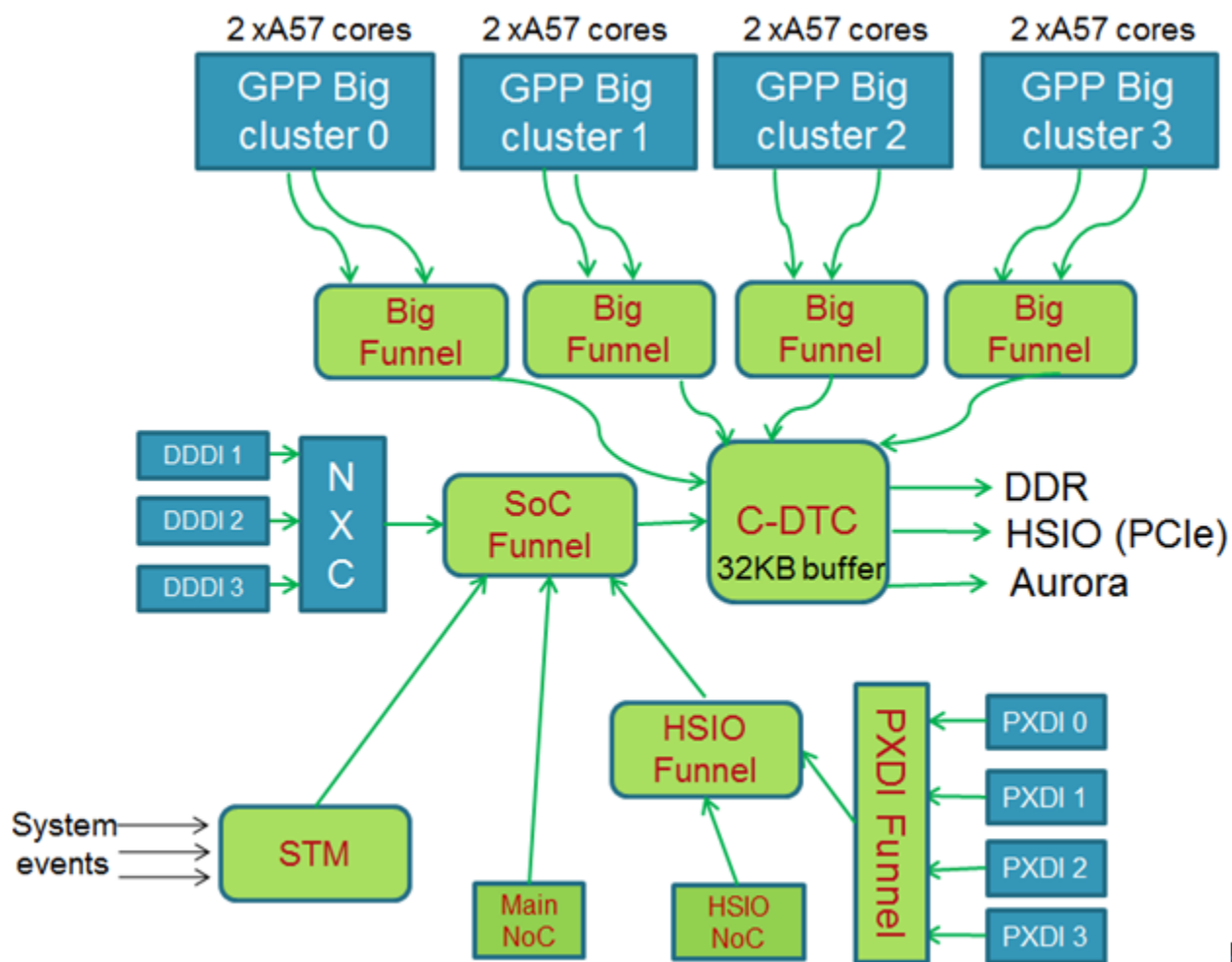


Figure 5. Core Trace path

The Trace Complex 2 Configurator is responsible with configuring the ETM cores (ETM - Embedded Trace Macrocell 4.0), Big Funnel (that makes the link between the active trace generator (active core) and the central DTC module), PXDI (n form 0 to 3) modules, PXDI Funnel, SoC Funnel (makes the link to central DTC module), DDDI (n form 1 to 3) modules, NXC, HSIO Funnel, Main NoC and HSIO NoC observers (each of these two observers has a number of trace generators probe) and STM. It also configures the C-DTC collector.

The C-DTC stores and forwards trace data using a dedicated RAM buffer. This reduces trace loss by absorbing spikes in trace data.

The destination of the raw trace is either the internal trace buffer of the C-DTC module or a user defined buffer in DDR.

The following settings can be configured per core:

- **Enable/Disable** To enable or disable trace generation
- **Trace Scenario:** option for this setting is Program Trace
- **Timestamp:** enabled or disabled
- **Executable file:** To add/remove executable files

The following settings can be configured per DDDI:

- **Enable/Disable** Trace Generation
- **Trace Mode:** DDDI messages can be Terse or Verbose. Terse mode sacrifices visibility for reducing bandwidth and Verbose mode maximizes visibility at the cost of increased bandwidth. The following settings are available:
 - Verbose (highest bandwidth)
 - Terse1

- Terse2
- Terse3 (lowest bandwidth).
- **Timestamp**: enabled or disabled.

The following settings can be configured per PXDI:

- **Enable/Disable** Trace Generation
- **Debug Probe Select**: is a number from 0 to 15, which specifies the set of signals that will be generated as trace data. The description of these sets can be found in the documents that depict the PXDI hardware modules.

The following settings can be configured per NoC Probe:

- **Enable/Disable** Trace Generation
- **Enable/Disable** Payload
- **Enable/Disable** Intrusive Mode

The following settings can be configured for STM:

- **Enable/Disable** Trace Generation
- **Enable/Disable** Timestamp

The following settings can be configured per data stream:

- **Trace Collection location**: options are DTC and DDR.
- **Trace Collection Mode**: single choice between **OneBuffer** and **Overwrite**.
 - If **OneBuffer** is selected, trace can only be collected only until buffer is filled. In this case only the first buffer size part of the trace is kept.
 - If **Overwrite** is selected, in case buffer is filled the pointer returns to the beginning of the buffer and overwrites older trace. In this case only the last buffer size part of the trace is kept.

In case trace is collected in DDR buffer, the user has to set in the configuration file for two more settings:

- **Trace Buffer Base Address**: sets the address in DDR where the trace buffer starts
- **Trace Buffer Size**: sets the size of the DDR trace buffer

The DTC internal trace buffer is a DDR zone dedicated to this purpose.

2.7 Conclusions

Standalone trace configurator tools are used to configure individual trace sources or the platform trace aggregators (Platform Configurator). More than one configurator should be run in order to activate all the trace modules for a trace session. For example, if you want to activate NoC trace, you need to run NoC Platform Configurator.

3 Decoders

3.1 Introduction

This chapter describes the following standalone decoders:

- [ARM v8 DDDI decoder](#)
- [ARM v8 PXDI decoder](#)
- [ARM v8 STM decoder](#)

Decoders

- [ARM v8 ETMv4.0 decoder](#)
- [ARMv8 NoC decoder](#)

3.2 ARM v8 DDDI decoder

This component offers complete decoding services that can be used to interpret any DDDI trace for an ARM v8 target transforming the raw data into useful information.

The main advantages of this tool are:

- the size, contains only what is needed,
- It is fast, all the services required are encapsulated into one application and all the calls are processed without any delays that may be caused by cross languages requests,
- It is having a simplified API and can be effortlessly integrated into any testing framework.

3.2.1 Component details

The component details include:

- [Options](#)
- [Sample trace](#)

3.2.1.1 Options

The following are the available options and their description.

Usage

```
dddi.decoder [Options] <input_file> [output_file]
```

Table 5. Available options

Option	Description
-h [--help]	Displays help message.
-p [--show-progress]	Shows the progress periodically.
-v [--version]	Shows the product version.
-a [--dddi-desc]	Specifies the path to the DDDI decoder description file.
[--input-file]	Input file
[--output-file]	Output file

The application uses the output file extension to determine the format, “.txt” for text and “.csv” for comma separated values. If this parameter is not specified the trace events are redirected to standard output.

Table 6. Hidden options

Option	Description
-f [--output-format]	Output format, supported txt and csv.
-b[--big-endian]	Specifies whether the Nexus trace data is in big endian format. If you do not specify this option, the parser will consider the input file as being in little endian format.
-c [--chunk-size]	Specifies the processing chunk size in bytes. Default value is 10240 bytes.
-s [--include-sync]	Specifies whether the formatted stream includes synchronization messages or not. If this option exists, the unformatting algorithm starts after the first synchronization packet. Once synchronized, all subsequent synchronization packets will be discarded.
-d [--nexus-description]	File that describes the format of Nexus messages generated by the Nexus trace module.

NOTE

The hidden options cover particular scenarios and are for testing purposes until the hardware trace is available.

ATB ID is an ID used to identify into a stream of trace formatted ATB each component. The traces collected from multi cores are mixed into a single stream that is in ATB formatted. To make a distinction between the 2 traces, ATB IDs are required. The ATB source id for DDDI is “41”.

3.2.1.1.1 Execution flow

The execution flow is described below.

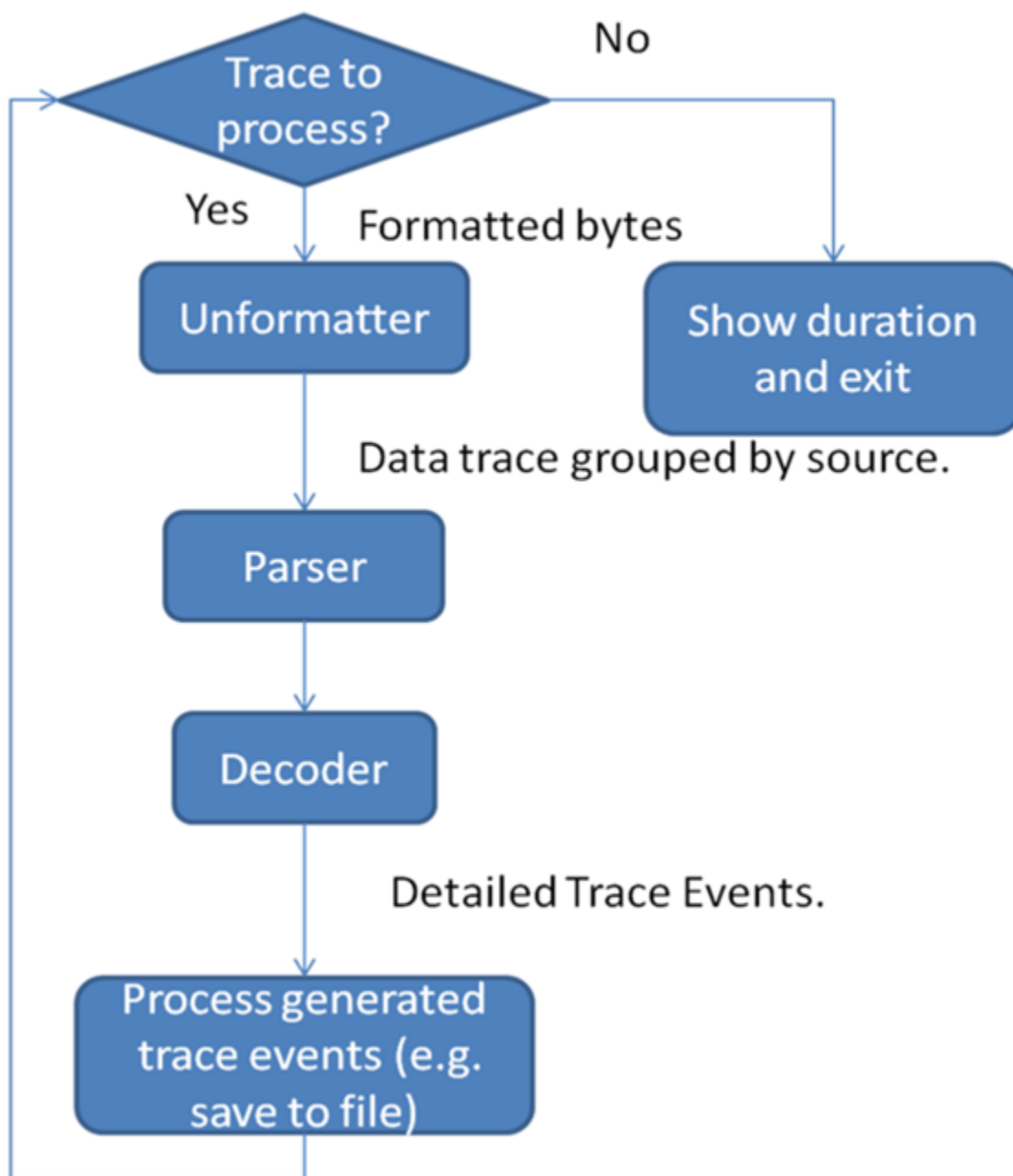


Figure 6. Execution flow

3.2.1.2 Sample trace

The product archive contains one trace files created using the following process. First we collected DDDI raw data trace from a LS2085A board, using ATB id 41.

To decode `dddi_trace.dat` you need to run the following command:

```
dddi.decoder dddi_trace.dat csv.csv -s -b -p -d nexus_dddi.xml -a  
TEMPLATEConfigFileLS2085A.xml
```

dddi_trace.dat is the input file, csv.csv is the output file, “-s” is used because when the trace was encapsulated into ATB format an ATB sync frame was inserted, “-b” because the trace is in big endian format and “-p” to show the progress. The “-d” option is used to provide the location of the file that describes the format of DDR Nexus Messages generated by the DDDI module, by default the application searches in startup directory for “nexus_dddi.xml” but since we used the trace which has couple of nexus messages different we need to specify the xml file. All this xml files are located in the archive near the executable or in a CodeWarrior environment in “\${CodeWarriorInstallDir}/Config/SA/data/fsl.configs.sa.ls.parsers”.

The output is redirected to cvs.csv file which will have the following format:

```
Index(n),Source(t),Type(t),Description(lt),Address(nh),Destination(nh),Timestamp(n),
1,DDDI,Custom,"Port: DDDI1. Verbose mode. Transaction source = GPP0. Transaction type: Read.
Transaction size = 256 bits. DDR EC bits = 0x1. Address = 0x001234560. ",,,5043814,
2,DDDI,Custom,"Port: DDDI1. Verbose mode. Transaction source = GPP0. Transaction type: Read.
Transaction size = 256 bits. DDR EC bits = 0x1. Address = 0x001234560. ",,,5044691,
3,DDDI,Custom,"Port: DDDI1. Verbose mode. Transaction source = GPP0. Transaction type:
Write. Transaction size = 256 bits. DDR EC bits = 0x1. Address = 0x001234564. ",,,5071268,
4,DDDI,Custom,"Port: DDDI1. Verbose mode. Transaction source = GPP0. Transaction type: Read.
Transaction size = 256 bits. DDR EC bits = 0x1. Address = 0x001234560. ",,,5247622,
```

3.3 ARM v8 PXDI decoder

This component offers complete decoding services that can be used to interpret any PXDI trace for a ARM v8 target transforming the raw data into useful information. **Pex Debug Trace Interface (PXDI)** is used to capture debug trace messages about PEX client operations.

The main advantages of this tool are:

- the size, contains only what is needed,
- is fast, all the services required are encapsulated into one application and all the calls are processed without any delays that may be caused by cross languages requests,
- simplified API, can be effortlessly integrated into any testing framework.

3.3.1 Component details

The component details include:

- [Options](#)
- [Sample trace](#)

3.3.1.1 Options

The following are the available options and their description.

Usage

```
pxdi.decoder [Options] <input_file> [output_file]
```

Table 7. Available options

Option	Description
-h [--help]	Displays help message.

Table continues on the next page...

Table 7. Available options (continued)

Option	Description
-p [--show-progress]	Shows the progress periodically.
-n [--pxdi-set-number] arg (=0)	PXDI set number. Used only if source is PXDI.
-d [--pxdi-desc]	Specifies the path to the PXDI decoder description file.
-v [--version]	Shows the product version.
[--input-file]	Input file
[--output-file]	Output file

Table 8. Hidden options

Option	Description
-f [--output-format]	Output format, supported txt and csv.
-c [--chunk-size]	Specifies the processing chunk size in bytes. Default value is 10240 bytes.
-s [--include-sync]	Specifies whether the formatted stream includes synchronization messages or not. If this option exists, the unformatting algorithm starts after the first synchronization packet. Once synchronized, all subsequent synchronization packets will be discarded.
-x [--pxdi-xml]	Path to the PXDI sets description file.

NOTE

The hidden options cover particular scenarios and are for testing purposes until hardware trace is available.

3.3.1.1.1 Execution flow

The execution flow is describe below.

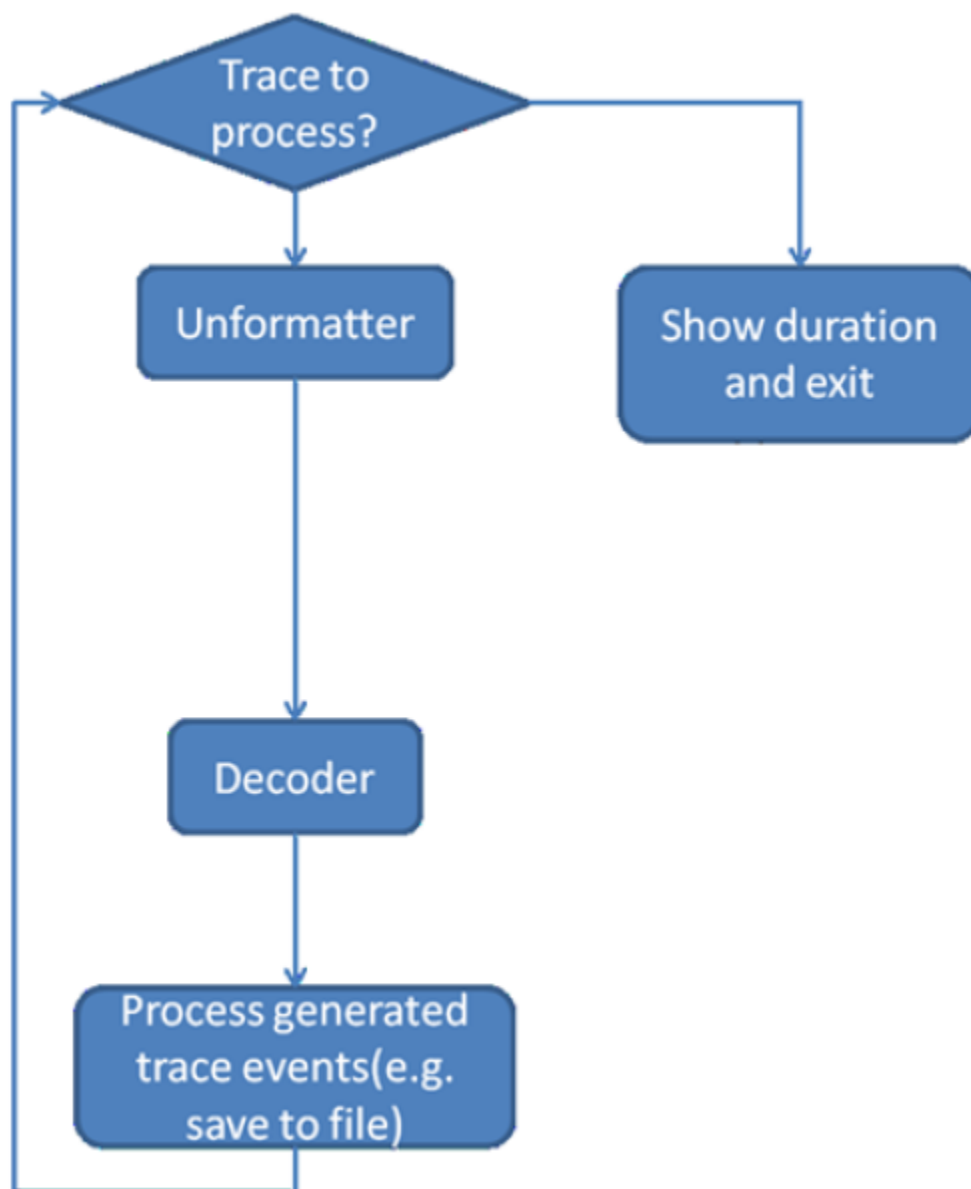


Figure 7. Execution flow

3.3.1.2 Sample trace

The product archive contains one trace files created using two frames, one containing normal data 0x00300001 and the second one contain couple of error bits, 0x80300000 form_filt_hv field should be 0 and unexpected_cpl_err set to 1. The error message is inserted only after 10 normal frames.

To decode the demo trace file that we used to exercise the application, pxdi_trace.dat, you need to run the following command:

```
pxdi.decoder pxdi_trace.dat csv.csv -n 3 -p -s -x pxdi_sets_description.xml -d  
PXDI_Decoder.xml
```

pxdi_trace.dat is the input file, csv.csv is the output file, “-n” is used to set the PXDI set number, “-s” is used because when the trace was encapsulated into ATB format an ATB sync frame was inserted, “-p” to show the progress, and “-d” to define the PXDI description file for decoder, used to read ATB IDs from it. By default the application searches in startup directory for pxdi_sets_description.xml but the location can be provided by the user with -x option. All this xml files are located in the archive near the executable or in a CodeWarrior environment in “\${CodeWarriorInstallDir}/Config/SA/data”, in this case in *fsl.configs.sa.ls.decoders* folder. All the trace sample files are located in *fsl.sample.trace* directory.

The output is redirected to cvs.csv file which will have the following format:

```
PXDI Set = 0x3. form_filt_hv = 0x0. form_filt_hdr_7_1 = 0x0. form_filt_hdr_31_22 = 0x0.
rtlh_radm_dllp_err = 0x0. rtlh_radm_malform_tlp_err = 0x0. int_rtlh_radm_ecrc_err = 0x1.
flt_q_header_destination = TRGT0. flt_q_header_cpl_status = Successful Completion(SC).
flt_q_cpl_abort = 0x0. flt_q_cpl_last = 0x0. cpl_mlf_err = 0x0. cpl_ur_err = 0x0. cpl_ca_err
= 0x0. unexpected_cpl_err = 0x1.
PXDI Set = 0x3. form_filt_hv = 0x1. form_filt_hdr_7_1 = 0x0. form_filt_hdr_31_22 = 0x0.
rtlh_radm_dllp_err = 0x0. rtlh_radm_malform_tlp_err = 0x0. int_rtlh_radm_ecrc_err = 0x1.
flt_q_header_destination = TRGT0. flt_q_header_cpl_status = Successful Completion(SC).
flt_q_cpl_abort = 0x0. flt_q_cpl_last = 0x0. cpl_mlf_err = 0x0. cpl_ur_err = 0x0. cpl_ca_err
= 0x0. unexpected_cpl_err = 0x0.
```

The 11th trace event has the unexpected_cpl_err bit set, this is an indication that the trace was decoded correctly because we inserted an error frame at each 10th message.

3.4 ARM v8 STM decoder

This component offers complete decoding services that can be used to interpret any STM trace (System Trace Macrocell), formatted in ATB frames, transforming the raw data into human readable information (csv format). This will allow a user to see the individual data and events that were generated by hardware STM masters or directly from the user application or a Linux kernel module.

STP (System Trace Protocol) allows higher level protocols (multi-character or multi-word) to be implemented on top of STM data using markers and flags. This would allow a powerful usecase such as on-target debug-print functionality, which means emitting binary general purpose data and character streams similar to printf. The decoder output can be fed to other processing tools, such as a logger or formatter which would reconstruct or format the data into strings or custom debug messages.

This decoder implements the MIPI STP 2.1 standard.

The main advantages of this tool are:

- the size, contains only what is needed,
- is fast, all the services required are encapsulated into one application and all the calls are processed without any delays that may be caused by cross languages requests,
- simplified API, can be effortlessly integrated into any testing framework.

3.4.1 Functional details

The STM decoder is part of a mixed tool containing an ATB unformatter and a STM decoder. The processing flow starts from raw STM Data, which for ARMv8 is packaged in ATB format, and will end with decoded data in human readable format, which is also ready to be displayed in a trace viewer or used as input by other logger or formatter clients.

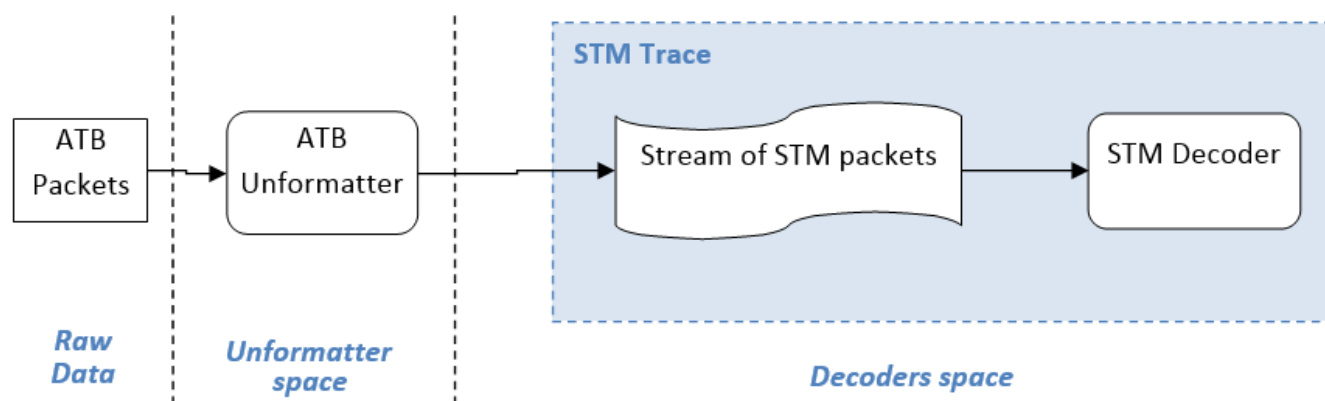


Figure 8. STM trace path

3.4.2 STP packets, according to MIPI STP standard v.2.1

MIPI STP v.2.1 protocol defines the following packets (termination “TS” means the packet contains timestamp):

Packet	Description	Generated Output
NULL	Filler packet (shortest, one ‘0’ nibble)	-
NULL_TS	Filler packet, time-stamped	-
M8 / M16	Master ID, 8/16 bit (8 bit contains only the lsb, msb is left unchanged)	-
MERR	Master error	ErrorEvent
C8 / C16	Channel, 8/16 bit (8 bit contains only the lsb, msb is left unchanged)	-
D4 / D8 / D16 / D32 / D64	Data, 4/8/16/32/64 bit	InfoEvent
D4M / D8M / D16M / D32M / D64M	Data, 4/8/16/32/64 bit with marker	InfoEvent
D4[M]TS / D8[M]TS / D16[M]TS / D32[M]TS / D64[M]TS	Data, 4/8/16/32/64 bit, time-stamped, with or without marker	InfoEvent
FLAG / FLAG_TS	Flag to mark a point of interest, with or without timestamp	InfoEvent
VERSION	Protocol version	-
USER / USER_TS	User-defined, with or without timestamp	InfoEvent
TIME / TIME_TS	Time synchronization, with or without timestamp	-
TRIG / TRIG_TS	Trigger, with or without timestamp	InfoEvent
FREQ / FREQ_TS	Time stamp clock frequency, with or without timestamp	InfoEvent
XSYNC / XSYNC_TS	Cross-synchronization, with or without timestamp	InfoEvent
GERR	Global error	ErrorEvent
ASYN	Alignment synchronization	-

Decoders

Only some of the packets generate output data. The others are used internally by the decoding algorithm (e.g. for synchronization, setting current trace source (master/channel), etc.).

The output data is formatted as a SA Engine Trace Event, containing a Trace Source (formatted as “master_id[15:0] : channel_id[15:0]”), a Description and a Timestamp.

STP is not for program trace. STP does not contain addresses, only full data and time information (which can be full or delta, depending on the VERSION packets).

3.4.3 Component details

The STM decoder product archive contains the decoder binary under internal directory ‘bin’, as well as a STM sample trace. In CodeWarrior, they are deployed to <CodeWarrior_install_dir>/ARMv8/sa_ls/bin. However they can be unpacked to any other path.

The Decoder binary:

- stm.decoder.exe (Windows) or stm.decoder (Linux)

The Decoder example:

- STMTraceData.dat – contains a generated STM Trace, containing ASYNC, VERSION and DATA (Dxx – no marker, no timestamp) packets.
- README file.
- STM_Decoder.xml file.

The trace file was created using a STM generated trace (from an internal STM trace generator) formatted into ATB frames (with an internal ATB formatter tool)

3.4.3.1 Usage

Below is the list of command line options that can be use to run the tool.

Usage:

```
stm.decoder [Options] <input_file> [output_file]
```

Options:

Table 9. Available options

Option	Description
-h [--help]	Displays help message.
-p [--show-progress]	Shows the progress periodically.
-v [--version]	Shows the product version.
<input_file>	Specifies the name of the binary trace file containing STM trace, generated by a Layerscape target. The trace format is STM encapsulated in ATB frames
<output_file>	Specifies the name of the .csv file, generated by the tool. The application outputs the format in csv format. If this parameter is not specified the trace events are redirected to standard output.

Table 10. Hidden options

Option	Description
-f [--output-format]	Output format, supported txt and csv.
-c [--chunk-size] arg(=1024)	Specifies the processing chunk size in bytes. Default value is 10240 bytes.
-s [--include-sync]	Specifies whether the formatted stream includes synchronization messages or not. If this option exists, the unformatting algorithm starts after the first synchronization packet. Once synchronized, all subsequent synchronization packets will be discarded.
[--HELP]	Displays all options (including hidden).
-t [--text]	Reconstruct the original strings instead of displaying the raw data.
-x [--stm-xml] path	Specifies the path to the STM description file

NOTE

All hidden options are created to cover couple of particular scenarios and for testing purposes until hardware trace is available.

3.4.4 Output files

The results are output in csv format in the file `stm_out.csv`.

	A	B	C	D	E
1	1	STM_0:0	Flag.	Info	0
2	2	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
3	3	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
4	4	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
5	5	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
6	6	STM_0:0	User Data = 0x00000000.	Info	0
7	7	STM_0:0	Flag.	Info	0
8	8	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
9	9	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
10	10	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
11	11	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
12	12	STM_0:0	User Data = 0x00000001.	Info	0
13	13	STM_0:0	Flag.	Info	0
14	14	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
15	15	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
16	16	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
17	17	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
18	18	STM_0:0	User Data = 0x00000002.	Info	0
19	19	STM_0:0	Flag.	Info	0
20	20	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
21	21	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
22	22	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
23	23	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
24	24	STM_0:0	User Data = 0x00000003.	Info	0
25	25	STM_0:0	Flag.	Info	0
26	26	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
27	27	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
28	28	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
29	29	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
30	30	STM_0:0	User Data = 0x00000004.	Info	0
31	31	STM_0:0	Flag.	Info	0
32	32	STM_0:0	Data = 0x77202c6f6c6c6548. Size = 64 bit.	Info	0
33	33	STM_0:0	Data = 0x646c726f. Size = 32 bit.	Info	0
34	34	STM_0:0	Data = 0x0a21. Size = 16 bit.	Info	0
35	35	STM_0:0	Data = 0x00. Size = 8 bit.	Info	0
36	36	STM_0:0	User Data = 0x00000005.	Info	0

Figure 9. Output

3.5 ARM v8 ETMv4.0 decoder

This component offers completed decoding services that can be used to interpret any ETMv4.0 trace for a ARM v8 target transforming the raw data into useful information. An **Embedded Trace Macrocell (ETM)** is a real-time trace module providing instruction and data tracing of a processor.

The main features of an ETM are:

- Trace generation - Outputs information that helps users understand the operation of the processor.
- Triggering and filtering facilities - An extensible specification enables users to control tracing by specifying the exact set of triggering and filtering resources required for a particular application. Resources include address comparators and data value comparators, counters, and sequencers.

The advantages of the ARM v8 standalone decoder tool are:

- Size - contains only what is needed;
- Speed - all the services required are encapsulated into one application and all the calls are processed without any delays that may be caused by cross languages requests;
- Simple API - can be effortlessly integrated into any testing framework.

3.5.1 Component details

ETM_4.0 decoder will be delivered as an independent product without any other libraries. Below we listed the options available and a short description.

```
etm40.decoder [Options] <input_file> <output_file>
```

Table 11. Available options

Option	Description
-h [--help]	Displays help message.
-i [--input-file] path	Input file path
-o [--output-file] path	Output file path
-p [--show-progress]	Show the progress periodically
-x [--ls2-xml] path	Path to the LS2 description file
-v [--version]	Displays the product version.

The application uses the output file extension to determine the format, “.txt” for text and “.csv” for comma separated values. If this parameter is not specified the trace events are redirected to standard output.

Table 12. Hidden options

Option	Description
--HELP	Displays all options (including hidden)
-f [--output-format]	Output format, supported txt and csv.
-c [--chunk-size] arg (=10240)	Specifies the processing chunk size in bytes. Default value is 10240 bytes.
-s [--include-sync]	Specifies whether the formatted stream includes synchronization messages or not. If this option exists, the unformatting algorithm starts after the first synchronization packet. Once synchronized, all subsequent synchronization packets will be discarded.

NOTE

All hidden options were created to cover couple of particular scenarios and for testing purposes until ARM v8 hardware trace will be available.

ATB IDs used for ARM v8 are:

- Core#0 - 1
- Core#1 - 2
- Core#2 - 3
- Core#3 - 4
- Core#4 - 5
- Core#5 - 6
- Core#6 - 7
- Core#7 - 8

The decoder has an xml file ETM4_0Decoder.xml with the description of the most important attributes need by the decoder. It contains 8 blocks as CORE X correspondent to 8 ARM v8 cores.

```
<ConfigBlock>
  <Name>Trace Generators</Name>
  <ConfigBlock>
    <Name>CORE</Name>
    <ConfigBlock>
      <Name>Core 0</Name>
      <BoolAttribute>
        <Name>Has ETM A-sync</Name>
        <DefaultValue>true</DefaultValue>
      </BoolAttribute>
      <UInt32Attribute>
        <Name>ATB ID</Name>
        <DefaultValue>1</DefaultValue>
      </UInt32Attribute>
      <StringAttribute>
        <Name>Decoder ID</Name>
        <DefaultValue>ETM</DefaultValue>
      </StringAttribute>
      <ConfigBlock>
        <Name>Target Images</Name>
        <Mutable>true</Mutable>
      </ConfigBlock>
    </ConfigBlock>
  </ConfigBlock>
</ConfigBlock>
```

For each Core is possible to add the path for the executables, in the attribute: <Name>**Target Images**</Name> .

3.5.1.1 Execution flow

The execution flow is describe below.

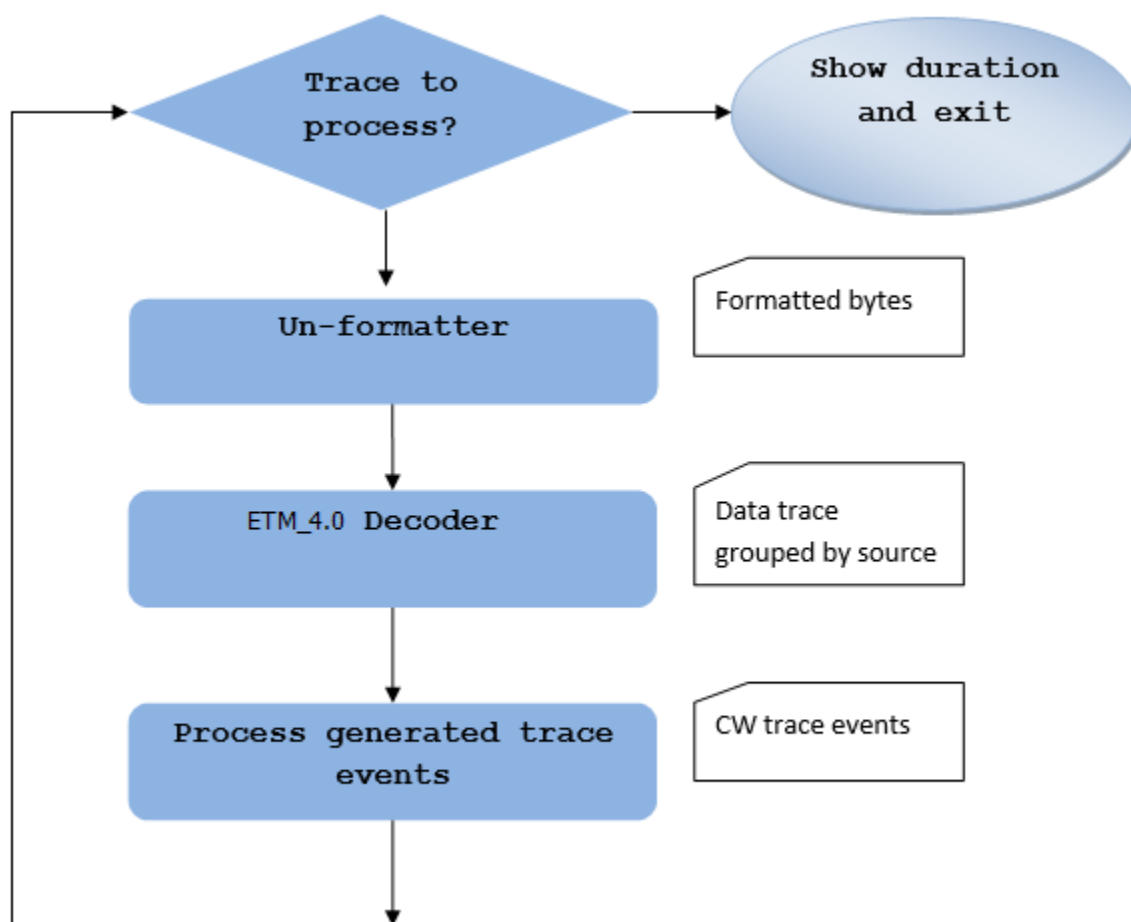


Figure 10. Flowchart for ETM_4.0 Decoder

3.5.2 Decoder example

The ARM v8 decoder product archive contains sample traces under the internal directory CodeWarrior_Examples. They are intended to be deployed to <CodeWarrior_install_dir>/ARMv8. However they can be unpacked to any other path.

The contents of this example directory:

- An ATB formatted sample trace file:
 - TraceDataArmv8_handmade.dat
 - TraceDataArmv8_handmade_dualcore.dat
- An elf file for decoding addresses encoded in the trace file:
 - Test1_cw.elf
 - Test1_cw_core1.elf
- An xml description file:
 - ETM4_0Decoder.xml
 - ETM4_0Decoder_dualcore.xml
- Disassembly for Test1_cw.elf:Test1.lst

3.5.2.1 Run the example

Follow the commands to run the ETM4.0 decoder:

1. Open a command prompt/shell and change to the example directory (<deploy_path>)
2. For Linux, add the binaries directory to the libraries search path by running: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: <deploy_path>/sa_ls/bin`
3. Run the command for - [ETM4_0 trace]:
 - For Windows: `<deploy_path>\sa_ls\bin\etm40.decoder.exe -p -i TraceDataArmv8_handmade.dat -o TraceDataArmv8_handmade.csv -x ETM4_0Decoder.xml`
 - For Linux: `<deploy_path>/sa_ls/bin/etm40.decoder -p -i TraceDataArmv8_handmade.dat -o TraceDataArmv8_handmade.csv -x ETM4_0Decoder.xml`
4. Open output files `TraceDataArmv8_handmade.csv` and observe the results.
5. Run the command for [ETM4_0 dual core trace]:
 - For Windows: `<deploy_path>\sa_ls\bin\etm40.decoder.exe -p -i TraceDataArmv8_handmade_dualcore.dat -o TraceDataArmv8_handmade_dualcore.csv -x ETM4_0Decoder_dualcore.xml`
 - For Linux: `<deploy_path>/sa_ls/bin/etm40.decoder -p -i TraceDataArmv8_handmade_dualcore.dat -o TraceDataArmv8_handmade_dualcore.csv -x ETM4_0Decoder_dualcore.xml`
6. Open output files `TraceDataArmv8_handmade_dualcore.csv` and observe the results.

3.5.2.2 Output files

The result output appears in csv format.

The results look like this:

```
Index(n),Source(t),Type(t),Description(lt),Address(nh),Destination(nh),Timestamp(n),
1,CORE 0,Info,"SYNC packet - ETM",,,,0,
2,CORE 0,Info,"Trace On packet - ETM -> start tracing after a (possible) trace gap",,,,0,
3,CORE 0,Info,"Context packet - ETM",,,,0,
,,, " exception level: 2"
,,, " state: 64-bit"
,,, " security state: non-secure"
,,, " VMID: 46"
,,, " ContextID: 1454120766"
4,CORE 0,Software Context,"software context id = 1454120766",,,,0,
5,CORE 0,Linear,Function main,0x400954,,,0,
,,, "0x400954 stp x29, x30, [sp, #-32]!"
,,, "0x400958 mov x29, sp"
,,, "0x40095c str wzr, [x29, #28]"
,,, "0x400960 ldr w0, [x29, #28]"
,,, "0x400964 cmp w0, wzr"
6,CORE 0,Linear,Function main,0x400968,,,0,
,,, "0x400968 b.ne #20"
7,CORE 0,Linear,Function main,0x40096c,,,0,
,,, "0x40096c movz w0, #1"
,,, "0x400970 str w0, [x29, #28]"
,,, "0x400974 ldr w0, [x29, #28]"
8,CORE 0,Branch,Branch from main to fa,0x400978,0x400910,0,
,,, "0x400978 bl #-104 --> 0x400910"
9,CORE 0,Linear,Function fa,0x400910,,,0,
,,, "0x400910 stp x29, x30, [sp, #-48]!"
,,, "0x400914 mov x29, sp"
,,, "0x400918 stp x19, x20, [sp, #16]"
,,, "0x40091c str w0, [x29, #44]"
,,, "0x400920 movz w20, #0"
,,, "0x400924 movz w19, #1"
,,, "0x400928 ldr w0, [x29, #44]"
,,, "0x40092c add w20, w0, #1"
,,, "0x400930 sub w19, w19, w20"
,,, "0x400934 mov w0, w20"
,,, "0x400938 mov w1, w19"
10,CORE 0,Branch,Branch from fa to fb,0x40093c,0x4008bc,0,
```

```

,,,"0x40093c bl #-128 --> 0x4008bc"
11,CORE 0,Linear,Function fb,0x4008bc,,0,
,,,"0x4008bc sub sp, sp, #48"
,,,"0x4008c0 stp x19, x20, [sp]"
,,,"0x4008c4 str w0, [sp, #28]"
,,,"0x4008c8 str w1, [sp, #24]"
,,,"0x4008cc movz w20, #3"
,,,"0x4008d0 movz w19, #7"
,,,"0x4008d4 mul w0, w20, w19"
,,,"0x4008d8 ldr w1, [sp, #28]"
,,,"0x4008dc add w0, w1, w0"
,,,"0x4008e0 str w0, [sp, #28]"
,,,"0x4008e4 ldr w0, [sp, #28]"
,,,"0x4008e8 add w0, w0, w19"
,,,"0x4008ec str w0, [sp, #44]"
,,,"0x4008f0 ldr w0, [sp, #28]"
,,,"0x4008f4 mul w1, w20, w0"
,,,"0x4008f8 ldr w0, [sp, #44]"
,,,"0x4008fc mul w0, w19, w0"
,,,"0x400900 add w0, w1, w0"
,,,"0x400904 ldp x19, x20, [sp]"
,,,"0x400908 add sp, sp, #48"

```

or for dual core:

```

Index(n),Source(t),Type(t),Description(lt),Address(nh),Destination(nh),Timestamp(n),
1,CORE 0,Info,"SYNC packet - ETM",,,,0,
2,CORE 0,Info,"Trace On packet - ETM -> start tracing after a (possible) trace gap",,,,0,
3,CORE 0,Info,"Context packet - ETM",,,,0,
,,," exception level: 2"
,,," state: 64-bit"
,,," security state: non-secure"
,,," VMID: 46"
,,," ContextID: 1454120766"
4,CORE 0,Software Context,"software context id = 1454120766",,,,0,
5,CORE 1,Info,"SYNC packet - ETM",,,,0,
6,CORE 1,Info,"Trace On packet - ETM -> start tracing after a (possible) trace gap",,,,0,
7,CORE 1,Info,"Context packet - ETM",,,,0,
,,," exception level: 2"
,,," state: 64-bit"
,,," security state: non-secure"
,,," VMID: 46"
,,," ContextID: 1454120766"
8,CORE 1,Software Context,"software context id = 1454120766",,,,0,
9,CORE 0,Linear,Function main,0x400954,,0,
,,,"0x400954 stp x29, x30, [sp, #-32]!"
,,,"0x400958 mov x29, sp"
,,,"0x40095c str wzr, [x29, #28]"
,,,"0x400960 ldr w0, [x29, #28]"
,,,"0x400964 cmp w0, wzr"
10,CORE 0,Linear,Function main,0x400968,,0,
,,,"0x400968 b.ne #20"

```

3.6 ARMv8 NoC decoder

This component offers complet decoding services that can be used to interpret any NoC trace for a ARMv8 target, transforming the raw data into useful information. NoC (Network on Chip) trace module is a real-time trace module providing information on transactions in NoC. There are 2 NoCs that could generate trace: Main (for LDPAA transactions) and HSIO (High-speed I/O).

The main features of a NoC trace are:

- Trace generation - Outputs information that helps users understand the transactions' flow inside NoC
- Statistics generation - Outputs statistical counters

Decoders

The advantages of the ARMv8 standalone decoder tool are:

- Size - The size of the decoder tool is appropriate enough to get integrated and updated in any automatic environment.
- Speed - All the services required are encapsulated into one application and all the calls are processed without any delays that may be caused by cross languages requests
- Simple API - It can be effortlessly integrated into any framework.

This section explains the following:

- [Component details](#)
- [Decoder example](#)

3.6.1 Component details

ARMv8 standalone decoder is delivered as an independent product without any other libraries. Below listed are the options available and their short descriptions.

```
noc.decoder [Options] <input_file> <output_file>
```

The following table lists the available options and their respective descriptions:

Table 13. Available options details

Options	Details
-h [--help]	Displays help message
-i [--input-file] path	Displays the input file path
-o [--output-file] path	Displays the output file path
-p [--show-progress]	Shows the progress periodically
-x [--ls2-xml] path	Shows the path to the ARMv8 description file
-v [--version]	Displays the product version

The application uses the output file extension to determine the format, ".txt" for text and ".csv" for comma separated values. If this parameter is not specified the trace events are redirected to standard output.

The following table lists the hidden options and their respective descriptions:

Table 14. Available options details

Options	Details
--HELP	Displays all options (including hidden)
-f [--output-format]	Displays the output format. The supported formats are .txt and csv.
-c [--chunk-size] arg (=10240)	Specifies the processing chunk size in bytes. Default value is 10240 bytes.
-s [--include-sync]	Specifies whether the formatted stream includes synchronization messages or not. If this option exists, the unformatting algorithm starts after the first synchronization packet. Once synchronized, all subsequent synchronization packets will be discarded.

NOTE

All hidden options were created to cover couple of particular scenarios and for testing purposes until ARMv8 hardware trace will be available.

For ARMv8, NoC networks have associated the IDs described in text. ATB IDs used for ARMv8 are:

- Main NoC - 0x50 (80 in decimal)
- HSIO NoC - 0x51 (81 in decimal)

The decoder has an xml file NoC_Decoder.xml with the description of the most important attributes required by the decoder.

Example of NoC_Decoder.xml file

```
<ConfigBlock>
  <Name>MAIN_NOC</Name>
  <BoolAttribute>
    <Name>Has Cell sync</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <UInt32Attribute>
    <Name>ATB ID</Name>
    <DefaultValue>80</DefaultValue>
  </UInt32Attribute>
</ConfigBlock>
<ConfigBlock>
  <Name>HSIO_NOC</Name>
  <BoolAttribute>
    <Name>Has Cell sync</Name>
    <DefaultValue>>false</DefaultValue>
  </BoolAttribute>
  <UInt32Attribute>
    <Name>ATB ID</Name>
    <DefaultValue>81</DefaultValue>
  </UInt32Attribute>
</ConfigBlock>
```

3.6.1.1 Execution flow

The following figure illustrates the execution flow of NoC decoder:

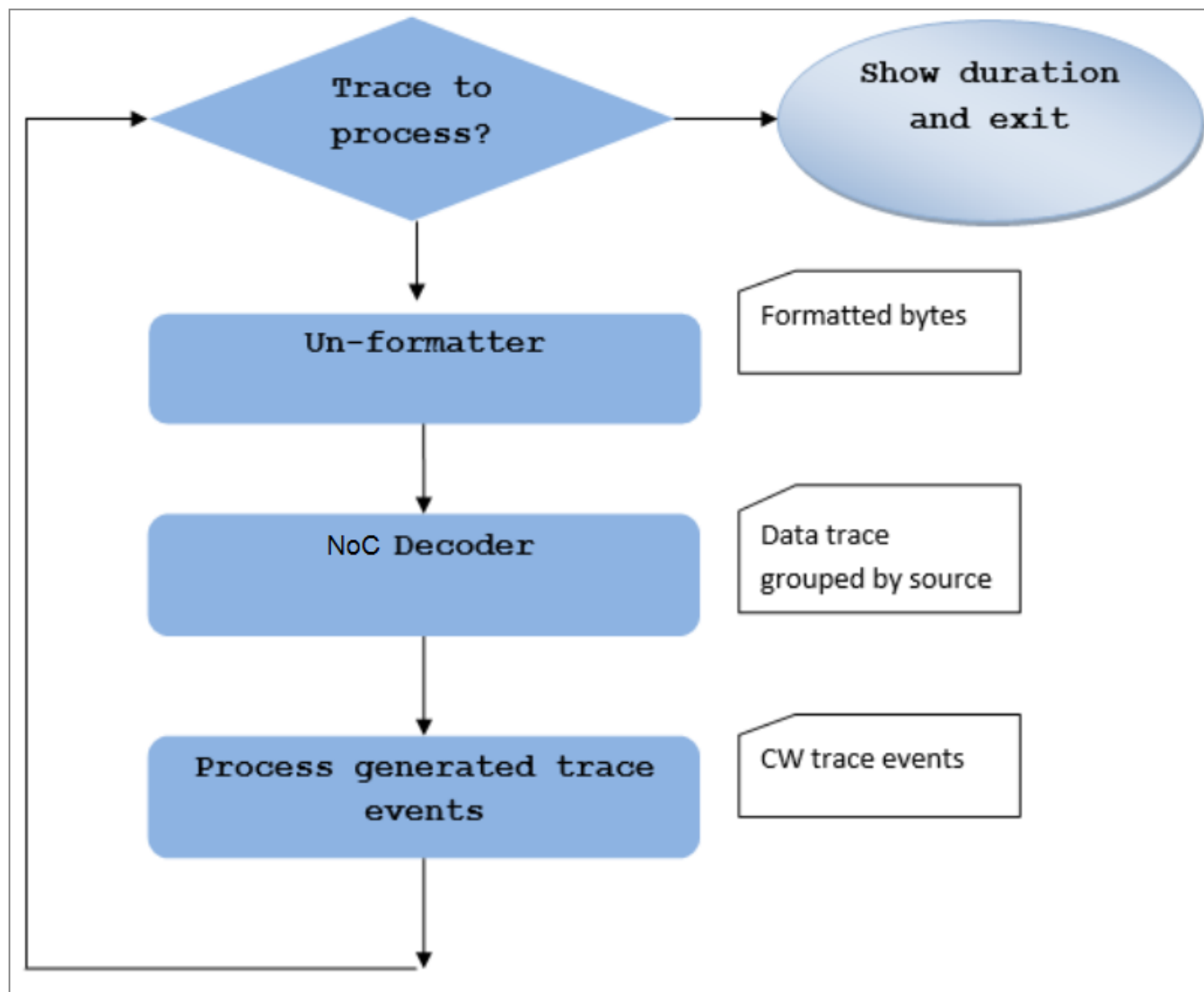


Figure 11. Execution flow

3.6.1.2 Decoder example

The ARMv8 decoder product archive contains sample traces under the internal directory CodeWarrior_Examples. They are intended to be deployed to <CodeWarrior_install_dir>/ARMv8.

NOTE

However, the files can be unpacked to any other path.

The contents of this directory are:

- ATB formatted sample trace file: It contains the following files:
 - hsio_atbid_0x51.dat
 - main_atbid_0x50.dat
 - main_and_hsio_atbid_0x50_0x51.dat
- An xml description file: The NoC_Decoder.xml file contains the description of trace data.

This section explains:

- [Run the example](#)
- [Output files](#)

3.6.1.2.1 Run the example

Follow the steps to run the NoC decoder:

1. Open the command prompt and change to the example directory (<deploy_path>)
2. For Linux, add the binaries directory to the libraries search path by running: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: <deploy_path>/sa_ls/bin`
3. Run the command
 - For Windows: `<deploy_path>\sa_ls\bin\noc.decoder.exe -p -ihsio_atbid_0x51.dat -o NoC_HSIO_handmade_trace.csv -xNoC_Decoder.xml`
 - For Linux: `<deploy_path>/sa_ls/bin/noc.decoder -p -ihsio_atbid_0x51.dat -o NoC_HSIO_handmade_trace.csv -xNoC_Decoder.xml`
4. Open output files `NoC_HSIO_handmade_trace.csv` and view the results.

3.6.1.2.2 Output files

The result of the output files appear in csv format.

Example of output files in csv format

```
Index(n),Source(t),Type(t),Description(lt),Address(nh),Destination(nh),Timestamp(n),
1,HSIO_NOC,Info,"SYNC packet - NoC",,,0,
2,HSIO_NOC,Info,"HSIO NoC Trace packet",,,0,
,,," probeId=0x0"
,,," lockBit=0x0"
,,," initFlow=0x7"
,,," targFlow=0xc"
,,," targSubRange=0x0"
,,," seqId=0x0"
,,," opcode=0x4"
,,," status=0x2"
,,," length=0xa"
,,," addr=0x4"
,,," user=0x7abcdef00"
,,," errCode=0x0"
,,," debug=0x0"
3,HSIO_NOC,Info,"HSIO NoC Statistics packet",,,0,
,,," probeId=0x1"
,,," isCounterWidth16=0x0"
```

3.6.2 Conclusion

Standalone decoders can be integrated into automated testing frameworks, for example, compare the output of an application using a golden file, or by developers to get an overview of an application, squeeze the best performance from the system or to investigate issues.

How to Reach Us:**Home Page:**freescale.com**Web Support:**freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, CodeWarrior, and QorIQ are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Layerscape is trademark of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. ARM, Cortex, Cortex-A53, Cortex-A57, and TrustZone are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2014, Freescale Semiconductor, Inc.