Architecture Overview for Debug

White Paper

Version 1.3

19 March 2021

MIPI Board Approved for Public Distribution 21 April 2021

Contents

Figures

Tables

Release History

Date	Version	Description
14 February 2014	v1.0	Board approved release
29 September 2016	v1.1	Board approved release
29 August 2018	v1.2	Board approved release
19 March 2021	v1.3	Board approved release

1 Overview

1.1 Scope

Recent technological developments have resulted in a quantum leap in the complexity of SoCs. Systems that were formerly deployed on one or more PCBs are now being instantiated as single discrete devices. While this trend is in general a boon to manufacturers and consumers of various systems, it has greatly increased the complexity of system debug and optimization. Signals and interfaces that used to be visible at test points on a PCB are now deeply embedded inside a SoC. The use of tried and true methods of probing buses and signals with dedicated Debug and Test equipment is now virtually impossible.

This increase in debug complexity is being addressed by IP vendors, SoC developers, OEMs and tools vendors. New technologies are being deployed that provide the visibility required in these complex and deeply embedded designs. In order to maximize the utility and efficiency of debug, converging on common interfaces and protocols used by these new technologies is essential.

To meet this need, the MIPI Debug Working Group (Debug WG) are developing a portfolio of standards and other documents that address the particular needs of debug. Some of the areas of focus are listed below.

- Minimizing the pin cost and increasing the performance of the basic debug interface
- Increasing the bandwidth, capability and reliability of the high-performance interfaces used to export high bandwidth, unidirectional debug data (e.g., processor trace data) to the debug tools
- Deploying debug connectors that are physically robust and have the performance required for the high bandwidth demands of modern debug technologies
- Developing generic trace protocols that allow many different on chip trace sources to share a single trace data flow to the debug tools
- Maximizing debug visibility in fielded systems by reusing some of the functional interfaces/connectors for debug
- Utilizing the new high bandwidth functional interfaces being deployed on various systems as a transport for debug

This document provides an overview of the efforts to address these goals and provides summaries of the documents that address them in detail.

2 Terminology

2.1 Definitions

1149.1: Short for IEEE 1149.1. See ***[IEEE01]***.

1149.7: Short for IEEE 1149.7. See ***[IEEE02]***.

Application Function: All functions of the TS other than Debug and Test Functions.

Application Processor: A programmable CPU (or CPU-based system on a chip (SoC) which may include other programmable processors such as DSPs), primarily, but not necessarily exclusively, programmed to coordinate the application processing and user interface processing.

Application Software: Used here to mean the target resident code that runs on the target processor. This includes the operating system as well as the program(s) running with the OS.

Basic Debug Communication: Debug communication needed through an 1149.1 (or equivalent) port only to manage basic debug communication functions such as run control, hardware breakpoints and watchpoints, and examining system state.

Boundary Scan: A production test mechanism where interconnects between chips or logic blocks in an SoC are verified by forcing known test patterns into the system via a serial scan interface, activating a test mode, and then scanning out the resultant values to test against expected results.

Built-in Self-Test (BIST): On-chip logic function that verifies all or a portion of the internal functionality of a SoC during production tests. BIST logic requires minimal interaction with external test infrastructures and speeds up verification of complex SoCs.

Debug: To detect, trace, and eliminate SW mistakes. Also used to get an insight into an embedded processor system for performance measurements and debug of system level hardware. Used in this document in an inclusive way that encompasses stop/start/break/step debugging as well as non-halting methods such as trace.

Debug Access and Control Subsystem (DACS): The subsystem that provides a path for the DTS to obtain direct access to application visible system resources (registers and memory).

Debug and Test Controller (DTC): The hardware system that is responsible for managing communications with a system being debugged (the Target System).

Debug and Test Function: A block of on-chip logic that carries out a debug function such as run control, providing debug access to system resources, Processor Trace, or test capability.

Debug and Test Interface (DTI): The interface between the Debug and Test System (DTS) and the Target System (TS). The interface enables access to basic debug communication, the trace port, streaming data (input and output), and other debug or test capabilities.

Debug and Test System (DTS): The combined HW and SW system that provides a system developer debug visibility and control when connected to a Target System. The system incorporates:

- A host PC, workstation or other processing system, running the debug or test software, controlling the Debug and Test Controller. See also: Debug and Test Controller (DTC).
- Debugger: The debugger software, part of the Debug and Test System. It interacts with the Debug and Test Controller and provides the (graphical) user interface for operating the Debug and Test Controller (like commanding single step, setting breakpoints, memory display/modify, trace reconstruction, etc.)

Debug and Test Target (DTT): A component in the Target System that implements one or more Debug and Test Functions. The interfaces to Debug and Test Targets, where different from the DTI, are not within the scope of this specification. Examples include the debug control module on a CPU, debug interface to system memory, or the configuration interface to an on-chip trace module.

Debug Instrumentation and Visibility Subsystem (DIVS): The subsystem that provides communication and storage of data generated by debug instrumentation modules (like processor and system trace) in the target system.

Debug Physical Interfaces (DPI): The chip and board level interfaces used to connect the DTC to the on-chip debug functions.

Double Data Rate (DDR): A parallel data interface that provides valid data on both the rising and falling edge of the interface clock.

Electrical: The definition of:

- Signal voltage levels, current drain and drive strength on inputs, outputs, and bi-directional pins
- Rise and fall times and expected loads for device pins.

Function Assignment: The mapping of functions to signals (and thus to pins as per the current Pin Assignment, e.g., Debug port pin [5] = CLK = TRACECLK.)

Function Select: The method by which the Basic Debug Communication channel can be switched between use for Debug Functions and use for operations needed to configure the debug system.

Gigabit Trace (GbT): A system architecture that supports transporting trace data over high-speed networks and transports. See ***[MIPI04a]***.

Gigabit Debug (GbD): A set of network-specific adaptor specifications for mapping SneakPeek and Gigabit Trace to various functional networks.

Hardware Protocol: The signal protocol required for a Debug and Test Controller to correctly move control or data information between the DTC and Target System.

High Bandwidth Connection: A Mating Connection, Pin Assignment and Electrical specification for full functionality, high frequency, higher pin count connection between the Target System and the Debug and Test Controller / TPA.

High-speed Trace Interface (HTI): The transport specification that defines the electrical and timing characteristics of high-speed serial trace export interfaces. See ***[MIPI09]***.

IEEE 1149.7 (basic debug communication): Enhanced IEEE1149.1 Debug and Test communication standard, configurable from 4 to 2 pins. The IEEE 1149.7 interface can be viewed as providing functionality enhanced compared to 1149.1 for Basic Debug Communication and test and with fewer pins. A two-way communication channel for exclusive Debug and Test uses. See ***[IEEE02]***.

Intellectual Property (IP): any patents, patent rights, trademarks, service marks, registered designs, topography or semiconductor mask work rights, applications for any of the foregoing, copyrights, unregistered design rights, trade secrets and know-how and any other similar protected rights in any country. Any IP definition by MIPI Bylaws will supersede this local one.

Low Pin Count Connection: A Mating Connection, Pin Assignment and Electrical specification for Basic Debug Communication and limited Trace Port functionality, lower frequency, low pin count connection between the Target System and the Debug and Test Controller / TPA.

Mating Connection: The connector to be used, defined by specific manufacturer and part number. The required keep out area for board design to enable unobstructed connector mating. The definition of cable characteristics and terminations may include the characteristics of a connection from the point it leaves an output buffer in a chip on the target or host side, routing on a printed circuit board on the DTC or Target System side, cabling between the signal source and destination, and any connections (via connectors) in the signal path.

Min-Pin: An interface for Basic Debug Communication with a minimal number of pins (2), using either IEEE 1149.7, SWD or I3C.

Mode Select: A method for selecting a different Mating Connection, a different operating mode, a different electrical mode or a combination of these, for example switching between 1149.1 and 1149.7.

Narrow Interface for Debug and Test (NIDnT): A signal-mapping specification that defines how to reuse the functional interfaces commonly available on fielded systems for debug. See ***[MIPI05]***.

Nexus: An IEEE-ISTO 5001™ standard interface for embedded processor debug. The Nexus standard includes support for Basic Debug Communication as well as instruction and data tracing. See ***[ISTO01]***.

Other Debug: Debug functions not covered by 1149.1, 1149.7 or the Trace Port for example off-chip memory emulation.

Parallel Trace Interface (PTI): The interface specification that defines the electrical and timing characteristics of trace export interfaces that consist of a single clock and multiple data signals. See ***[MIPI02]***.

Pin Assignment: The mapping of signals to pins, e.g., SIGNAL_NAME on pin number N. This may include restrictions on allowable pin assignments.

Processor Trace: The non-intrusive capture and logging of the activity of an embedded processor and the subsystem in which the processor resides. Processor trace generally consists of one or more of the following trace types, but it is not limited to these:

- Instruction (PC) Trace – Application execution flow can be reconstructed by processing the logged information
- Data Trace – Data access activity is captured at the processor boundary

The captured data is encoded for efficiency and this data is stored on-chip for later upload or immediately transmitted through a chip interface to an off-chip receiver.

Return Test Clock (RTCK): A non-standard extension to 1149.1 that provides a feedback path for pacing transaction on the interface.

Serial Wire Debug (SWD): An interface used for Basic Debug Communication. See ***[ARM01]***.

Series Scan Topology: A connection scheme where the control signals on the debug interfaces are connected in parallel, but the data signals are daisy chained.

Silicon Test Subsystem (STS): This subsystem supports communication between the DTS and the on-chip logic used for production test (boundary scan, BIST, etc.).

Star Scan Topology: A connection scheme where both the control and data signals on the debug interfaces are connected in parallel.

System Software Trace (SyS-T): A format for transporting software traces and debugging information between a target system (TS) running embedded software, and a debug and test system (DTS), typically a computer running one or more debug and test applications (debuggers and trace tools).

System Trace Module (STM): A system trace interface with capabilities to export SW (printf type) and HW generated traces (e.g., PC trace and memory dumps). Typical implementation is 4-bit parallel double data rate. The STM uses a nibble-oriented protocol called STP. See ***[MIPI03]***.

System Trace Protocol (STP): The protocol used with STM. See ***[MIPI03]***.

System on a Chip (SoC): An electronic system in which all (or most of) the functional modules are integrated on a single silicon die and packaged as a single chip.

System Trace: In the context of this document, system trace refers to SW Instrumentation Trace and HW Instrumentation Trace.

- SW Instrumentation Trace:Message output from instrumented application code.
- HW Instrumentation Trace: Messages triggered by transactions/events on the SoC infrastructure(s) and other HW modules in the system.

Target System (TS): The system being debugged, up to the Debug and Test Interface (DTI). The TS might be a discrete device (a chip) or a collection of 1 to N discrete devices grouped on a board or collection of boards. The TS might also contain 0 to N individual Debug and Test Targets.

Test Access Port (TAP): The on-chip interface to Debug and Test resources. Both 1149.1 and 1149.7 support the concept of a Test Access Port.

Timing: The AC characteristics of debug signals at the pins of the target device. Includes skew, jitter, rise and fall times, data/clock alignment, set-up and hold times. While this is shown to be common between all connectors, there will likely be some variation, for example the Gigabit connector might not have separate clock and data pins.

Trace: A form of debugging where processor or system activity is made externally visible in real-time or stored and later retrieved for viewing by an applications developer, applications program, or, external equipment specializing observing system activity.

Trace Channel: A group of one or more signals and a clock that move trace information from the TS to the DTS. There may be more than one Trace Channel between the TS and DTS.

Trace Data Protocol: The implementation-specific encoding of a particular type of trace by a particular module.

Trace Port: An output port for the transmission of real-time data indicating the operation of the target (e.g., program execution and/or data bus transactions). Data transmitted across the Trace Port may be generated by hardware, software instrumentation, or by a mixture of the two. This does not include trace collected on-chip for later upload.

Trace Port Analyzer (TPA): An external hardware unit for collecting data transmitted from the Trace Port. The data might be stored locally in real time before uploading to the host debug tools for later analysis by the user, e.g., a logic analyzer or a unit customized to record trace information would both qualify.

Trace Wrapper Protocol (TWP): A protocol that wraps trace from different sources into a single stream for simultaneous capture by a single TPA. See ***[MIPI04]*** and ***[MIPI04a]***.

Trigger: An indication that a specific system event has occurred. A trigger may be an input to the TS, a signal within the TS, or an output from the TS. The response to the trigger is determined by the entity to which the trigger is sent.

2.2 Abbreviations

e.g.	For example (Latin: exempli gratia)
i.e.	That is (Latin: id est)

2.3 Acronyms

AC	Alternating Current
BIST	Built-in Self-Test
CCC	Common Command Code
CPU	Central Processing Unit
DACS	Debug Access and Control Subsystem
DDR	Double Data Rate
DFT	Design for Test
DIP	Data Integrity Package
DIVS	Debug Instrumentation and Visibility Subsystem
DNI	Debug Network Interfaces
DPI	Debug Physical Interfaces

DSP	Digital Signal Processor
DTC	Debug and Test Controller
DTI	Debug and Test Interface
DTS	Debug and Test System
DTT	Debug and Test Target
GbD	Gigabit Debug
GbT	Gigabit Trace
HTI	High-speed Trace Interface
HW	Hardware
I3C	Improved Inter Integrated Circuit
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
IPS	Internet Protocol Sockets
IPR	Intellectual Property Rights
ISTO	Industry Standards and Technology Organization
JTAG	Joint Test Action Group
microSD	Micro Secure Digital
MMC	MultiMediaCard
NIDnT	Narrow Interface for Debug and Test
nTRST	Not Test Reset
OFM	Original Functional Mode
OS	Operating System
PC	Personal Computer or Program Counter
PCB	Printed Circuit Board
PHY	Physical Interface
POR	Power on Reset
PTI	Parallel Trace Interface
RF	Radio Frequency
RTCK	Return Test Clock
SIM	Subscriber Identity Module
SoC	System on a Chip
SPP	SneakPeek Protocol
SPTB	SneakPeek Transfer Block
STM	System Trace Module

STP	System Trace Protocol
STS	Silicon Test Subsystem
SW	Software
SWD	Serial Wire Debug
SyS-T	System Software Trace
TAP	Test Access Port
TCK	Test Clock
TCKC	Test Clock Compact
TCP	Transmission Control Protocol
TDI	Test Data Input
TDIC	Test Data Input Compact
TDO	Test Data Output
TDOC	Test Data Output Compact
TDP	Trace Data Protocol
TMS	Test Mode Select
TMSC	Test Mode Select Compact
TPA	Trace Protocol Analyzer
TS	Target System
TWP	Trace Wrapper Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
WG	Working Group

2.4 Use of Inclusive Language

The MIPI Alliance is committed to the use of inclusive language in its specifications and documentation. Terms such as “master” and “slave” are being replaced with better, more descriptive terms and the MIPI Alliance is actively updating its documentation to use the new inclusive language. This document will no longer include these non-inclusive terms and the reader will notice the new language.

Note that at the time of publication, not all the documents referenced in this document were updated. The table below is a list of the legacy terms that have been updated in this document with the new inclusive language.

Referenced Specification	Legacy Term	New Term
Debug for I3C	Master	Controller
	Multi-Mastering	Multiple Controller-capable
	Slave	Target
STP	Master	Major Source

3 References

Note:

Public Release Editions are available for most of the following MIPI Alliance adopted specifications, see links in respective sections.

[MIPI01] *MIPI Alliance Recommendation for Debug and Trace Connectors*, version 1.10.00 and higher, MIPI Alliance, Inc., 16 March 2011.

[MIPI02] *MIPI Alliance Specification for Parallel Trace Interface (PTI^{SM})*, version 2.0, MIPI Alliance, Inc., 3 May 2011.

[MIPI03] *MIPI Alliance Specification for System Trace Protocol (STP^{SM})*, version 2.2, MIPI Alliance, Inc., 21 August 2015.

[MIPI04] *MIPI Alliance Specification for Trace Wrapper Protocol (TWP^{SM})*, version 1.00.00, MIPI Alliance, Inc., 23 February 2010.

[MIPI04a] *MIPI Alliance Specification for Trace Wrapper Protocol (TWP^{SM})*, version 1.1, MIPI Alliance, Inc., 3 September 2014.

[MIPI05] *MIPI Alliance Specification for Narrow Interface for Debug and Test (NIDnT^{SM})*, version 1.2, MIPI Alliance, Inc., 17 August 2017.

[MIPI06] *MIPI Alliance Specification for SneakPeek*^{SM} *Protocol (SPP^{SM})*, version 2.0, MIPI Alliance, Inc., 21 May 2019.

[MIPI07] *MIPI Alliance Specification for Gigabit Debug for USB*, version 1.1, MIPI Alliance, Inc., 12 October 2017.

[MIPI08] *MIPI Alliance Specification for Gigabit Debug for Internet Protocol Sockets*, version 1.0, MIPI Alliance, Inc., 20 May 2016.

[MIPI09] *MIPI Alliance Specification for High-Speed Trace Interface (HTI^{SM})*, version 1.0, MIPI Alliance, Inc., 10 March 2016.

[MIPI10] *MIPI Alliance Specification for System Software Trace (SyS-T^{SM})*, version 1.0, MIPI Alliance, Inc., 1 December 2017.

[MIPI11] *MIPI System Software Trace (MIPI SyS-T) – Example Code*, https://github.com/MIPI-Alliance/public-mipi-sys-t, MIPI Alliance, Inc., last accessed 19 March 2021.

[MIPI12] *MIPI Alliance Specification for Debug for I3C*, version 1.0, MIPI Alliance, Inc., 21 April 2020.

[MIPI13] *MIPI Alliance Specification I3C*^{SM}, version 1.1, MIPI Alliance, Inc., 27 November 2019.

[MIPI14] *MIPI Alliance Specification I3C Basic*^{SM}, version 1.0, MIPI Alliance, Inc., 19 July 2018.

[IEEE01] IEEE Std 1149.1™-2013, *Standard for Test Access Port and Boundary-Scan Architecture*, Institute of Electrical and Electronic Engineers, 2013.

[IEEE02] IEEE Std 1149.7™-2009, *Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary Scan Architecture*, Institute of Electrical and Electronic Engineers, 2009.

[ISTO01] IEEE-ISTO 5001™-2012, *The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface*, version 3.0.1, IEEE- Industry Standards and Technology Organization, 2012.

[ARM01] *ARM® CoreSight™ Architecture Specification*, version 3.0, ARM Limited, 2017.

[AUR01] *Aurora 8B/10B Protocol Specification*, SP002 (v2.3), Xilinx, Inc., 145 http://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_protocol_spec_sp002.pdf, 1 October 2014.

[USB01] *USB 3.1 Device Class Specification for Debug Device*, Revision 1.0, http://www.usb.org, 14 July 2015.

This page intentionally left blank.

4 Debug System

4.1 System Framework

The modern systems on a chip often have complex Debug and Test architectures. In a simplistic view, the modern SoC Debug and Test architecture can be broken down into the following major subsystems:

- **Debug Access and Control Subsystem (DACS)** – This subsystem provides a path for the DTS to obtain direct access to application visible system resources (registers and memory). It also provides bidirectional communication for configuration and control of debug specific modules in the TS. The communication between the debug and the DACS is generally implemented via one of the following (this is not an exhaustive list):
  - Serial scan via a dedicated Debug and Test interface on the device
  - Memory mapped using a dedicated debug interconnect or in some cases the application visible system interconnect
  - A proprietary communication protocol and interface on the device boundary
- **Debug Instrumentation and Visibility Subsystem (DIVS)** – This subsystem provides communication and storage of data generated by debug instrumentation modules (like processor and system trace) in the target system. DIVS communication path to the DTS is usually via high-speed serial or trace interfaces and is generally unidirectional.
- **System Test Subsystem (STS)** – This subsystem supports communication between the DTS and the on-chip logic used for production test (boundary scan, BIST, etc.). Access to the STS is generally accomplished via serial scan.
- **Debug Physical Interfaces (DPI)** – The physical interfaces that support debug at the SoC boundary and on the PCB.
- **Debug Network Interfaces (DNI)** – The internal interfaces that allow debug and trace data to be transmitted to and from the DTS on functional networks. This communication is with dedicated intelligent resources (sometimes called the *Debug Butler*) that possibly:
  - Enable *bare metal* debug on systems where the normal functional communication management is not yet functioning
  - Allow debug to minimize or eliminate the use of functional resources for managing debug communications

Figure 1 provides a top-level view of how all the pieces of the Debug and Test architecture are integrated on a device.

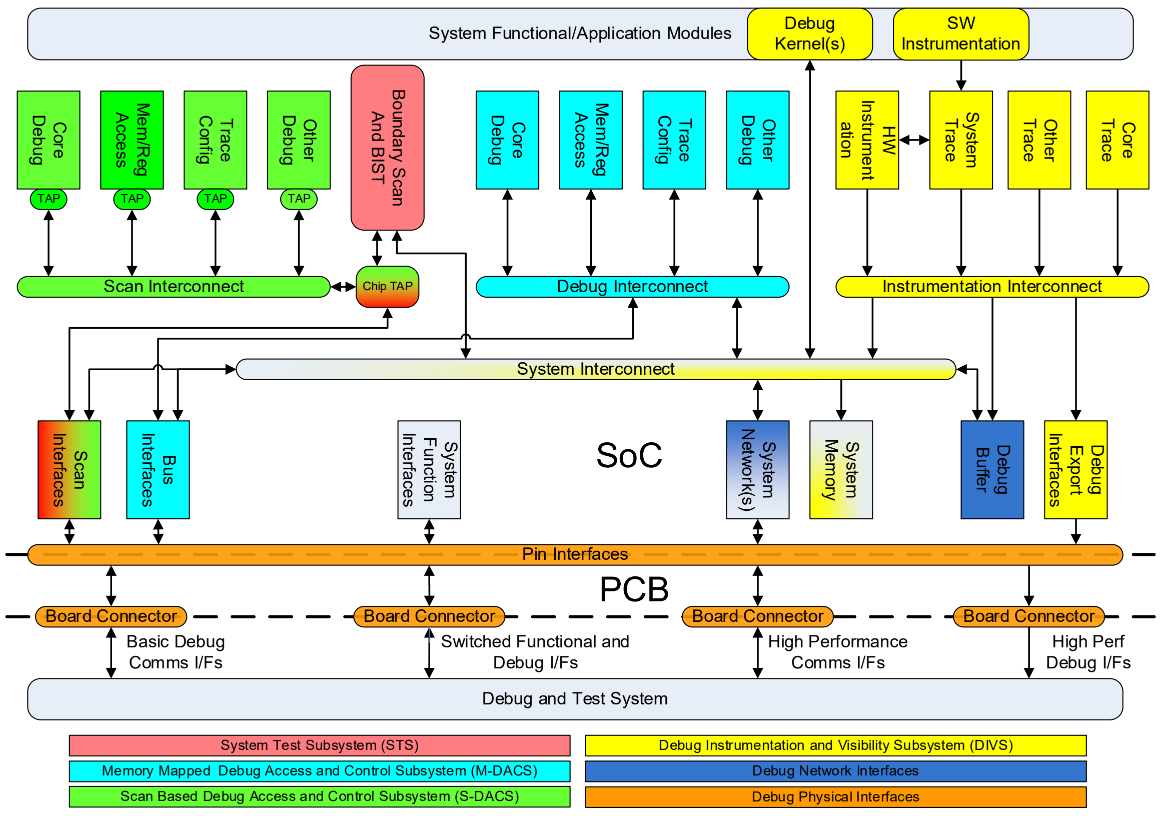


Figure 1 MIPI Debug Generic System Framework

4.2 The MIPI Debug and Test System

The MIPI Debug WG effort does not address all the functional blocks in the generic framework. The Debug WG standards and recommendations focus on device and board interfaces and protocols. There is also an effort to standardize on communications for debug instrumentation (i.e., trace protocols), but with a generic approach that maintains compatibility with protocols that already exist. ***Figure 2*** illustrates the areas of the framework that are targeted by the various MIPI Debug specifications and recommendations addressed in this document.

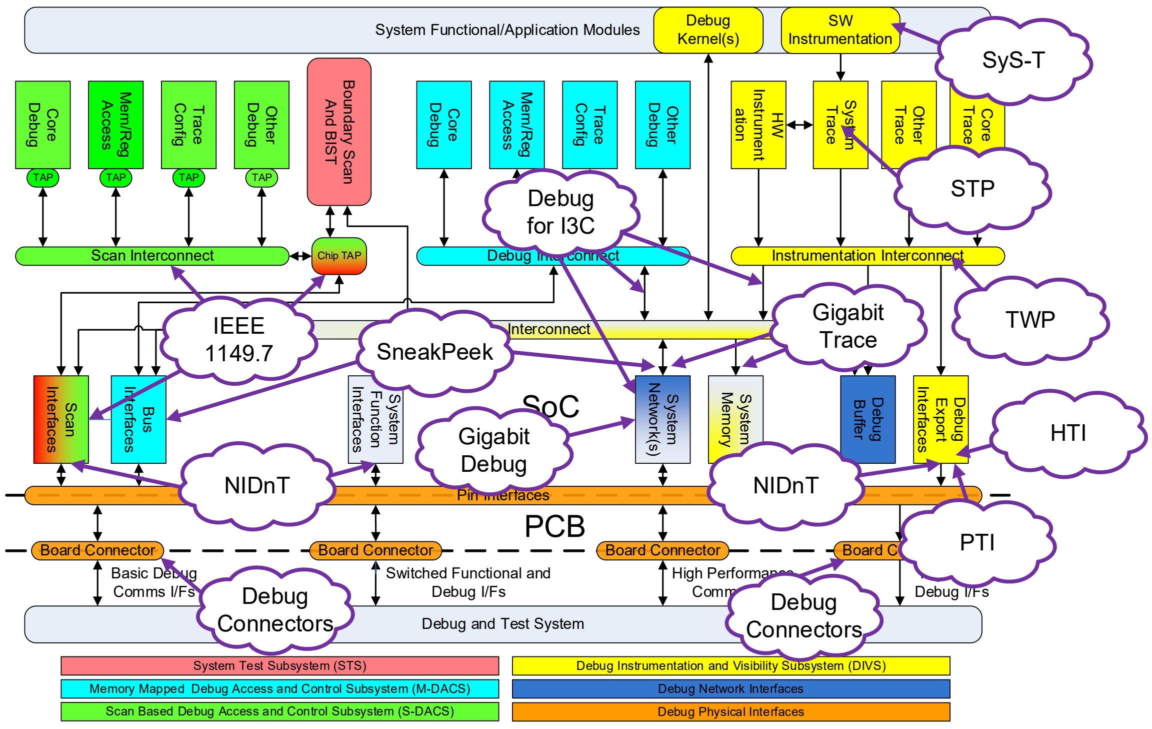


Figure 2 MIPI Debug Documentation and the Debug Architecture

Figure 3 shows a more detailed block diagram showing how the generic debug framework can be realized across an entire multiple-chip system. The devices share the basic debug, trace and functional interfaces. Basic run control can be provided via the shared debug connection. Trace transport can utilize a shared link dedicated to trace or a standard application visible network. In all cases, the footprint of the debug interface to the tools is greatly reduced.

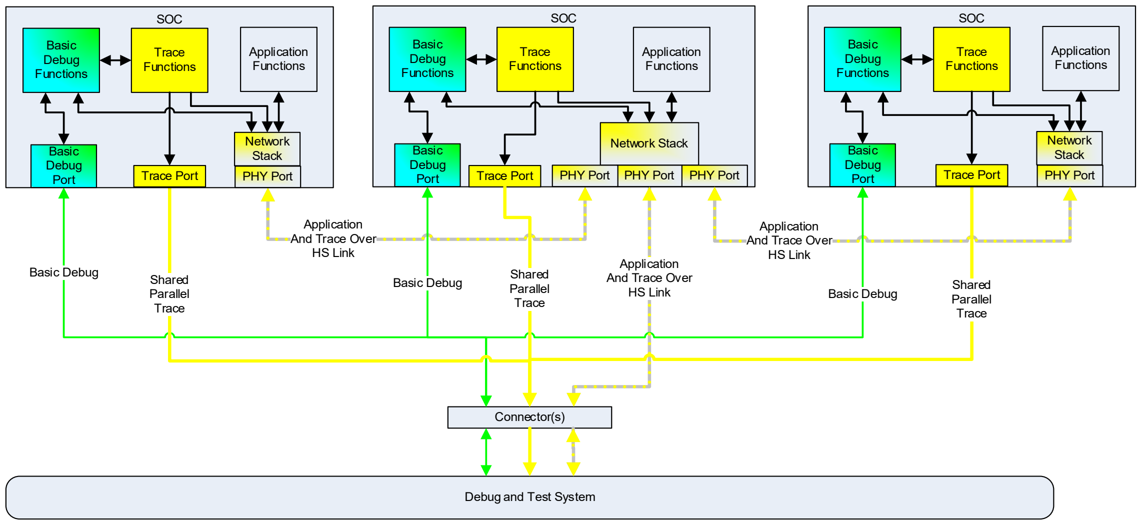


Figure 3 Example MIPI System Overview

5 Debug Physical Interfaces (DPI)

5.1 Parallel Trace Interface (PTI) Specification

5.1.1 Trace and Debug Overview

It has become an accepted axiom that as the complexity of an embedded system increases, the need for system designers and developers to obtain visibility into the behavior of the system increases proportionally. One of the most common methods for providing this visibility is to provide a streaming interface on an embedded System on a Chip. This interface can be used to export data about system functionality and behavior to a host system for analysis and display. Since the data exported on this interface often allows developers to reconstruct (or "trace") some portion of system activity, these types of interface have commonly been referred to as Trace Interfaces or Trace Ports. Examples of trace data include:

- The instruction execution sequence for one or more embedded processors. This is commonly referred to as Program Counter (PC) Trace.
- Data bus transactions made by an embedded processor core. This is commonly referred to as Data Trace.
- Snapshots of transactions on the system interconnect(s). This is commonly referred to as System Trace.
- Streaming output from instrumented application code. This is commonly referred to as Instrumentation Trace.

The bandwidth requirements for the common trace functions listed above often compel system designers to implement the trace interface as a parallel interface with multiple data signals and a clock. For purposes of this document, the trace interface will subsequently be referred to as the Parallel Trace Interface or PTI.

5.1.2 Relationship to MIPI Debug Architecture

Figure 4 shows the standard MIPI debug architecture highlighting the functional areas addressed by the PTI specification.

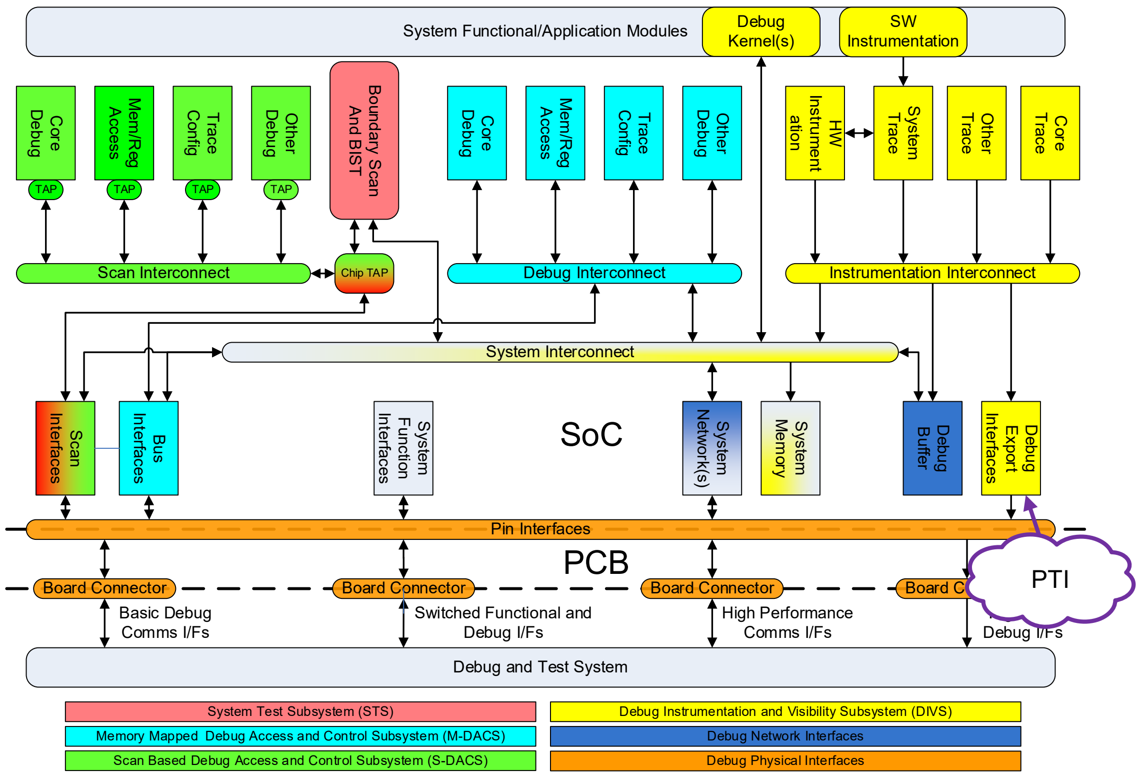


Figure 4 PTI in the MIPI Debug Architecture

5.1.3 Trace Scenarios

A typical embedded system may have one or more HW modules that produce trace data. The typical flow is outlined below and illustrated in ***Figure 5***.

- Debug and Test Targets (DTTs) reside in the Target System (TS).
- Trace modules inside a DTT contain one or more HW sub-modules that capture the system transactions with the required data. See the Trace Collect block in ***Figure 5***.
- One or more HW modules encode or compress the data into an implementation specific encoding(s). These encoding(s) are called the Trace Data Protocols (TDPs). See the Trace Format block in ***Figure 5***.
- One or more HW modules export the encoded data to the DTC using device pins. The interface used to transfer this data is the Parallel Trace Interface or PTI. See the Trace Export block in ***Figure 5***.
- The DTC captures the data.
- The data is decoded and analyzed using the DTS.

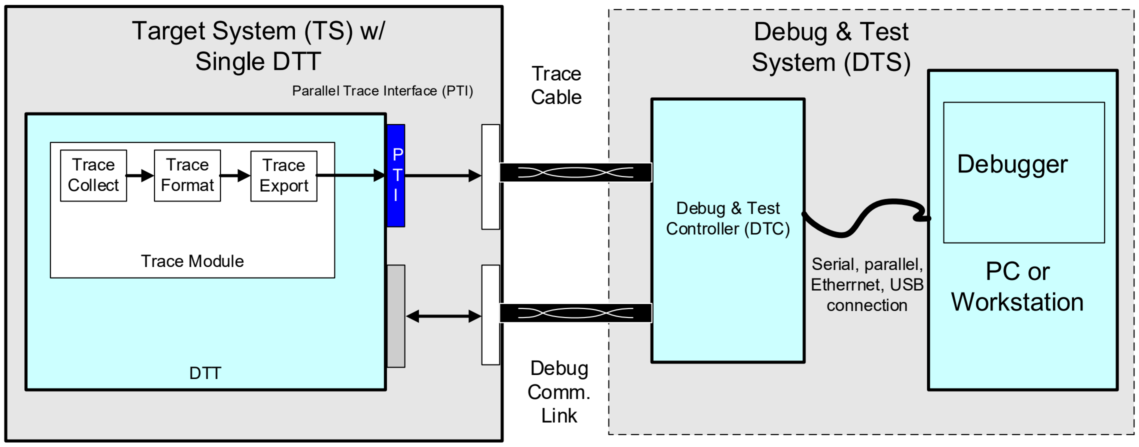

Target System (TS) w/ Single DTT
Parallel Trace Interface (PTI)
Trace Collect
Trace Format
Trace Export
Trace Module
PTI
DTT
Trace Cable
Debug Comm. Link
Debug & Test System (DTS)
Debug & Test Controller (DTC)
Serial, parallel, Etherrnet, USB connection
Debugger
PC or Workstation

Figure 5 Example System with PTI

Note that only HW modules directly responsible for producing the data and clock of a PTI are required to implement a PTI. ***Figure 6*** shows how the PTI implementation is dependent upon system configuration.

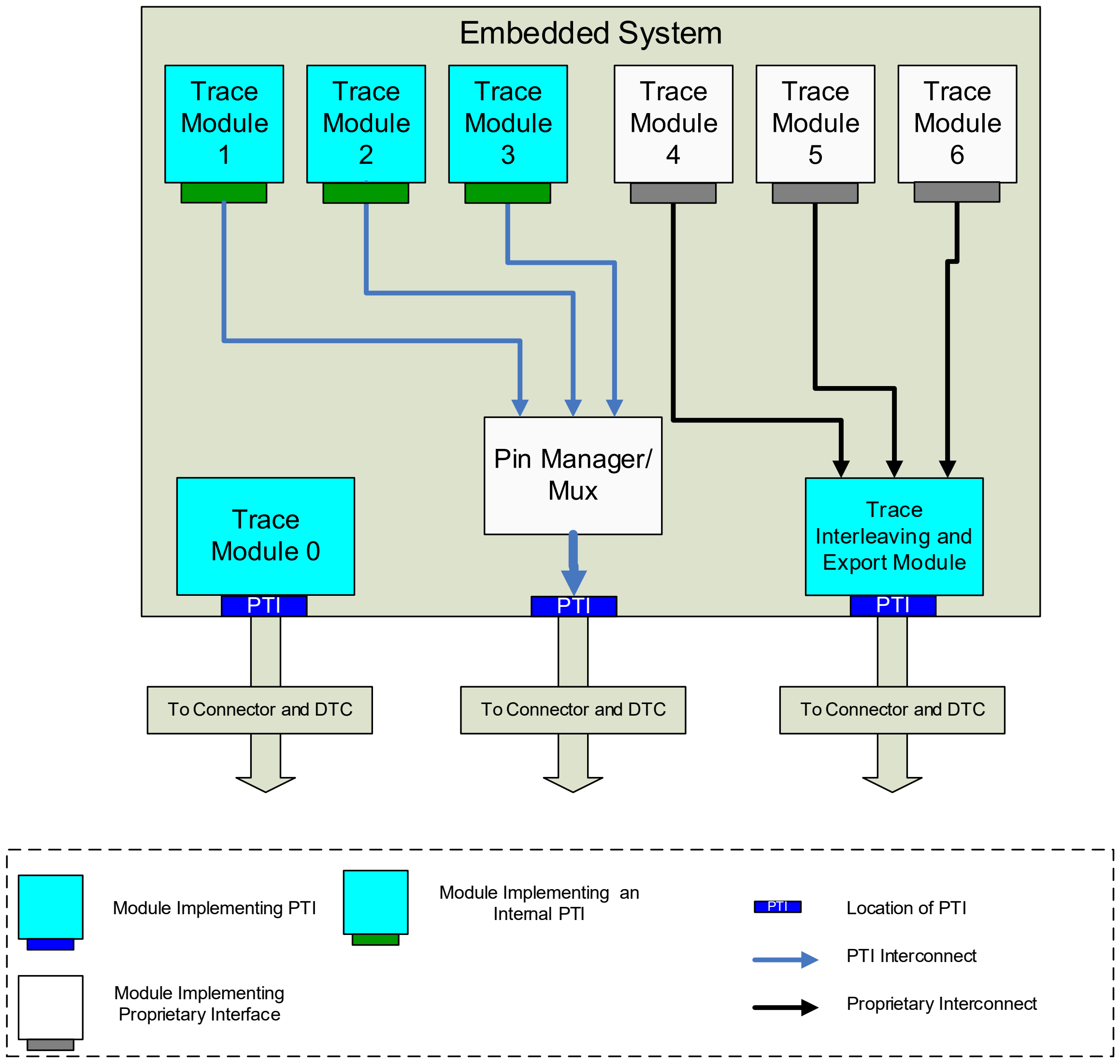


Figure 6 PTI Layers within a System

The scenario for Trace Module 0 is reasonably straightforward. The module itself is directly connected to a dedicated PTI on the device boundary and the module is responsible for implementing the PTI.

The scenario for Trace Modules 1–3 is slightly more complex. Here multiple modules export trace through a device level pin manager or mux. This management logic is only responsible for controlling which pins on the device PTI are assigned to the device internal trace clients. It does not produce the data and clock signals for the PTI but only routes them from the various trace modules. Thus, the individual trace modules are required to implement the PTI. Since the pin manager routes the internal PTI signals to the device boundary, there is also a PTI at the device pins.

The scenario for Trace Modules 4–6 shows a system where multiple trace modules provide data over a proprietary trace interconnect. This system allows data to be combined or interleaved in some fashion before export. The interleaving and export module implements the PTI and the individual trace modules communicate using implementation specific protocols that are beyond the scope of this document.

5.1.3.1 Multi-Point Trace Connections

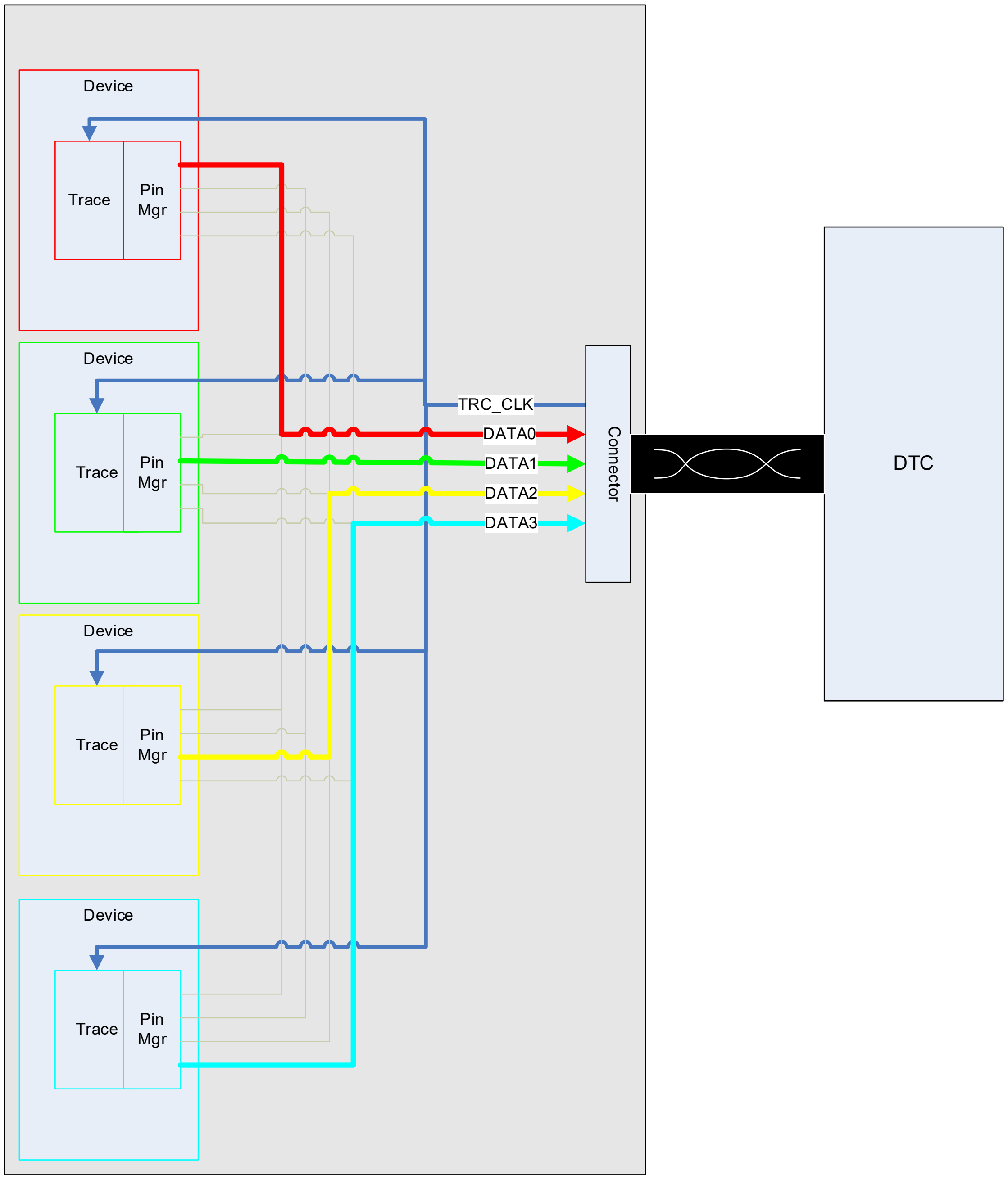
Version 2 of the PTI specification expands the interface description to include a shared trace connection where multiple PTI interfaces are merged through a single connector on a PCB board. Multi-point PTIs are very useful for supporting trace on fielded systems that have multiple trace-enabled ASICs but only a single connector (with limited data pins) for interfacing to an external DTC. A standard example would be a mobile terminal with an application and modem SoC and a single MIPI NIDnT connection.

Devices can be configured to drive data on a subset of the PTI signals on their boundaries. The PTI signals are merged at the connector, but only one PTI is driving any given data signal. The clock for all the interfaces is supplied from an external source (generally the DTC). ***Figure 7*** shows an example with four devices (each with 4-pin PTIs) sharing a connector with each of them only exporting on a single pin.

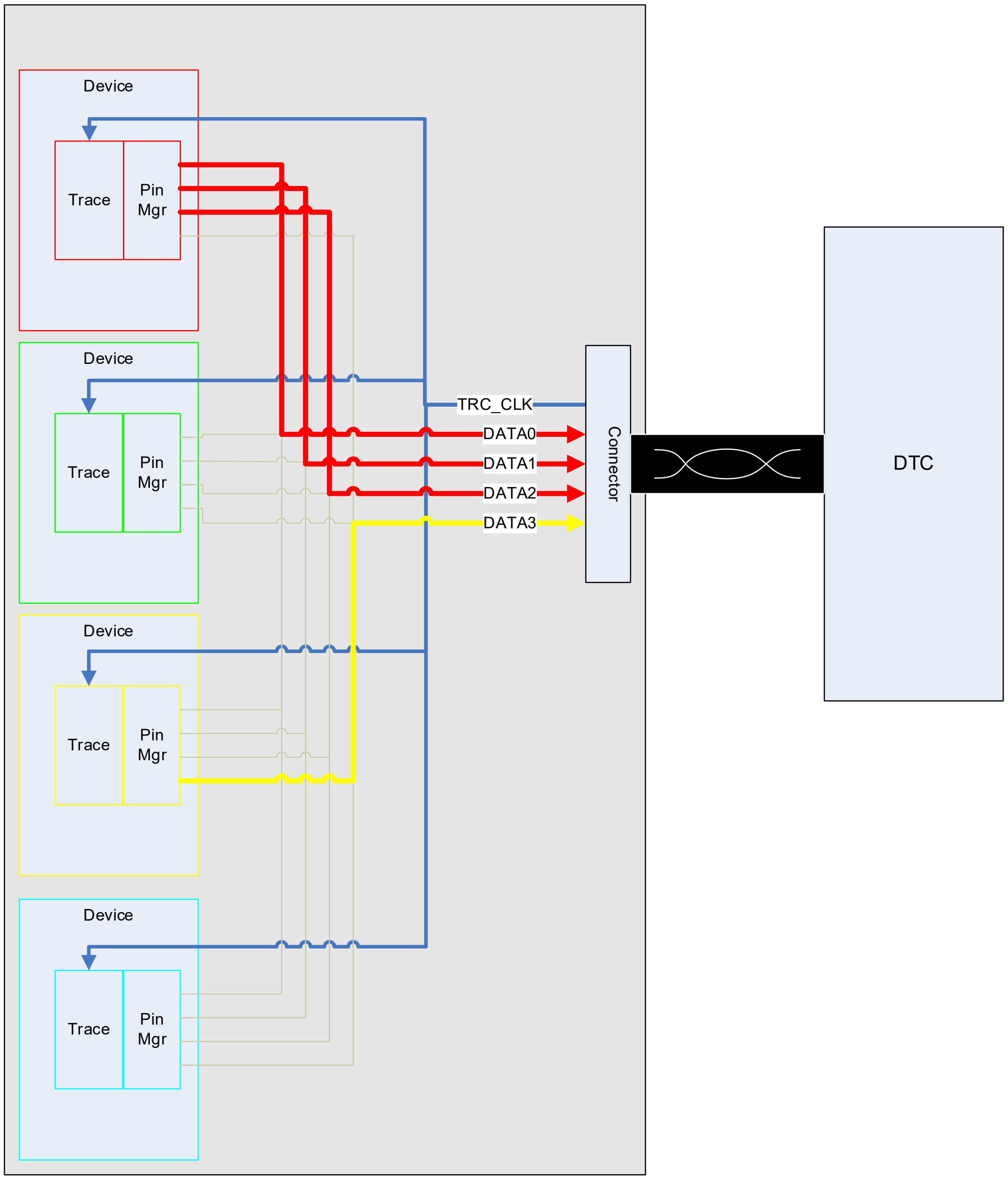
A similar configuration is shown in ***Figure 8***, but in this scenario only two devices are active and the port is shared as 3 pins and 1 pin. These are just examples, and the multi-point routing scheme defined in this document supports varying PTI widths and numbers of devices.

Providing these enhanced features requires new operating modes for the clock and data portions of a PTI.

- Clock Modes
  - PTI-out-clock Mode: The PTI sources the clock along with the data
  - PTI-in-clock Mode: The clock for the PTI is an input to the module driving the PTI data
- Data Modes
  - Point-to-point Data Mode: Data indexes are fixed on the PTI
  - Multi-point Data Mode: Data indexes may shift across the PTI



420 **Figure 7 Multi-Point PTI with 4-Pin Trace and Four Devices Sharing the Connector**



421 **Figure 8 Multi-Point PTI with 4-Pin Trace and Two Devices Sharing the Connector**

5.1.4 Detailed Specification

422 For details of the MIPI PTI, consult the document: MIPI Alliance Specification for Parallel Trace Interface,
423 ***[MIPI02]***. This specification is available to MIPI members and to the public through the MIPI website. The
424 public version of the specification can be found at: http://resources.mipi.org/mipi-pti-download.

5.2 High-speed Trace Interface (HTI) Specification

5.2.1 Overview

Transferring data off-chip from high performance embedded microprocessor cores requires a data port with sufficient trace data bandwidth. Parallel port implementations such as MIPI Parallel Trace Interface (PTI), ***[MIPI02]***, employ a clock synchronous parallel interface, using as many as 32 parallel data lines to provide the required bandwidth. Increasing CPU clock speeds and use of multiple processor cores demand increasing data port bandwidth, while at the same time the number of I/O pins used for the data port is being reduced to facilitate lower cost and a higher level of SOC/ASIC integration.

MIPI High-speed Trace Interface (HTI) is a serial implementation of the data port, taking advantage of available high-speed serial interface technology used in interfaces such as PCI Express®, DisplayPort™, HDMI®, or USB, provides higher transmit bandwidth with fewer I/O pins compared with a parallel implementation. Unlike protocol specifications in the MIPI Gigabit Debug portfolio, such as ***[MIPI08]***, HTI is not designed to be used over the high-level protocols implemented by interfaces such as PCI Express, but is intended to re-use the low-level physical high-speed portions of those interfaces, in a bare-metal environment.

5.2.2 Relationship to the MIPI Debug Architecture

Figure 9 shows the standard MIPI debug architecture highlighting the functional areas addressed by the HTI specification.

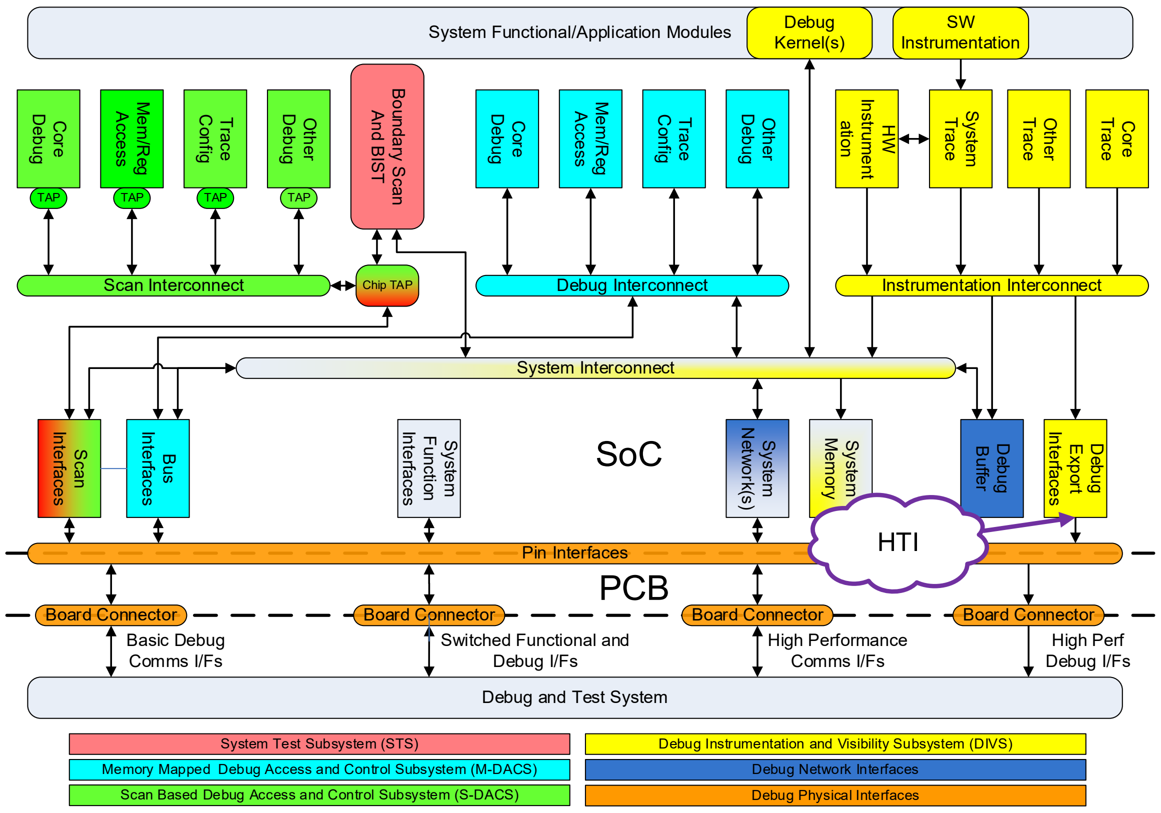


Figure 9 HTI in the MIPI Debug Architecture

5.2.3 HTI Details

HTI defines a method to transport a single stream of trace information over a channel consisting of one to eight high-speed serial lanes, using the Aurora 8B/10B protocol ***[AUR01]***. HTI uses the serial simplex mode of Aurora to transmit data in one direction from TS to DTS.

The HTI specification supports transmission of either the MIPI STP ***[MIPI03]*** protocol or MIPI TWP ***[MIPI04a]*** protocol over an HTI channel.

The HTI specification consists of the following aspects:

- The LINK layer, which defines how the trace is packaged into the Aurora 8B/10B protocol.
- The PHY layer, which defines the electrical and clocking characteristics.
- A programmer's model for controlling HTI and providing status information.

In addition to the trace information, the HTI LINK layer provides the ability to include:

- Optional CRC data, to assist in detecting errors in the trace transmission.
- Optional User Flow Control messages, to indicate additional information about the trace data stream.

5.2.4 Detailed Specification

For details on HTI, consult the MIPI Alliance Specification for High-speed Trace Interface (HTI), ***[MIPI09]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-hti-download.

5.3 Debug Connector Recommendations

5.3.1 Dedicated Debug Connector Overview

Board developers, debug tools vendors and test tool vendors all benefit when the number of connectors and connector pin mappings used to support Debug and Test is minimized. To this end, MIPI Alliance is promoting a set of connectors and mappings that address a wide variety of debug use scenarios.

5.3.2 Relationship to the MIPI Debug Architecture

Figure 10 shows the standard MIPI debug architecture highlighting the functional areas addressed by the connector recommendation.

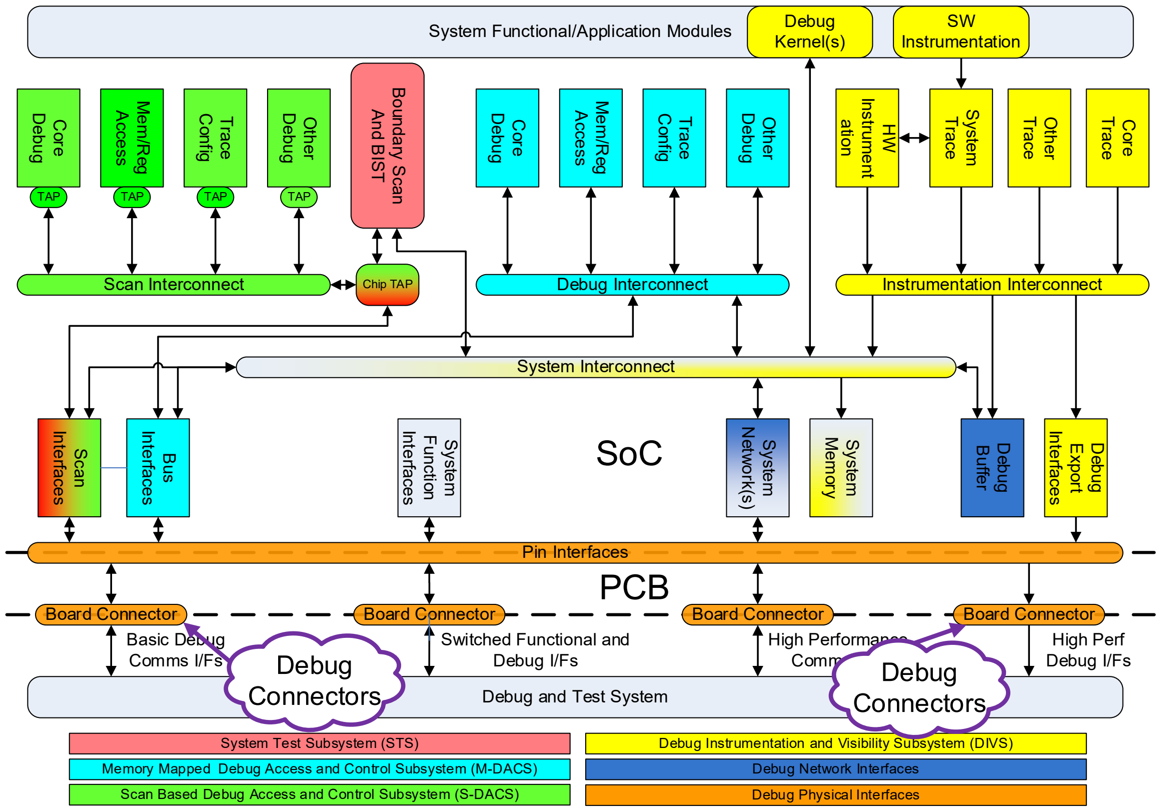


Figure 10 Connectors in the MIPI Debug Architecture

5.3.3 Basic Debug Connectors

As the connector was not part of the original IEEE 1149.1 JTAG standard, a large number of different JTAG connectors have emerged. The MIPI recommendation of standard connectors promotes convergence toward a minimum set of debug connectors. The scalable 0.05 inch Samtec FTSH connector family provides a cheap, small and robust target connection and is available in many variants (including lockable ones) from multiple connector vendors. The pin-out allows scaling of the debug connection to meet different requirements. This includes very small footprint connections (down to 10 pins), legacy JTAG support (including vendor specific pins) and system level trace support (STM).

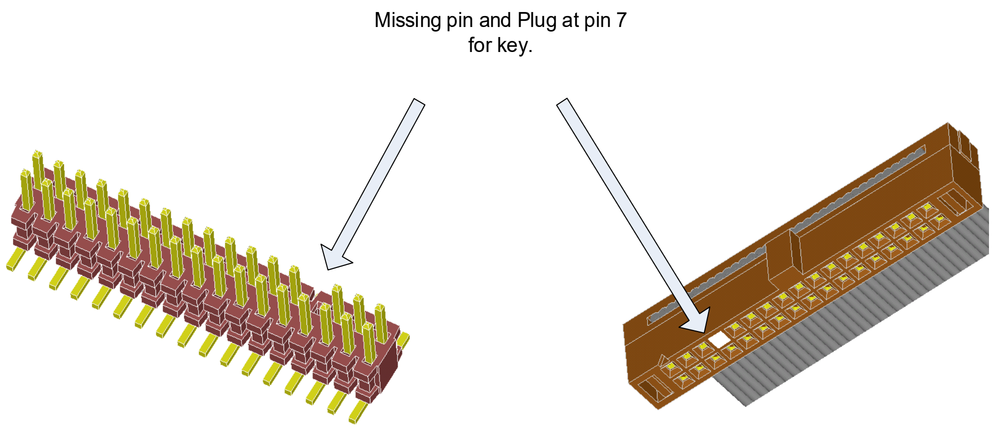


Figure 11 Basic Debug PCB (left) and Cable End Connector (34-pin Samtec FTSH)

5.3.4 High-Speed Parallel Trace Connectors

Many debug tools vendors support target systems with high-speed trace interfaces. These tools utilize a number of different mating connectors.

The *MIPI Alliance Recommendation for Debug and Trace Connectors*, ***[MIPI01]***, document defines two connectors for supporting high-speed trace and basic debug. The first connector is only intended for backwards-compatible designs. The second connector is recommended for new designs. The goal is to have this recommendation define a "de facto" industry standard for the trace connection and thus lessen the burden on target system and tools developers that need to support a large number of different mating connections.

The recommended trace connector is a 60 pin Samtec QSH/QTH connector. The signal to pin mapping, which is defined in the recommendation, supports one run control and several trace configurations. The different trace configurations use up to 40 data signals and up to 4 clock signals. To minimize complexity, the recommendation defines four standard configurations with one, two, three or four trace channels of varying width.

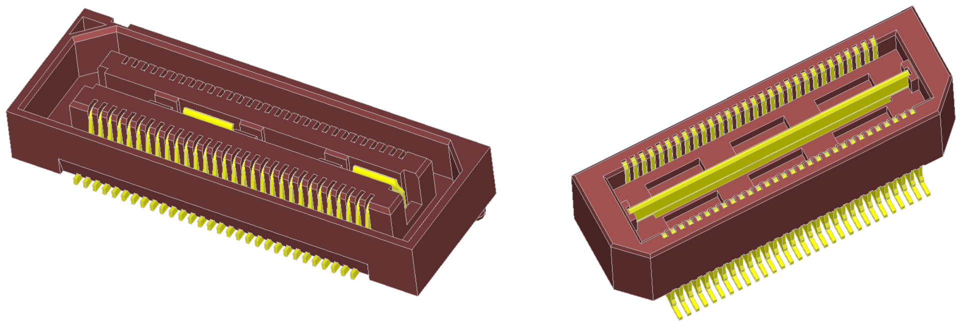


Figure 12 Recommended Samtec QSH/QTH Connector

5.3.5 Detailed Documentation

For details of the MIPI recommended connectors and connector pin mappings, consult the document: MIPI Alliance Recommendation for Debug and Trace Connectors, ***[MIPI01]***. This document is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: https://mipi.org/sites/default/files/MIPI-Alliance-Recommendation-Debug-Trace-Connectors.pdf.

5.4 Narrow Interface for Debug and Test (NIDnT) Specification

5.4.1 Overview

The MIPI Debug Working Group has standardized a way to utilize functional interfaces for debug or test. This technology is called NIDnT (Narrow Interface for Debug and Test). It allows better debug support in production or near-production units.

NIDnT technology defines low pin count, reliable, and high performance, debug interfaces that can be used in deployed systems. These interfaces provide access to basic debug, trace of application activity, and HW test capability by reusing already existing functional interfaces. In some cases, these interfaces are accessible at the packaged boundary. This technology provides the means to use functional interfaces for either functional or debug purposes. One or more functional interfaces (e.g., MMC card slot for trace and USB for basic debug) may be used to provide debug capability. NIDnT technology does not aim to replace current technologies such as debugging via a serial interface (e.g., GDB using a UART, or on-device debug agent).

5.4.2 Relationship to the MIPI Debug Architecture

Figure 13 shows the standard MIPI debug architecture highlighting the functional areas addressed by the NIDnT specification.

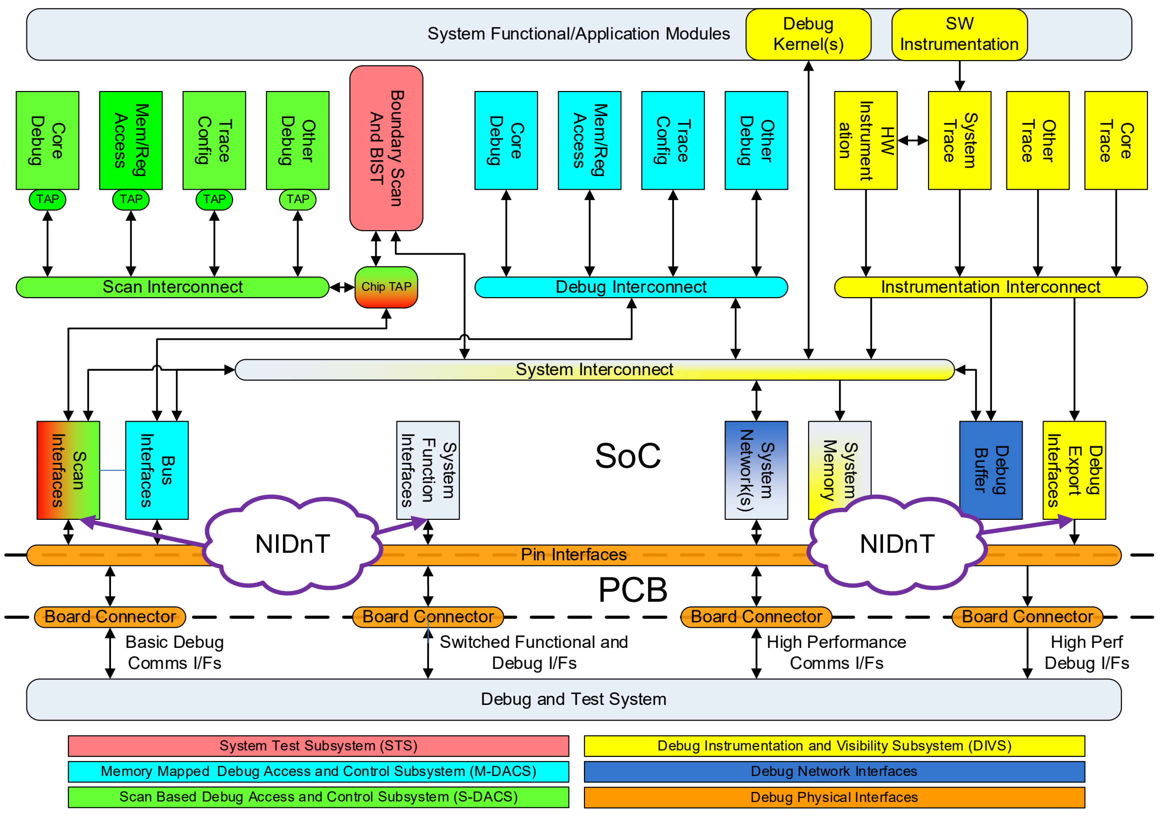


Figure 13 NIDnT in the MIPI Debug Architecture

5.4.3 NIDnT Details

NIDnT technology has the potential for changing the product development paradigm as it provides for the use of one or more of a product's functional interfaces for debug. This can extend the availability of the debug capabilities used in the early stages of product development to the latter stages. This is especially valuable when these interfaces are available at the boundary of the product's actual physical enclosure in the product's final form factor. This change in the product development paradigm is described in the following paragraphs.

During the early stages of product development, IEEE 1149.1/1149.7/SWD/I3C based basic debug, trace of application activity, and software messages sent over simple streaming interfaces like serial ports are typically used for debug. Historically, much of this product development is performed using test or development boards. These boards provide dedicated and readily accessible Debug and Test interfaces for connecting the tools. A system with a dedicated debug interface is shown in ***Figure 14***.

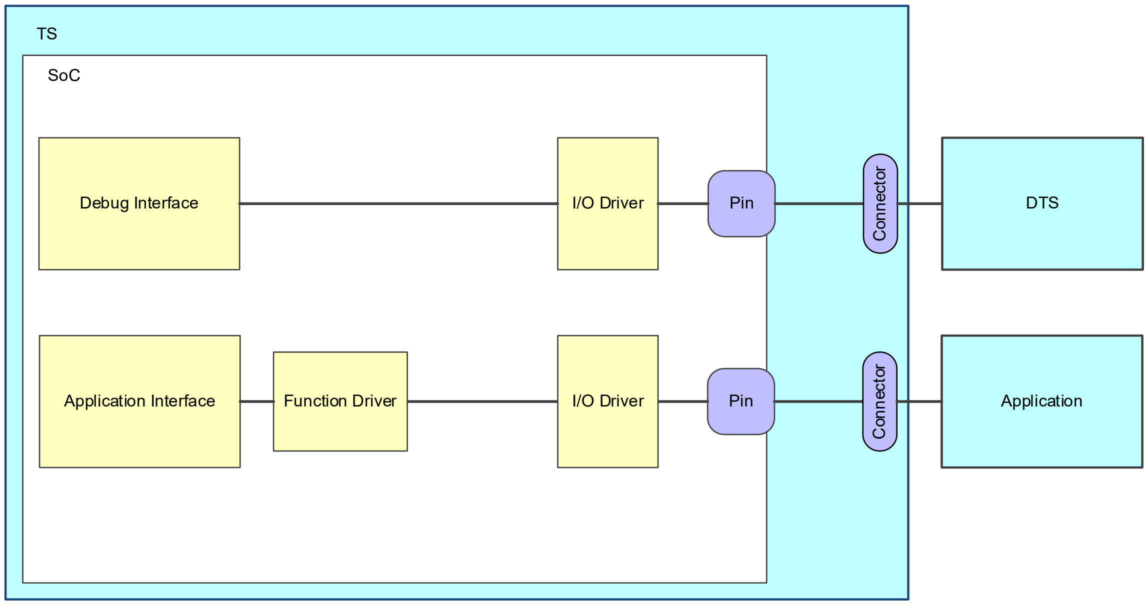


Figure 14 Example of System with a Dedicated Debug Interface

In most cases, a product's final form factor does not have dedicated Debug and Test interfaces as these interfaces are not propagated to the boundary of the product's physical enclosure. This hampers the identification of bugs present at this point in the product development.

A product might include a proprietary JTAG connector that requires some disassembly (e.g., removing the battery cover and battery) and the use of a test fixture. The physically invasive process of accessing this connector could itself cause bugs or RF performance issues to disappear, or new ones to appear.

Figure 15 shows how NIDnT technology extends the use of functional interfaces for Debug and Test purposes. It creates a dual use functional interface by multiplexing the debug signals with the normal function signals within the SoC in a manner that is similar to a switch. Connecting either the normal function or the debug function to the interface connects that function's inputs and outputs to the interface. Disconnecting either the normal function or debug function from the interface connects its inputs to inactive default values that create the function's inert operation while leaving its outputs unused. For example, a SoC could multiplex an IEEE 1149.7 Test Access Port (TAP) and a Parallel Trace Interface (PTI) over the functional I/Os that normally provide a microSD interface. In this case, the IEEE 1149.7 TAP could be used for both basic debugging and as a control channel for the trace function that utilizes the PTI interface.

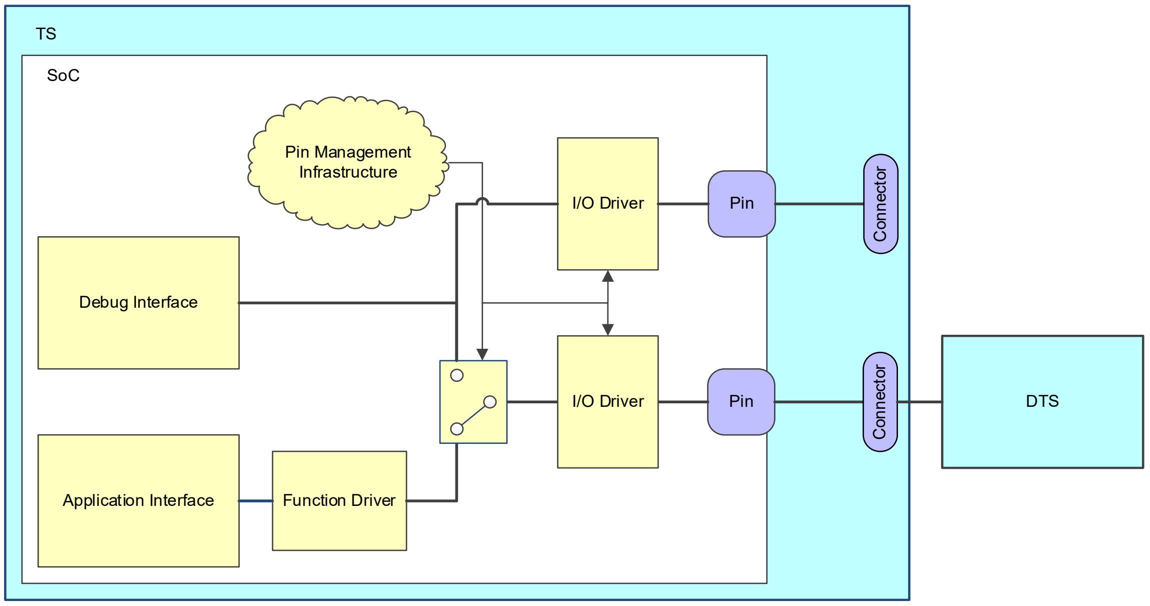


Figure 15 Example of System with NIDnT Capability

It is expected that adapters will be used to connect a product's NIDnT Interface (e.g., microSD interface, or USB) to the MIPI Debug Connectors (as defined in ***[MIPI01]***). The use of an adapter provides for debugging the product in its final form factor with standard debug tools, as the adapter remaps the signals presented by the tools on these standard debug connectors to the appropriate positions on the functional connectors.

5.4.4 Debug and Test Capabilities Supported by NIDnT Overlay Modes

A NIDnT Interface supports an operating mode that provides all functional operation of the interface (Overlay Mode 0, also called the Original Functional Mode (OFM)) and one or more non-OFM Overlay Modes (Overlay Modes 1 through n) providing debug and test capability.

The debug and test capabilities that can be supported with these Overlay Modes are listed below with their associated pin counts shown in parenthesis. These capabilities might be mixed and matched to provide one or more combinations of debug and test capability within the limitations (pin count and drive characteristics) of a specific functional interface or combination of interfaces. The combinations supported for a specific NIDnT Interface are outlined in interface-specific sections of the NIDnT specification.

Table 1 Summary of Test/Debug Capabilities Supported by NIDnT

Capability	Interface with Single-Ended Electricals	Interface with Differential Electricals
Basic Debug	**<u>2-pin (Min-Pin) Debug</u>** • IEEE 1149.7 ***[IEEE02]*** • Serial Wire Debug ***[ARM01]*** • UART • I3C • Vendor Defined Single-Ended Debug	**<u>4-pin High-Speed Debug</u>** • Vendor Defined Differential Debug
	<u>5-pin Legacy Debug</u> • IEEE 1149.1 ***[IEEE01]***	
	<u>6-pin Modified Legacy Debug</u> • Modified IEEE 1149.1 Standard with return clock (deprecated)	
Trace	**<u>Single-Ended Trace</u>** • Parallel Trace Interface ***[MIPI02]*** • Vendor Defined Single-Ended Trace	**<u>High-Speed Trace</u>** • High-Speed Trace Interface (HTI) ***[MIPI09]*** • Vendor Defined Differential Trace
User Defined	Vendor Defined Single-Ended	Vendor Defined Differential

The trace function can either run with a clock shared with the 2-pin Min-Pin debug interface or run with an independent clock. If the focus is on maximum trace bandwidth, a shared clock provides the largest number of trace data pins but ties the data rate of each data pin to the clock rate of the 2-pin Min-Pin debug interface.

Non-OFM Overlay Modes that support debug, i.e., that switch some of the NIDnT Interface pins to being used for Basic Debug signals, are called Debug Overlay Modes (see table in the NIDnT Specification, ***[MIPI05]***).

5.4.5 Functional Interfaces that are NIDnT Candidates

The current version of the NIDnT Specification addresses the reuse of the following interfaces:

- microSD
- USB (USB 2.0 and USB Type-C^{TM})
- Display (HDMI and DisplayPort (DP))

Future versions of the NIDnT Specification might support other interfaces including, but not limited to:

- SIM (smart card)
- UniPro

5.4.6 Detailed Specification

For details of NIDnT technology, consult: MIPI Alliance Specification for Narrow Interface for Debug and Test (NIDnT), ***[MIPI05]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-nidnt-download.

6 Debug Access and Control Subsystem (DACS)

6.1 IEEE 1149.7 Debug and Test Interface Specification

The IEEE 1149.7 standard ***[IEEE02]*** supports the needs of both Debug and Test. It is a superset of the IEEE 1149.1 standard ***[IEEE01]*** and represents a natural evolution of this standard. This approach preserves the industry's hardware and software investments in the IEEE 1149.1 standard since its inception. While this is not a MIPI specification, the min-pin debug effort started in MIPI, so it is included here to help complete the debug framework. The standard:

- Provides a substantial, yet scalable set of additional debug related capability
- Supports multiple connection topologies
  - Four-wire series or star
  - Two-wire star
- Halves the width of the interface in two-wire star configurations while maintaining performance

Six capability classes (T0-T5) are supported, with the implementer selecting the capability class implemented. A class defines both mandatory and optional capability. Class capability increases progressively, with the capability of a class including the capability of all lower numbered classes.

Capability classes T0-T2 support operation with the four-wire Test Access Port (TAP) (defined by the IEEE 1149.1 standard) connected in a four-wire series topology. Each of these classes incrementally extends the IEEE 1149.1 capability while using only the Standard Protocol defined by the IEEE 1149.1 standard.

Capability classes T3 additionally supports deployment in a four-wire star topology.

Capability classes T4-T5 provide for implementing devices with either a four-wire TAP (IEEE 1149.1 style) or a two-wire TAP (unique IEEE 1149.7 style). Devices with the four-wire TAP configuration can be operated in all connection topologies. Devices with the two-wire TAP configuration can be operated only in a two-wire scan topology.

The T4-T5 classes incorporate the Advanced Protocol. The Advanced Protocol provides for the joint use of the TAP for real-time system instrumentation, classic debug, and test, using only the TCKC and TMSC signals as it:

- Redefines the functionality of the IEEE 1149.1 TCKC and TMSC signals
- Eliminates the need for the TDIC and TDOC signals
- Allows the use of the TAP for both scan and non-scan data transfers

The combination of a two-wire TAP and use of the Advanced Protocol provides the capability of a five-wire IEEE 1149.1 TAP using only two signals, plus additional system debug capability.

A high-level view of the IEEE 1149.7 interface connectivity between a DTS and TAPs within the TS is shown in ***Figure 16***. Both the four-wire (wide) and two-wire (narrow) TAP configurations are shown with an optional test reset signal. A deprecated non-standard return clock signal is also comprehended with the four-wire configuration (the use of this and other non-standard signals is strongly discouraged by the standard).

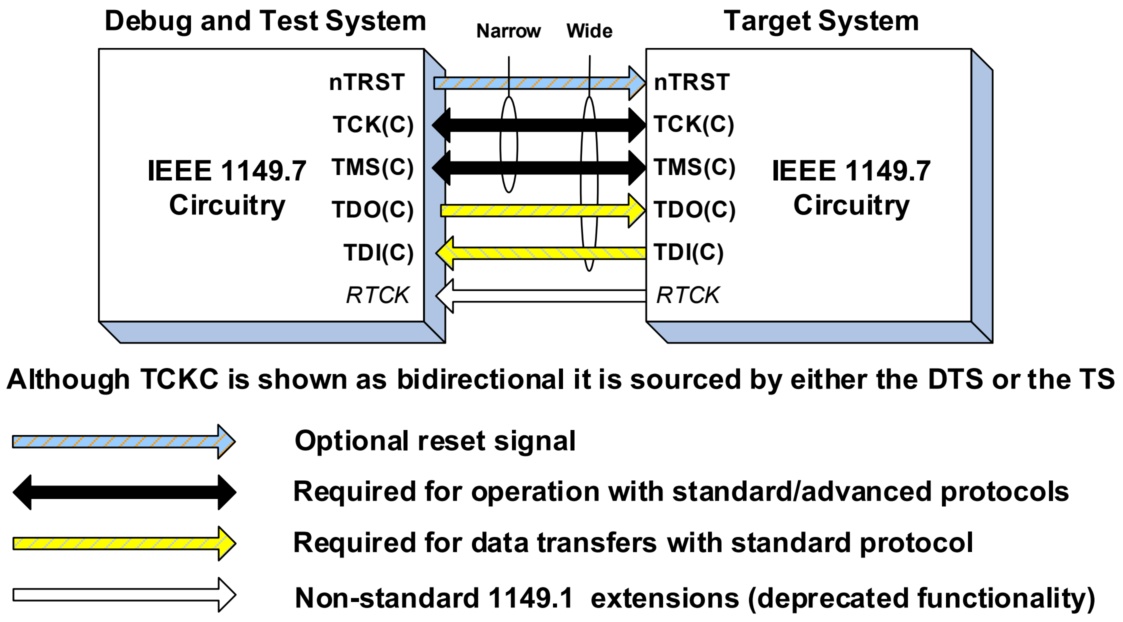

Debug and Test System
Narrow
Wide
Target System
IEEE 1149.7 Circuitry
nTRST
TCK(C)
TMS(C)
TDO(C)
TDI(C)
RTCK
nTRST
TCK(C)
TMS(C)
TDO(C)
TDI(C)
RTCK
IEEE 1149.7 Circuitry
Although TCKC is shown as bidirectional it is sourced by either the DTS or the TS
Optional reset signal
Required for operation with standard/advanced protocols
Required for data transfers with standard protocol
Non-standard 1149.1 extensions (deprecated functionality)

Figure 16 DTS to TS Connectivity

All capability classes begin operation using the Standard Protocol. IEEE 1149.7 operation is compatible with IEEE 1149.1 from power-up, with the function of TCK(C) and TMS(C) signals providing the functionality (or a superset thereof) of the TCK and TMS signals that is specified by the IEEE 1149.1 standard.

All IEEE 1149.7 based devices may be implemented in a manner that allows their use in system configurations where there is:

- A mix of components implementing different capability classes
- A mix of connection topologies

The DTS can use facilities defined by the standard to determine the following:

- The types of connection topologies deployed within the TS
- The component mix with the TS:
    - 1149.1 components
    - 1149.7 components + class of each component

6.1.1 Relationship to MIPI Debug Architecture

Figure 17 shows the standard MIPI debug architecture highlighting the functional areas addressed by the IEEE 1149.7 standard.

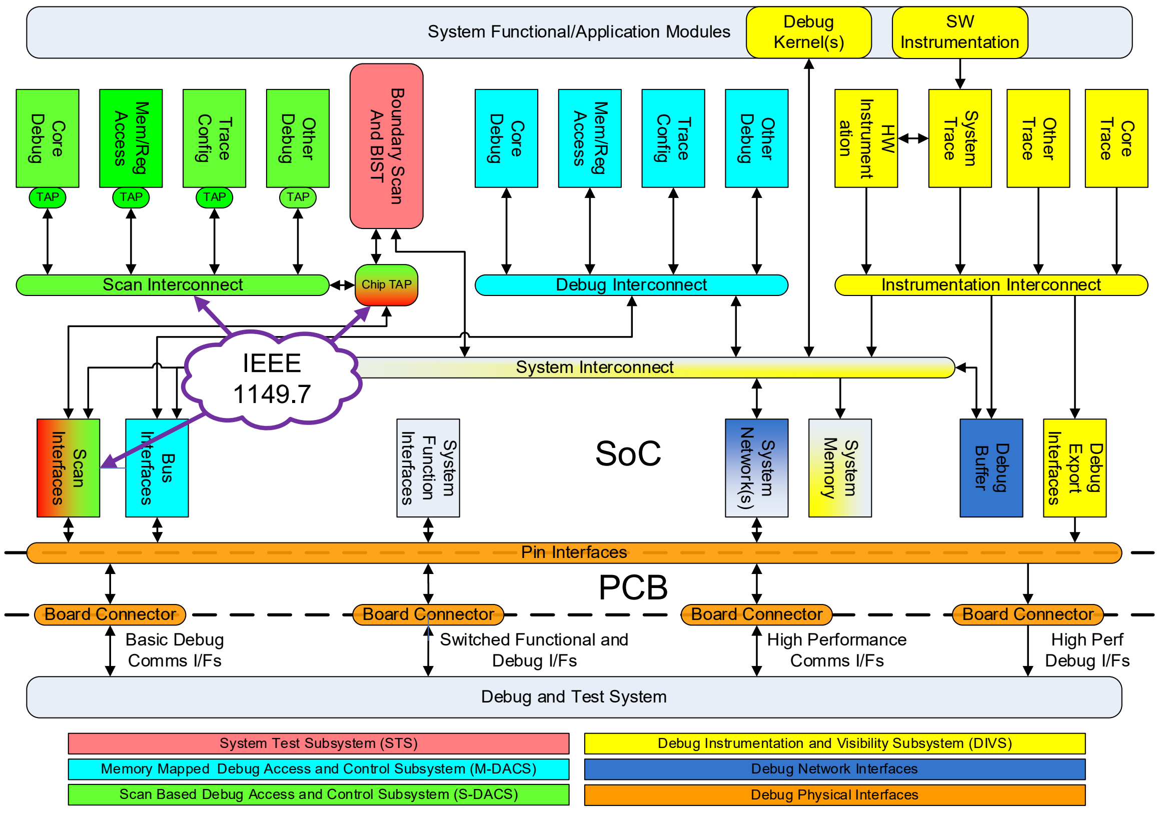


Figure 17 IEEE 1149.7 in the MIPI Debug Architecture

6.1.2 Detailed Specification

For details of the 1149.7 specification, consult the document: IEEE 1149.7 Standard for Reduced-pin and Enhanced-functionality Test Access Port and Boundary Scan Architecture ***[IEEE02]***.

6.2 SneakPeek Specification

The SneakPeek framework is intended to enable debugging of a Target System via standard network connection. This is accomplished by moving a portion of the Debug and Test Controller function onto the SoC. These embedded DTC functions can be reached by network communication links that previously have not been leveraged for *DTC-like* debug. SneakPeek also leverages a significant portion of the on-chip debug infrastructure. As a result, DTC tools that previously used dedicated debug links (e.g., 1149.7 or PTI) can easily be ported to work in a SneakPeek framework through simple network adaptor layers. The identical capabilities realized via the dedicated debug interfaces should be available via SneakPeek (with possible performance penalties).

6.2.1 Relationship to MIPI Debug Architecture

Figure 18 shows the standard MIPI debug architecture highlighting the functional areas addressed by the SneakPeek specification.

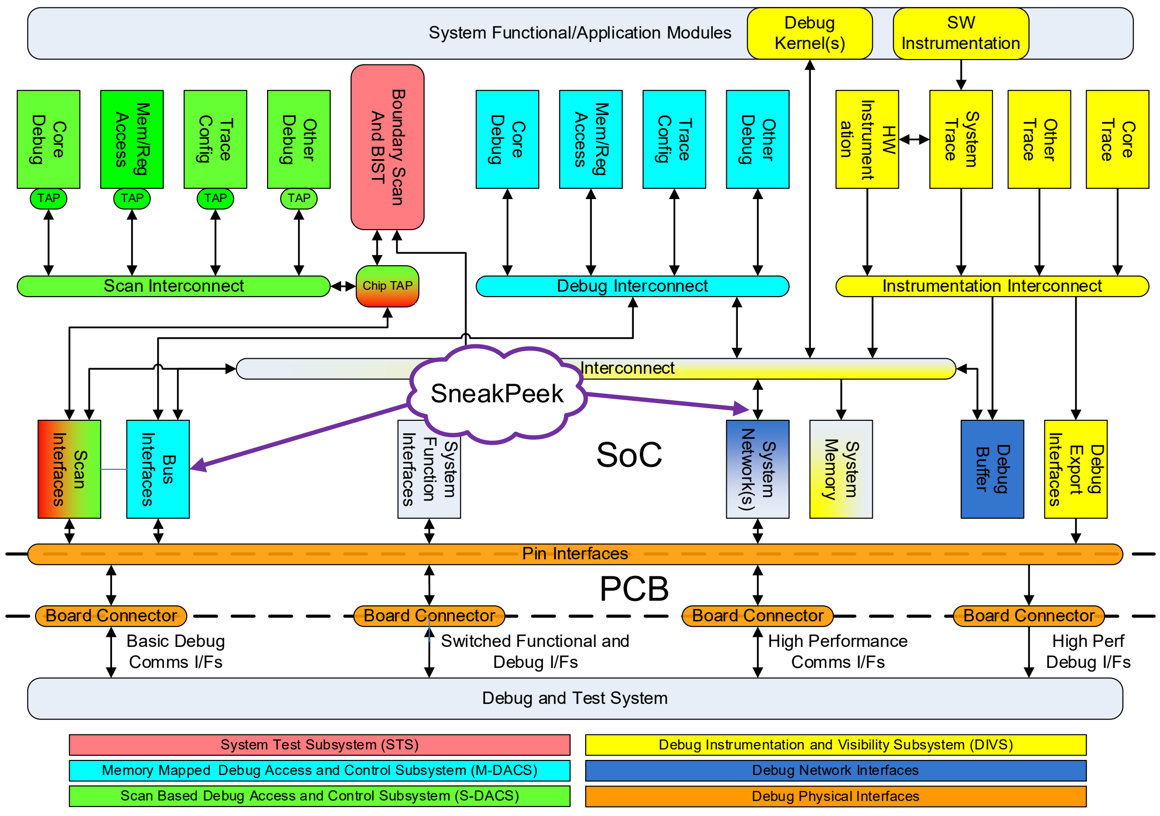


Figure 18 SneakPeek in the MIPI Debug Architecture

6.2.2 Overview

The SneakPeek Protocol (SPP) is used to communicate between a Debug Test System (DTS) and a Target System (TS). This communication facilitates using Debug Applications (typically software) within the DTS to debug the operation of the TS.

The SneakPeek Protocol abstracts the system designer from dedicated debug communication interfaces such as JTAG and replaces them with the familiar mechanism of address-mapped read and write transactions to enable the Debug Applications to observe, interrogate and adjust the Target System. These transactions might be addressed to main system memory, special function memories, or address-mapped peripherals within the TS.

If the system requires legacy dedicated debug communication interfaces to be used internally within part of a system, then these could be constructed by a dedicated address-mapped peripheral within the Target System that is then accessed by the DTS via SneakPeek.

Figure 19 illustrates the route by which one or more debug software applications in a DTS utilize SneakPeek Memory Agents within a TS to perform address-mapped transactions for them.

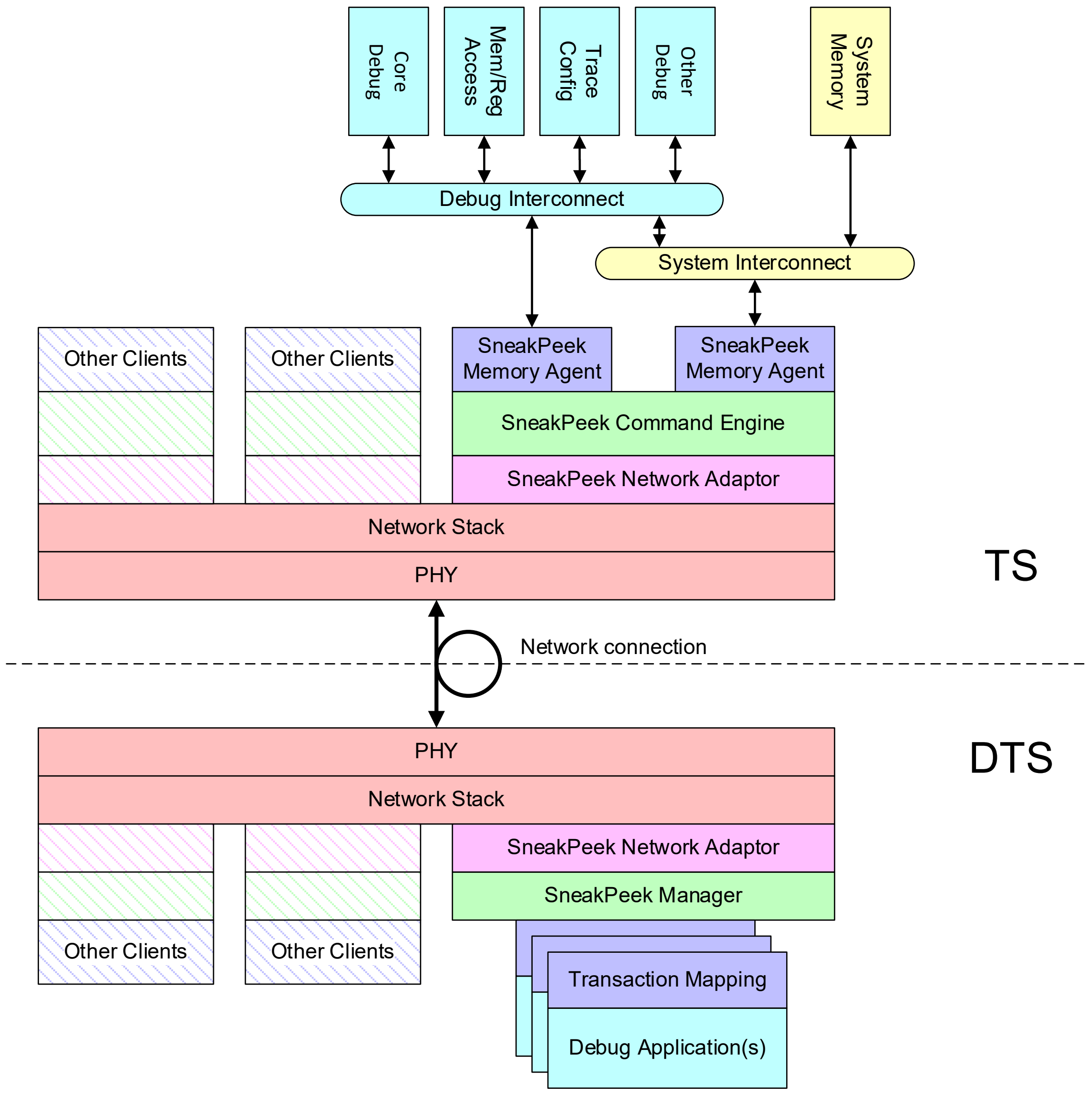


Figure 19 Overview of SneakPeek System

The basic communication units used by SneakPeek are SneakPeek Command Packets sent from the DTS to the TS, and SneakPeek Response Packets sent from the TS to the DTS. To provide more efficient interactions with the communication network, the DTS packs typically many Command Packets into a single SneakPeek Transfer Block (SPTB) before handing this over to the network driver for transmission to the TS. Similarly, the TS packs typically many Response Packets into a single SPTB for transmission to the DTS.

Figure 20 shows how the SneakPeek Protocol is built on top of existing network infrastructure.

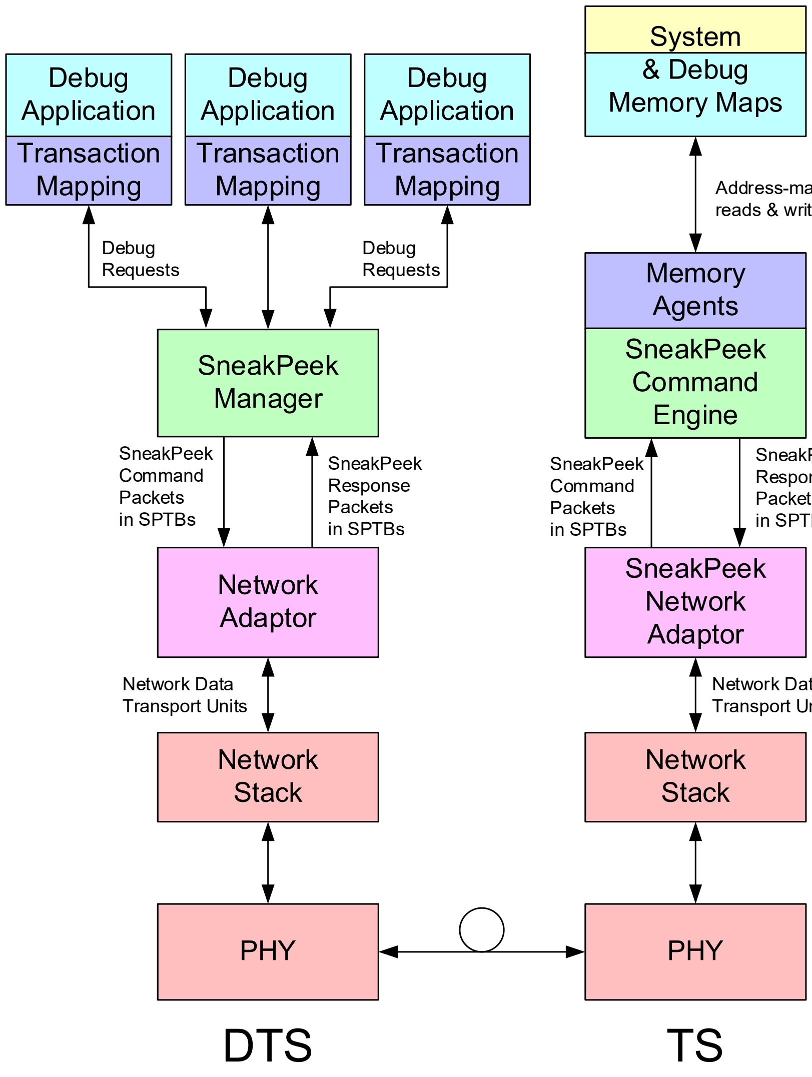

Debug Application
Transaction Mapping
Debug Application
Transaction Mapping
Debug Application
Transaction Mapping
System & Debug Memory Maps
Address-mapped reads & writes
Debug Requests
Debug Requests
Memory Agents
SneakPeek Manager
SneakPeek Command Engine
SneakPeek Command Packets in SPTBs
SneakPeek Response Packets in SPTBs
SneakPeek Command Packets in SPTBs
SneakPeek Response Packets in SPTBs
Network Adaptor
SneakPeek Network Adaptor
Network Data Transport Units
Network Data Transport Units
Network Stack
Network Stack
PHY
PHY
DTS
TS

Figure 20 SneakPeek Protocol and Network Stacks in DTS and TS

In summary:

- The DTS sends SneakPeek Command Packets grouped into SPTBs to the TS over a data communication network.
- These Command Packets cause an action or effect in the TS, typically an address-mapped read or write transaction. The Command Engine generates a Response Packet corresponding to each Command Packet (with some special case exceptions).
- The TS sends SneakPeek Response Packets grouped into SPTBs to the DTS over the data communication network.
- The SneakPeek Packets in a stream have a defined order at their source and are interpreted in this order at their destination. The SneakPeek Protocol is not concerned with actual transmission order

over the physical or other layers of the network stack but assumes that the network reconstructs the original order before handing off the SneakPeek Packets at their destination.

6.2.3 Protocol Styles

SPP version 2.0 introduces TinySPP, an "optimized" style of the SneakPeek Protocol focusing on low bandwidth interfaces and "tiny" implementations. TinySPP provides a reduced feature-set (e.g., no sequence number, reduced access space, reduced direct addressing space, and no access size variability) and "coexists" with SPP version 1.0, or FullSPP. Reducing the size of the Command and Response Packets is done by assuming certain behaviors and by placing some restrictions on these interfaces. These restrictions and assumptions are usually acceptable as a tradeoff for a smaller and simpler implementation more tailored for lower bandwidth and/or half-duplex interfaces.

The main differences between the TinySPP and FullSPP styles are shown in ***Table 2***.

Table 2 Comparison of SneakPeek Protocol Styles

Feature	**TinySPP**	**FullSPP**
Number of Access Spaces	8 (3-bit field)	32 (5-bit field)
Sequence Number	N/A. Additional requirements to network: • Messages stay in order • Guaranteed Delivery • DTS/TS has to do book-keeping for request/ response	Present
Standard Size Field	Chosen by TS	Different options
Size of Transaction Byte Count (TBC) Field	7 bits	16 bits
Short Addressing	6-bit address replacement supported. Short addressing is required in a TinySPP implementation	N/A
Packet Alignment	Byte aligned	16-Byte aligned
Shortest Packet Length	4 Bytes	16 Bytes
Usage in MIPI Specifications	Debug for I3C	GbD for USB, GbD for IPS

6.2.4 Detailed Specifications

For details of the SneakPeek Protocol, consult the document: MIPI Alliance Specification for SneakPeek Protocol, ***[MIPI06]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-spp-v2-download.

This page intentionally left blank.

7 Debug Instrumentation and Visibility Subsystem (DIVS)

7.1 Instrumentation and Visibility Subsystem Overview

The DIVS is basically a network or interconnect that allows trace data to flow from various sources to the trace data sink (generally the DTS). The DIVS architecture provides a rich set of features that can be utilized to effect this purpose:

- Trace protocols such as the System Trace Protocol (STP) that provide a standard encoding for trace from multiple different HW and SW sources.
- Trace merge protocols such as the Trace Wrapper Protocol (TWP) that can be used to combine many different trace streams into a single stream of data for easy transport management.
- Trace network protocols like the Gigabit Trace (GbT) and network adaptor specifications that define how trace data should be formatted for transport over standard network links.

7.2 System Trace Protocol (STP) Specification

Real-time trace has become an indispensable tool for debugging and optimizing embedded systems. This trace can come from a variety of sources, including:

- Trace components monitoring processor instruction and data flow.
- Instrumentation in the software running on a processor.
- Trace components monitoring activities outside the processor.

Each trace source has its own protocol, and these protocols share a number of common required features. The System Trace Protocol (STP) is a base protocol which provides these common features.

The advantages of this shared approach are:

- Reuse reduces the time and cost of designing new protocols, as well as IP and tools supporting them.
- Commonality of features enables greater interoperability, for example by providing time correlation between multiple trace streams.
- A robust base protocol ensures common protocol design mistakes are avoided.

The STP specifications were developed to leverage the advantages listed above. STP was not intended to supplant or replace the highly optimized protocols used to convey data about processor program flow, timing or low-level bus transactions. It is anticipated that STP data streams will exist side by side with these optimized protocols as part of a complete debug system.

7.2.1 Relationship to MIPI Debug Architecture

Figure 21 shows the standard MIPI debug architecture highlighting the functional areas addressed by the STP specifications.

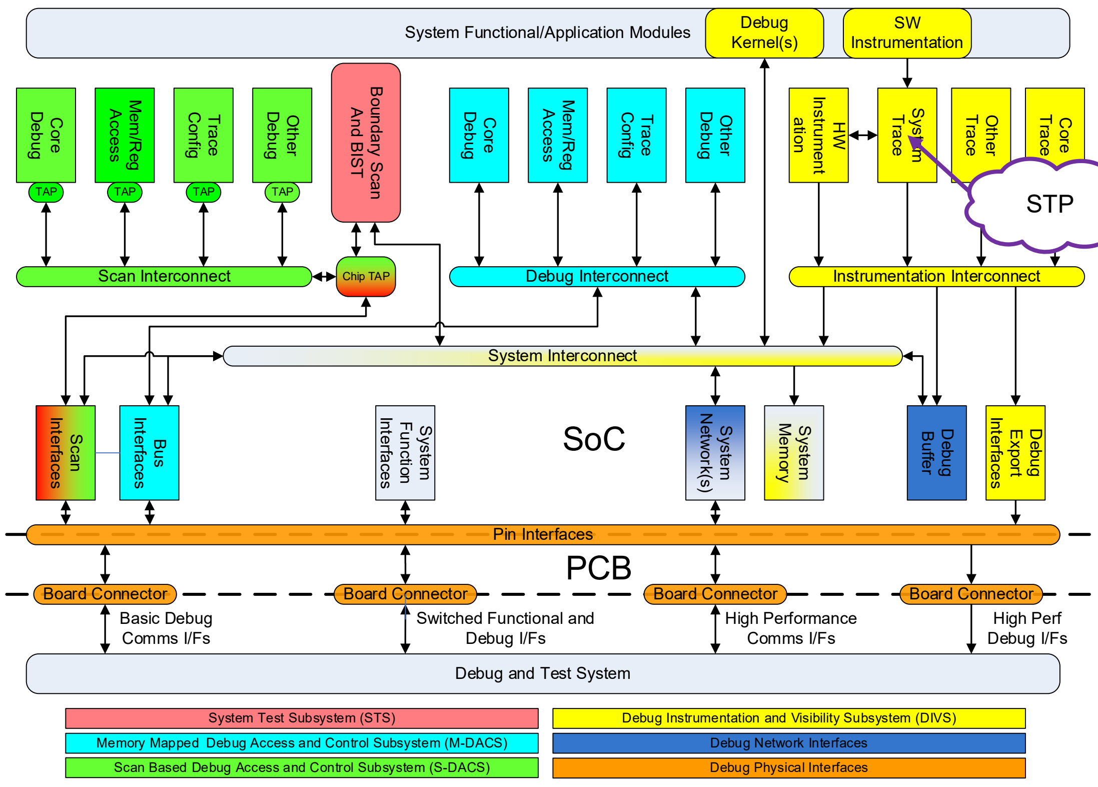


Figure 21 STP in the MIPI Debug Architecture

7.2.2 Protocol Overview

STP was developed as a generic base protocol that can be shared by multiple, application-specific trace protocols. STP was not intended to supplant or replace the highly optimized protocols used to convey data about processor program flow, timing or low-level bus transactions. STP is designed so that its data streams coexist with these optimized protocols as part of a complete debug system. The STP protocol is now in its second generation (STPv2) which is backward compatible with the first generation.

STPv2 includes the following features:

- A trace stream comprised of 4-bit frames (nibbles)
- Support for merging trace data from up to 65536 independent data sources
- Up to 65536 independent data Channels per Major Source
- Basic trace data messages that can convey 4, 8, 16, 32, or 64-bit wide data
- Time-stamped data packets using one of several time stamp formats including:
  - Gray code
  - Natural binary
  - Natural binary delta
  - Export buffer depth (legacy STPv1 timestamp that requires DTC support)
- Data packet markers to indicate packet usage by higher-level protocols
- Flag packets for marking points of interest (for higher-level protocols) in the stream

- Packets for aligning time stamps from different clock domains
- Packets for indicating to the DTC the position of a trigger event, which is typically used to control actions in the DTC; for example, to control trace capture
- Packets for cross-synchronization events across multiple STP sources
- Support for user-defined data packets
- Facilities for synchronizing the trace stream on bit and message boundaries
- Optional support for data integrity protection of the trace stream
- Add data integrity package (DIP) to facilitate error detection over noisy connections
- Platform Description ID (PDID) packet types to carry payload information identifying the platform the trace was captured and describe the contained trace data formats to enable processing tools to auto-detect the format of the individual traces received from the trace stream itself.

Figure 22 shows the conceptual hierarchy of the different terms described in this specification. The clouds are elements from the data model.

A stream of STP packets generally contains data from a number of different Major Sources, which in turn may each have a number of different Channels. These two levels of hierarchy may be used, for example, to distinguish different software applications (Channels) running on different processors (Major Sources).

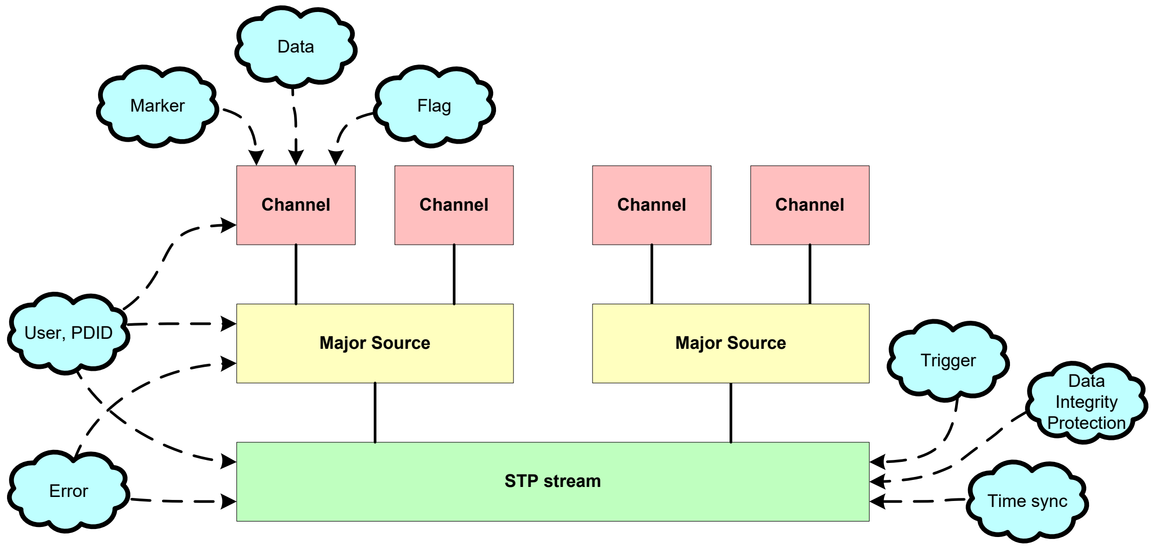


Figure 22 Conceptual Hierarchy of STP Major Sources and Channels

Figure 23 shows an example of a target system that utilizes a module implementing the System Trace Protocol. In this example, the STP data is transferred to the DTC across a PTI.

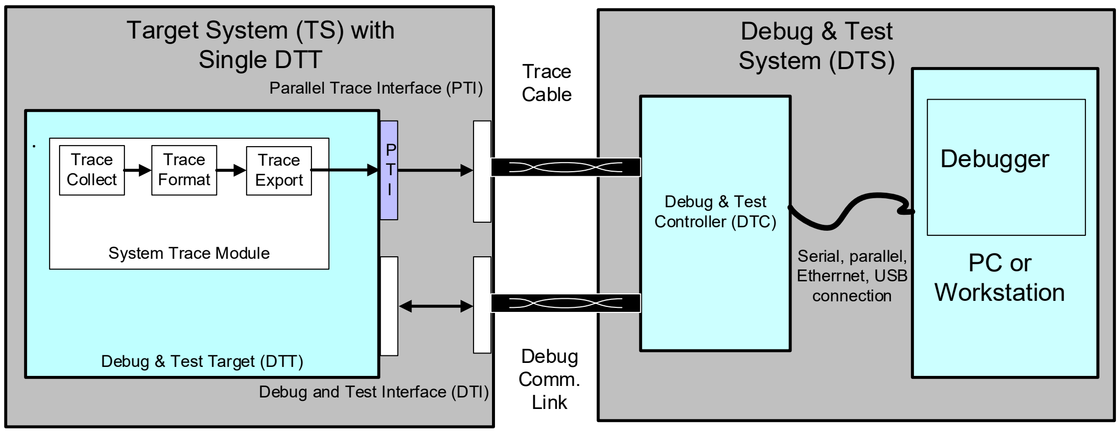

Target System (TS) with Single DTT
Parallel Trace Interface (PTI)
Trace Collect
Trace Format
Trace Export
PTI
System Trace Module
Debug & Test Target (DTT)
Debug and Test Interface (DTI)
Trace Cable
Debug Comm. Link
Debug & Test System (DTS)
Debug & Test Controller (DTC)
Serial, parallel, Ethernet, USB connection
Debugger
PC or Workstation

Figure 23 STM in a Target System

The timing diagram in ***Figure 24*** shows an example of the STP packets that might be transferred to the DTC in such a system. This example shows the end of a synchronization sequence followed by a series of 16-bit data packets on Channel 4 of Trace Source (Major Source) 3.

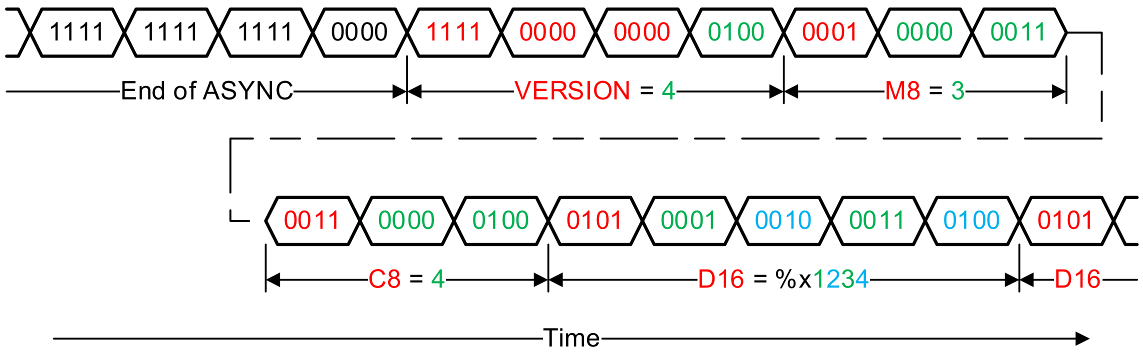


Figure 24 Example STP Packet Sequence

7.2.3 Detailed Specification

In addition to the current version 2.2 of the MIPI STP Specification, version 2.3 is under development in the MIPI Debug Working Group and is expected to be available in 2021. Version 2.3 will add a new Platform Description ID packet for identifying the correct decoder(s).

For details of MIPI STP, consult the document: MIPI Alliance Standard for System Trace Protocol Specification Version 2.2, ***[MIPI03]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-download-system-trace-protocol.

7.3 Trace Wrapper Protocol (TWP) Specification

7.3.1 Overview

The Trace Wrapper Protocol (TWP) enables multiple source trace streams to be combined (merged) into a single trace stream. The basic principle is that the source trace streams (byte streams) can be assigned system unique IDs. A wrapping protocol is then used to encapsulate all the streams in the system identifying them with these IDs. This protocol also includes provisions for synchronizing the merged output stream and providing inert packets for systems that cannot disable continuous export of data. It has optional facilities for indicating to the Debug and Test Controller (DTC) the position of a trigger event, which is typically used to control actions in the DTC, for example to control trace capture.

This specification is complementary to the MIPI Alliance Specification for Parallel Trace Interface (PTI), ***[MIPI02]***, and to the MIPI Gigabit Debug network adaptor specifications, such as ***[MIPI07]***. It is intended to be used by any module or layer that merges multiple trace data streams. The ultimate destination of the merged streams might include:

- Host debug tools via a dedicated trace export interface (PTI)
- On-chip capture into a dedicated trace buffer
- On-chip capture into general system memory
- Host debug tools via a functional network (GbD)

This specification is also complementary to the MIPI Alliance Specification for System Trace Protocol, ***[MIPI03]***, enabling a trace output to be shared between sources that implement STP and logic that implements other trace protocols.

This specification is equivalent to the Trace Formatter Protocol specified in the Arm® CoreSight™ Architecture Specification, ***[ARM01]***.

7.3.2 Relationship to MIPI Debug Architecture

Figure 25 shows the standard MIPI debug architecture highlighting the functional areas addressed by the TWP specification.

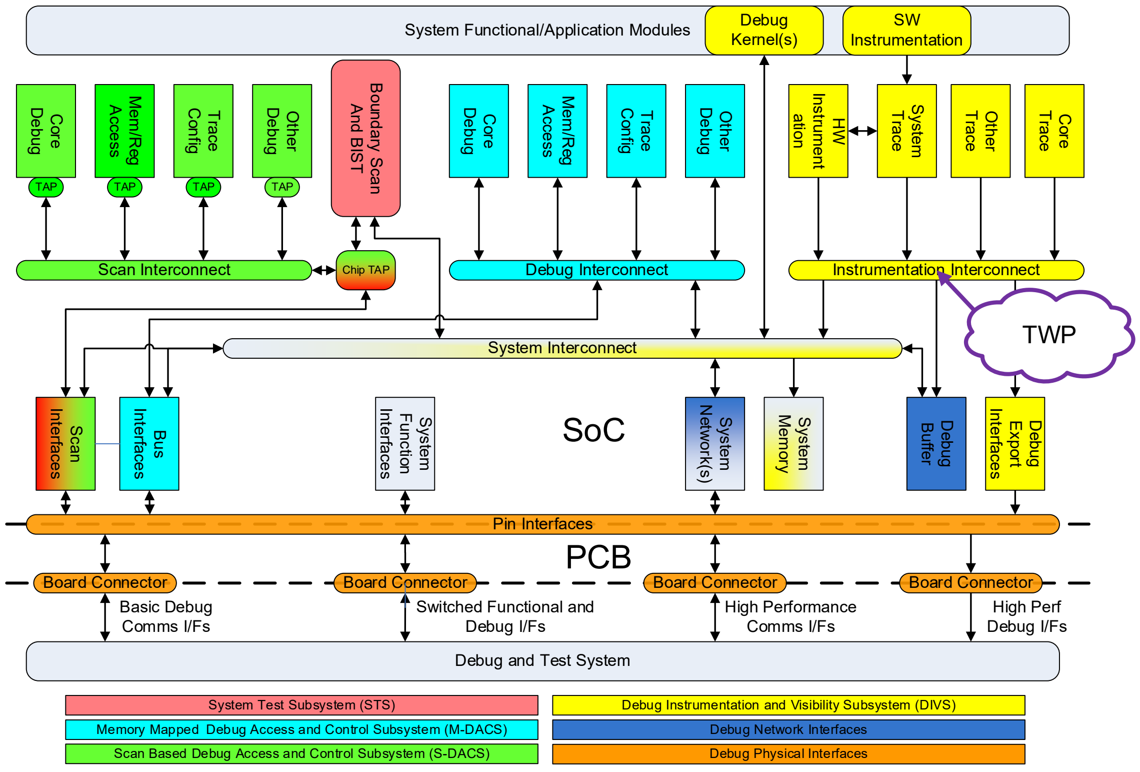


Figure 25 TWP in the MIPI Debug Architecture

7.3.3 TWP Features

The features of TWP are summarized below:

- Allows up to 111 source trace streams to be represented as a single stream and later separated by either hardware or software.
- Requires low additional bandwidth.
- Minimizes the amount of on-chip storage required to generate the protocol.
- Permits any source trace stream to be used, regardless of its data format.
- Is suitable for high-speed real-time separation of the component trace streams.
- Is a bit stream that can be exported using any transport that supports bit stream data.
- Can be efficiently stored to memory whose width is a power of two for later retrieval.
- Has facilities for synchronization points so decode can be accomplished even if the start of the trace is lost.
- Has facilities for indicating to the Debug and Test Controller (DTC) the position of a trigger event, which is typically used to control actions in the DTC, for example to control trace data capture.
- Has facilities for padding the data output for scenarios where a transport interface cannot be idled, and valid data is not available.

7.3.4 TWP Description

Each trace source, whose output is to be wrapped by TWP, is given a 7-bit trace source ID. The trace consists of a number of Trace Fragments, each consisting of an ID indicating the source of the trace and at least one byte of data.

If the source trace stream cannot be naturally represented using a stream of bytes, then an additional protocol specific to the source trace stream has to be implemented in order to convert the source trace stream into a stream of bytes.

7.3.5 Layers

TWP is split into the following layers:

- Layer T1: Flow Control. This layer enables TWP to be used over a connection which requires continuous communication, for example PTI in situations where the clock cannot be stopped.
- Layer T2: Alignment Synchronization. This layer enables the alignment of frames in Layer T3 to be determined.
- Layer T3: Data. This layer conveys trace data using 128-bit frames.

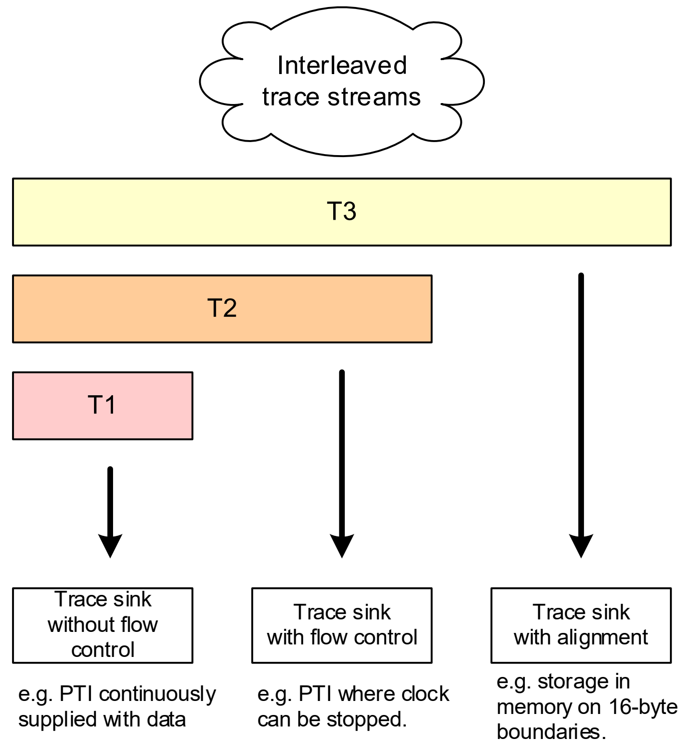

Interleaved trace streams
T3
T2
T1
Trace sink without flow control
Trace sink with flow control
Trace sink with alignment
e.g. PTI continuously supplied with data
e.g. PTI where clock can be stopped.
e.g. storage in memory on 16-byte boundaries.

Figure 26 Example Use Cases for Layers T1, T2 and T3

7.3.6 Detailed Specification

For details of MIPI TWP, consult the document: MIPI Alliance Specification for Trace Wrapper Protocol, ***[MIPI04]*** and ***[MIPI04a]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-twp-download.

7.4 Gigabit Trace (GbT)

7.4.1 Summary

One of the primary functions of the DIVS is to provide means to organize on-chip data and transport it to an external Debug and Test System for analysis. Historically, this data path used dedicated interfaces on the SoC boundary (the Parallel Trace Interfaces introduced earlier). In some system scenarios, however, it is desirable to transport the trace data via networks and interfaces which are shared with traffic sent by the mission mode (normal) functions of the device. Leveraging functional interfaces and transports for debug enhances the capabilities of the debug systems in scenarios where debug over dedicated interfaces is difficult or impossible. Gigabit Trace (GbT) focuses on the sharing of standard communication channels for debug.

The GbT architecture is a layered system. The GbT System facilitates packaging trace data as a stream of GbT Network messages suitable for transport over a shared network and/or interconnect. It defines a network independent set of data packets that are shared (but not required) by all network transports.

A Gigabit Trace system also requires a Network Adaptor that consumes GbT Network Messages and produces a message stream compatible with the targeted transport. The network adaptor layers are generally called Gigabit Debug Adaptors since they often support other network capable debug protocols like SneakPeek. The goal is to define MIPI Gigabit Debug network adaptor specifications for all the common transports found in different systems.

7.4.2 Relationship to MIPI Debug Architecture

Figure 27 shows the standard MIPI debug architecture highlighting the functional areas addressed by the Gigabit Trace specifications.

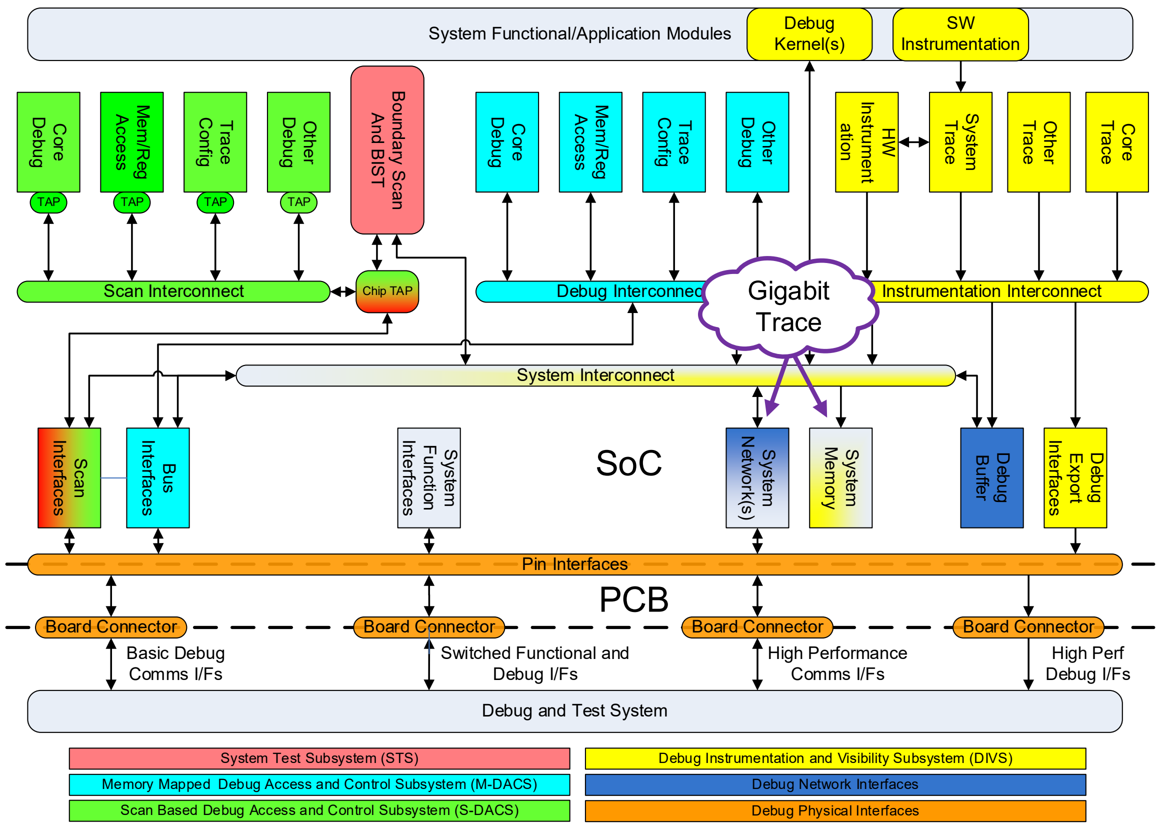


Figure 27 Gigabit Trace and the MIPI Debug Architecture

7.4.3 Gigabit Trace System Overview

The TWP has facilities to readily adapt trace streams for export over the high-speed network interfaces present on different systems. As these functional interfaces are now supporting extremely high data rates, the term Gigabit Trace (GbT) has been coined. In a GbT system, the trace stream can co-exist on the network link with other (functional) data traffic and the debug tooling is an application layer client on the network. This approach enables trace capture in fielded systems where dedicated debug connections are not available. It also enables trace capture in the DTS using any host system (such as a high-performance PC) that supports a high-speed network interface and can store data at high data rates.

Figure 28 and ***Figure 29*** show a typical GbT system and the data flow in the TS and the DTS. These figures are abstract functional diagrams that illustrate data flow through the system. The individual blocks only define functions that likely exist in the system, not HW or SW modules with defined interfaces and behaviors.

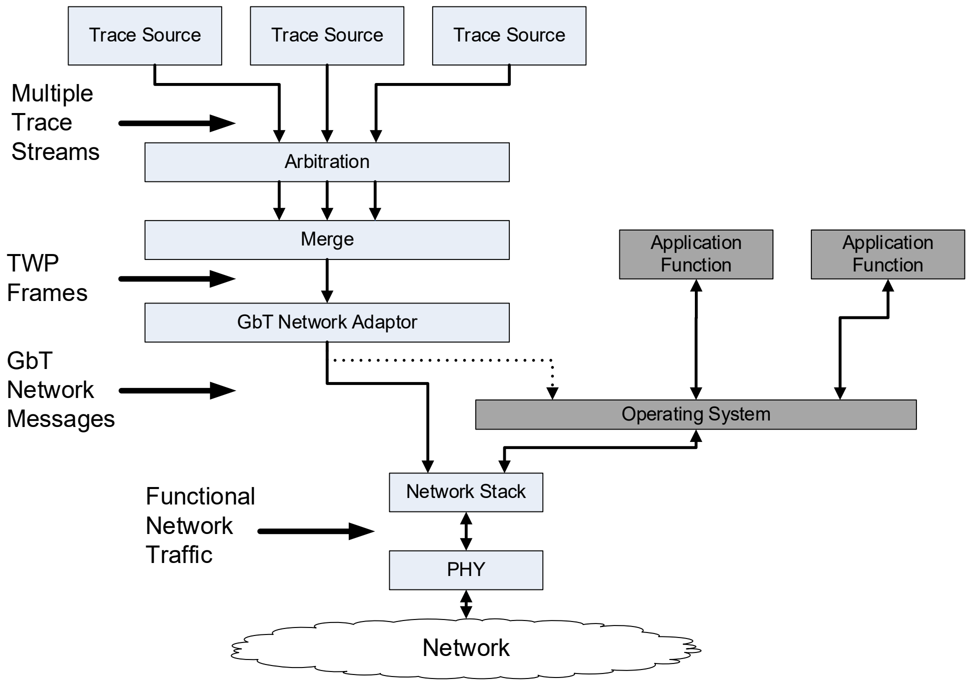


Figure 28 Typical GbT Configuration and Data Flow (TS)

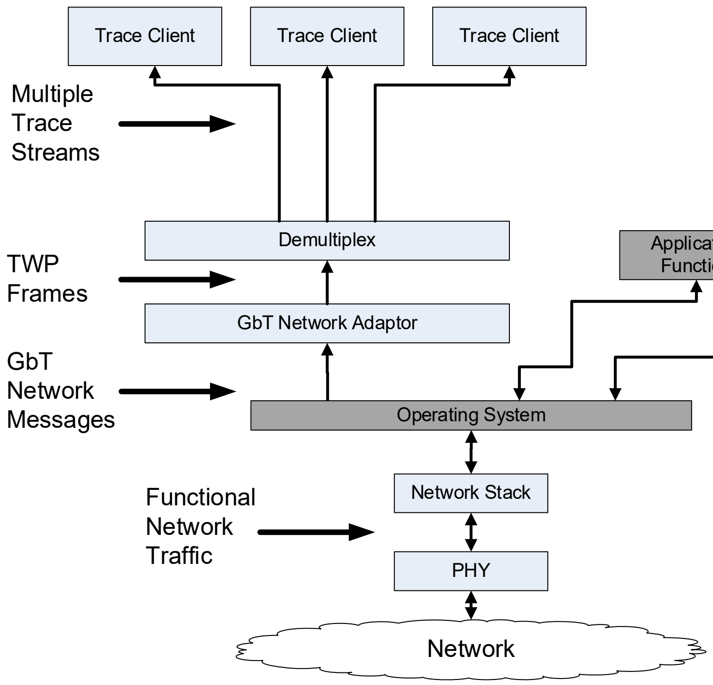


Figure 29 Typical GbT Configuration and Data Flow (DTC and DTS)

Note that in the TS, the GbT data path may optionally use the low-level OS to merge trace data with other (functional) network streams. This is obviously more intrusive to the system function than a direct data path to the lower levels of the network stack (also shown). A more SW-intensive system might ease the complexity of the HW required to support GbT and it is anticipated that both approaches will be utilized.

The MIPI GbT solution builds on the MIPI TWP data primitives. The MIPI GbT solution uses a GbT Network Adaptor (in the TS and DTS) to isolate generic GbT from the properties of a specific Network Stack. A typical GbT system might adapt trace for export over a USB interface (USB 2.0 or 3.0 depending on bandwidth requirements).

The MIPI Debug Working Group will produce independent specifications defining how a GbT system can be realized on various transport networks. These *Adaptor* specifications will provide the details on how to map the GbT framework outlined in this Annex to specific constraints and capabilities of a particular transport network.

7.4.4 Requirements Summary

A GbT system generally addresses the following requirements:

- Provides a mechanism to convey high bandwidth trace data over a transport network.
- Compatible with a variety of transport networks.
  - Packages trace data streams into network-independent messages.
- Builds on existing network protocol specifications (referred to as the functional or transport network).

7.4.5 Detailed Specification

The details of the Gigabit Trace framework are outlined in an annex to the MIPI Alliance Specification for Trace Wrapper Protocol, version 1.1.

For details of the Gigabit Trace framework, consult the document: MIPI Alliance Specification for Trace Wrapper Protocol, ***[MIPI04a]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-twp-download.

7.5 STP and TWP in the DIVS

At first inspection, it might seem that STP and TWP have significant functional overlap. Both support the merging of trace data from multiple trace sources. They also have facilities for stream synchronization and alignment. A more detailed analysis, however, reveals that the protocols are optimized for different capabilities and the differences in the protocols complement each other in a complex trace infrastructure.

TWP has a very uniform packet structure that is optimized for encoding and decoding interleaved byte streams. The protocol can be implemented easily in HW and the ability to switch active trace streams on any byte boundary decreases the amount of buffering required to support frame creation. The fixed data frame also simplifies mapping TWP to some other transport protocol payload (the GbT scenario). TWP is thus ideal for trace data paths where many high-bandwidth trace sources are merged before export on a high-performance link.

These high-throughput requirements extend into the DTS as well. The fixed frames of TWP enable efficient decode of the captured trace stream. The DTC hardware can remove lower-level link maintenance packets (synchronization and padding) before the higher-level data is stored. This type of filtering is highly desirable when supporting systems where constraints dictate that the trace interface cannot be halted (e.g., a multi-point data mode PTI).

While STP also supports merging of trace streams, the protocol also provides features that assist high-level trace protocol (e.g. time stamps, frame markers, and hierarchical source IDs). These features greatly decrease the complexity at the trace source. These sources do not have to worry about supporting their own methods of time stamping or frame marking within their own protocols. The hierarchical IDs enable support for complex trace topologies (e.g., software message traces from multiple processes on multiple CPUs). Supporting these features increases the complexity of a module merging the data from various sources into an STP stream. Since the interleaving boundary for STP is the non-fixed STP message boundary, the modules implementing STP might require significant buffering and pipelining to achieve high throughput. STP is thus ideal for trace data paths that might not have extreme bandwidth requirements but support many trace sources (such as SW threads or small HW modules) generating trace.

Figure 30 shows an example of a DIVS architecture that uses the various MIPI protocols and specifications in a layered approach to trace export. SW and HW messages, encoded as STP messages (comprised of STP packets) are transferred on the trace interconnect. High-bandwidth processor trace byte streams are also present on this interconnect. These various trace byte streams are interleaved using TWP and the packets are either exported directly to the pins or collected into GbT Network Messages for adaptation to a functional network protocol.

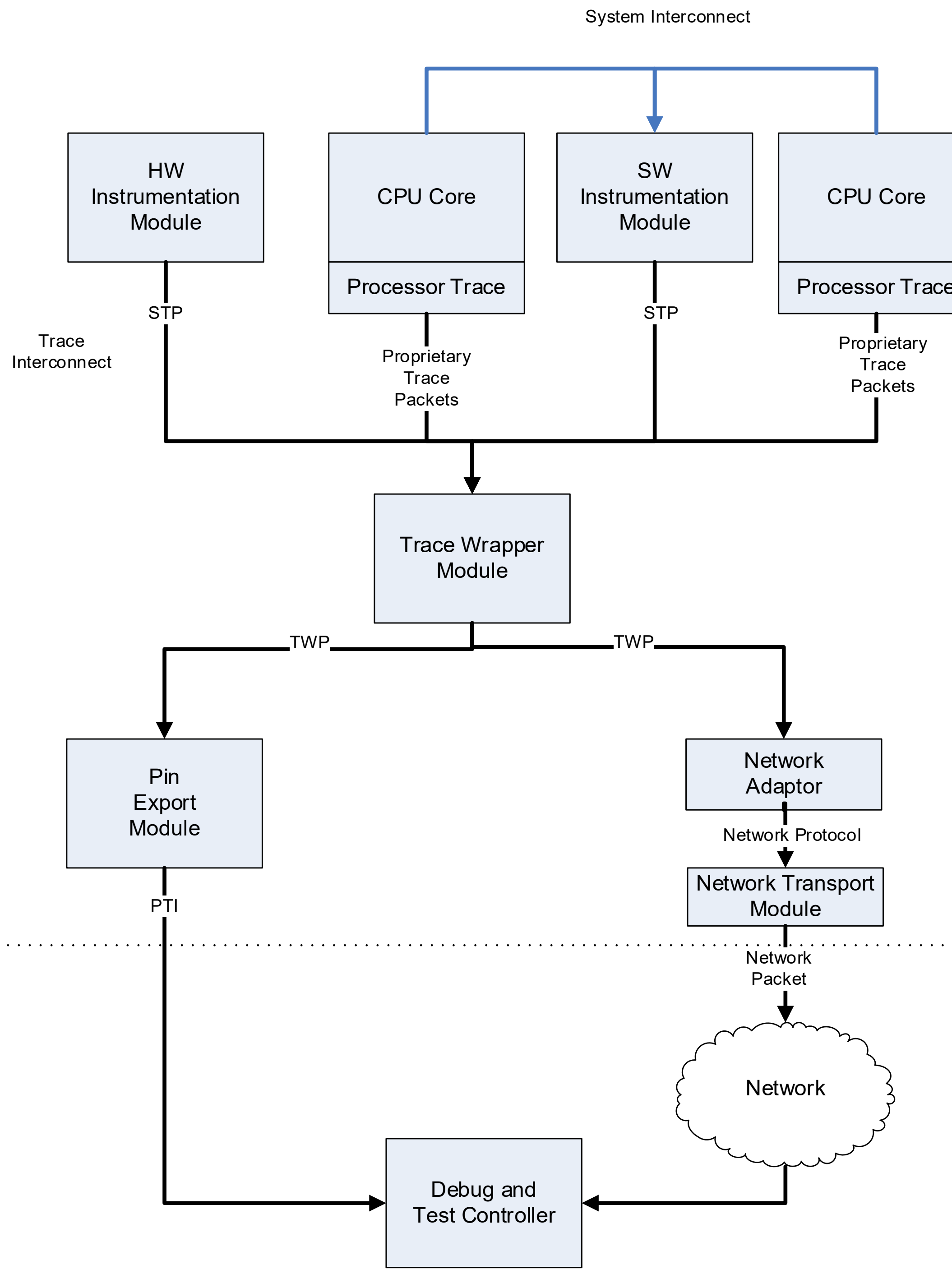


Figure 30 Example Trace Architecture

7.6 System Software Trace (SyS-T) Specification

7.6.1 Overview

System Software Trace (SyS-T) is a format for transporting software traces and debugging information between a target system (TS) running embedded software, and a debug and test system (DTS), typically a computer running one or more debug and test applications (debuggers and trace tools). SyS-T is primary an OS independent software tracing protocol, but it can also be used on bare-metal or OS environments.

The purpose of SyS-T is to provide a common trace format to exchange information between a TS and a DTS. SoCs contain many different software agents. For different operating systems there exist different, specific tracing solutions. There is no common solution existing across different software/firmware and hardware agents. MIPI SyS-T is aiming to fill this gap.

7.6.2 Relationship to MIPI Debug Architecture

Figure 31 shows the standard MIPI debug architecture highlighting the functional areas addressed by the SyS-T specification.

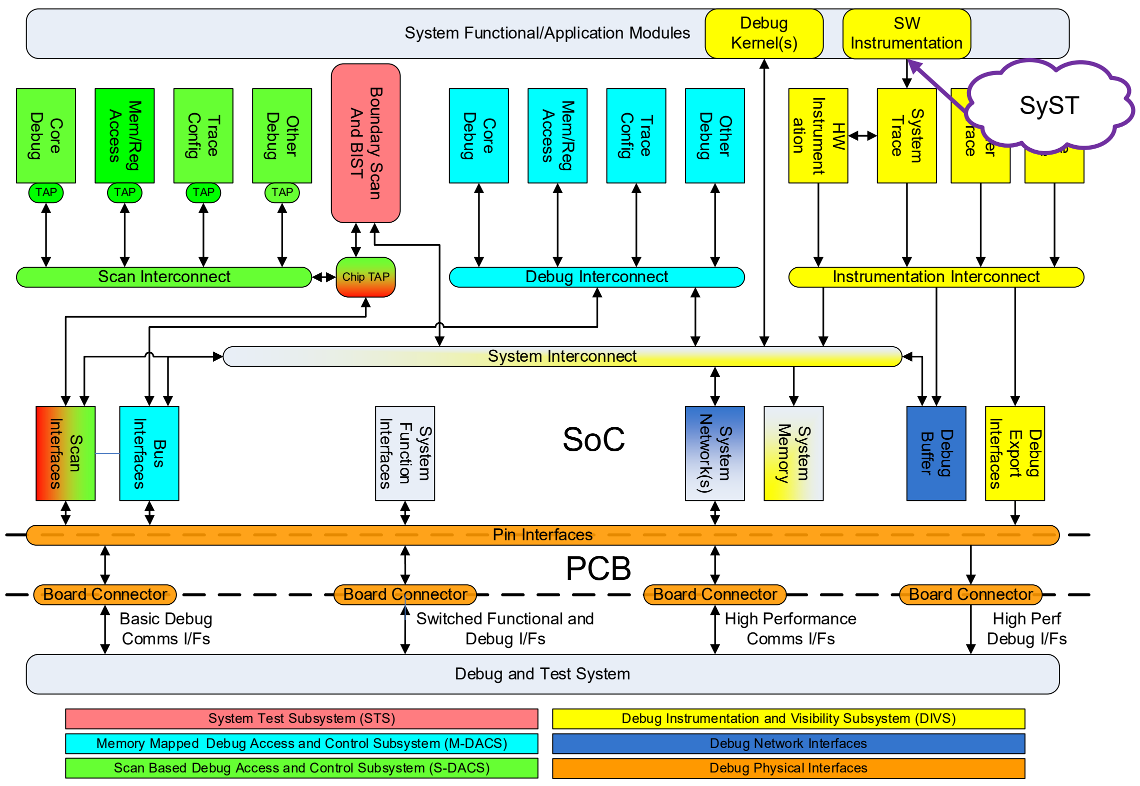


Figure 31 SyS-T in the MIPI Debug Architecture

7.6.3 Usage

SyS-T provides a platform independent general purpose trace protocol and software instrumentation library. SyS-T defines a variety of trace messages ranging from simple UTF-8 based text and printf() style messages to complex binary data payloads. SyS-T is suitable for trace data generation from non-OS, bare-metal environments, as well as OS kernel and user mode software. The SyS-T specification enables vendor independent trace debug tools development for environments that don't already provide an established trace standard. It does so by separating the trace generation on the TS from the decoding on the DTS into independent tasks.

Today's platforms or SoCs contain multiple agents that are producing traces send from a TS. Different agents can be seen as independent from each other regarding trace generation. Additional logic like a trace arbiter is used to combine the agents trace data fragments together into a single platform level data stream. SyS-T does not replace the trace arbiter step. SyS-T is used directly inside the agents for generating the SyS-T trace data the gets send to the trace arbiter. A system implementing SyS-T therefore owns one to many independent SyS-T instances, depending on how many agents are using the SyS-T tracing method.

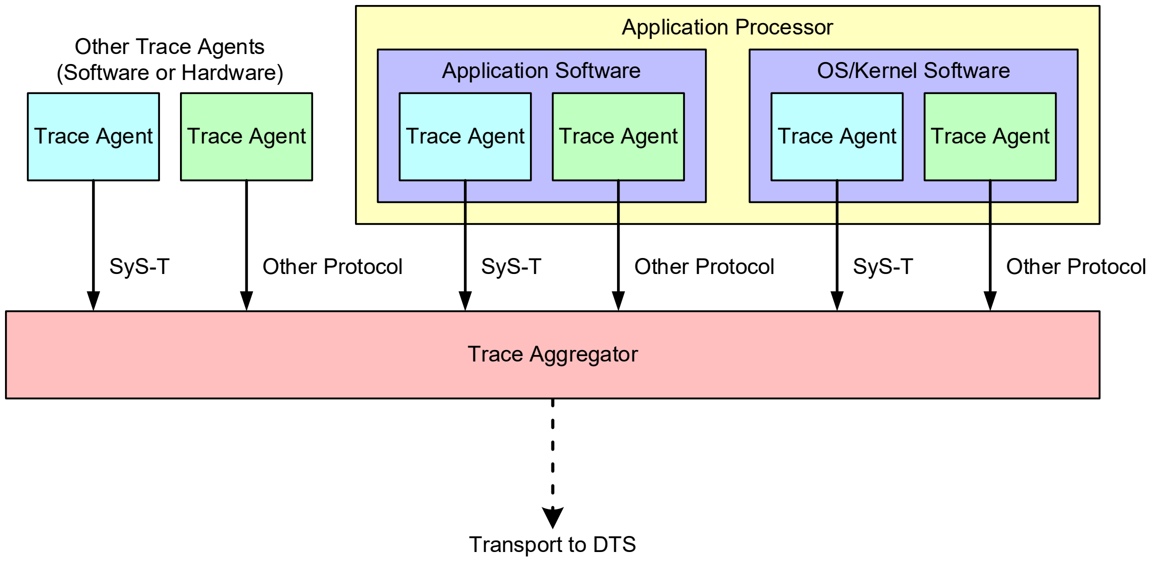


Figure 32 SyS-T Instances in a Target System

7.6.4 SyS-T Instrumentation Library

The SyS-T Data Protocol generation is provided by a portable "C"-Language based software library called SyS-T Instrumentation Library ***[MIPI11]***. The SyS-T Instrumentation Library provides a function style API (referred to as the SyS-T API) to software using pre-processor macros. This library serves as the reference implementation for a SyS-T Data Protocol generator. The usage of this library is optional. Vendor-specific implementations are allowed as long as the output is compatible with the SyS-T Data Protocol.

7.6.5 Detailed Specification

In addition to the current version 1.0 of the MIPI SyS-T Specification, version 1.1, is under development in the MIPI Debug Working Group and is expected to be available in 2021. Version 1.1 will include an additional MIPI SyS-T packet type to support sending pure binary data packets.

For details of SyS-T technology, consult: MIPI Alliance Specification for System Software Trace (SyS-T), ***[MIPI10]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-sys-t-download. In addition to the specification, the Open Source code for the SyS-T Instrumentation Library with an example implementation is posted on GitHub, ***[MIPI11]***.

This page intentionally left blank.

8 Debug Network Interfaces (DNI)

8.1 Gigabit Debug (GbD) Specification

8.1.1 Overview

Gigabit Debug (GbD) is the blanket terminology for mapping debug capabilities to a particular functional network. Unlike NIDnT, the network interface and protocol stack function normally. Gigabit Debug just defines how to adapt the SneakPeek and Gigabit Trace functions so that they can co-exist with other network traffic (as normal application layer functions). While the goal of a GbD system is to minimize intrusiveness of debug on regular system functions, it is acknowledged that some debug capabilities (like trace) may require significant network bandwidth and will thus have the potential for significant impact to the normal system.

The current effort focuses on mapping the network independent MIPI SneakPeek Protocol and Gigabit Trace framework to networks commonly found in many systems. Some of the items addressed in Gigabit Debug Specifications include:

- Connection/session initialization and de-initialization
- Network link management
- Packaging of MIPI protocol messages into network messages
- Mapping aspects of Basic Debug and Trace functionality to network features
- Network error handling

Figure 33 shows how the Gigabit Debug Adaptor specifications complement the specific MIPI debug protocol specifications.

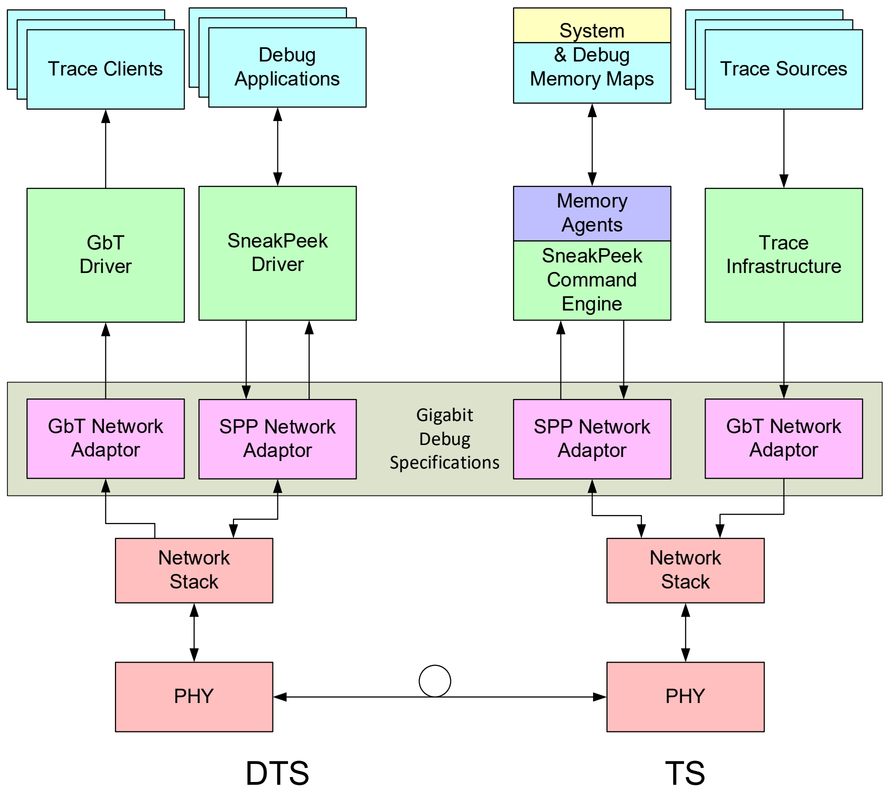


Figure 33 Gigabit Debug Functional Block Diagram

One of the fundamental features of GbD functionality is that it co-exists with non-debug network clients and can operate quite effectively in multi-node networks that are commonplace today. ***Figure 34*** illustrates how GbD and non-debug network traffic are integrated in such networks.

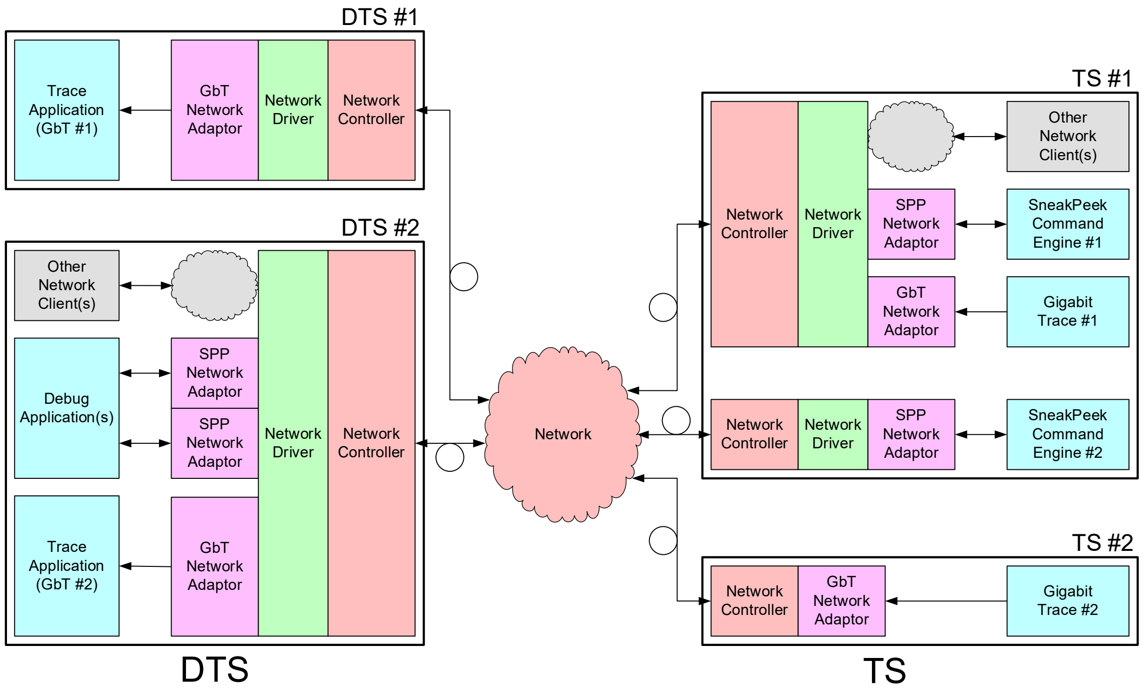


Figure 34 GbD in a Multiple-Node Network

8.1.2 Relationship to MIPI Debug Architecture

Figure 35 shows the standard MIPI debug architecture highlighting the functional areas addressed by the Gigabit Debug specifications.

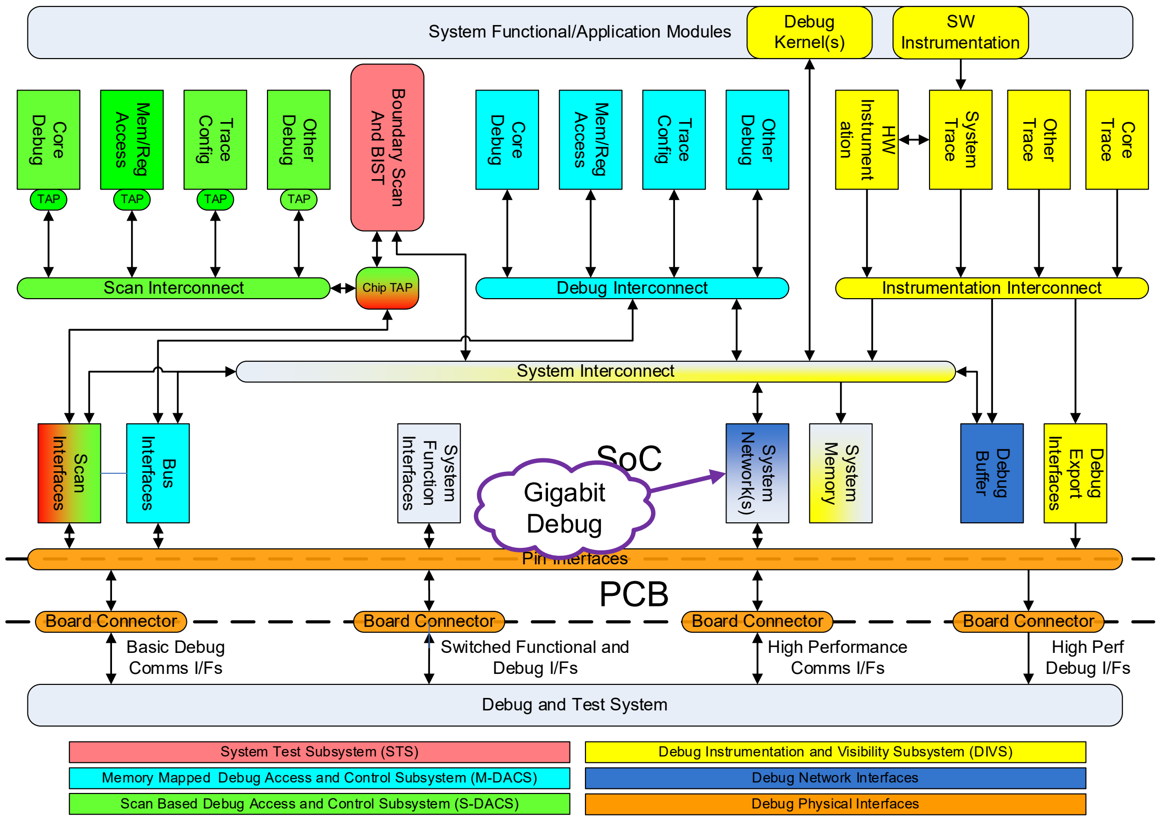


Figure 35 Gigabit Debug and the MIPI Architecture

8.1.3 Detailed Specifications

Currently, the Gigabit Debug Specification addresses the following functional networks:

- USB
  - For details of the Gigabit Debug adaptors for USB, consult the document: MIPI Alliance Specification for Gigabit Debug for USB, ***[MIPI07]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-gbd-usb-download.
  - Supports both a MIPI-defined extension to standard USB Descriptors and the Debug Device Class Descriptor as given by the Device Class Specification for Debug, ***[USB01]***.
- TCP and UDP over Internet Protocols
  - For details of the Gigabit Debug adaptors for Internet Protocol (IP) sockets, consult the document: MIPI Alliance Specification for Gigabit Debug for Internet Protocol Sockets, ***[MIPI08]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: http://resources.mipi.org/mipi-gigabit-debug-for-ips-download.

Other functional networks will be addressed in future Gigabit Debug Specifications.

8.2 Debug for I3C

8.2.1 Overview

The Debug for I3C Specification describes methods for using the Improved Inter Integrated Circuit (I3C) as a minimal-pin interface to transport debug controls and data between a DTS and a TS. Current debug solutions, such as JTAG ***[IEEE01]*** and Arm® Serial Wire Debug ***[ARM01]***, are statically structured which leads to limited scalability regarding the accessibility of debug components/devices. Also, when looking at the new requirements of near future technologies, such as 5G, and environments/markets, such as IoT, there are gaps that need to be addressed. The Debug for I3C Specification targets these gaps and shortcomings by using the capabilities of I3C to handle debug connectivity on buses that are dedicated for debug or shared with functional transfers, handling the debug network topology in a dynamic fashion.

The Debug for I3C Specification is designed in accordance with either v1.0 or greater of the MIPI Specification for I3C ***[MIPI13]***, or v1.0 or greater of the I3C Basic Specification ***[MIPI14]***. Implementations can be used with I3C interfaces that implement either specifications. The Debug for I3C Specification relies on the multi-controlling and multi-drop capabilities of the I3C Specification. The Debug for I3C Specification uses the existing common command codes (CCC) as defined by ***[MIPI13]*** as well as defining debug-specific CCC to handle debug communication and trace messaging. In-band interrupts (IBI) are also used and debug-specific Mandatory Data Byte (MDB) values are defined as a method of debug event and other communications initiated by the TS.

The Debug for I3C Specification allows for different designs where the I3C bus could be shared with non-debug communication. Whether the I3C bus is shared or dedicated for debug, the specification also allows for different DTS access points and allows for an externally connected DTS. The ability for an I3C bus to have multiple Controller-capable devices allows the DTS to be connected as either the Primary Controller (usually with dedicated debug I3C buses) or as a Secondary Controller (usually with shared I3C buses).

8.2.2 Relationship to MIPI Debug Architecture

Figure 36 shows the standard MIPI debug architecture highlighting the functional areas addressed by the Debug for I3C Specification.

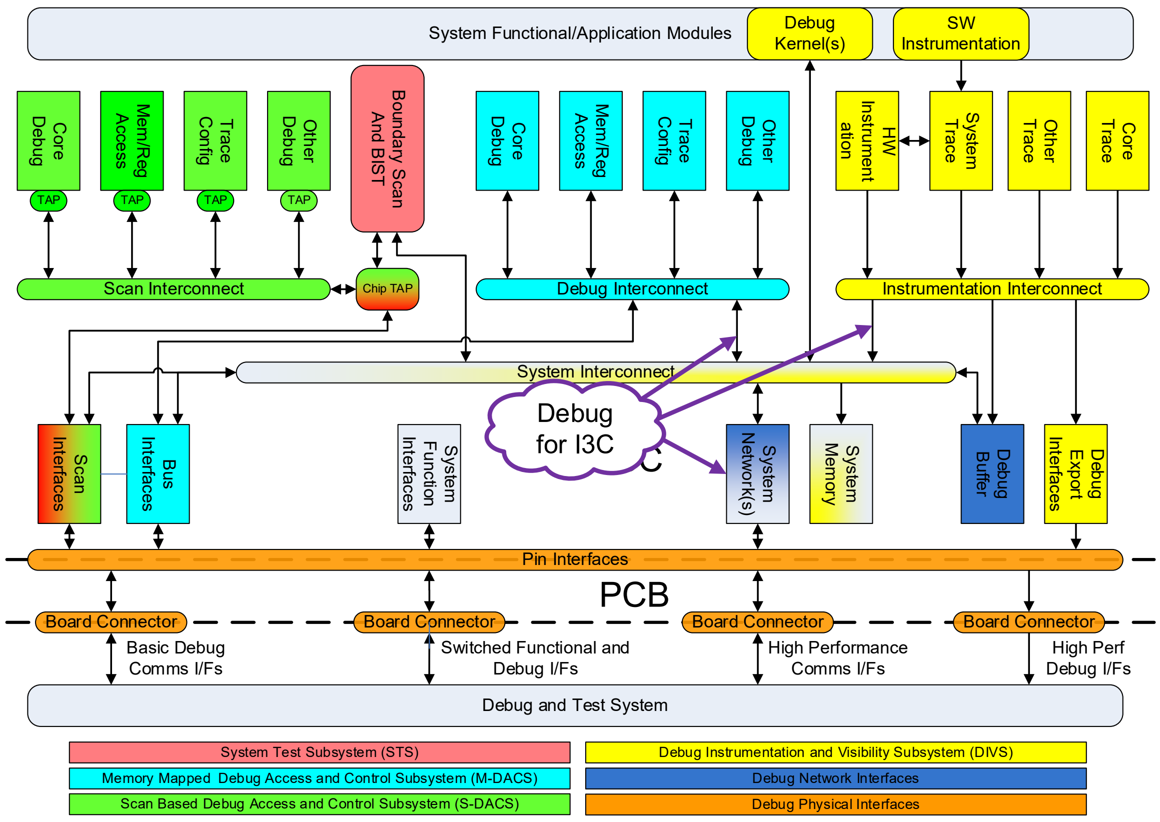


Figure 36 Debug for I3C in the MIPI Debug Architecture

8.2.3 Target System Implementation Overview

The Debug for I3C Specification allows for variations of adaptors for the different debug protocols. This specification maps the following protocols:

- **SneakPeek:** Used for debug communication with a SneakPeek Command Engine (TinySPP).
- **TWP:** Used for output of trace formatted using Trace Wrapper Protocol.
- **STP:** Used for output of trace formatted using System Trace Protocol.
- **Simplified Address-Mapped (SAM):** Simple address-mapped access to TS resources.
- **UART:** Virtual UART for character-oriented communication, e.g., `scanf()` and `printf()`, or a GDB debug monitor.
- **ImpDef:** Implementation-Defined communication protocol.

The primary function of these adaptors (referred to as a Network Adaptor) is facilitating the transport of data between debug application layer entities at opposite ends of an I3C bus. The Network Adaptor maps data objects that have meaning at the debug application layer to transport mechanisms provided by an I3C bus. The Network Adaptor takes data from a debug application layer function and passes it to an I3C protocol stack, or vice versa. This mapping preserves the relevant properties of the application data. A Network Adaptor also provides control functions connected with system setup, initialization, and operation.

Figure 37 shows how the Debug for I3C Specification complements the specific MIPI debug protocol specifications with Applications and Debug Protocols above it, and I3C transport networks below it.

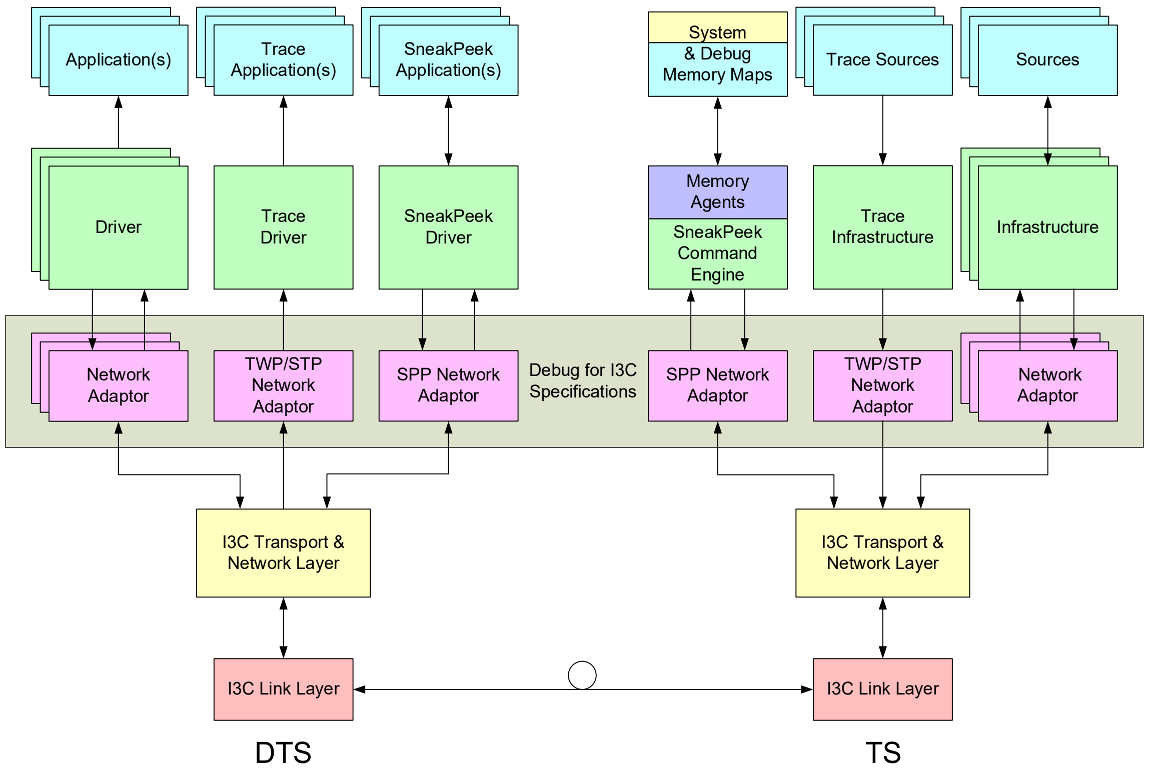


Figure 37 Debug for I3C Functional Block Diagram

Figure 38 shows an example Target System. This figure shows the standard I3C Target behavior (colored yellow), the global debug functions (colored red), four Network Adaptors (colored purple), and the accompanying debug functions (colored green for functions that are typically hardware and blue for those that are typically software).

The Network Adaptors in this system are:

- Index 0 is a SneakPeek Protocol (SPP) Network Adaptor transporting bytes formatted as Command and Response Packets for a Command Engine using the TinySPP style.
- Index 2 is a System Trace Protocol (STP) Network Adaptor transporting data from an STM which contains trace produced by a SyS-T client.
- Index 4 is a Simplified Address-Mapped (SAM) Network Adaptor transporting bytes formatted as Commands and Responses for a Debug Engine using the SAM Protocol.

- Index 5 is a Trace Wrapper Protocol (TWP) Network Adaptor transporting trace data from two trace sources that has been combined into a single stream using TWP.
- Index 14 is a UART Network Adaptor carrying character streams to and from `scanf()` and `printf()` functions in an application program.

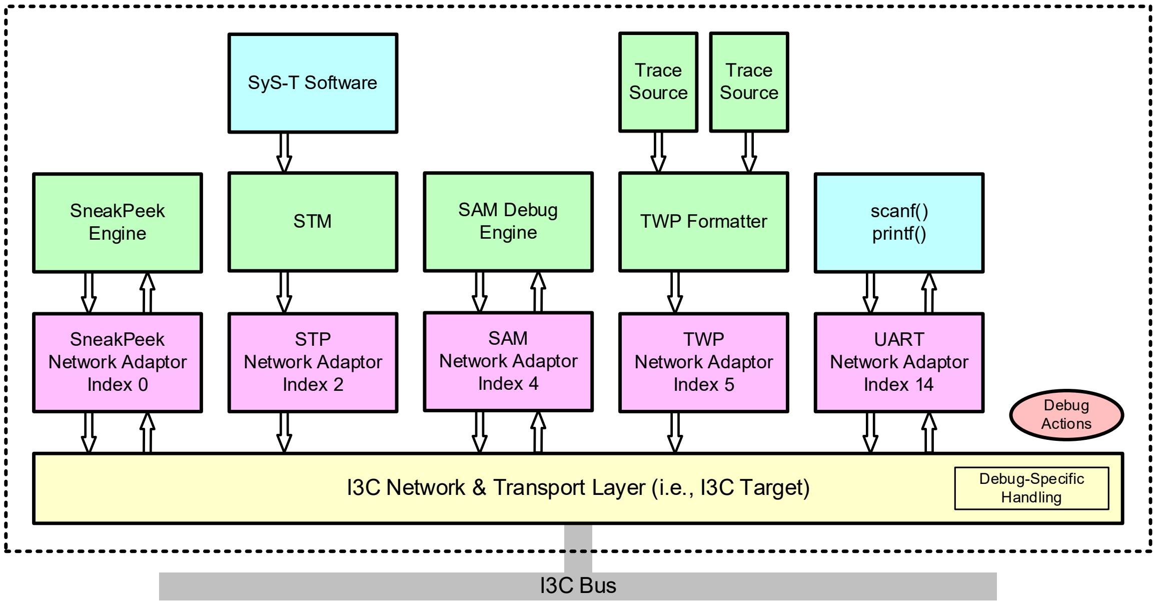

SyS-T Software
Trace Source
Trace Source
SneakPeek Engine
STM
SAM Debug Engine
TWP Formatter
scanf() printf()
SneakPeek Network Adaptor Index 0
STP Network Adaptor Index 2
SAM Network Adaptor Index 4
TWP Network Adaptor Index 5
UART Network Adaptor Index 14
Debug Actions
I3C Network & Transport Layer (i.e., I3C Target)
Debug-Specific Handling
I3C Bus

Figure 38 Example Target System with Multiple Network Adaptors

8.2.4 Detailed Specification

For details of the MIPI Debug for I3C, consult the document: MIPI Alliance Specification for Debug for I3C, ***[MIPI12]***. This specification is available to MIPI members and to the public through the MIPI website. The public version of the specification can be found at: https://resources.mipi.org/mipi-debug-i3c-download.

This page intentionally left blank.

Participants

The following list includes those persons who participated in the Working Group that developed this Supporting Document and who consented to appear on this list.

Bruce Ableidinger, SiFive
Christian Boenig, Lauterbach GmbH
Enrico Carrieri, Intel Corporation
Gary Cooper, Texas Instruments Incorporated
John Horley, Arm Limited
Jason Kirschenbaum, Intel Corporation
Rolf Kuehnis, Intel Corporation
Stephan Lauterbach, Lauterbach GmbH
Jason Peck, Texas Instruments Incorporated
Radu Pitigoi-Aron, Qualcomm Incorporated
Matthew Schnoor, Intel Corporation
Eric Upson, Intel Corporation
Dan Wetzel, Western Digital Technologies, Inc.

Past contributors to v1.2:

Eddie Ashfield, MIPI Alliance (Staff)
Enrico Carrieri, Intel Corporation
Gary Cooper, Texas Instruments Incorporated
Patrik Eder, Intel Corporation
John Horley, ARM Limited
Rolf Kuehnis, Intel Corporation
Stephan Lauterbach, Lauterbach GmbH
Andrea Martin, Lauterbach GmbH
Laura Nixon, MIPI Alliance (Staff)
Jason Peck, Texas Instruments Incorporated
Norbert Schulz, Intel Corporation

This page intentionally left blank.